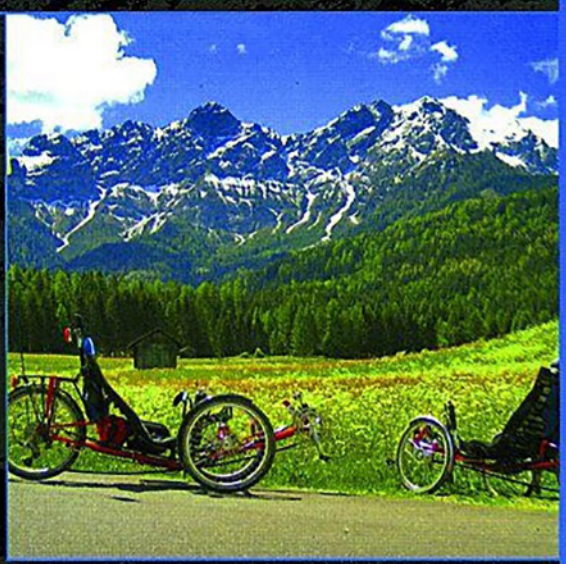
The Addison-Wesley Signature Series

A Kent Beck Signature Book

Бережливое производство программного обеспечения

От идеи до прибыли

Мэри и Том Поппендик



Предисловия Джеффа Сазерленда и Кента Бека

Бережливое производство программного обеспечения

От идеи до прибыли

Implementing Lean Software Development

From Concept to Cash

Mary and Tom Poppendieck

ADDISON-WESLEY

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokio • Singapore • Mexico City

Бережливое производство программного обеспечения

От идеи до прибыли

Мэри и Том Поппендик

ВИЛЬЯМС

Москва · Санкт-Петербург · Киев
2010

ББК 32.973.26-018.2.75
П58
УДК 681.3.07

Издательский дом "Вильямс"

Зав. редакцией *С.Н. Тригуб*

Перевод с английского и редакция *О.А. Меженного*

По общим вопросам обращайтесь в Издательский дом "Вильямс" по адресу:
info@williamspublishing.com, http://www.williamspublishing.com

Поппендик, Мэри, **Поппендик**, Том.
П58 Бережливое производство программного обеспечения: от идеи до прибыли. : Пер. с англ. — М. : ООО "И.Д. Вильямс", 2010. — 256 с. : ил. — Парал. тит. англ.

ISBN 978-5-8459-1538-2 (рус.)

ББК 32.973.26-018.2.75

Научно-популярное издание

Мэри Поппендик, Том Поппендик

Бережливое производство программного обеспечения: от идеи до прибыли

Литературный редактор *Е.П. Перестюк*
Верстка *О.В. Романенко*
Художественный редактор *В.Г. Павлютин*
Корректор *Л.А. Гордиенко*

Подписано в печать 26.03.2009. Формат 70x100/16.
Гарнитура Times. Печать офсетная.
Усл. печ. л. 20,64. Уч.-изд. л. 16,9.
Тираж 1000 экз. Заказ № 0000.

Отпечатано по технологии CtP
в ОАО "Печатный двор" им. А. М. Горького
197110, Санкт-Петербург, Чкаловский пр., 15.

ООО "И. Д. Вильямс", 127055, г. Москва, ул. Лесная, д. 43, стр. 1

ISBN 978-5-8459-1538-2 (рус.)
ISBN 0-321-43738-1 (англ.)

Оглавление

Предисловие **15**
Введение **19**

Глава 1. История **23**
Глава 2. Принципы **41**
Глава 3. Ценность **63**
Глава 4. Потери **85**
Глава 5. Скорость **113**
Глава 6. Люди **131**
Глава 7. Знание **159**
Глава 8. Качество **181**
Глава 9. Партнеры **209**
Глава 10. Путешествие **223**

Список литературы **245**
Предметный указатель **252**

Содержание

Предисловие **15**

Введение **19**

Глава 1. История **23**

Взаимозаменяемые детали 23
Взаимозаменяемые люди 24
Семейство Тойода 24
Производственная система Toyota 26
Тайити Оно 26
Поставки “точно вовремя” 27
Автоматизация 27
Сигаэо Синго 27
Производство без запасов 28
Производство без проверок 28
“Точно вовремя” 28
Концепция бережливости 32
Бережливое производство и бережливые операции 33
Бережливая цепь поставок 33
Бережливая разработка продуктов 34
Бережливый подход к созданию программного обеспечения 38
Попробуйте это 38

Глава 2. Принципы **41**

Принципы и практика 41
Разработка программного обеспечения 42
Разработка 42
Программное обеспечение 43
Семь принципов бережливого подхода при разработке ПО 44
Принцип 1: ликвидировать потери 44
Миф: созданная заранее спецификация сокращает потери 46
Принцип 2: встраивать качество 47
Миф: цель тестирования — выявление дефектов 49
Принцип 3: создавать знание 50
Миф: прогнозы обеспечивают предсказуемость 51
Принцип 4: откладывать необратимые решения 52
Миф: план — это обязательство 53

Принцип 5: доставлять быстро 54
Миф: спешка ведет к браку 54
Принцип 6: уважать людей 55
Миф: существует наилучший метод 57
Принцип 7: оптимизировать целое 57
Миф: оптимизировать путем декомпозиции 59
Попробуйте это 60

Глава 3. Ценность 63

Бережливые решения 63
От концепции до прибыли 66
Концепция 66
Осуществимость проекта 66
Пробная версия 67
Прибыль 68
Восхищенные потребители 68
Глубокое понимание потребителя 69
Фокусироваться на работе 70
Ориентированная на потребителя организация 71
Руководство 72
Главный инженер 72
Команда руководителей 74
Совместное руководство 74
Кто за все отвечает 75
Многофункциональные коллективы 76
Технологичность 77
Разработка на заказ 78
От проектов к продуктам 78
Сотрудничество между ОИТ и бизнесом 80
Ответственность 81
Попробуйте это 82

Глава 4. Потери 85

Меньше программного кода 85
Zara 85
Сложность 87
Каждая функциональная возможность должна быть оправдана 87
Минимально-полезные наборы функций 89
Не следует автоматизировать запутанные процессы 89
Семь непроизводительных расходов 90
Частично выполненная работа 91
Избыточные функциональные возможности 92
Повторное приобретение знания 93
Передача работы 93
Переключение между задачами 94
Задержки 96

Дефекты 97
Картирование потока создания ценности 98
Подготовка 99
Выбор процесса 99
Выбор начала и останова отсчета времени 99
Необходим ответственный 100
Не нужно усложнять 100
Примеры 101
Пример 1 101
Пример 2 103
Пример 3 105
Пример 4 107
Итоги 109
Перспективные карты потока создания ценности 110
Попробуйте это 110

Глава 5. Скорость 113

Поставлять быстро 113
PatientKeeper 113
Время — универсальная валюта 116
Теория очередей 117
Закон Литтла 118
Вариация и загрузка 118
Сокращение времени цикла 119
Добиться ритмичности поступления работы 120
Свести к минимуму количество одновременно выполняющейся работы 122
Минимизировать размеры объектов, одновременно находящихся в производстве 123
Создать постоянный ритм работы 124
Объем работы не должен превышать возможности организации 125
Использование планирования с “вытягиванием” 127
Резюме 129
Попробуйте это 130

Глава 6. Люди 131

Система менеджмента 131
Boeing 777 131
У. Эдвардс Деминг 134
Почему хорошие начинания оказываются неудачны 137
Коллективы 138
Что такое коллектив? 139
Компетентность 141
Руководство 143
Основанные на ответственности планирование и управление 144
Визуальная рабочая среда 146
Самоуправляющийся производственный процесс 147
Канбан 148

Андон 149
Наглядное информирование 150
Побудительные мотивы 151
Оценка эффективности 151
Система оценок 152
Вознаграждение 153
Совет 1: обеспечить, чтобы обоснованность повышений была неоспорима 153
Совет 2: не придавать особого значения ежегодным повышениям зарплаты 153
Совет 3: награждать в зависимости от вклада в общий результат, а не за личные достижения 154
Совет 4: найти лучшую мотивацию, чем деньги 155
Попробуйте это 156

Глава 7. Знание 159

Создание знания 159
В чем ваша проблема? 159
Научный метод мышления 160
Сохранение знания 161
Отчеты A3 163
Век Интернета 164
“Точно вовремя ” 165
Параллельная разработка альтернатив 165
Пример 1: интерфейс для медицинского устройства 167
Пример 2: компенсация эффекта красных глаз 168
Пример 3: сменные интерфейсы 168
Почему это не потери 169
Рефакторинг 169
Унаследованные системы 171
Решение проблем 172
Упорядоченный подход 173
1. Обозначить проблему 173
2. Проанализировать ситуацию 174
3. Создать гипотезу 175
4. Провести эксперименты 176
5. Оценить результаты 176
6. Стандартизировать метод и многократно его применять 177
Кайдзен-мероприятия 177
Решение проблем с участием больших коллективов 177
Попробуйте это 179

Глава 8. Качество 181

Обратная связь 181
Программа Polaris 181
Планирование релизов 183
Архитектура 185
Итерации 186

Подготовка 187
Планирование 189
Реализация 190
Оценка 191
Вариация: интерфейс пользователя 192
Дисциплина 192
Пять “S” 194
Стандарты 196
Экспертиза программного кода 197
Парное программирование 198
Защита от ошибок 199
Автоматизация 200
Разработка через тестирование 201
Блочные тесты 201
Тесты “историй” 202
Тесты на простоту использования и диагностические тесты 203
Тестирование качества 203
Управление конфигурацией 203
Непрерывная интеграция 204
Вложенная синхронизация 205
Попробуйте это 206

Глава 9. Партнеры 209

Совместные усилия 209
Чрезвычайное происшествие! 209
Open Source 211
Глобальные сети 212
Аутсорсинг 215
Инфраструктура 215
Транзакции 216
Разработка 216
Контракты 218
Соглашение T5 218
Контракт PS 2000 219
Реляционные контракты 220
Попробуйте это 221

Глава 10. Путешествие 223

Куда бы вам хотелось отправиться ? 223
Компьютер на колесах 224
Долговременная перспектива 225
Люди важнее всего 227
Чему мы научились ? 228
Шесть сигм 228
Руководители производства и неофициальные лидеры 229
Средства и результаты 229

Теория ограничений 229
Критическая цепь 231
Привычка приспосабливаться 232
Гипотеза 233
Обучение 233
Стремление улучшать процессы 235
Критерии 236
Время цикла 237
Финансовая отдача 238
Удовлетворение заказчика 239
Дорожная карта 240
Попробуйте это 241
Оптимизировать целое 241
Уважать людей 241
Доставлять быстро 242
Не торопиться брать обязательства 242
Создавать знание 243
Встраивать качество 243
Ликвидировать потери 244

Список литературы 245

Предметный указатель 252

Нашим родителям:
Джону и Мардж Браст,
а также
Элмеру и Рут Поппендик

Отзывы о книге

“Эта замечательная книга объединяет практические советы, готовые к использованию методы и глубокое понимание, почему именно так следует создавать программное обеспечение. Мне приходилось наблюдать коллективы, полностью трансформировавшиеся благодаря идеям этой книги.”

—Майк Кон (Mike Cohn), автор книги *Agile Estimating and Planning*

“Я, сам практиковавший бережливый подход, обращался к первой книге этих авторов не раз. Когда вышла вторая книга, я был восхищен: она оказалась еще лучше. Если вас интересует, как принципы бережливости могут быть использованы в организации, занятой разработкой программного обеспечения, эта книга как раз для вас. Мэри и Том Поппендик предлагают читателям приятную смесь истории, теории и практики.”

—Алан Шеллоуэй (Alan Shalloway), соавтор книги *Design Patterns Explained*

“Я с большим удовольствием прочел данную книгу. Кажется, она еще лучше первой книги Тома и Мэри, хотя та книга была исключительно хороша! Мэри обладает глубокими познаниями в применении бережливых методов при разработке продукции и в производстве. Это редкий случай, когда рассматривается применение данных методов к разработке программного обеспечения. Этого не встретишь ни в одной другой книге (за исключением первой книги этих авторов).”

—Бас Водд (Bas Vodde)

“Эта новая книга Мэри и Тома Поппендик является хорошо написанным и понятным введением в принципы бережливости и доказавшую свою эффективность практику для менеджеров и инженеров, занятых созданием программного обеспечения. Упомянутые принципы и практика иллюстрируются очень удачными примерами успешного их применения. Я с удовольствием прочел данную книгу.”

—Роман Пичлер (Roman Pichler)

“В своей новой книге Мэри и Том Поппендик рассматривают более глубоко темы, с которыми они познакомили читателя в их первой книге *Lean Software Development*. Они начинают с убедительной и увлекательной истории о применении бережливых принципов, а затем переходят к обсуждению таких ключевых тем, как потребительская ценность, потери и люди. Каждая глава включает упражнения, призванные помочь читателю научиться применять изложенные методы. Если вы хотите лучше понять, как принципы бережливости используются при создании программного обеспечения, это книга для вас.”

—Билл Уэйк (Bill Wake), независимый консультант

Предисловие

Джефф Сазерленд (Jeff Sutherland)

Впервые я начал применять методологию Scrum[1] в 1993 году в Easel Corporation в Барлингтоне (Burlington), Массачусетс[2], в сотрудничестве с нашим генеральным директором (CEO). В 1995 году я начал работать с Кеном Швайбером (Ken Schwaber), который формализовал процесс дистрибуции вновь созданного программного обеспечения по всему миру[3]. В четырех компаниях, где мне пришлось работать после Easel Corporation, я использовал природу самоорганизации Scrum для перехода от Type A Scrum к Type B Scrum и далее к Type C Scrum.

В 2000 году мне удалось применить методологию Scrum в компании PatientKeeper[4] и, как выяснилось, данная методология прекрасно сочетается с бережливым процессом разработки. С годами мы сократили время цикла до устраивающего заказчиков срока: неделя для срочных работ, месяц для мелких дополнений программного обеспечения и три месяца для значительных новых продуктов.

В PatientKeeper вся компания функционирует с применением методологии Scrum; каждый понедельник имеют место собрания MetaScrum[5], на которых принимаются решения, влияющие на управление, адаптацию, самоорганизацию и развитие компании. Подобным собранием всегда руководит Product Owner (или менеджер продукта) и на них присутствуют все учредители компании, а также генеральный директор. Бережливые концепции здесь тщательно рассматриваются и итерации Sprint начинаются, останавливаются или изменяются только на таких собраниях. В соответствии с решениями, принимаемыми на этих еженедельных собраниях MetaScrum, могут иметь место глобальные изменения, затрагивающие всю компанию (а также заказчиков, которых это касается).

Журнал предстоящей работы (product backlog) подготавливает менеджер продукта вместе с коллективом. Пятнадцатиминутные ежедневные совещания Scrum of Scrums проводит главный ScrumMaster. На подобных совещаниях рассматривается, что коллектив делал вчера, чем он будет заниматься сегодня и что мешает выполнению работы? Благодаря этим кратким собраниям, а также автоматизированной системе слежения за состоянием журналов предстоящей работы одновременных параллельных Sprint, нам удавалось осуществлять 45 релизов программного обеспечения ежегодно, предназначен-

---

[1] Scrum — одна из самых популярных методологий гибкой разработки программного обеспечения.

[2] Sutherland J. Agile Development: *Lessons Learned from the First Scrum*, Cutter Agile Project Management Advisory Service: Executive Update, 5(20), pp. 1-4. — *2004.*

[3] Schwaber K. *Scrum Development Process*, OOPSLA Business Object Design and Implementation Workshop, Springer: London, *1997.*

[4] Более подробно об этой компании речь пойдет в главе 5.

[5] Sutherland J. Future of Scrum: *Parëallel Pipelining of Sprints in Complex Projects*, AGILE 2005 Conference. *2005.*

ных для крупных, решающих важные задачи предприятий. Даты (очередных релизов) сообщались заказчикам до начала Sprint, и тысячи веб-пользователей и сотни врачей с PDA имели возможность оценить новое ПО к концу каждой Sprint. Для программы PatientKeeper интервал от концепции (или от журнала предстоящей работы) до получения прибыли (или продукта у потребителей) составил один месяц. Нам пришлось ликвидировать потери повсеместно — до, во время и после процесса реализации.

В 1993 году бережливый подход к разработке программного обеспечения не пользовался большой популярностью ни в одной отрасли. Методология Scrum явилась первой конкретной реализацией бережливого мышления для разработки ПО, позволившей различным компаниям организовать работу коллективов разработчиков на основе принципов бережливости и с использованием легкой в понимании стандартной модели. Основная трудность при этом заключалась в том, что трудно было объяснить, как реализовать эту модель, чтобы добиться стабильного повышения качества и производительности.

Сегодня печатные издания и семинары от Мэри и Тома Поппендик предоставляют проверенный практикой набор принципов, призванных помочь организациям внедрять предлагаемые средства, приемы и методы в своей уникальной среде и с учетом своих возможностей. Мы теперь можем объяснить, как использовать методологию Scrum для применения принципов бережливости в разработке программного обеспечения. Помимо моей компании, PatientKeeper, я использовал процедуры и процессы, описанные в этой книге для обучения (как оптимизировать Scrum) специалистов-практиков по всему миру.

Бережливый подход к созданию ПО рассматривает все гибкие (agile) методы в качестве эффективных, доказавших свою пригодность применений бережливого мышления. Однако данный подход предоставляет более широкую перспективу, позволяющую гибким методам быстро развиваться. Во-первых, бережливый подход позволяет держать под контролем всю цепочку создания ценности (от концепции до прибыли), а также выявлять все случаи потерь и задержек, имеющих место до и после разработки кода. Во-вторых, он создает менеджмент-среду, делающую возможным развитие гибких методов создания программного обеспечения. В-третьих, бережливый подход предоставляет набор доказавших свою эффективность принципов, которые каждая организация может использовать для внедрения предлагаемых средств, приемов и методов в своей уникальной среде и с учетом своих возможностей.

Каждая глава этой книги иллюстрирует набор принципов, применение которых позволит организовать более продуктивную работу коллективов разработчиков. Если вы хотите быть так же эффективны, как компания Toyota, где производительность вчетверо выше, чем у конкурентов (а качество в двенадцать раз лучше), вам не обойтись без этих принципов. Использование этих принципов сделает тщетной конкуренцию с вами и бесспорной вашу победу на рынке. Отдача от инвестиций при использовании практики, описанной в данной книге, может быть очень высокой.

Чтобы воспользоваться преимуществами бережливого подхода, мы полагаем, вам следует систематически применять соответствующие принципы на практике. Иными словами, вы должны глубоко понять японские концепции Muri, Mura и Muda. Авторы, Мэри и Том, представили их в своей книге в виде семи принципов бережливости и семи основных случаев потерь при разработке программного обеспечения, чтобы помочь читателю понять, как они работают и что нужно делать.

Концепция Muri связана с загрузкой системы, а Mura предписывает никогда не оказывать давление на индивидуума, систему или процесс. Тем не менее многие менеджеры

стремятся загрузить разработчиков на 110%. Они прилагают все усилия, чтобы создать атмосферу "срочности" и разработчики "работали напряженнее". Они пытаются контролировать каждый шаг коллективов, что вредит их самоорганизации. Эти трудно постижимые идеи часто порождают задержки, необходимость переделывать уже сделанную работу и неудачные проекты.

Когда я спрашивал таких менеджеров, пытаются ли они загрузить свой ПК или ноутбук на 110 процентов, они дружно отвечали: "Конечно нет. Мой компьютер перестанет работать, если попытаться это сделать.". Из-за перегрузки людей проекты часто задерживаются, программное обеспечение получается некачественным и его трудно поддерживать, и дела, в целом, идут все хуже. Необходимо понять, что в компании Toyota и в методологии Scrum используется система с "вытягиванием" (pull system), предписывающая избегать стрессов (Mura) и устранять "узкие места" (Muri). Разработчики берут из журнала предстоящей работы (Product Backlog) то, что им нужно, и когда нужно. Они выберут из журнала только ту работу, которая готова для Sprint, и они никогда не возьмут больше, чем они смогут выполнить в данной Sprint. Ускорить работу можно, выявляя и устраняя помехи и работая с менеджментом с целью ликвидации потерь (Muda). Если устранить потери, это уменьшит объем ожидающей выполнения работы. В результате производительность возрастает, и у разработчиков появляется время для повышения качества программного обеспечения и уделения большего внимания нуждам заказчиков.

Таким образом, оптимизируя загрузку системы (Muri) и избегая стрессов (Mura), вы сокращаете время цикла. Это делает очень заметными недостатки, вызывающие потери (Muda). Если устранить эти недостатки, коллектив разработчиков будет работать быстрее, будет делать больше при меньших затратах, повысит качество и создаст как раз такой продукт, который нужен заказчику.

Я рекомендую читателю постоянно держать книгу Мэри и Тома Поппендик на рабочем столе и пользоваться ею регулярно при реализации принципов бережливости. При некоторой настойчивости вы быстро сможете удвоить производительность, устраняя потери (и уменьшая при этом объем работы), а затем удвоите ее еще раз, устраняя разного рода помехи. Достигнув четырехкратного увеличения производительности (с помощью описанных в книге методов), вы быстро добьетесь и двенадцатикратного повышения качества, как компания Toyota.

— Джефф Сазерленд, Ph.D
Технический директор (Chief Technology Officer) компании PatientKeeper,
сертифицированный ScrumMaster,
один из создателей методологии разработки Scrum,
июль, 2006 год.

Кент Бек (Kent Beck)

Разработка программного обеспечения (ПО) — это цепь со множеством звеньев; повышение общей эффективности требует пристального взгляда на всю цепь. Данная книга предназначена для людей, имеющих отношение к созданию ПО, — не только для программистов, но также менеджеров, спонсоров, заказчиков, тестировщиков и проектировщиков. Изложенные здесь принципы в конечном счете касаются всех, связанных с разработкой программного обеспечения. Книга обращена к тем, кто готов взглянуть на процесс разработки как на единое целое.

Чтобы идеи представляли какую-то ценность, они должны основываться как на теории, так и на практике. Данная книга переносит хорошо зарекомендовавшие себя на производстве теорию и практику бережливости в разработку программного обеспечения. Многие фундаментальные проблемы производства являются также проблемами разработки ПО. Речь идет о неопределенностях и изменениях, постоянно совершенствующихся процессах и создании ценности.

Что эта книга может предложить? Во-первых, многообразие теорий, предоставляющих альтернативные подходы к совершенствованию процесса разработки ПО. Во-вторых, множество историй о применении этих теорий к реальным проектам. В-третьих, провоцирующие вопросы, призванные помочь читателю применять изложенные теории.

Мэри обладает уникальной квалификацией для написания этой книги. Она является опытным экспертом в области бережливого производства, знакомым с предметом не понаслышке (однажды, оказав помощь в преобразовании использовавшихся технологических процессов, ей удалось спасти от закрытия завод). Она также опытный разработчик программного обеспечения (как программист и менеджер). Мне однажды пришлось совместно с Мэри проводить семинар,и было удовольствием наблюдать, силу и уверенность, с которыми она подавала материал. Ее голос как будто слышится на страницах этой книги.

По сравнению с предыдущей книгой Мэри и Тома Поппендик, *Lean Software Development* данная книга предлагает новый взгляд на реализацию данного подхода. Здесь повторено изложение теории из предыдущей книги, однако с акцентом на применение идей к реальным ситуациям. В результате мы получили интересно написанное практическое руководство по применению идей, потенциально способных помочь вам трансформировать ваш процесс разработки.

Данная книга предназначена для практиков, стремящихся, чтобы их действия имели максимальный эффект. Если вы относите себя к этой категории, я рекомендую вам эту книгу в качестве источника идей и энергии для перемен к лучшему.

— Кент Бек,
Three Rivers Institute,
июль, 2006 год.

Введение

Идея *бережливости* пришла к нам из 90-х годов и нашла свое отражение в книге *Lean Software Development*, которую мы написали в 2003 году. Мы заметили, что для передовых идей, нашедших применение в производстве и логистике, часто требуется десятилетие или два, чтобы быть использованными в процессах разработки. Поэтому мы подумали, что еще не поздно воспользоваться хорошо себя зарекомендовавшими концепциями 80-х и 90-х годов, чтобы объяснить, почему гибкие (agile) методы являются очень эффективным подходом в разработке программного обеспечения.

Данная стратегия сработала. Книга *Lean Software Development* представляет собой описание набора методов, базирующихся на бережливом мышлении, которое руководители и менеджеры продолжают находить полезным для понимания гибких методов создания программного обеспечения. Книга была приобретена многими разработчиками, которые затем дали ее почитать своим менеджерам, и многие менеджеры затем распространили множество ее копий среди коллег, надеясь, что это поможет осуществить переход к бережливым и гибким (lean/agile) методам разработки ПО.

Тем временем с идеей произошло нечто необычное. За последние пару лет бережливые инициативы пережили небывалый рост популярности. Слово *lean* (бережливость), которое первоначально приобрело популярность в 90-х годах, использовалось для обозначения японского подхода к производству автомобилей[1]. В последние годы деятельность компаний Honda и Toyota на рынке США была очень успешной, в то время как автопроизводители из Детройта были заняты реструктуризацией. Например, прибыли Toyota возросли с 8 млрд. долл. в фискальном году, который закончился 31 марта 2003 года, до более чем 10 млрд. долл. в 2004 году, 11 млрд. долл. в 2005 году и 12 млрд. долл. В 2006 году. Все это заставило многие компании еще раз присмотреться к концепции *бережливости*, чтобы попытаться понять, что сто*и*т за таким устойчивым и непрерывным успехом.

Бережливые инициативы редко берут начало в подразделениях компании, занимающихся разработкой программного обеспечения или продуктов (изделий), но можно привести множество примеров того, как успешные бережливые инициативы заимствовались для разработки у производства или логистики. Однако практику бережливости в том виде, как она сложилась на производстве, не так то легко адаптировать к среде разработки. В то время как никто не оспаривает действенности принципов бережливости, часто оказывается очень непросто применить соответствующую практику и критерии в среде разработки. В случаях, когда бережливые инициативы “стопорились” в среде разработки, многие компании обнаруживали, что книга *Lean Software Development* предоставляет солидную основу для переосмысления подходов и адаптации идей бережливости к условиям организации, занятой разработкой.

Преимущества бережливого и гибкого подхода к разработке ПО в последние несколько лет приобрели широкую известность и признание, и многие организации изменили

---

[1] Womack J., Jones D., Roos D. *The Machine That Changed the World*, Rawson Associates, *1990.*

свои методы создания программного обеспечения. Мы путешествовали по всему миру, посещая многие организации во время, когда делались попытки внедрить новые подходы, и мы многому научились в результате общения с людьми, упорно работающими над тем, чтобы изменить свои методы создания ПО. По мере того как рос объем наших знаний, то же происходило и со спросом на умение реализовать бережливый подход к разработке программного обеспечения. Мы осознали, что новая книга позволит нам поделиться тем, что мы узнали, с гораздо бо́льшим количеством людей, чем это возможно через личные контакты. Поэтому мы суммировали наш опыт в этой книге.

Это не поваренная книга, содержащая рецепты по реализации бережливого подхода. Так же как и наша предыдущая книга, это набор интеллектуальных методов, позволяющих подойти к адаптации принципов бережливости в вашей среде. Мы начали там, где закончилась первая книга, и окунулись глубже в вопросы и проблемы, с которыми приходится сталкиваться при попытке реализовать бережливые и гибкие методы в разработке ПО. Данная книга может рассматриваться как продолжение *Lean Software Development*. Вместо повторения того, что было в первой книге, здесь мы рассматриваем соответствующие вопросы под иным ракурсом. Мы исходим из того, что читатель убежден, что бережливый подход к разработке программного обеспечения — это хорошая идея, и сосредотачиваемся на основных вопросах его успешной реализации. Мы рассматриваем ключевые аспекты реализации и обсуждаем, что важно, что нет и почему. Нашей целью является помочь организациям перейти к более эффективной разработке программного обеспечения.

Первая глава книги посвящена истории бережливого подхода, а во второй рассматриваются семь принципов бережливого подхода к разработке ПО, представленных в *Lean Software Development*. Далее следуют главы, посвященные *ценности*, *потерям*, *скорости*, *людям*, *знанию*, *качеству*, *партнерам* и предстоящему *путешествию*. Каждая из этих восьми глав начинается с истории, иллюстрирующей, как некая организация сумела решить стоящую перед ней проблему. После этого следует обсуждение основных вопросов, которые мы (авторы) сочли важными; здесь же приводятся краткие истории, иллюстрирующие эти вопросы, а также содержащие ответы на типичные вопросы, которые мы часто слышим. Каждая глава завершается набором упражнений, призванных помочь читателю исследовать тему более глубоко.

Благодарности

Мы рады случаю поблагодарить всех, кто посещал наши курсы и семинары, кто приглашал нас в свои компании и особенно тех, кто делился с нами опытом и задавал интересные вопросы. В данной книге мы делимся знанием, почерпнутым от всех вас.

Большая благодарность Джеффу Сазерленду и Кенту Беку за их предисловия, в которых много добрых слов об этой книге, а также нашему редактору Грегу Дунчу (Greg Doench), редактору проекта Туррелу Албо (Tyrrell Albaugh) и особенно редактору Нэнси Хендрикс (Nancy Hendryx). Особая благодарность тем, кто сделал вклад в данную книгу в виде идей и историй: Джилл Аден (Jill Aden), Бреду Эпплтону (Brad Appleton), Ральфу Бонету (Ralph Bohnet), Майку Кону (Mike Cohn), Бенту Йенсену (Bent Jensen), Брайану Морику (Brian Marick), Клер Кроуфорд-Мейсон (Clare Crawford-Mason), Риану Мартенсу (Ryan Martens), Джеральду Месзаросу (Gerard Meszaros), Линн Миллер (Lynn Miller), Кенту Шнайцу (Kent Schnaith), Яну Шиммингсу (Ian Shimmings), Джоэлю Спольски (Joel Spolsky), Джеффу Сазерленду, Нэнси Ван Шундерверт (Nancy Van Schooenderwoert), Биллу Уэйку (Bill Wake), Вернеру Вильду (Werner Wild), а также руководителю производства и нашему другу, который пожелал, чтобы его имя не упоминалось.

Мы благодарим всех, кто прочел черновик данной книги и внес свои предложения, в частности Глена Аллемана (Glen Alleman), Бреда Эпплтона, Даниэля Бролунда (Daniel Brolund), Боба Коррика (Bob Corrick), Аллана Келли (Allan Kelly), Кента Шнайца, Дэйва Симпсона (Dave Simpson), Алана Шеллоуэя (Alan Shalloway), и Виллема ван дер Енде (Willem van den Ende). Мы благодарим обозревателей Ю Элв (Yi Lv), Романа Пичлера (Roman Pichler), Билла Уэйка (Bill Wake) и особенно Баса Водда (Bas Vodde) за их детальные и профессиональные комментарии. Особая сердечная благодарность Майку Кону, который ознакомился с обоими черновиками книги, учитывая при этом стоящие перед авторами жесткие сроки и стараясь не задерживать нас. Его комментарии и советы помогли сделать книгу лучше.

— Мэри и Том Поппендик,
июль, 2006 год.

Глава 1

История

Взаимозаменяемые детали

Париж, Франция, июль 1785 года. Миновало восемнадцать месяцев после завершения освободительной войны в Америке и оставалось четыре года до начала французской революции. Потребность в оружии занимала умы многих, когда Оноре Бланк (Honore Blanc) пригласил высокопоставленных военных и дипломатов в свою оружейную мастерскую в Париже. Он разобрал пятьдесят ружейных замков и разложил отдельные их детали по ящикам. Удивленные посетители произвольно брали нужные детали из ящиков, собирали из них замк*и*, а затем присоединяли их к мушкетам. Обнаружилось, что все детали были совместимы. Впервые была продемонстрирована возможность собирать ружья из взаимозаменяемых деталей.

Томас Джефферсон (Thomas Jefferson), в то время дипломат в Париже, присутствовавший на этой демонстрации, увидел способ решения одной из проблем своей беспокойной страны. Соединенные Штаны испытывали в то время нехватку оружия для своей защиты и расширения границ. Если наладить изготовление взаимозаменяемых деталей, тогда сравнительно неквалифицированные рабочие смогли бы за невысокую плату собирать большое количество ружей — настоящий бум для молодой страны, у которой не было ни денег, ни квалифицированной рабочей силы для производства оружия.

К задаче создания технологического процесса, достаточно точного, чтобы производить взаимозаменяемые детали ружей, обратился Эли Уитни (Eli Whitney), недавно запатентовавший устройство для очистки хлопка. В 1798 году Уитни получил государственный заказ на изготовление 10000 ружей в течение двух лет. На выполнение данного заказа ушло десять лет, и при этом расходы в несколько раз превзошли запланированные, но детали ружей не были полностью взаимозаменяемыми. Тем не менее Уитни считается разработчиком *американской производственной системы*, позволяющей неквалифицированным рабочим, используя соответствующие инструменты и шаблоны, изготавливать взаимозаменяемые детали, из которых затем собираются готовые изделия.

В течение XVII века Соединенные Штаты переживали бурный экономический рост, которым в значительной мере были обязаны новой производственной системе. В то же время в Европе идея отказаться от ремесленного производства натолкнулась на сильное сопротивление. Во Франции работа Оноре Бланка была прекращена правительством, которое опасалось потерять контроль над производством в условиях, когда неуправляемые работники смогут собирать мушкеты. В Англии изобретатели машин, автоматизировав-

ших процессы прядения и ткачества, подверглись нападениям гневных толп, побуждаемых страхом потерять работу. Однако в Америке работы было мало, так же как и ремесленных традиций, поэтому идея производства взаимозаменяемых деталей укоренилась и расцвела.

Взаимозаменяемые люди

Детройт, США, январь 1914 года. Генри Форд, после запуска сборочного конвейера для своей знаменитой Модели Т, поднял зарплаты рабочим с 2,4 долл. за девятичасовый рабочий день до 5 долл. за восьмичасовый. Пресса полагала, что он сошел с ума, однако это был прозорливый ход. Форд забирал более 85% стоимости произведенного автомобиля, поэтому он мог позволить себе удвоить зарплаты своим рабочим. Он уже резко снизил цену своего автомобиля. Теперь он повысил зарплаты и сократил рабочий день, чтобы создать средний класс, у которого имелись бы свободное время и деньги на приобретение автомобилей.

Ранее на сборку автомобиля уходило 12 часов; теперь на это требовалось 90 минут. Куда девалось так много времени? Менеджеры Форда, при организации рабочих мест на конвейере, применили идеи Фредерика Уинслоу Тейлора (Frederick Winslow Taylor) об эффективности труда. Тейлор полагал, что работники с фиксированными зарплатами тратят рабочее время, стараясь придумать, как работать помедленнее, поскольку эффективный труд не увеличивал зарплату и грозил сокращением рабочих мест. Его подход состоял в том, чтобы процесс сборки на конвейере разделить на мельчайшие операции, занимающие минимальное время. При этом предполагалось, что работник на данном рабочем месте сам определит для себя, как лучше выполнять данную операцию.

Работа на конвейере была неинтересной, периодически повторяющейся и строго контролируемой. Рабочих учили, как именно следует выполнять данную работу и за сколько времени. Новый работник мог быть обучен в течение десяти минут. Подобно взаимозаменяемым деталям веком ранее, взаимозаменяемые работники были в центре новой индустриальной модели: массового производства.

При этом предполагалось, что недостаток разнообразия и независимости возместят высокие зарплаты, и некоторое время так и было. Сначала дела у Форда шли прекрасно. Продажи взлетели до небес и автомобиль Форда царил на рынке. Однако вскоре Модель Т устарела, а все более процветающий средний класс испытывал все больше желания сменить свои старые автомобили на более современные модели. Форд не отреагировал вовремя на новые требования рынка, поскольку его система производства была наиболее эффективна при производстве только одной модели. В то же время Альфред П. Слоун (Alfred P. Sloan) в General Motors организовал производство, ориентированное на создание множества моделей, предназначенных для различных сегментов рынка. А система производства Форда, по мере роста требований к разнообразию и сложности ассортимента, становилась все более неуклюжей.

Также, с течением времени рабочие все более ощущали, что попали в ловушку. Они привыкли к высоким жизненным стандартам и не имели шансов найти работу с аналогичной зарплатой в другом месте. Широко распространившиеся рабочие волнения в 30-х годах XX столетия часто объясняют системой производства, исключающей уважение к работникам, которых легко было заменить.

Семейство Тойода

Керия, Япония, 1927 год. Компания Toyoda Automatic Loom Works организовала для инженеров-текстильщиков демонстрацию своего нового ткацкого станка. Сначала посетители могли ознакомиться с процессом производства данного оборудования, а затем их пригласили в прядильно-ткацкий цех, где они наблюдали одновременную работу 520 ткацких станков компании Toyoda. Ничего подобного до этого не существовало. Данные станки, которые обслуживались только 20 ткачами, выполняли 240 операций в минуту. Они были полностью автоматические и могли работать без участия человека всю ночь, в соответствии с законом о запрете ночного труда. Когда на челноке, снующем по всему ткацкому станку, оканчивалась нить, его тут же, без остановки работы, сменял новый челнок. Если даже рвалась одна из сновальных нитей или оканчивалась уточная нить, станок немедленно останавливался и подавал сигнал о возникшей проблеме.

Чтобы вникнуть в производственную систему Toyota[1], важно понять, как трудно было разработать и изготовить "идеальный ткацкий станок". Сакичи Тойода (Sakichi Toyoda) создал свою первую модель в 1896 году, а устройство автоматической смены челноков было изобретено только через семь лет. Было проведено тестирование для сравнения 50 ткацких станков Тоеды (с автоматической сменой челнока) с таким же количеством станков из Европы. Результаты оказались неутешительными. Эти "ранние" станки Toyoda были сложными, громоздкими машинами, с низкой точностью и трудными в обслуживании.

Сакичи Тойода нанял технических специалистов и пригласил американского инженера Чарльза Э. Фрэнсиса (Charles A. Francis) для внедрения в своей компании американской производственной системы. Фрэнсис усовершенствовал конструкцию оборудования и построил мастерскую для его производства. Он разработал спецификации стандартов, создал стандартные шаблоны и приспособления и реконструировал процесс производства. В течение последующего десятилетия, в то время как Европа и Америка были отвлечены войной, ткацкие станки Toyoda продавались очень хорошо.

Хотя Сакичи Тойода охотно использовал точные взаимозаменяемые детали, в текстильном производстве не было места для взаимозаменяемых работников. Автоматические ткацкие станки — сложные и высокоточные машины, чувствительные к смене материала и сложные в обслуживании. Поэтому для одновременной работы каждых 25–30 станков требовался высококвалифицированный ткач. Если для обслуживания работающего ткацкого станка была необходима высокая квалификация, разработка и производство таких станков предъявляли еще бо́льшие требования. Однако Сакичи Тойода умел выбирать среди выпускников японских технических университетов нужных ему людей. И даже будучи занят созданием новых компаний, он всегда полагался (в разработке и производстве новых моделей ткацких станков) на свой коллектив конструкторов, состав которого всегда поддерживался на высоком уровне.

В 1921 году сын Сакичи Тойода, Кийтиро, приступил к работе в компании своего отца и сосредоточился на проблеме автоматизации станков. В 1924 году они (отец и сын) совместно получили патент на усовершенствованный механизм автоматической смены

[1] Буква "д" в фамилии Тойода была заменена на "т", когда была создана компания Toyota Motor Company. Соответствующие японские иероглифы сходны, только для "Тоеты" (по сравнению с "Тоедой") требовалось двумя мазками кисти меньше.

челноков. Кроме того, были разработаны методы выявления проблем и остановки ткацкого станка с тем, чтобы работающие станки можно было оставлять без присмотра на ночь. Кийтиро Тойода подумывал о строительстве фабрики по производству новых ткацких станков и установил 520 из них на экспериментальной ткацкой фабрике компании. После презентации этих "идеальных ткацких станков" заказы на них посыпались как из рога изобилия. Кийтиро использовал прибыли для основания нового бизнеса по производству автомобилей. Он посетил Детройт и провел там несколько лет, учась, как изготавливать двигатели. Первый автомобиль компании Toyota появился на рынке в 1936 году, однако его производство вскоре было прервано войной.

Производственная система Toyota

Куруме, Япония, октябрь 1949 года. Отменены ограничения на производство легковых автомобилей[2], но еще в 1945 году Кийтиро Тойода поставил перед своей компанией задачу "догнать Америку". Однако было ясно, что Toyota не сможет "догнать", используя американскую модель массового производства. Массовое производство предполагает получение прибылей за счет увеличения объемов производства, однако сырья в то время не хватало, заказов было мало, а требования к разнообразию ассортимента были высокие. В этих условиях получать прибыли, увеличивая объемы производства, было просто невозможно.

Основная идея Кийтиро Тойода состояла в том, что детали на сборочную линию должны поступать "точно вовремя", именно тогда, когда в них возникнет надобность. Иными словами, детали для сборки должны браться не со склада, а быть произведены непосредственно накануне. Для реализации этой идеи потребовалось время, однако в 1962 году, через десять лет после смерти Кийтиро Тойода, производственная система Toyota (Toyota Production System) была принята во всей компании.

Тайити Оно

Тайити Оно (Taiichi Ohno) был менеджером в компании Toyota, сумевшим реализовать идею Кийтиро Тойода и создать то, что позже стало известно как производственная система Toyota. Он изучил систему производства Форда, однако озарение наступило, когда он познакомился с тем, как в американских супермаркетах организован учет товара. К этому добавились его глубокие познания в технологии прядильно-ткацкого производства, а также идеи его подчиненных. Годы экспериментирования ушли на то, чтобы постепенно создать производственную систему Toyota — процесс, который Оно представлял себе как никогда не прекращающийся. Данная идея постепенно получила распространение во всей компании, а на Тайити Оно, по мере этого возлагалась все бо́льшая ответственность.

В своей книге *Toyota Production System*[3] (Производственная система Toyota) Оно называет ее "системой, предусматривающей полное устранение потерь". Он объясняет, что данная система покоится на двух столпах: поставках "точно вовремя" и автоматизации.

---

[2] Это ограничение ввела Америка (с целью избежать конкуренции), оккупировавшая Японию после войны. — *Примеч. ред.*

[3] Данный раздел основан на книге Taiichi Ohno *Toyota Production System: Beyond Large-Scale Production*, Productivity Press, Japan, *1978*. (На английский язык переведена в 1988 г.) Великолепная книга, которая легко читается и которую можно рекомендовать даже сегодня.

Поставки "точно вовремя"

Важно заметить, что данный принцип полностью противоречил всем общепринятым представлениям того времени. Сопротивление усилиям Оно было огромным, и он добился успеха только потому, что пользовался поддержкой Эйдзи Тойода (Eiji Toyoda), возглавившего компанию после того, как его двоюродный брат Кийтиро вышел в отставку в 1950 году. Оба Тойода блестяще предвидели, что перспективы кроются вовсе не в наращивании объемов производства, а в том, чтобы научиться сохранять рентабельность при расширении ассортимента. При увеличении объема вдвое затраты сокращаются на 15–25% на единицу продукции; в то же время затраты растут на 20–35% каждый раз, когда ассортимент удваивается.[4] Однако поставки "точно вовремя" снижают действие факторов, обеспечивающих основной вклад в рост затрат при расширении ассортимента. В действительности это единственная модель производства (среди существующих на сегодняшний день), которая позволяет иметь рентабельное производство при широком ассортименте.[5]

Автоматизация

Автоматизированные ткацкие станки Тоеды могли работать без участия ткачей, поскольку были в состоянии выявлять возникновение неполадки и автоматически прекращать работу. Автоматизация означает организацию производственного процесса таким образом, чтобы любая аномалия немедленно обнаруживалась, работа останавливалась и источник проблемы устранялся прежде, чем работа будет продолжена.

*О*но называл это "автоматизацией с человеческими свойствами". Наши тела содержат в себе автоматическую нервную систему, которая управляет рефлексами, такими как дыхание, биение сердца и переваривание пищи. Если мы прикоснемся рукой к чему-либо горячему, наша нервная система автоматически даст команду отдернуть ее (руку), не дожидаясь, пока мозг оценит происходящее и пошлет соответствующий сигнал. Автоматизация в общем смысле означает, что некая система содержит средства, позволяющие реагировать на события немедленно и адекватно, не дожидаясь указаний из управляющего центра.[6]

Сигаэо Синго

Сигаэо Синго (Shigeo Shingo) был консультантом, который помогал Тайити Оно внедрять производственную систему Toyota в одноименной компании, а позже помогавший компаниям по всему миру понять эту систему и реализовать ее у себя. Менеджеры, занимавшиеся внедрением концепции поставки "точно вовремя" в 80-х годах XX столетия, с благодарностью вспоминают "зеленую книгу"[7], — первое издание, посвященное данной концепции, опубликованное на английском языке. Перевод был не особенно хорош, а материал труден для понимания и наполнен техническими терминами, однако это оказалась удивительно прозорливая книга.

---

[4] Stalk G. *Time—The Next Source of Competitive Advantage*, Harvard Business Review, *July, 1988.*

[5] См. статью Balle F., Balle M *Lean or Six Sigma*, `www.lean.org/library/leanorsigma.pdf`.

[6] Taiichi Ohno, Там же, *p. 46.*

[7] Shingo S. *Study of 'Toyota' Production System*, Productivity Press, *1981.*

В этой книге Синго рассматривает две основные темы: производство без запасов и без контроля. Тот, кто внимательно познакомится с данной книгой, заметит, что это, по сути, описание практической реализации двух основных идей Оно из производственной системы Toyota.

Производство без запасов

Подход "точно вовремя" предполагает отказ от запасов (деталей, материалов и т.п.), которые неизбежны при стремлении получить экономический эффект за счет расширения производства. Основная идея в том, чтобы производить все мелкими партиями, а для этого необходима возможность очень быстро перенастраивать оборудование с производства одной детали на производство другой. В разработке программного обеспечения в качестве "запасов" можно рассматривать незавершенную работу. В этом случае снизить объем незавершенной работы можно, осуществляя частые релизы (или выпуски) создаваемого ПО. Однако в некоторых организациях между релизами проходят недели и месяцы и каждый релиз содержит большое количество новых функциональных возможностей. В результате приходится выполнять большие объемы работы по тестированию и вводу в эксплуатацию вновь созданного ПО для каждого релиза. Подобный подход неприемлем не только с экономической точки зрения. Например, пользователи вправе ожидать обновления антивирусного программного обеспечения на их компьютерах через несколько часов после появления новой угрозы. В этом случае изменения ПО будут небольшими, а потому трудозатраты, связанные с вводом его в эксплуатацию, будут незначительны.

Производство без проверок

Идея, лежащая в основе автоматизации, состоит в том, что система должна быть создана с защитой от ошибок. Во время работы никто не будет искать поломки в машине или проверять качество продукции. Система со встроенной защитой от ошибок не нуждается в проверках. Например, вам не удастся подключить разъем кабеля монитора к компьютеру перевернутым. И нет никакой необходимости, чтобы кто-нибудь проверял, правильно ли был подключен кабель, поскольку подключить его неправильно невозможно. Защита от оплошности предполагает, что любая ошибка, которая может быть совершена, в конце концов непременно будет совершена, поэтому изначально следует сделать так, чтобы данная оплошность была невозможна.

"Точно вовремя"

Поскольку в то время компании росли быстро и могли продать все, что было произведено, производственная система Toyota в основном игнорировалась (даже в Японии), пока не разразился нефтяной кризис 1973 года. Однако замедление экономики, вызванное нефтяным кризисом, быстро все расставило по своим местам, и система Toyota сразу же вызвала интерес. Другие компании начали изучать производственную систему Toyota и внедрять у себя многие ее элементы. Уже через десять лет Америка и Европа начали ощущать серьезную конкуренцию со стороны японских компаний. Например, я (Мэри)[8] в 80-х годах работала на предприятии по производству видеокассет, когда продукция

[8] Один из авторов этой книги. — *Примеч. ред.*

японских компаний появилась на рынке по невероятно низким ценам. Исследование показало, что японские компании использовали новый подход, известный как "точно вовремя". На моем предприятии, чтобы сохранить конкурентоспособность, данный подход был изучен и использован.

Картинку, которая использовалась нами для графического изображения этого подхода, можно увидеть на рис. 1.1.

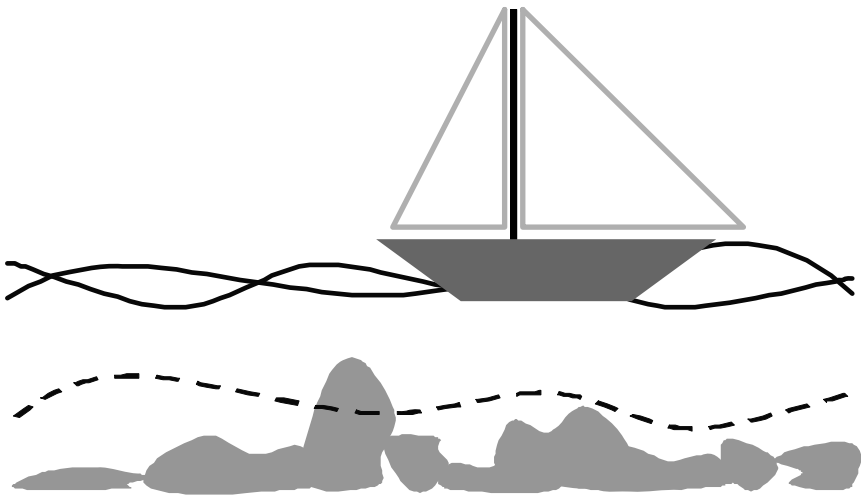


Рис. 1.1. Чтобы проблемы проявились, достаточно снизить уровень запасов

Объем запасов соответствует уровню воды в реке, и когда он высок, множество больших подводных камней (или проблем) скрыты. Однако, если понизить уровень воды, камни начнут появляться на поверхности. Необходимо удалить эти камни, иначе ваша лодка наскочит на них. После ликвидации самых крупных камней можно еще больше понизить уровень запасов, обнаружить новые камни, удалить их из реки и продолжать повторять это, пока на дне не останется одна галька.

Почему бы просто не поддерживать уровень запасов на высоком уровне и игнорировать камни? Дело в том, что камни — это разного рода проблемы (не выявленные дефекты в готовых изделиях, выходящие из-под контроля процессы, произведенные товары, которые не будут куплены, до того как истечет срок их годности, системы учета запасов, в которых царствует путаница — и т.п.). Также камни — это скрытые потери, о которых вы ничего не будете знать, пока не понизите уровень запасов.

Основной урок, вытекающий из принципа производства "точно вовремя", состоит в том, что следует отказаться от попыток довести до максимума эффективность на каждом рабочем месте. На нашем предприятии имелось множество дорогого оборудования, из которого, как полагало руководство, следует выжать все, что только можно. Однако подобный подход приводил только к росту объема запасов, которые накапливались перед каждым станком с тем, чтобы обеспечить его беспрерывную работу, и после него, поскольку произведенные с помощью данного станка детали в этот момент просто некуда было девать (рис. 1.2). Когда был реализован принцип производства "точно вовремя" (и прекратились попытки выжать из оборудования все возможное), горы запасов исчезли, и мы с удивлением обнаружили, что, в целом, выпуск продукции *увеличился*.

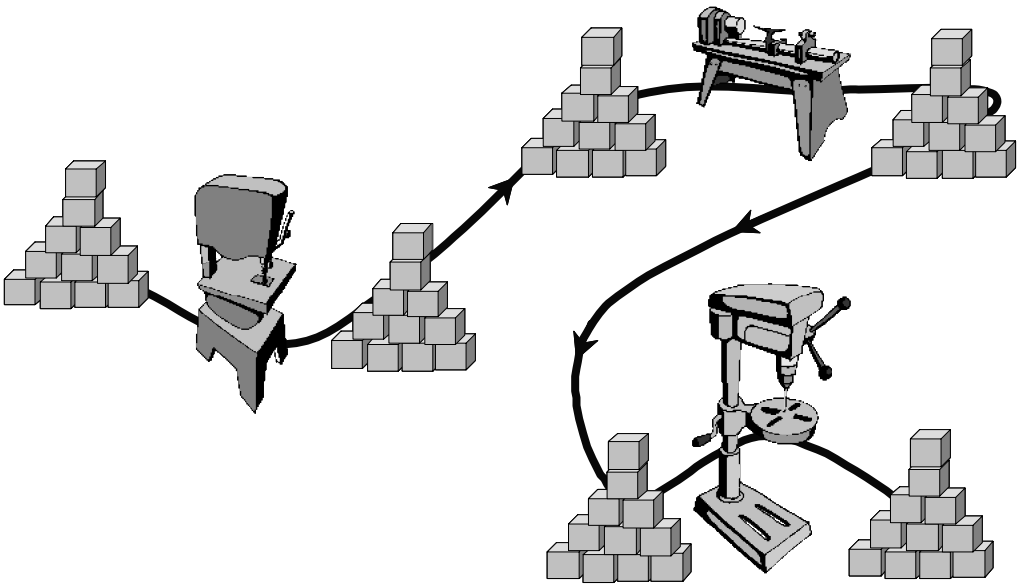


Рис. 1.2. Следует прекратить попытки довести эффективность на каждом рабочем месте до максимума

Останов линии и безопасность

Одним из элементов концепции ''точно вовремя'', который легко внедрить, является останов производственной линии в случае инцидента (или сбоя в работе). На нашем предприятии по производству видеокассет использовались в основном хрупкие материалы, поэтому культивировалась достаточно агрессивная программа безопасности. Согласно этой программе считалось важным расследовать даже мельчайший инцидент, поскольку мелкие инциденты, если они игнорируются, обычно ведут к крупным инцидентам.

Книга *Managing the Unexpected*[9] (''Как справляться с неожиданностями'') авторов Weick (Вейка) и Sutcliffe (Сатклиффа) рассказывает, что организации, подобные нашему предприятию, создают среду, где большое внимание уделяется безопасности. Подобный подход имеет четыре аспекта.

1. **Выявление возможных сбоев в работе.**

   Мы тратили много времени, думая о том, что в производственном процессе может случиться непредвиденного, и старались предусмотреть ответные меры.

2. **Понимание производственных операций.**

   Предполагалось, что каждый менеджер на предприятии должен провести некоторое время, работая на одном из рабочих мест производственной линии.

3. **Стремление учиться на ошибках.**

   Расследовался даже мельчайший инцидент, чтобы исключить его повторение.

4. **Уважение к опыту.**

   Каждый менеджер знал, что только люди, выполняющие реальную работу, по-настоящему понимают, как функционирует предприятие.

[9] Weick K.E., Sutcliffe K.M. *Managing the Unexpected: Assuring High Performance in an Age of Complexity*, Jossey-Bass, *2001.*

Оставалось простое дело — превратить культуру безопасности в культуру останова производственной линии в случае инцидента. Для этого необходимо было помимо сбоев в работе обратить также внимание на возможный брак. Для того чтобы сделать ненужным контроль инцидентов, на каждом этапе каждой операции была предусмотрена защита от оплошности. Когда случался брак, работа приостанавливалась, и работники старались определить, что именно вызвало появление проблемы. Если возникший на определенном этапе брак не приводил к останову производственной линии, мы старались определить, как сделать так, чтобы исключить повторения подобного. Говоря "мы", я имею в виду наших рядовых рабочих, поскольку именно они создали производственный процесс. (Мэри Поппендик)

Когда было принято решение внедрить на предприятии принцип производства "точно вовремя", под рукой не оказалось консультантов, которые могли бы подсказать, что именно для этого следует делать, поэтому нам пришлось до всего додумываться самим. Мы создали модель предприятия: огромный стол был покрыт большим листом бумаги, на котором были изображены все производственные процессы. Далее мы создали так называемые "канбан-карточки"[10] (на полосках бумаги были сделаны надписи, обозначающие различные типы запасов, и каждая такая полоска была помещена в отдельный кофейный стаканчик — рис. 1.3). При этом каждый стаканчик представлял тележку с соответствующими деталями. Далее мы напечатали бланки недельных заказов на упаковку готовых изделий и попытались имитировать систему с "вытягиванием" (pull system). Мы имитировали выполнение заказов на упаковку готовых изделий, используя в качестве игровой доски большой лист бумаги и стаканчики, представляющие предприятие. Когда стаканчик с партией готовых изделий "был упакован", полоска бумаги (канбан-карточка) перемещалась в начало процесса, где она использовалась в качестве сигнала о необходимости произвести больше соответствующих деталей.[11]

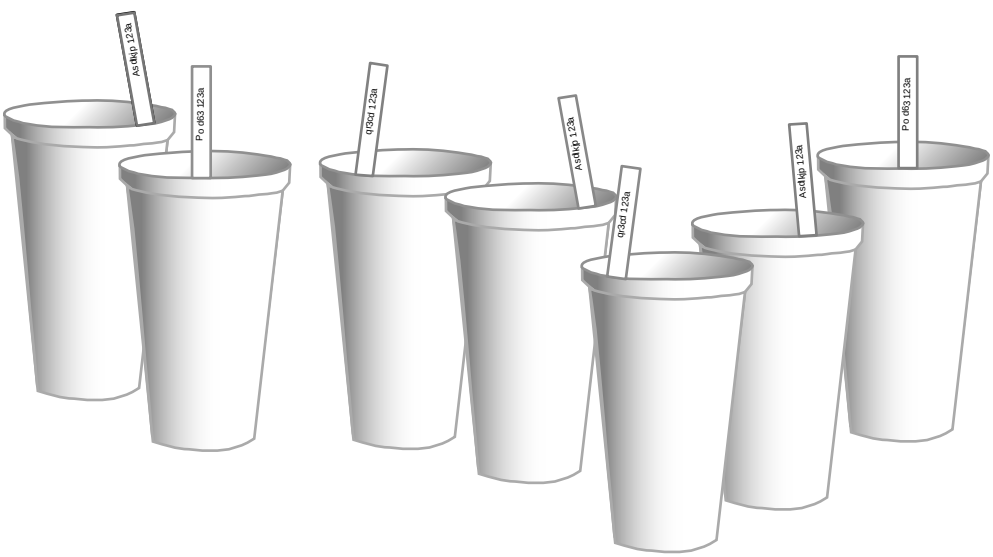


Рис. 1.3. Стаканчики имитируют тележки с деталями

[10] Подробнее об этом см. в главе 6.

[11] Данный подход к планированию обозначается термином "канбан".

Используя описанную выше модель, мы продемонстрировали концепцию системы с "вытягиванием" сначала производственным менеджерам, затем руководству предприятия, затем руководителям смен. Наконец, руководители смен использовали данную модель для проработки внедряемой концепции с каждым рабочим на каждом участке. Каждому участку было предоставлено самостоятельно определить детали функционирования новой системы с "вытягиванием" в их среде. Скрупулезная подготовка заняла несколько месяцев, однако в конце концов все было готово. В один прекрасный уикэнд, после внесения необходимых изменений, коллектив затаил дыхание. Компьютерное планирование было полностью отключено. Его заменило ручное планирование, организованное с помощью канбан-карточек, о которых шла речь выше. Наша новая система "точно вовремя" имела оглушительный успех, в основном благодаря тому, что все ее нюансы определялись самими рабочими, которым было проще устранить мелкие недочеты и легче совершенствовать процесс впоследствии.

Концепция бережливости

В 1990 году в книге *The Machine That Changed the World*[12] ("Машина, изменившая мир") было предложено новое имя для того, что до этого было известно как поставки "точно вовремя", или производственная модель Toyota. С этого момента принцип, на котором построено изготовление автомобилей в этой компании, стал называться **бережливым производством** (lean production). В течение нескольких последующих лет многие компании старались внедрить у себя бережливое производство, однако это оказалось удивительно сложным делом. Как и со всеми предыдущими новыми моделями производства, сопротивление со стороны тех, кто инвестировал в старую модель, было яростным.

Многие сочли концепцию бережливого производства противоречащей интуиции и недостаточно убедительной для того, чтобы менять укоренившиеся привычки. Очень часто компании внедряли только часть системы, например, поставки "точно вовремя" без второй составляющей, известной как "останов линии". В этих компаниях не учли, что по-настоящему бережливым можно назвать производство *"... где максимальное количество задач и ответственности передаются рабочим, которые в действительности создают автомобиль на сборочной линии и поощряет внедрение системы выявления брака, которая быстро определяет источник обнаруженной проблемы."*[13]

Несмотря на трудности, связанные с внедрением нового (противоречащего интуиции) принципа, многие попытки оказались чрезвычайно успешными. Были созданы по-настоящему бережливые производства, которые неизменно процветали. Концепция бережливости была принята в других областях хозяйственной деятельности (помимо производства), таких различных, как обработка заказов, розничная продажа и обслуживание самолетов. Указанная концепция была также применена в организации цепи поставок, разработке изделий и разработке программного обеспечения (рис. 1.4).

---

[12] Womack J., Jones D., Roos D. *The Machine That Changed the World*, Rawson Associates, *1990.*
[13] Там же, *p. 99.*

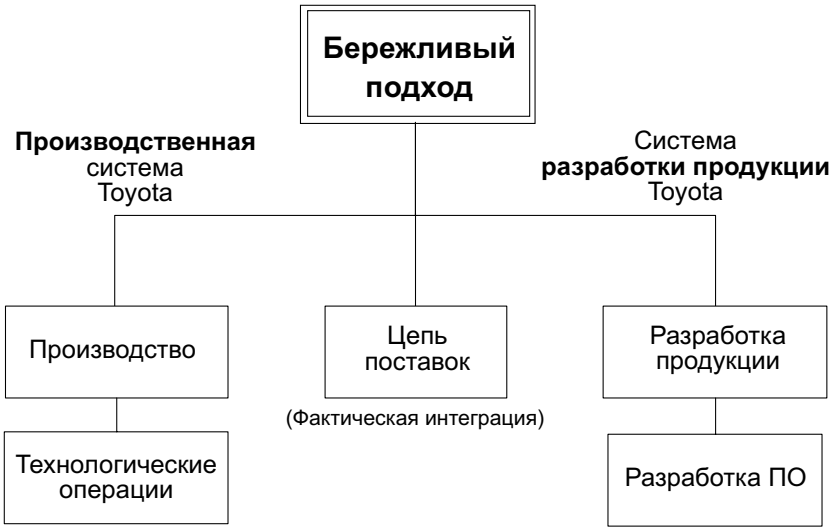


Рис. 1.4. Применение концепции бережливости

Бережливое производство и бережливые операции

Бережливое производство способно определять стандарты для дисциплины, эффективности и производительности. В действительности использование принципов бережливости на производстве позволяет получить значительное преимущество, которому конкурентам часто нечего противопоставить. Например, система выполнения заказов компании Dell Computer регулярно поставляет клиентам созданные на заказ компьютеры в течение нескольких дней — достижение, которое трудно повторить конкурентам, не желающим отказаться от своих систем дистрибуции (ориентированных на поставку крупных партий готовых изделий). Принцип бережливости нашел применение также в непроизводственной сфере. Например, компания-авиаперевозчик Southwest Airlines сосредоточилась на перевозке пассажиров непосредственно из пункта А в пункт Б на сравнительно небольших машинах, в то время как конкурентам трудно отказаться от привычного ориентирования на перевозку пассажиров большими группами. В некоторых отраслях, таких как быстрая доставка посылок, многие компании были перестроены на основе принципов бережливости, но только те из них, где используются бережливые операции (в последнем случае одноразовая доставка), способны выжить.

Бережливая цепь поставок

Когда принципы бережливого производства утверждаются в стенах предприятия, их приходится распространять и на поставщиков, поскольку принципы массового и бережливого производств очень плохо сочетаются. В компании Toyota поняли это рано и помогли своим поставщикам адаптировать у себя принципы бережливого производства. По оценке Питера Друкера (Peter Drucker) сеть поставщиков Toyota обеспечивает снижение издержек

на 25–30%.[14] Когда Toyota в 80-х годах прошлого века открыла свои предприятия в Соединенных Штатах, она организовала здесь и аналогичную сеть поставщиков. Удивительно, но американские компании-поставщики для автомобильной промышленности часто имеют в своей структуре подразделения, ориентированные на бережливое производство, специально предназначенные для обслуживания Toyota, в то время как прочие подразделения этих компаний организованы на "традиционных" принципах, поскольку другие компании-производители автомобилей не могут сотрудничать с "бережливыми поставщиками".[15] Бережливая цепь поставок также чрезвычайно важна для Dell, так как эта компания собирает свои компьютеры из узлов, разработанных и произведенных другими компаниями. Поскольку компания Dell сотрудничает с партнерами так, словно они ее составная часть (свободный обмен информацией с тем, чтобы вся цепь поставок смогла адаптировать принципы бережливости), можно сказать, что имеет место "виртуальная (или фактическая) интеграция".

В бережливых цепях поставок компании учатся работать так, чтобы не возникало проблем с партнерами; при этом в отдельных компаниях появляется понимание, что их главные интересы не отделимы от интересов всей цепи поставок. Для организаций, участвующих в совместной разработке программного обеспечения, управление цепями поставок предоставляет хорошо опробованную модель того, как отдельным компаниям следует формулировать и осуществлять договорные взаимоотношения на основе принципов бережливости.

Бережливая разработка продуктов

"Истинная разница между Toyota и прочими производителями автомобилей состоит не в производственной системе Toyota, а в системе разработки продукции Toyota", — говорил Косаку Ямада (Kosaku Yamada), главный конструктор модели Lexus ES 300.[16] Процесс разработки изделия очень отличается от процесса его производства, и методы управления, успешные в производстве, часто совершенно не годятся для разработки. Тем не менее эпохальная книга *Product Development Performance*[17] ("Организация разработки продукции") авторов Clark (Кларка) и Fujimoto (Фуджимото) показывает, что эффективный процесс разработки имеет много общего с производством. Сходства двух процессов, описанные Кларком и Фуджимото, суммирует табл. 1.1.

Среди компаний, которым удалось уловить суть производственной системы Toyota и должным образом применить ее к процессу разработки продукции, прежде всего следует назвать саму компанию Toyota. Поэтому не удивительно, что в конце 90-х годов прошлого века стало очевидно, что Toyota использует уникальный и успешный подход к разработке своих изделий. Подход Toyota оказался одновременно противоречащим интуиции и пророческим. Очень мало было попыток непосредственно использовать в процессе разработки продукции элементы производственной системы Toyota (изначально ориентированные на производство), однако лежащие в основе соответствующего процесса разработки принципы явно были заимствованы из этой системы.

---

[14] Drucker P. *Management Challenges for the 21st Century*, Harper Business — *2001, p. 33.*

[15] Dyer J. *Collaborative Advantage: Winning Through Extended Enterprise Supplier Networks*, Oxford University Press, *2000.*

[16] Vasilash G.S. *Engaging the ES 300,* Automotive Design and Production, *September, 2001.*

[17] Clark K. B. и Fujimoto T., *Product Development Performance: Strategy, Organization, and Management in the World Auto Industry*, Harvard Business School Press, *1991.*

Таблица 1.1. Сходства между бережливыми процессами производства и разработки продукции[18]

Бережливое производство	Бережливая разработка
Частые изменения производственного процесса	Частые изменения конструкции (для программного обеспечения — частые релизы)
Жесткие сроки на производство	Жесткие сроки на разработку
Сокращенные запасы деталей между технологическими операциями	Сокращенные объемы информации между этапами разработки
Частые перемещения малых партий деталей от одной технологической операции к другой	Частые интенсивные потоки от одного этапа разработки к другому
Сокращенные запасы требуют меньше ресурсов и большего потока информации между технологическими операциями	Сокращенные сроки на разработку требуют меньше ресурсов и потоков информации между этапами разработки
Способность адаптироваться к изменениям в объемах производства, ассортименте и конструкции	Способность адаптироваться к изменениям в конструкции, сроках разработки и заданной цене продукции
Отказ от строгой регламентации работы (для рабочих) ведет к более высокой производительности	Отказ от строгой регламентации работы (для инженеров или разработчиков) ведет к более высокой производительности
Приоритет быстрому решению проблем и непрерывному совершенствованию процесса	Приоритет частым последующим инновациям и непрерывному совершенствованию продукции и процесса
Одновременное улучшение качества, сокращение сроков поставки и увеличение производительности	Одновременное улучшение качества, сокращение сроков разработки и увеличение производительности

Изделие, полученное в результате процесса разработки, может быть очень удачным или посредственным. Оно может иметь элегантный дизайн и прийтись как раз впору потребителям либо оно может оказаться не отвечающим ожиданиям ни потребителей, ни его создателей. Модели автомобилей Toyota, как правило, попадают в первую категорию. Обозреватели объясняют это организацией руководства проектами в компании Toyota: за коммерческий успех той или иной модели несет ответственность главный инженер соответствующего проекта. Главный инженер проекта одновременно должен быть способен оценить требования рынка и иметь техническую эрудицию, необходимую для обеспечения высокого качества конструкции созданной модели. Джеффри Лайкер (Jeffrey Liker), рассказавший в своей книге *The Toyota Way*[19] историю создания моделей Lexus и Prius, особо подчеркивает, как эти два очень успешных изделия были представлены на рынок в рекордные сроки под руководством двух выдающихся главных инженеров.

Разработку изделия можно рассматривать как процесс создания знания (knowledge creation process). Система разработки продукции Toyota создает знание, широко исследуя различные варианты конструкции, экспериментируя со множеством прототипов и организуя регулярные совместные совещания, на которых производится оценка разрабаты-

[18] Clark K. B. и Fujimoto T, *Product Development Perfor-mance*, *p. 172.*

[19] Liker J. *The Toyota Way: 14 Management Principles from the World's Greatest Manufacturer*, McGraw Hill, *2004.*

ваемой конструкции и принимаются решения на основе максимально подробной информации. Знание, созданное в процессе разработки и производства (чтобы оно не потерялось), оформляется в виде кратких и полезных одностраничных информационных листков, которые делают его (знание) доступным для всех. Создание и представление знания в удобном для будущего использования виде является отличительной чертой *системы разработки продукции* Toyota.

Национальный центр основ технологии (National Center for Manufacturing Sciences — NCMS) провел многогодичное исследование системы разработки продукции Toyota, и выявленные при этом оригинальные подходы, методы и приемы были описаны Майклом Кеннеди (Michael Kennedy) в книге *Product Development for the Lean Enterprise*[20] . В этой книге Кеннеди сформулировал четыре важнейших принципа системы разработки продукции Toyota (рис. 1.5).

Рис. 1.5. Важнейшие принципы системы разработки продукции Toyota[21]

[20] Kennedy M. *Product Development for the Lean Enterprise: Why Toyota's System Is Four Times More Productive and How You Can Implement It*, Oaklea Press, *2003.*

[21] Данный рисунок из той же книги (автор Michael Kennedy), p. 120. Использовано с разрешения автора.

Система разработки продукции Toyota

Эта система основана на четырех важнейших принципах.

1. **Разработка продукции под руководством лидера-предпринимателя.**

   Главный инженер проекта компании Toyota несет ответственность за коммерческий успех созданной им модели. Он очень опытный инженер, вполне способный создать конструкцию будущего автомобиля на системном уровне. В то же время он обязан развить в себе глубокое понимание сегмента рынка, для которого предназначен создаваемый автомобиль, и создать модель, которая порадует покупателей. Главный инженер создает прообраз нового изделия, который он регулярно передает команде разработчиков, обсуждая с ними повседневные решения. В случае несогласия он защищает этот прообраз или идет на компромисс. Он определяет сроки и вносит изменения в процесс с тем, чтобы все было завершено вовремя.

2. **Технические специалисты высшей квалификации.**

   Со времен Сакичи Тойода компании Toyoda и Toyota всегда нанимали технических специалистов высшей квалификации, которые разрабатывали для этих компаний сложные технические изделия. Годы уходят на то, чтобы инженер стал настоящим экспертом в своей области, и в Toyota инженеры не переводятся на административные должности (и такие переходы не мотивируются) до того, как они полностью не овладели всеми тонкостями своей профессии. Менеджер — это учитель, ставший настоящим знатоком своего дела на участке, за который он отвечает; он обучает новых инженеров и ''ведет'' их от статуса ученика до подмастерья и далее до мастера-инженера.

3. **Планирование и контроль, основанные на ответственности.**

   Главный инженер проекта на два-три месяца вперед определяет график разработки нового автомобиля, где предусмотрены ключевые точки синхронизации. (Точка синхронизации — мероприятие в рамках разработки проекта, когда представители разных коллективов, независимо разрабатывающих отдельные элементы и узлы проекта, собираются и решают вопросы совместимости их разработок.) Инженеры знают, чего от них ждут в предстоящей точке синхронизации, и они предоставляют нужные результаты без необходимости контроля. Если инженерам для выполнения работы требуется какая-либо информация, предполагается, что они должны ее получить из соответствующего источника. Недавно главные инженеры проектов Toyota положили начало практике использования т.н. ''общих комнат'', где члены коллектива разработчиков могут работать и где весь коллектив собирается регулярно. В этой комнате, помимо прочего, развешаны разного рода диаграммы и графики, иллюстрирующие различные аспекты выполняемого проекта.

4. **Параллельная разработка альтернатив.**

   Подобный метод разработки проекта означает исследование множества возможных вариантов будущей конструкции и принятие оптимального решения путем постепенного исключения всех прочих вариантов. Что это означает на практике? Это значит проявлять осторожность и не принимать решений до тех пор, пока не возникнет абсолютная необходимость в их принятии. Это в свою очередь предполагает постоянное изучение всех оставшихся вариантов. В результате решение будет принято на основе максимально возможного количества информации. Парадокс состоит в том, что параллельная разработка альтернатив может показаться связанной с неоправданными затратами. Однако при взгляде на систему в целом, подобный подход к разработке проекта позволяет коллективу разработчиков принять наиболее оптимальное решение гораздо быстрее, чем в случае, если бы от многих возможных вариантов будущей конструкции (возможно, на первый взгляд нерациональных), чтобы не тратить на них время, отказывались быстро.[22]

[22] Больше информации о параллельной разработке альтернатив можно найти в главе 7.

Бережливый подход к созданию программного обеспечения

Программное обеспечение представляет собой одну из разновидностей продукта. В действительности большинство используемого ПО можно приобрести так же, как любой товар. Программное обеспечение, которое не было создано в виде самостоятельного изделия, может быть встроено в аппаратуру или являться составной частью игровой либо поисковой программы. Некоторое программное обеспечение, включая большинство пользовательского ПО, используемого на работе, создается как часть производственного процесса. Пользователи не покупают программное обеспечение, создаваемое в большинстве компаний, специализирующихся на создании ПО. Они покупают игры или текстовые процессоры, или поисковые средства, или устройства аппаратуры. В этом смысле большинство полезного ПО является частью чего-то большего.

Изделие или процесс, куда предполагается встроить программное обеспечение, представляет собой реальный продукт. Поэтому в практическом смысле разработку ПО можно назвать частью разработки продукта. Иными словами, чтобы понять бережливый подход к созданию программного обеспечения, было бы полезно разобраться, что собой представляет совершенный процесс разработки продукта.

В основе бережливого подхода к созданию программного обеспечения лежат все те же принципы, на которых основана производственная система Toyota и система разработки продукции Toyota. Поэтому первый шаг в бережливом подходе к созданию ПО состоит в том, чтобы понять эти принципы, которым посвящена следующая глава.

Попробуйте это

1. Посетите веб-сайт компании Toyota и просмотрите имеющееся здесь видео об автоматизации, (`www.toyota.co.jp/en/vision/production_system/video.html`)[23]. Видео о поставках “точно вовремя” и производственной системе Toyota также стоит посмотреть.
2. Склонны ли вы делить сложную работу на мелкие операции? Если требуется отправить 100 писем, как вы их сложите, разместите по конвертам, наклеите бумажки с адресами и марки? Будете ли вы обрабатывать каждое письмо от начала и до конца, или каждую операцию по очереди выполните для всей группы писем? Почему? Попробуйте оценить время, требуемое для каждого метода, и определите, какой из них быстрее. Если у вас есть дети, спросите их, какой подход они используют.
3. В табл. 1.1 перечислены сходства между процессом производства и процессом разработки продукции. Обсудите эту таблицу в вашем коллективе — каждую строку по очереди. Принято ли в вашей среде думать о незавершенной работе как о непроизводительном запасе? Имеют ли смысл прочие аналогии? Аналогии — это обоюдоострый меч. В каких случаях аналогии между производством и разработкой продуктов могут сбивать с толку?

[23] Данный веб-сайт был опубликован в апреле 2006 года. Упомянутую страницу также можно найти по адресу `www.toyota.co.jp/en/`, где следует последовательно перейти по следующим ссылкам: Top Page⇨Company⇨Vision & Philosophy⇨Toyota Production System⇨Video Introducing the Toyota Production System.

4. Вы работаете в коллективе умных людей. Попробуют ли эти люди обойти проблему или они воспримут ее как повод остановить работу и найти источник проблемы? Составьте список десяти наибольших производственных проблем, возникших в вашем коллективе за последнюю неделю. После описания каждой проблемы укажите метод, которым она была решена. Оцените каждую проблему по шестибалльной шкале (от 0 до 5). Оценка 5 означает, что вы уверены в выявлении этой проблемы, в устранении ее причин и в том, что после этого она вряд ли когда-либо возникнет снова. Оценка 0 означает, что нет сомнения в том, что проблема возникнет снова. Каков общий счет?
5. Если члены вашего коллектива инстинктивно стараются обойти проблему, отложив ее решение на пото*м*, значит, у них развиты *неправильные рефлексы*! Подумайте, что потребуется для того, чтобы нетерпимость к проблемам сделать привычной. На одном отдельно взятом участке вашей работы постарайтесь перейти от игнорирования проблем и поиска обходных путей к принципу “останова производственной линии”. Постарайтесь, чтобы это стало привычкой. Затем повторите это для другого участка.

Глава 2

Принципы

Принципы и практика

Принципы — это исходные истины, которые не изменяются во времени и пространстве, в то время как практика — это применение данных принципов в определенной ситуации. Практика может и должна изменяться при переходе из одной среды в другую, а также по мере развития ситуации в данной среде.

Предположим, вы решили изменить практику разработки программного обеспечения, поскольку она приносит не очень хорошие результаты, однако не ясно, какую именно новую практику следует внедрить. Иными словами, вы хотите уйти от чего-то, однако не ясно, куда именно следует направиться. Как лучше потратить время при поиске нового подхода к разработке программного обеспечения — попытаться понять основополагающие принципы или изучить применяемую в разных сферах практику?

Можно учиться в процессе работы: принять какую-либо хорошо зарекомендовавшую себя практику (или опыт) и работать, надеясь, что это, в конце концов, позволит понять основополагающие принципы. И можно попытаться предварительно понять основополагающие принципы, а затем применять их с учетом конкретных условий. Вероятно, наилучшие результаты можно получить, сочетая два подхода. Копирование чужого опыта без понимания основополагающих принципов очень часто приводит к посредственным результатам. Однако, когда принципы изучены и поняты, полезно заимствовать удачный опыт организации, подобной вашей, внеся в него необходимые изменения, чтобы он лучше подходил к вашей среде. В результате внедрение новых принципов вызовет минимум проблем.

Например, когда на предприятии по производству видеокассет, на котором работала Мэри, было решено внедрить подход поставки “точно вовремя”, оказалось невозможно просто скопировать его в виде, в каком он применялся у Toyota, поскольку это предприятие не просто собирало кассеты, а изготавливало их. Поэтому руководству предприятия пришлось тщательно продумать, что означает в местных условиях подход поставки “точно вовремя”. В результате было решено использовать систему сигнализации (канбан)[1] и реализовать планирование с “вытягиванием” (pull-sheduling). Но руководство понимало, что простого сокращения запасов недостаточно для достижения нужного снижения себестоимости, поэтому для проработки всех нюансов нового производственного процесса были привлечены рядовые рабочие и создана “культура останова произ-

[1] Подробнее об этом см. в главе 6.

водственной линии". Подобное сочетание понимания принципов и творческого заимствования существующего опыта привело к оглушительному успеху.

Многие компании, просто скопировавшие систему сигнальных карточек компании Toyota (Toyota's Kanban system) в 90-х годах, получили посредственные результаты. В этих компаниях не поняли, что принцип бережливости должен стать элементом системы управления, нацеленным на ликвидацию потерь (или непроизводительных расходов). Мы подозреваем, что во многих компаниях, где попытались внедрить систему сигнальных карточек (и не получили нужных результатов), просто не сумели выявить упомянутые потери (и ликвидировать их). Аналогично, если компания внедрит у себя гибкую (agile) практику разработки программного обеспечения, но при этом не выявит потерь и не сможет управлять потоками материалов (вернее, в данном случае информации) так, чтобы ликвидировать их (потери), а также не будет культивировать у себя глубокое уважение к рядовым рабочим и партнерам, результаты могут оказаться такими же посредственными.

Разработка программного обеспечения

Практику применения принципа бережливости в управлении производством и цепью поставок не так-то легко внедрить для разработки программного обеспечения, поскольку как разработка, так и программное обеспечение всегда отличаются от производства и снабжения. Давайте рассмотрим оба эти понятия — разработка и программное обеспечение — и проанализируем, в чем заключается их уникальность.

Разработка

Разработка — это процесс трансформирования идей в продукты. Существуют два подхода к осуществлению подобной трансформации. Один подход можно назвать детерминистским, а второй — эмпирическим. Детерминистский подход предполагает в качестве первого шага создание полного описания нужного продукта, за которым следует реализация этого описания. При эмпирическом подходе сначала создается высокоуровневая концепция продукта (т.е. без определения его отдельных нюансов), а затем устанавливаются тщательно определенные обратные связи, регулирующие усилия по разработке таким образом, чтобы получить оптимальную интерпретацию концепции.

Система разработки продукции Toyota полностью "исповедует" эмпирический подход. Сначала создается концепция будущей модели (а не ее полное описание), а затем эта концепция, в процессе разработки, эмпирически дополняется различными "подробностями" до тех пор, пока не будет получен готовый продукт. Например, концепция продукта для модели автомобиля Pirus не упоминает гибридный двигатель, а просто определяет, что на одном литре горючего будущий автомобиль должен проехать не менее двадцати километров (47,5 миль на одном галлоне). Концепция модели Pirus также предусматривает вместительную кабину для пассажиров, однако не содержит размеров будущего автомобиля. Только в процессе разработки коллектив конструкторов определил, что жестким требованиям к расходу горючего лучше всего соответствует гибридный двигатель.[2]

---

[2] Liker J. The *Toyota Way*, McGraw-Hill, *2004*. См. главу 6, посвященную созданию модели автомобиля *Prius*.

Мы полагаем, что любой процесс разработки продукции, который предполагается корректировать в будущем, должен быть эмпирическим, поскольку этот подход предоставляет наилучшие возможности адаптации к изменениям. Как будет указано в следующем разделе, качественное программное обеспечение должно создаваться таким образом, чтобы без проблем адаптироваться к изменениям как в процессе разработки, так и эксплуатации.

Что такое "водопад"

Я не сталкивалась с термином "водопад" (применительно к разработке ПО) вплоть до 1999 года, когда начала работать над государственным проектом. Я была (очень хорошим) программистом в 70-х годах, создавая в то время программное обеспечение, управляющее оборудованием. Сегодня это, наверное, назвали бы встроенным программным обеспечением, однако в те дни компьютеры были слишком громоздки, чтобы их можно было куда-то встроить. Я работала над некоторыми крупными проектами, включающими создание и запуск сложных технологических линий по производству магнитной ленты.[3] Моими коллегами по этим проектам были инженеры с многолетним опытом, которые разрабатывали крупные, сложные, линии по производству ленты. Эти проекты управлялись опытными экспертами, которые знали, как добиться утверждения бюджета и планирования, а затем, основываясь на них, организовать разработку технологической линии. При этом все прекрасно сознавали, что хотя бюджет и планирование важны, все же главной целью является создание оборудования, способного производить высококачественный продукт.

Когда я стала менеджером информационной системы на предприятии по производству видеокассет, я использовала старые подходы (которым научилась в свою бытность программистом, создающим программное обеспечение), всегда помня, что главной целью моего отдела является поддержка производства. Позже я занималась разработкой продуктов, где мы использовали точный, но легко настраиваемый процесс, обеспечивающий прибыльность новых продуктов.

Таким образом мне удавалось избегать "водопадной" методологии разработки вплоть до 1999 года, когда я приступила к работе над государственным проектом. Я была просто озадачена подобным подходом, потому что не могла понять, как это работает; и в действительности это не работало. Когда я сравнивала мой опыт работы над сложными, но успешными проектами с предложенным к использованию "водопадным" подходом, который оказался неуспешным даже в процессе работы со сравнительно небольшими проектами, я решила написать книгу о том, что в действительности работает.[4] В той книге мы сформулировали семь принципов разработки программного обеспечения, которые перечислены ниже. (Мэри Поппендик)

Программное обеспечение

Встроенное программное обеспечение представляет собой составную часть изделия, которая, как предполагается, будет часто изменяться. Если бы эту часть не требовалось изменять, ее вполне можно было бы реализовать аппаратными средствами. Программное обеспечение, применяемое на предприятии, является частью производственного процесса, причем основную сложность этого процесса целесообразно переложить на программное обеспечение. Если регулярно требуется осуществлять сложный расчет или в производственном процессе имеют место сложные информационные потоки, которыми нужно управлять, подобные задачи возлагаются на программное обеспечение. На ПО также возлагаются задачи управления взаимодействием пользователей, различные срочные задачи, а также задачи прогнозирования. В силу

[3] Оборудование для производства магнитной ленты несколько напоминает автоматизированные ткацкие станки, производимые Toyoda Automatic Loom в 20-х годах.

[4] Poppendieck M., Poppendieck T. *Lean Software Development: An Agile Toolkit*, Addison-Wesley, *2003.*

этого хорошая архитектура программного обеспечения сводится к возможности без проблем его изменять.[5] Не удивительно поэтому, что значительно больше половины всего ПО разрабатывается после первого релиза (или выпуска)[6].

Со временем изменения в производственном программном обеспечении имеют тенденцию становиться все более трудными и дорогостоящими. Изменения в ПО ведут к увеличению его сложности, а сложность, в свою очередь, делает программный код менее устойчивым и более склонным к сбоям. Очень часто компании начинают обнаруживать, что их инвестиции в программное обеспечение в итоге материализуются в виде клубка неуправляемого программного кода. Однако этого вполне можно избежать. Лучшие программные продукты используются уже в течение десятилетия или больше и каждый полезный продукт подвергался регулярным изменениям в ходе его использования. При создании таких продуктов в них была заложена архитектура и использован процесс разработки, предусматривающие толерантность к изменениям программного кода.

Семь принципов бережливого подхода при разработке ПО

В данном разделе суммируется информация о семи принципах бережливого подхода к разработке программного обеспечения, которые легли в основу нашей предыдущей книги. Некоторые читатели, возможно, заметят, что описания некоторых принципов перефразированы, однако суть осталась прежней. После описания каждого принципа мы рассказываем о превалирующем мифе, делающим данный принцип противоречащим интуиции (или "контринтуитивным" — counterintuitive) для тех, кто верит в мифы.

Принцип 1: ликвидировать потери

Тайити Оно называл производственную систему Toyota системой менеджмента, ориентированной на "полную ликвидацию потерь".[7] Когда его спрашивали, как это достигается, он объяснял: "Все сводится к совершенствованию графика выполнения работ с момента получения заказа и до получения оплаты за выполненный заказ. При этом сроки сокращаются за счет ликвидации всех ненужных затрат."[8] В целом, именно к этому сводится и бережливое производство. Что же касается процесса создания программного обеспечения, в котором реализованы принципы бережливости, то он преследует все ту же цель (ликвидация потерь), однако начало и окончание процесса разработки могут быть изменены. Отсчет времени начинается с получения заказа (на разработку продукта, удовлетворяющего запросам заказчика — что бы это ни значило в ваших конкретных ус-

---

[5] Например, в статье *Quality With a Name* (автор Jim Shore) хорошая архитектура программного обеспечения определяется следующим образом: "Хорошая архитектура ПО своидт к минимуму время, требуемое для создания, модификации и поддержки программного обеспечения при приемлемом быстродействии (run-time performance)." См.: `www.jamesshore.com/Articles/Quality-With-a-Name.html`.

[6] На "поддержку" программного обеспечения приходится от 40 до 90% его общей стоимости. См. Kajko-Mattsson M., Westblom U., Forssander S., Andersson G., Medin M., Ebarasi S., Fahlgren T., Johansson S. E., Tornquist S., Holmgren M. *Taxonomy of Problem Management Activities*. Материалы Пятой Европейской Конференции по поддержке и восстановлению программного обеспечения (Fifth European Conference on Software Maintenance and Reengineering), *March 2001, p. 1-10.*

[7] Ohno T. *Toyota Production System: Beyond Large Scale Production*, Productivity Press, *1988,* p. 4.

[8] Там же, p. ix.

ловиях) и оканчивается, когда программное обеспечение, соответствующее этим требованиям, поставляется заказчику и начинает им использоваться. Процесс разработки программного обеспечения, основанный на принципах бережливости, фокусируется на сокращении сроков выполнения заказа путем ликвидации всех потерь.

Для ликвидации лишних затрат (или потерь) прежде необходимо их выявить. Поскольку непроизводительная затрата — это все, что не увеличивает потребительскую ценность, первый шаг в ликвидации подобных затрат состоит в том, чтобы понять, что собой представляет потребительская ценность. Также чрезвычайно важно выработать глубокое понимание, что больше всего удовлетворит заказчика после того, как он начнет использовать новое программное обеспечение. Это не так-то просто, поскольку заказчик ПО часто сам не до конца понимает, чего он хочет. Кроме того, после того как заказчик видит вновь созданное программное обеспечение в действии, представление о том, что ему нужно, обычно меняется. Тем не менее успешные компании-разработчики ПО прилагают огромные усилия, стараясь понять, что заказчику нужно, и очень часто удовлетворяют их. Вспомните Google: это поисковое средство давно и неизменно восхищает пользователей всего мира.

После того как понимание, что собой представляет потребительская ценность выработано, следующий шаг состоит в том, чтобы по-настоящему научиться распознавать потери. Это понятие охватывает все, что не помогает предоставить заказчику то, что ему нужно, где ему нужно и когда ему нужно. Все, что мы делаем, если это не работает на то, чтобы наиболее полным образом удовлетворить требования заказчика, является источником непроизводительных затрат. Сюда же можно отнести любую задержку выполнения заказа.

В производстве источником непроизводительных затрат являются запасы (деталей, узлов и материалов), которыми необходимо управлять (перемещать, складировать, вести их учет и т.п.). Это связано не только с затратами времени и усилий, но также увеличивает сложность процесса — важный фактор роста стоимости. Запасы (деталей, узлов и материалов) теряются, устаревают, скрывают в себе проблемы с качеством и омертвляют вложенные средства. Поэтому одна из целей производства — поддерживать запасы на минимальном уровне.

Запасами в процессе разработки программного обеспечения является частично выполненная (или незавершенная) работа. Незавершенному ПО присущи все пороки, свойственные запасам в производстве (неиспользуемый программный код теряется, устаревает, содержит проблемы с качеством и "омертвляет" вложенные средства). Более того большинство рисков, связанных с созданием ПО, кроется как раз в незавершенной работе.

Значительная доля потерь в разработке программного обеспечения связана с необходимостью переделывать то, что уже было сделано. Например, нередко приходится переделывать уже созданное ПО из-за того, что пока оно создавалось, изменились требования заказчика. Это достаточно типичная ситуация, и от 30 до 50% требований, сформулированных задолго до начала разработки ПО, к концу процесса обычно изменяются. Также необходимость переделывать уже сделанную работу часто возникает, когда тестирование производится со значительной задержкой. (Если при разработке в программный код был внесен дефект, который сразу не был обнаружен и устранен, созданный после этого код (писавшийся с учетом имеющегося дефекта), когда дефект будет устранен, может оказаться неработоспособным и его придется переделывать.) К сожалению, эти случаи часто являются только предтечей еще более крупных потерь, связанных с необходимостью повторного выполнения уже сделанной работы, когда имеет место отложенная (или небрежная) интеграция.

Однако значительно больший источник непроизводительных затрат при разработке программного обеспечения — это избыточные функциональные возможности. Только около 20% функциональных возможностей в типичном пользовательском программном обеспечении используется регулярно, и около двух третей возможностей используется редко.[9] Мы не имеем в виду обязательные средства безопасности. Речь идет о функциях, особой необходимости в которых нет изначально. Добавление избыточных функциональных возможностей в программное обеспечение обходится чрезвычайно дорого. Они увеличивают сложность программного кода, затрудняя его последующее совершенствование, и тем самым сокращают "срок его жизни".

Миф: созданная заранее спецификация сокращает потери

Причины появления в программном обеспечении лишнего кода кроются в игре, в которую мы часто играем с заказчиками. Причем правила для этой игры устанавливаем мы сами.

> Будьте так любезны, драгоценный заказчик, предоставьте нам список того, что программное обеспечение должно делать. И будьте добры расписаться внизу. После этого, если потребуется что-либо изменить или добавить, вам придется пройти через запутанный процесс, называемый "управление изменениями", чтобы добиться одобрения каждого изменения. Поэтому, если вы хотите получить хорошее программное обеспечение, постарайтесь все предусмотреть заранее, поскольку нам нужно знать обо всем этом до начала процесса разработки.

Не удивительно, что наши заказчики включают в этот список все, что только можно. И очень часто меры, призванные ограничивать масштабы создаваемого программного обеспечения, дают противоположные результаты. Подобно изготовлению лишних деталей, которое Тайити Оно называл наихудшими потерями в производстве, неиспользуемые функциональные возможности в программном обеспечении можно назвать так же. Каждая избыточная частичка программного кода, увеличивающая его сложность, "отравляет" код на всю его оставшуюся жизнь. Неиспользуемый код также нуждается в тестировании, документировании и поддержке. Он вносит свою лепту в повышение неустойчивости всего программного кода, трудность его понимания и совершенствования в будущем. Расходы, связанные со сложностью, превосходят все прочие расходы по поддержке программного кода; кроме того, ненужные функциональные возможности — главный фактор, снижающий быстродействие программного обеспечения.

Целесообразно сначала полностью разработать 20% программного кода, которые обеспечивали бы 80% потребностей заказчика, и только после этого переходить к разработке следующих наиболее необходимых функциональных возможностей. Не следует заранее составлять список всех возможностей, которые могут когда-либо понадобиться, особенно если этот список исходит от заказчиков, которые не очень хорошо знают, что им нужно.

---

[9] Когда Джим Джонсон (Jim Johnson), председатель *Standish Group*, сообщил об этих цифрах на конференции по экстремальному программированию в 2002 году на Сардинии, тогда они основывались на неполных данных. С тех пор мы спрашивали почти в каждой нашей группе, соответствуют ли данные цифры их опыту, особенно если основное занятие слушателей заключалось в создании программного обеспечения на заказ. Ответы всегда подтверждали, что эти цифры, если и не абсолютно точны, все же приблизительно соответствуют истине.

Принцип 2: встраивать качество

Мы должны повышать дисциплину часто говорили нам настроенные скептически менеджеры. Мы возражали, что бережливый подход к разработке программного обеспечения как раз и предполагает соответствующий уровень дисциплины. Просто на дисциплину следует взглянуть несколько иначе. Целью с самого начала является встраивание качества в программный код (build quality into the code), а не тестирование этого кода после его создания. Следует сделать так, чтобы дефекты просто не попадали в код. Очевидно, что это можно осуществить только в организации, где дисциплина на очень высоком уровне.

Списки дефектов

Мы посетили организацию, у которой был сертификат CCM Level 4 и которая вскоре должна была получить сертификат CCM Level 5. На нас произвели впечатление люди и дисциплина; было не удивительно, что их программные продукты пользовались хорошей репутацией.

Но что-то было не так. Недавно срок выпуска очередного программного продукта у них растянулся с шести недель до четырех месяцев и они просили нас разобраться почему. Мы создали графическое представление их процесса разработки и в конце его обнаружили четырехнедельный период тестирования. "Не удивительно, что вы не уложились в шесть недель. Четыре недели ушло только на тестирование вашего продукта", — сказала им я.

"Я знаю, мы должны автоматизировать процесс тестирования, — ответил менеджер по качеству, — но это представляется упрощенным подходом." Я согласилась. Это была первоклассная организация. Но ее проблемы, вероятно, не отграничивались только тем, что могло быть решено с помощью автоматического тестирования.

"Давайте сделаем анализ Парето[10] четырехнедельного периода тестирования, — предложила я. "Чем вы были заняты в эти четыре недели; что заняло наибольшее время?"

"Устраняли дефекты", — последовал немедленный ответ.

"А, так в действительности вы не занимались тестированием все четыре недели, вы устраняли дефекты!" — сказала я. "Тогда неправильно это называть тестированием. Предположим, никаких дефектов, которые бы потребовалось исправлять, не нашлось. Сколько времени в этом случае заняло бы тестирование?"

"Возможно, два-три для", — был ответ.

"Если вы смогли бы выполнять тестирование за два-три дня, тогда можно ли было бы вернуться к шестинедельным срокам на разработку ваших продуктов?" — спросила я.

"Без сомнения."

"Похоже, что тестирование вы проводите слишком поздно. Вам бы следовало выявить все эти дефекты гораздо ранее", — предложила я.

"Но мы так и делаем, — он ответил, — все эти дефекты зафиксированы в нашей системе выявления дефектов."

"Вы хотите сказать, что знаете о них, но откладываете их устранение до самого последнего момента?" — удивленно спросила я.

---

[10] В основе анализа Парето лежит правило "важных несколько и мелких много", так же известное, как правило "80/20", которое впервые было популяризовано Дж. М. Джураном (J.M. Juran) — гуру в области управления качеством. Анализ осуществляется следующим образом: разделите большую проблему на несколько более мелких проблем, определите самую крупную из этих мелких проблем, найдите источник этой проблемы и устраните его. После ликвидации источника крупнейшей из этих мелких проблем, примените данный подход к оставшимся проблемам. Более подробный пример данного анализа можно найти в главе 7.

"Да", — последовал робкий ответ.

"ОК, вы знаете, что вам следует делать", — сказала я. "Не занимайтесь учетом дефектов при их обнаружении, а тут же устраняйте их! Сократите завершающее тестирование до пары дней, заставив всех поверить, что к началу завершающего тестирования программный код должен работать." У меня не было сомнений, что этот очень профессиональный коллектив сможет справиться с данной целью, как только люди поймут, что это важно. (Мэри Поппендик)

Согласно Сигаэо Синго, возможны две разновидности контроля: контроль с целью обнаружения дефекта (когда он уже имеется) и контроль с целью его предотвращения.[11] Если вам по-настоящему нужно качество, вы будете контролировать условия с тем, чтобы изначально предотвратить появление дефектов, а не устраивать контроль (или тестирование) после того как факт имел место,. А если это не возможно, вы будете проверять продукт после каждого небольшого этапа разработки с тем, чтобы дефекты выявлялись немедленно после их появления. По обнаружении дефекта вы "остановите производственную линию", выясните его первопричину, а затем немедленно устраните дефект.

Системы обнаружения дефектов — это очереди незавершенных работ; если угодно, очереди работ, нуждающихся в исправлении. Очень часто мы полагаем, что, поскольку дефект помещен в очередь, все в порядке, он уже никуда не денется. Однако с точки зрения концепции бережливой разработки программного обеспечения очереди — это коллекторы непроизводительных затрат. Целью является не иметь в очереди ни одного дефекта, а еще более "целесообразная цель" — не иметь такой очереди совсем. Если вы считаете, что это невозможно представить, познакомьтесь с трехгодичным опытом работы Нэнси Ван Шундерворт (Nancy Van Schooenderwoert) над проектом по разработке сложного и часто изменяющегося встроенного программного обеспечения.[12] За три года был в целом обнаружен 51 дефект после блочного (или модульного) тестирования, причем каждый раз выявлялось не более двух дефектов сразу. Кому нужна система отслеживания для всего двух дефектов?

Сегодня существуют средства, позволяющие устранять дефекты из программного кода в ходе его создания. Группа разработчиков, используя разработку через тестирование (test-driven-development), создает блочные тесты и приемочные тесты до создания соответствующего кода. Разработчики интегрируют код и тесты в систему так часто, как это возможно (каждый час или около того) и инициируют запуск тестовой нагрузки (test harness), чтобы убедиться, что со времени предыдущей проверки дефекты в коде не появились. Если тест не выполняется, они не создают новый код, пока проблема не будет устранена. В конце дня запускается более сложная и полная тестовая нагрузка. В конце недели система проверяется с помощью еще более сложной тестовой нагрузки. В организациях, где используется такой подход, дефекты возникают редко и их причины устраняются очень быстро.

[11] Shingo S. *Study of 'Toyota' Production System*, Productivity Press, *1981*, Chapter 2.3.

[12] Schooenderwoert N., Morsicato R. Taming the Embedded Tiger—*Agile Test Techniques for Embedded Software*, Proceedings Agile Development Conference, Salt Lake City, Июнь, 2004, and Schooenderwoert N. *Embed-ded Agile Project by the Numbers with Newbies*, Proceedings Agile 2006 Conference, Minneapolis, *Juky 2006*.

Рост производительности[13]

Разработка через тестирование (Test Driven Development — TDD) — это удивительно эффективный подход к повышению качества программного кода. В феврале 2004 года я предложил внедрить этот метод в компании, где качество программного обеспечения превратилось в проблему. Хотя ни одна "порция" кода здесь не была создана более чем пять лет назад, тем не менее, подобные приложения рассматривались как "унаследованные" (legacy). В среднем, в каждом релизе (которые следовали с интервалом в шесть месяцев) обнаруживалось десять дефектов на каждую тысячу строк (без учета комментариев) программного кода (написанного на Java).

Низкое качество программного обеспечения этой компании отрицательно сказывалось на ее репутации и затрудняло последующее совершенствование этого ПО.

Чтобы исправить положение, в компании сначала использовали гибкий (agile) процесс разработки Scrum[14], а вскоре начали применять и разработку через тестирование (TDD). После применения TDD количество выявляемых дефектов сократилось до менее чем трех на тысячу строк программного кода. Однако на практике это 70-ное сокращение числа дефектов — показатель преуменьшенный, поскольку никто не выяснял, когда данный дефект попал в код — до или после внедрения TDD. Реальное сокращение числа дефектов (по скромной оценке) было бы от 80 до 90%.

Теперь коллектив разработчиков тратил гораздо меньше времени на выявление дефектов (как с помощью отладчика во время разработки, так и позже, когда сообщение об обнаруженном дефекте поступало от пользователей). В результате производительность после внедрения TDD возросла втрое. (Майк Кон (Mike Cohn), президент Mountain Goat Software)

Миф: цель тестирования — выявление дефектов

Целью процесса тестирования (а также коллектива, разрабатывающего тесты и проводящего тестирование) должно быть *предотвращение* дефектов, а не поиск их. В организации, где качеству уделяется должное внимание, работа должна быть ориентирована на удаление дефектов из программного кода в процессе его создания (если угодно, это можно назвать встраиванием качества в код), а не на проверку качества кода позже. Авторы не хотят сказать, что заключительная проверка кода излишня. Просто выявление дефектов во время заключительной проверки должно быть исключением, а не правилом.

На предприятии, где работала Мэри, полагали, что 80% дефектов, виновниками которых якобы являются люди, в действительности исходят от системы, допускающей появление этих дефектов. Иными словами, подавляющее большинство дефектов — это в действительности проблема менеджмента. В ходу был лозунг: "Делать сразу правильно", который интерпретировался как необходимость изжить привычку откладывать устранение дефектов на потом. Для этого было необходимо использовать защиту от оплошности на каждом этапе производства с тем, чтобы было легко предотвращать появление дефектов.

Лозунг "Делать сразу правильно" в разработке программного обеспечения должен означать широкое применение разработки через тестирование (Test Driven Development — TDD), чтобы сразу же обеспечить нужное функционирование программного кода. К сожалению этот лозунг часто трактуется совершенно иначе: как только код создан, он уже никогда не должен изменяться. Подобная интерпретация поощряет разработчиков использовать одну из наиболее порочных практик, возможных при разработке сложных

[13] Огромная благодарность Майку Кону за то, что он поделился с нами опытом. Использовано с разрешения.

[14] Хорошо известная итерационная методология разработки программного обеспечения. Schwaber K., Beedle M. Agile Software *Development with SCRUM*, Prentice Hall, *2001;* Schwaber K. *Agile Project Management with Scrum*, Microsoft, *2004.*

систем. Полагать, что программное обеспечение, будучи созданным, больше никогда не должно меняться — это опасный миф.

Давайте вспомним наилучший способ сократить непроизводительные расходы при разработке программного обеспечения: создавать минимальный код. Чтобы программный код получился минимального объема, следует определить, какие 20% функций (из всех возможных) будущего программного обеспечения могут удовлетворить 80% требований заказчика и создать соответствующий код первым. После этого следует добавлять новые функциональные возможности до тех пор, пока стоимость создания очередной функции не превысит ее полезность. При добавлении новых функций необходимо заботиться о простоте программного кода, иначе скоро станет невозможно с ним работать.

Принцип 3: создавать знание

Один из удивительных аспектов "водопадной" модели разработки состоит в том, что некоторое знание в форме "технических условий" создается до (и отдельно от) создания программного кода. Разработка ПО — это процесс создания знания. Хотя общая архитектурная концепция может быть обозначена до начала создания кода, подтверждение (или не подтверждение) правильности этой архитектуры имеет место всегда после. На практике реальная архитектура программного обеспечения всегда материализуется в процессе разработки, даже если подробный документ, описывающий эту архитектуру, был составлен предварительно. При предварительном определении архитектуры невозможно учесть ни всей сложности, с которой приходится сталкиваться в процессе ее реализации, ни обратной связи с процессом разработки. Что еще хуже, заданная предварительно и подробно архитектура не чувствительна к обратной связи с инвесторами и заказчиками. Процесс разработки, направленный на создание знания, предполагает, что архитектура программного обеспечения возникнет во время создания программного кода, и исключает затраты на его предварительное определение.

Алан Мак-Кормак (Alan MacCormack), профессор Гарвардской школы бизнеса (Harvard Business School), затратил много времени на изучение процесса приобретения опыта в организациях. Не так давно, когда он изучал практику разработки программного обеспечения в одной из компаний, к нему обратились с просьбой оценить два проекта — "хороший" и "плохой"[15]. "Хороший" проект осуществлялся строго по инструкции. Была создана оптимизированная архитектура, и полученная в результате система в высокой степени соответствовала заданной вначале спецификации. А "плохой" проект хаотически подвергался постоянным изменениям в ходе его создания, по мере того как работающий над ним коллектив старался понять и отреагировать на изменения требований рынка.

После оценки проектов с использованием критериев Мак-Кормака выяснилось, что "хороший" проект имеет низкое качество, недостаточное быстродействие и плохо соответствует требованием рынка, в то время как рыночный успех "плохого" проекта превзошел все ожидания. Однако ничего удивительного нет в том, что коллектив, старавшийся в процессе разработки уловить требования рынка, создал лучший продукт. Настоящим откровением явилось то, что менеджеры компании искренне полагали, что пытаться понять запросы рынка и учитывать их при создании программного продукта — это плохо.

---

[15] Эта история рассказана в статье *Creating a Fast and Flexible Process: Research Suggests Keys to Success* (автор Alan MacCormack), которую можно найти в Интеренете по адресу `www.roundtable.com/MRTIndex/FFPD/ART-maccormack.html`.

Мак-Кормак идентифицировал четыре подхода, ведущие к созданию успешного программного обеспечения.[16]

1. Ранний релиз с минимумом функциональных возможностей с тем, чтобы потребители могли его оценить и выразить свое мнение, пожелания и претензии.
2. Ежедневный выпуск сборок, их тестирование и учет (в дальнейшей разработке) результатов тестирования.
3. Наличие коллектива и (или) лидера с достаточным опытом и развитой интуицией, позволяющих принимать правильные решения.
4. Использование модульной архитектуры, поддерживающей возможность добавления новых функций.

Компании, долгое время демонстрирующие великолепные результаты в создании программного обеспечения, имеют одну общую особенность: они генерируют новое знание в процессе регулярного экспериментирования и сохраняют это знание в виде, удобном для доступа к нему в более крупных организациях. Такие компании аккумулируют не только явное знание, они находят способы неявное знание представить в более определенном виде и сделать его частью базы знаний организации.[17] В этих компаниях существует понимание, что, хотя приобретение опыта при создании программного продукта важно, сохранение этого знания в виде, удобном для использования в процессе создания новых продуктов, еще важнее.

Важно организовать процесс разработки, который поощряет систематическое и целенаправленное приобретение опыта, но так же важно систематически совершенствовать этот процесс. Иногда в попытках создать "стандартный процесс" реальные процессы разработки программного обеспечения ограничиваются рамками документации, что затрудняет коллективам разработчиков совершенствовать эти процессы. В организации, где внедрены принципы бережливости, знают, что необходимо постоянно совершенствовать используемые процессы, поскольку в комплексной среде проблемы неизбежны. Каждая аномалия должна инициировать поиск источника, вызвавшего проблему, а также поиск средств ее решения, а затем соответствующие изменения должны быть внесены в процесс разработки, чтобы предотвратить возникновение этой проблемы вновь. Усилия по совершенствованию процесса разработки должны стать стремлением и ответственностью коллектива разработчиков, и в каждом коллективе должно выделяться время специально для работы над совершенствованием этого процесса на регулярной основе.

Миф: прогнозы обеспечивают предсказуемость

Предсказуемые результаты — одно из основных требований, навязываемых компаниям и их высшему менеджменту рынком, и эти ожидания в конце концов оказывают влияние на разработку программного обеспечения. Однако (к сожалению) разработка программного обеспечения известна своей непредсказуемостью, поэтому на соответствующие организации постоянно оказывается давление с целью сделать их более предска-

---

[16] В дополнение к статье, о которой речь шла выше, см. также MacCormack A. *Product-Development Practices That Work: How Internet Companies Build Software*, MIT Sloan Management Review, *Winter 2001*, Vol. 40, № 2.

[17] Nonaka I., Takeuchi H. *The Knowledge Creating Company: How Japanese Companies Create the Dynamics of Innovation*, Oxford University Press, *1995,* p. 225.

зуемыми. Парадокс в том, что в стремлении увеличить предсказуемость разработки программного обеспечения часто имеет место практика, вызывающая обратный эффект. Сначала создается план, а затем вся последующая работа строится согласно этому плану, словно данный документ — это точное определение будущего. И все решения принимаются согласно этому плану, который трудно пересмотреть. При этом теряется способность гибко реагировать на изменения, когда прогнозы не оправдываются. Решение этой проблемы, может показаться, лежит в том, чтобы делать более точные прогнозы.

Мы забываем, что прогнозы обычно не оправдываются, если они:

- чересчур сложны;
- слишком подробны;
- относятся к отдаленному будущему;
- касаются неопределенных вещей.

Однако существуют проверенные способы получения надежных результатов, даже если они не были сформулированы заранее в виде прогноза. Секрет в том, чтобы перестать действовать так, словно предсказания (в действительности являющиеся всего лишь прогнозом) — это факт, который если еще не имеет места, то гарантированно станет реальностью в будущем. Взамен необходимо сократить время реагирования с тем, чтобы можно было корректно реагировать на события по мере того, как они происходят.

С целью увеличения предсказуемости результатов необходимо уменьшить количество прогнозов, на основе которых принимаются решения. Решения, основанные на фактах, а не на прогнозах, ведут к наиболее предсказуемым результатам. Как инженер по контролю, Мэри знает, что эмпирический подход к процессу разработки (который основывается на обратной связи) позволяет получить более предсказуемые результаты, чем детерминистский подход. В целом, организация, где хорошо развита способность реагировать быстро и корректно на события, даст более предсказуемые результаты, чем организация, в которой пытаются предсказать будущее.

Принцип 4: откладывать необратимые решения

Люди, чья профессия состоит в том, чтобы реагировать на чрезвычайные ситуации, специально обучаются адекватно вести себя в непредсказуемых и часто опасных обстоятельствах. Такие люди умеют быстро оценить ситуацию и определить, как долго можно ждать, прежде чем принять критическое решение. Определив время для такого решения, они учатся ждать до последней секунды, прежде чем предпринять ответные действия, поскольку именно тогда они будут располагать наиболее полной информацией.

Та же логика подходит для необратимых решений, которые приходится принимать в процессе создания программного обеспечения: отложить принятие подобного решения на последний момент, — т.е. до последнего момента, прежде чем будет слишком поздно. (Собственно, не следует торопиться с любыми решениями и следует попытаться сделать так, чтобы все решения были обратимы.) Однако, определяя заранее свойства системы, следует избегать принимать решения, которые трудно было бы изменить. Программная система не нуждается в абсолютной гибкости, однако необходимо, чтобы в определенных точках процесса (там, где, скорее всего, потребуется вносить изменения) сохранялась возможность выбора того или иного варианта. Коллектив и (или) руководитель с опытом в данной области должен развить у себя соответствующую интуицию с тем, чтобы ориентироваться, в какие моменты необходимо иметь выбор из нескольких вариантов.

Многие предпочитают принимать решения быстро с тем, чтобы сразу уменьшить неопределенность. Однако при неопределенности, особенно когда она сопровождается сложностью, наиболее успешный подход заключается в том, чтобы экспериментировать с различными вариантами, откладывая принятие решения до последнего момента. В действительности многие лучшие стратегии создания программного обеспечения изначально нацелены на то, чтобы оставлять возможность выбора одного из нескольких вариантов как можно дольше, чтобы необратимые решения принимались в последний момент.

Миф: план — это обязательство

"Я всегда полагал, что при подготовке к сражению планы бесполезны, однако планирование необходимо." Эта известная цитата Дуайта Эйзенхауэра (Dwight Eisenhower) позволяет понять разницу между планированием с целью изучения возможных вариантов и принятием обязательных для выполнения планов. Планирование — важное упражнение, позволяющее развить соответствующие рефлексы, и оно необходимо для определения высокоуровневой архитектуры сложной системы.

В то же время от плана, если он рассматривается как что-то обязательное, часто больше вреда, чем пользы. Вот что говорит по этому поводу Тайити Оно.[18]

> Планы изменяются очень легко. Реальная практика не всегда соответствует планам и решения, соответствующие изменившимся обстоятельствам, должны приниматься быстро. Если все время придерживаться однажды принятой идеи и не вносить изменений в планы, в такой обстановке бизнес не сможет существовать долго.
>
> Поговорка гласит, что чем здоровее позвоночник, тем легче он гнется. Гибкость важна. Если с позвоночником возникли проблемы и тело помещается в гипс (необходимая мера), позвоночник перестает функционировать. Строгое следование заданному плану — все равно, что лишить позвоночник гибкости. Это вредно для здоровья.

Барри Боум (Barry Boehm) и Ричард Тернер (Richard Turner)[19] изобрели понятие "методы разработки согласно плана" (plan-driven methods) для обозначения методов создания программного обеспечения, рассматривающих план в качестве принятого обязательства. Они определили процесс разработки согласно плана как имеющий "естественно присущее ему свойство ... обеспечивать запланированные результаты".[20] Также они указали, что методы разработки согласно плана возникли в мире государственных контрактов.[21] Действительно, сложно осуществлять жесткое управление контрактами, как это привыкли делать государственные организации, без создания подробного плана, который бы рассматривался как обязательство для компаний-исполнителей контрактов.

Однако в коммерческом мире компании, где существует понимание своей выгоды (такие как Toyota) осознали, что придерживаться подробно определенного плана плохо для бизнеса. Не следует поддаваться мифу, что планирование — это то же, что и принятие обязательств. Планировать следует тщательно, а брать обязательства осторожно.

---

[18] Ohno T. *Toyota Production System: Beyond Large Scale Production*, Productivity Press*, 1988,* p. 46.
[19] Boehm B., Turner R. *Balancing Agility and Discipline: A Guide for the Perplexed*, Addison-Wesley*, 2004.*
[20] Там же, p. 12.
[21] Там же, p. 10.

Принцип 5: доставлять быстро

Когда наши дети были такие маленькие, что их подбородки были на уровне стола (когда они сидели на стульях), всегда можно было найти пару толстых каталогов Sears в любом доме, где мы были в гостях, и, положив их на сиденья стульев, посадить детей повыше. В те времена все любили перелистывать такие каталоги, и мы часто заказывали по ним различные товары. Доставка занимала две-три недели, поэтому мы не покупали таким образом товары, которые можно было приобрести в местном супермаркете. Где-то в середине 80-х годов компания L.L. Bean (специализировавшаяся на заказах по почте) из Мэйна (Maine) решила конкурировать с Sears в сроках доставки. Ее целью было обеспечить доставку в течение 24 часов после оформления заказа. Это оказалось такой удачной идеей, что другие компании начали посещать дистрибуционный центр L.L. Bean, чтобы познакомиться, как компании удалось это.

Конкуренция продолжалась недолго, поскольку двух- или трехнедельная доставка вскоре начала казаться ужасно медленной. В результате приближающийся к своему 100-летию почтенный каталог Sears оказался неконкурентоспособным и закрыл свой магазин в 1993 году. Услуги L.L. Bean обходились намного дешевле, поскольку быстрая доставка исключала (почти) переоформление заказов. Только подумать: если мы заказывали желтую рубашку в Sears, можно было позвонить им неделю или две спустя и сказать: "Знаете, мы передумали: взамен желтой пришлите нам, пожалуйста, голубую рубашку". И компания бы внесла изменения в заказ. Однако если бы мы попробовали обратиться в L.L. Bean с той же просьбой, они бы, скорее всего, ответили: "Почему бы вам не открыть дверь и посмотреть на крыльцо? Заказ должен быть доставлен сегодня!"

Мораль этой истории в том, что программное обеспечение должно доставляться заказчикам так быстро, чтобы у них не оставалось времени на то, чтобы передумать.

Доставка за ночь была изобретена почти исключительно компанией FedEx, которая лидировала в своей области (даже несмотря на то, что вскоре появились аналогичные компании). Компания Dell создала систему срочных поставок (собиравшихся на заказ) компьютеров, которую оказалось очень трудно копировать конкурентам. Toyota смогла выпустить свою революционную модель Pirus с гибридным двигателем в рекордные 15 месяцев — скорость поставки продукта, приблизиться к которой смогли очень немногие компании.

Компании, которые побеждают конкурентов по срокам выполнения заказов, часто имеют существенное преимущество также и в ценах на свои услуги. Они сумели ликвидировать бо́льшую часть непроизводительных затрат, и это принесло им существенную экономию. Кроме того, у них очень низкий процент брака. Скорость невозможна без высокого качества. Более того, чтобы обеспечить быстрое выполнение заказа, необходимо хорошо понимать требования заказчика. Подобные компании работают так быстро, что могут позволить себе экспериментальный подход к поставке продуктов, пробуя новые идеи, чтобы определить, какие из них работают.

Миф: спешка ведет к браку

В индустрии создания программного обеспечения долго считалось, что для достижения высокого качества "необходимо снизить темпы и проявлять осторожность". Однако когда отрасль навязывает заказчикам подобный компромисс, отказавшаяся от компромисса ком-

пания приобретет существенное конкурентное преимущество.[22] Google и PatientKeeper, речь о которых пойдет далее в этой книге (в главах 3 и 5 соответственно), — только две компании из многих, доставляющих свое программное обеспечение быстро и с высоким качеством. А занятые разработкой программного обеспечения компании, продолжающие верить, что быстрота и качество несовместимы, имеют все шансы в не столь отдаленном будущем последовать по пути каталога Sears.

Предостережение: не приравнивайте оперативность к поспешности. Это совершенно разные понятия. Мобильный коллектив разработчиков должен выработать в себе развитую интуицию и культуру останова производственной линии. Причина очевидна: вы не сможете поддерживать высокую скорость работы, если не будете встраивать качество в создаваемые продукты.

В борьбе за дисциплину во многих организациях разрабатываются подробные планы, стандарты и спецификации для различных работ, а также описания всего процесса. Это часто осуществляется специальной группой, которая консультирует работников и следит за соблюдением всех стандартов и норм. Целью всех этих усилий является выработка стандартизированного, повторяющегося процесса, который, среди прочего, облегчает перемещение людей из одного проекта в другой.[23] Однако в условиях процесса, созданного таким образом, чтобы участвующих в нем людей было легко заменить, невозможно появление работников (обладающих соответствующими качествами и квалификацией), необходимых для функционирования быстрых, гибких производств.

Если вы хотите работать быстро, для этого необходимы заинтересованные, думающие люди, от которых можно ждать хороших решений. В мобильных, быстро воспринимающих инновации коллективах, работа организована таким образом, чтобы работники без указаний знали, что им следует делать каждую минуту, а также решали проблемы и адаптировались к изменениям, не ожидая разрешения. Организации, где внедрены принципы бережливости, работают в соответствии со стандартами, однако эти стандарты содержат в себе лучшее актуальное знание, как следует выполнять работу. Соответствующие бережливости стандарты не возбраняется ставить под сомнение и совершенствовать.[24]

Существует два пути достижения высокого качества. Можно работать медленно и осторожно. И можно воспитать людей, которые постоянно будут вносить совершенствования в свою работу, встраивать качество в создаваемые продукты и разовьют в себе способности, позволяющие быстро и надежно отвечать на требования заказчиков — гораздо быстрее конкурентов.

Принцип 6: уважать людей

Когда Джоэль Спольски (Joel Spolsky) завершил учебу в колледже и приступил к работе в Microsoft, перед ним была поставлена задача определить стратегию макроязыка для Excel. Он потратил некоторое время, изучая пожелания пользователей (относительно макросов), а затем написал спецификацию. К его большому удивлению, группа разра-

---

[22] Stalk G., Lachenauer R. *Hardball: Are You Playing to Play or Playing to Win*, Harvard Business School Press, *2004.*

[23] Boehm B., Turner R. *Balancing Agility and Discipline: A Guide for the Perplexed*, Addison-Wesley, *2004,* p. 11–12.

[24] Shook J. *Bringing the Toyota Production System to the United States in Becoming Lean*, Productivity Press, 2004, p. 59–60.

ботчиков, занятая созданием архитектуры будущего приложения, встретила в штыки его спецификацию и потребовала пересмотреть ее. Он подумал, что у них есть полезные советы для него, однако обнаружилось, что в маленькой группе (включавшей четырех докторов физики, а также босса, занимавшего шестое место в рейтинге компании, и личного друга Билла Гейтса) о макросах знают меньше его. Несмотря на поверхностные представления о том, как пользователи используют макросы, группа разработки архитектуры полагала, что это их работа определять, какой должна быть реализация макросов. Менеджеры Джоэля дали ясно понять, что решения в этом вопросе следует принимать ему, и его группа разработчиков поддержала Джоэля. Однако создателям архитектуры приложения это не понравилось и их босс созвал совещание, чтобы пожаловаться на то, как Джоэль вносит путаницу в создание стратегии макросов.

Если бы Microsoft была обычной компанией, нетрудно представить, чем бы окончился этот сценарий: разработчики архитектуры настояли бы на своем и Джоэлю пришлось бы или уступить, или уволиться. Однако случилось не так. На следующий день после совещания старший вице-президент компании встретился с Джоэлем в кафетерии и спросил, как ему работается с группой разработки архитектуры. Джоэль, конечно, ответил, что все обстоит прекрасно. Однако еще через день пополз слух, что группа разработки архитектуры распущена. На Джоэля этот случай произвел глубокое впечатление. "В Microsoft, — говорил он, — если вы отвечаете за стратегию макроязыка, даже если вы работаете в компании менее шести месяцев, вы Бог в том, что касается макросов Excel, и никто, в том числе и руководитель с шестым рейтингом в компании, не может становиться на вашем пути."[25]

История Джоэля иллюстрирует значение шестого принципа (уважать людей) в сфере разработки программного обеспечения с точки зрения тех, кто выполняет реальную работу.

Интересно, что три из четырех важнейших принципов системы разработки продукции Toyota (см. рис. 1.5)[26], касаются людей, занятых в процессе разработки. Знакомство с этими принципами позволит получить более глубокое понимание того, в чем состоит уважение к людям.

1. **Руководитель-предприниматель.** Люди предпочитают работать над успешными продуктами, а такие продукты обычно создаются под руководством превосходных руководителей. В компаниях, где развито уважение к людям, должное внимание уделяется воспитанию творческих руководителей. Подобные руководители затем возглавляют коллективы, и одной из их задач является поощрение заинтересованных, думающих людей и направление их усилий на создание выдающихся продуктов.
2. **Технические специалисты высшей квалификации.** Компания, рассчитывающая добиться преимущества над конкурентами в определенной области, должна воспитывать нужных ей специалистов. Компании, предпочитающие приглашать специалистов со стороны, неизбежно обнаруживают, что следуют за конкурентами. А компании, полагающие, что специалисты им не нужны, вскоре столкнутся с тем, что у них нет никакого преимущества перед конкурентами. В компаниях, где преобладает мудрость, уделяют внимание воспитанию своих специалистов и забо-

[25] Spolsky J. *Two Stories*, `http://www.joelonsoftware.com/articles/twostories.html`. Использовано с разрешения.

[26] Kennedy M. Product Development for the Lean Enterprise: Why Toyota's System Is Four Times More Productive and How You Can Implement It, Oaklea Press*, 2003,* p. 120. Использовано с разрешения.

тятся, чтобы все коллективы состояли из работников, способных достичь поставленных целей.

3. **Планирование и контроль, основанные на ответственности.** Уважение к людям означает, что коллективу дается общий план и определяются реальные задачи, а дальше коллективу предоставляется свобода действий в их выполнении.

Миф: существует наилучший метод

Книга *Cheaper by the Dozen*[27] ("Дюжина дешевле") — очень забавная и правдивая история о семье где 12 детей и отец — специалист по эффективности. Поскольку Мэри была четвертым ребенком в семье, где 11 детей, понятно, что она охотно прочитала эту книгу в детстве. Однако от нее ускользнуло (пока она не перечитала эту книгу недавно), что книга *Cheaper by the Dozen* дает нам уникальный взгляд на то, откуда берет начало наука об управлении. Отец семейства, Фрэнк Гилберт (Frank Gilbreth), консультант с мировым именем, настаивал, что для каждой задачи существует "наилучший метод" ее выполнения. Читая между строк, нетрудно догадаться, что (судя по почти военной дисциплине, введенной среди домочадцев) большинство рабочих вряд ли захотели бы быть объектом его усилий по повышению эффективности. Но если Фрэнк Гилберт страдает от (достойного юмора) отсутствия уважения к людям, то его коллега Фредерик Уинслоу Тейлор (Frederick Winslow Taylor) демонстрирует в своих произведениях едва прикрытое презрение к "рабочей силе".

После смерти Фрэнка Гилберта его жена и партнер Лилиан Гилберт (Lillian Gilbreth) взяла на себя управление бизнесом и стала одним из наиболее известных инженеров своего времени. Она ушла от идеи "наилучшего метода" как в управлении своей семьей[28], так и в профессиональной деятельности, где сосредоточилась на мотивации работников.

Однако ущерб уже был причинен. Задача поиска "наилучшего метода" (для выполнения каждой работы) и наделения его статусом закона была поставлена перед инженерами в большинстве отраслей промышленности Америки. Эти инженеры (часто без реального понимания сути работы) создавали стандарты, а затем навязывали их, даже мысли не допуская, что возможно существование еще более лучшего метода.

Не бывает "наилучших методов", поскольку не существует процесса, который нельзя было бы усовершенствовать. Чтобы убедиться в этом, потратьте некоторое время, молча наблюдая за работающими людьми. Вскоре вы заметите много мелочей, которые можно было бы улучшить. Процессы должны совершенствоваться теми, кто выполняет соответствующую работу. Чтобы справиться с проблемами (по одной за раз, начиная с самой большой) потребуются время и усилия. Подобный никогда не прекращающийся процесс совершенствования работы следует начать в каждой организации, занятой разработкой программного обеспечения.

Принцип 7: оптимизировать целое

Процесс разработки программного обеспечения известен своей тенденцией к оптимизации отдельных частей системы или стадий процесса.

---

[27] Frank B., Gilbreth Jr., Carey E. G. *Cheaper by the Dozen*, T.Y. Crowell Co., *1948*. Фильм под тем же названием имеет мало общего с книгой.

[28] Frank B. Gilbreth Jr. Carey E. G., *Belles on Their Toes*, T.Y. Crowell Co., *1950*.

- **Порочный круг №1 (естественно, этого не может случиться в вашей компании).**
  - Заказчик нуждается в некоторых новых функциях в своем программном обеспечении — "на вчера".
  - Разработчику говорят: "Сделай это быстро, любой ценой!"
  - Результат: в программный код вносятся поспешные изменения.
  - Результат: сложность кода возрастает.
  - Результат: число дефектов в коде возрастает.
  - Результат: экспоненциально увеличивается время, необходимое для внесения в код новых изменений.
- **Порочный круг №2 (этого также не может случиться в вашей компании).**
  - Тестировщики перегружены работой.
  - Результат: тестирование производится после создания программного кода.
  - Результат: разработчики лишены немедленной обратной связи.
  - Результат: в коде появляются новые дефекты.
  - Результат: тестирование занимает еще больше времени, системы содержат больше дефектов.
  - Результат: организация обратной связи еще больше откладывается. Далее следует повторение цикла.

Бережливое производство позволяет оптимизировать поток создания ценности с момента принятия заказа и до поставки готового продукта и удовлетворения запросов заказчика. Если организация сосредотачивается на чем-то меньшем, чем поток создания ценности в целом (т.е. оптимизирует отдельные его этапы), можно почти гарантировать, что этот самый поток в целом пострадает. Мы это наблюдали много раз на картах потока создания ценности на наших курсах: почти каждый раз, когда мы обнаруживали большую задержку выполнения заказа, это объяснялось ответственностью, которую никто не хотел на себя брать, одно подразделение организации пыталось переложить ее на другое и об интересах заказчиков в подобных случаях никто не заботился.

Организации обычно полагают, что они оптимизируют целое; в конце концов все знают, что оптимизация частностей — это плохо, поэтому никто не признает, что у них это имеет место. Тем не менее удивительно часто оптимизация частностей становится практикой и встраивается в систему оценок. Рассмотрим случай с компанией Fujitsu.[29] В 2001 году Fujitsu взяла на себя обязанность организации службы технической поддержки (help desk) для британской авиакомпании BMI. Были проанализированы вызовы, поступившие от служащих BMI и обнаружилось, что 26% вызовов обусловлены неисправными принтерами на стойках регистрации авиакомпании. Fujitsu оценила, как долго принтеры были в нерабочем состоянии и сообщила сумму понесенных убытков BMI. После этого Fujitsu убедила менеджмент BMI заменить используемые принтеры более надежными устройствами. Подобным же образом Fujitsu продолжала выявлять и искоренять причины, вызывающие неисправности (и как следствие — обращения к службе поддержки) и через 18 месяцев число обращений сократилось на 40%.

---

[29] Womack J. P. и Jones D. T., *Lean Consumption: How Companies and Customers Can Create Value and Wealth Together*, Free Press, *2005,* p. 58–63.

Казалось бы, это история с хорошим концом, однако что-то в ней не так. Оплата труда персонала большинства подобных служб поддержки определяется количеством обработанных обращений. Сократив число обращений на 40%, Fujitsu тем самым на столько же сократила оплату своих услуг! К счастью, результаты, полученные Fujitsu, были настолько впечатляющими, что этой компании удалось договориться с BMI об оплате на основе "потенциальных", а не реальных обращений. Согласно новому соглашению, оплата услуг Fujitsu в дальнейшем не зависела от числа обращений и эта компания могла продолжать помогать BMI по-настоящему решать проблемы ее бизнеса.

Данная история демонстрирует, что служба технической поддержки) при традиционной системе оплаты не имеет стимулов выявлять и устранять причины проблем клиентов.

Преодоление барьеров между организациями обходится дорого. Питер Друкер (Peter Drucker) указывал, что компания, поддерживающая единую систему управления для всего потока создания ценности, получает от 25 до 30% преимущества в цене перед конкурентами.[30] Таким образом, немалые возможности экономии кроются в структурировании контрактов, привлечении внешних подрядчиков и взаимодействии различных компаний при соответствующих стимулах с тем, чтобы целью всех участников процесса была оптимизация целого.

Миф: оптимизировать путем декомпозиции

Альфред П. Слоун (Alfred P. Sloan) изобрел организационную структуру, предназначенную управлять сложностью производственного процесса. Он создал децентрализованные подразделения и использовал денежные показатели для определения эффективности деятельности менеджеров. Это было существенным улучшением по сравнению с системой управления Форда; это помогло Дженерал Моторс производить широкий ассортимент автомобилей и заменить Форда в качестве лидера на рынке. С тех пор компании прилагали усилия, чтобы определить подходящие показатели для управления производительностью.

Хотя использование Слоуном показателей для управления сложностью процесса было блестящим, с применением данной концепции легко можно было переусердствовать. Люди склонны разделять сложную ситуацию на мелкие компоненты — вероятно, наследство от экспертов в области эффективности, которые разделяли сложную работу на мелкие операции. После подобного разложения (или декомпозиции) каждый компонент оценивался и оптимизировался. Напрашивается предположение, что после оптимизации всех компонентов вся система будет также оптимизирована. Однако это неверно. Если разделить поток создания ценности на элементарные части и оптимизировать их отдельно одна от другой, опыт показывает, что система в целом почти наверняка оптимизирована не будет.

В то же время, когда оценка производится с использованием слишком большого числа показателей, истинная цель может в этой ситуации теряться и не существует рецептов определения компромисса между ними. Решение состоит в том, чтобы поднять уровень показателей и *уменьшить* их число. Определите высокоуровневый показатель, использование которого вело бы к получению нужных результатов. Для примера с показателями проекта выше вместо добавления двух новых показателей следует использовать единственный показатель — тот, который действительно имеет значение. Доход от инвестиций (Return on

---

[30] Drucker P. *Management Challenges for the 21st Century*, Harper Business*, 1999,* p. 33.

investment — ROI) — возможный кандидат для проектов. Модели, учитывающие прибыли и потери, хорошо работают для продуктов.[31] Если оптимизировать по одному главному показателю, все прочие показатели в этом случае позаботятся о себе сами.

Подержанные автомобили[32]

Упражнения по использованию ROI на наших курсах пользуются популярностью, поскольку они помогают людям принимать компромиссные решения, которые действительно удовлетворяют клиентов. На одном из семинаров Ян и его группа создали ROI для компании, занимающейся перепродажей подержанных автомобилей. В результате удалось выявить, что сокращение времени пребывания автомобилей в компании с пяти дней до четырех обеспечит очень ощутимый финансовый эффект.

Однако это еще не вся история. Ян тогда только что познакомился с методологией программирования Scrum, когда ему было поручено поменять очень старую компьютерную систему в компании по перепродаже подержанных автомобилей. Он договорился с заказчиком, что новое программное обеспечение будет поставляться поэтапно каждый месяц, и если заказчик будет неудовлетворен на одном из этапов, он сможет аннулировать заказ.

Менеджер Яна был очень недоволен такой договоренностью. И Яну было дано понять, что если он потеряет заказ от компании по перепродаже подержанных автомобилей, то работы в консалтинговой фирме для него может больше не найтись. Поэтому Ян отнесся очень серьезно к задаче ежемесячно радовать своего заказчика.

В компании Ян выяснил, что сокращение времени реализации автомобилей должно оказать положительный финансовый эффект, а в нашем классе он смог немного попрактиковаться в оценке этого эффекта. Затем он получил от заказчика кое-какие реальные данные и ввел их в простую финансовую модель. Результаты помогли как Яну, так и заказчику ежемесячно концентрироваться на наиболее нужных функциональных возможностях для будущей системы.

Через несколько месяцев Ян написал нам, что заказчик был так доволен результатами, что консалтинговая фирма вскоре должна получить новые заказы, и что его менеджеры вполне довольны. Сегодня менеджеры Яна являются пылкими сторонниками гибкой (agile) методологии разработки программного обеспечения и в компании считают себя пионерами в применении этой методологии. (Мэри Поппендик)

Попробуйте это

1. Какой из семи мифов причиняет больше всего неудобств в вашей ситуации? Почему?
   - Созданная заранее спецификация сокращает непроизводительные затраты
   - Цель тестирования — выявление дефектов
   - Прогнозы обеспечивают предсказуемость
   - План — это обязательство
   - Спешка ведет к браку
   - Существует наилучший метод

[31] Poppendieck M., Poppendieck T. *Lean Software Development: An Agile Toolkit*, Addison-Wesley, *2003,* p. 83–92.

[32] Позвольте выразить благодарность Яну Шиммингсу (Ian Shimmings) за то, что он поделился с нами опытом. Использовано с разрешения.

- Оптимизировать путем декомпозиции

2. С чего начинается поток создания ценности в вашей организации? Что для вас означает “получить заказ”? В некоторых компаниях отсчет времени для разработки продукта начинается, когда концепция продукта одобрена для разработки (заказ — результат процесса одобрения). В других отсчет времени начинается с момента, когда маркетологи распознают потребность рынка и инициируют создание новой функциональной возможности. Какой подход кажется вам лучшим?
3. Повторное выполнение уже сделанной работы в вашей организации.
   а) Какой процент требований к будущему продукту (после того как он документирован) изменяется?
   б) Какой процент времени разработки тратится на тестирование программного кода и устранение его дефектов в конце цикла разработки?
   в) Что можно было бы сделать, чтобы сократить эти проценты?
4. Люди: насколько углубленное обучение получили руководители низшего звена в вашей организации, как управлять своими подразделениями? По каким критериям оценивается их деятельность? Насколько углубленное обучение получили менеджеры проектов низшего звена в вашей организации, как управлять своими коллективами? По каким критериям оценивается их деятельность? Есть ли разница? Почему?
5. Показатели: пусть каждый работник в вашей организации составит список на листе бумаги с перечнем всех основных показателей эффективности, по которым, как он полагает, оценивается их деятельность. Затем объедините все показатели в единый список, возможно, анонимно. Сколько показателей оказалось в этом списке? Подходящие ли это показатели? Отсутствуют ли в списке какие-либо ключевые показатели?

Глава 3

Ценность

Бережливые решения

Свою книгу *Lean Solutions* ("Бережливые решения") авторы Джеймс Вумэк (James Womack) и Дениэл Джонс (Daniel Jones) начинают с признания, что им самим приходится быть потребителями (или клиентами), которым необходимо решить какие-то проблемы, вследствие чего они часто бывают раздражены из-за множества трудностей, с которыми сегодня неизбежно связан статус клиента. С высоты своего положения (как потребителей) каждый из них хочет сказать следующее тем, кто предоставляет им товары и услуги.[1]

- Решите мою проблему кардинально.
- Не заставляйте меня тратить время.
- Дайте мне именно то, что я хочу.
- Предоставьте то, что мне требуется, именно там, где мне это нужно.
- Обеспечьте то, в чем я нуждаюсь, именно тогда, когда мне это нужно.
- Сократите до минимума число решений, которые я должен принять, чтобы решить свою проблему.

Давайте познакомимся с компанией, которая согласилась с этими пожеланиями клиентов, кажется, еще несколько лет назад.

Google

Google, возможно, единственная компания в мире, которая декларирует цель добиться, чтобы пользователи покидали их веб-сайт как можно скорее[2].

В 1999 году компания Google вышла на переполненный рынок поисковых средств с амбициозной целью сделать доступной для всех информацию всего мира. Компания решила, что для достижения поставленной цели необходимо сосредоточиться на двух моментах.

---

[1] Womack J., Jones D. *Lean Solutions: How Companies and Customers Can Create Value and Wealth Together*, Free Press, *2005*, p. 15.

[2] С веб-сайта Google, `www.google.com/corporate/tenthings.html`.

- Предоставлять более существенные результаты поиска, чем конкуренты.
- Сделать так, чтобы у тех, кто пользуется данным поисковым средством, оставались приятные впечатления.

Идея сработала. В своем последнем выпуске за 1999 год журнал *PC Magazine* присудил поисковому средству Google премию за техническое совершенство среди веб-приложений. Как отмечалось в *PC Magazine*: "Не требуется много раз обращаться к услугам Google, чтобы оценить преимущества этого поискового средства. И как можно не оценить средство, которое постоянно предоставляет хорошие результаты."[3] Премия привлекла внимание к Google, и в результате данная компания начала головокружительный подъем.

Корпоративная философия Google, опубликованная на веб-сайте компании, начинается со следующих четырех пунктов.[4]

1. Основное внимание — потребностям пользователя, а все остальное приложится.
2. Лучше сделать что-то одно, но по-настоящему качественно.
3. Демократия возможна и в веб.
4. Лучше быстро, чем медленно.

В начале своего существования компании Google пришлось прилагать значительные усилия, чтобы не изменить своим принципам. Всплывающие окна и реклама, столь популярные на других веб-сайтах, были строго запрещены в Google, поскольку они вызывали недовольство пользователей. Компания всегда очень ревностно относилась к быстродействию своих средств. Любая ощутимая задержка с выдачей результатов рассматривалась как "растранжиривание" времени клиента. В то время как большинство конкурентов создавали порталы, начальная страница Google всегда была предельно простой: единственный логотип, около тридцати слов текста, одно поле ввода и пара кнопок. В компании полагали неправильным отвлекать внимание пользователей излишествами.

В конце концов предоставляемые Google результаты поиска приобрели качество и полноту и с течением времени становились все лучше. Основатели компании создали уникальный метод классификации веб-страниц, представленных в результатах поиска, — вычислялось число "голосов", поданных за ту или иную страницу, т.е. число и значимость ссылок на нее. Страница с бо́льшим числом ссылок получала преимущество. В дальнейшем компания Google продолжала совершенствовать свой метод классификации веб-страниц.

В процессе создания своего продукта (в данном случае поискового средства) Google применяла все те же четыре принципа.

1. **Ценность: основное внимание — потребностям пользователя, а все остальное приложится.**

   Коллективы разработчиков имели систему поощрений, стимулирующую создание продуктов, ориентированных на интересы пользователей. При этом предполагалось, что, после того как продукт приобретет популярность, руководство компании сумеет извлечь из него прибыль.[5]

---

[3] PC Magazine, *December 14, 1999*, p. 104.

[4] С веб-сайта Google, `www.google.com/corporate/tenthings.html`.

5 Vise D. A *Google's Missing Piece*, Washington Post, *February 10, 2005*., p. E05.

2. **Совершенство: лучше сделать что-то одно, но по-настоящему качественно.**

   Компания Google, так же, как и ее знаменитое поисковое средство, великолепна в своем деле. Высокий технический уровень коллектива позволяет экспериментировать во многих направлениях одновременно.[6]

3. **Демократия: и в веб возможна демократия.**

   Новые идеи тут же получают поддержку в Google, привлекая критическую массу людей, стремящихся работать над ними. Сам факт, что продукт представлен на сайте Google Labs, популяризует его и привлекает к нему внимание пользователей.

4. **Скорость: лучше быстро, чем медленно.**

   То, на что у других компаний уходят годы, в Google осуществляется в течение нескольких недель или дней.[7]

Обратите внимание, что Google, к своему спектру услуг, добавила карты. В марте 2004 года была выпущена бета-версия Google Local — средства, позволяющего ограничить поиск определенной географической областью. В октябре того же года Google приобрела компанию Keyhole, что предоставило ей доступ к изображениям земной поверхности, получаемым с помощью спутников. В феврале 2005 года в Google Local были добавлены карты. А изображения из космоса появились в наборе функциональных возможностей Google в апреле 2005 года — всего через шесть месяцев после приобретения Keyhole! И всего двумя месяцами позже было выпущено Google Earth — средство, предоставляющее доступ к изображениям (земной поверхности) из космоса, картам и локальному поиску с помощью инновационного интерфейса на рабочем столе, имитирующего глобус.

Компания Google сосредоточилась на своей стратегической задаче: сделать доступной и полезной для всех информацию всего мира. Для ее решения Google создает как программные средства, используемые внутри компании, так и веб-продукты, которые включают изощренные алгоритмы, сложную аппаратуру и тщательно разработанное программное обеспечение. Продукты и средства разрабатываются быстро и поэтапно небольшими коллективами. При этом поощряется выпуск программных продуктов до того, как они будут полностью завершены — с целью создания обратной связи с пользователями. Программное обеспечение компании Google, которое неизменно восхищает пользователей, славится элегантным дизайном и высоким качеством.

По всем меркам Google — очень успешная организация, и, что более важно, ее продукты непрерывно совершенствуются в соответствии с тем, как изменяется мир. Google постоянно представляет новые продукты, несмотря на то, что эта компания не имеет долговременного плана создания новых продуктов. Компания определяет целесообразность создания того или иного продукта приблизительно так же, как она классифицирует веб-страницы в результатах поиска: приоритет определяется энтузиазмом коллектива разработчиков и интересом пользователей.

Авторы полагают, что компания Google добилась таких больших успехов благодаря тому, что она предоставляет пользователям именно ту информацию, которая им нужна, в нужное время и в нужном месте, без проблем, не заставляя пользователей ждать, и без надоедливой рекламы. Средства Google достаточно просты в использовании, даже для ма-

---

[6] Hammonds K.H, *How Google Grows...and Grows...and Grows*, Fast Company, Issue 69, *April 2003*., p. 74.
[7] `http://video.google.com/videoplay?docid=-8618166999532839788`.

ленького ребенка. И они позволяют решать проблему комплексно — проверить орфографию, осуществить перевод с одного языка на другой, конвертировать единицы измерения; кроме того, здесь имеются словарь, калькулятор и карты. Словом, все это в сумме полностью соответствует определению "бережливое решение".

От концепции до прибыли

Давайте присмотримся к графику работы над первым продуктом Google, чтобы познакомиться с функционированием этой компании.

Концепция

В 1996 году Ларри Пейдж (Larry Page) и Сергей Брин (Sergey Brin) объединили усилия для решения интересующей их задачи — поиска информации в Интернете. В то время они были студентами-выпускниками Станфордского университета (специальность — информатика), занимавшимися исследованием проблем поиска данных, поэтому они решили попробовать свои силы в поиске информации в Интернете. Двумя годами позже они опубликовали работу, в которой были изложены подробности концепции, которая позже легла в основу работы поискового средства Google.[8]

Предложить блестящую концепцию (которая может быть результатом или долгих исследований, или озарения) — это только начало. Компании могут создавать условия для появления инноваций, но не существует магической формулы для озарений. Однако обычно озарения недостаточно: "Инновации — это 5 процентов озарения и 95 процентов тяжелого труда"[9]. Проблема большинства компаний в том, что люди "с вдохновением" не имеют времени или возможностей реализовать его. В то же время компании, постоянно работающие над тем, чтобы превратить вдохновение в инновацию, предоставляют время и создают возможности для людей и коллективов с хорошими идеями, чтобы эти идеи смогли созреть и превратиться в концепции.

Осуществимость проекта

Компания Google, подобно Dell, "начиналась" в комнате студенческого общежития. Когда первый интерфейс поискового средства Google появился на веб-сайте Станфордского университета, компьютеры, с которых можно было осуществлять поиск данных в Интернете (с помощью нового средства), располагались в комнате Пейджа. Вскоре Пейдж и Брин оставили учебу ради бизнеса. После этого худшее, что с ними могло бы случиться, это что им придется вернуться в университет, чтобы завершить свое образование. Компания Google была основана в 1998 году, однако почти в течение года продукт новой компании имел статус бета-версии, в то время как Пейдж и Брин исследовали осуществимость проекта. В это время компания начала привлекать внимание средств массовой информации, трафик рос и Пейдж с Брином смогли обеспечить необходимое финансирование. Наконец пометка "бета" была удалена и исследовательский проект стал служить основой реального продукта.

---

8 Brin S., Page L. *The Anatomy of a Large-Scale Hypertextual веб Search Engine*, Computer Networks and ISDN Systems, 30(1–7):107–117, *April 1998*.

9 Цитата Томаса Эдисона.

Стадия определения осуществимости в процессе разработки продукта важна, поскольку она предоставляет возможности для экспериментирования. Следует заметить, что стадия определения осуществимости — это не кабинетные (или бумажные) исследования. Концепция — это прекрасно, но ничто не заменит ее проверки в реальных условиях.

Изготовленные вручную образцы

На предприятии по производству видеокассет, где мне пришлось работать, из этапа определения осуществимости проекта мне больше всего запомнились образцы нового продукта (или изделия), которые были произведены в лаборатории вручную с помощью самого примитивного оборудования. Но если не вспоминать о примитивном оборудовании, эти изготовленные вручную образцы были очень близкой имитацией конечного продукта.

Мы израсходовали множество этих образцов, к каждому из которых был приклеен аккуратный ярлык. Мы различным образом испытывали эти образцы — на механическое разрушение, воздействие солнца или влаги или, может быть, отдавали их пользователям для испытания в реальных условиях.

Для меня стадия определения осуществимости связана с большим количеством экспериментов, позволяющих выяснить (в реальных условиях и у реальных пользователей), что в новом продукте хорошо и что плохо. (Мэри Поппендик)

Это также время принятия решений, определяющих конструкцию (или архитектуру) будущей системы, — критически важная задача, которую многие компании игнорируют. Каковы основные свойства бизнес-процесса или ключевых модулей аппаратуры? Какие интерфейсы будут использоваться в системе и как они будут взаимодействовать? Как архитектура программного обеспечения будет поддерживать продукт (или изделие)? Где в системе узкие места и какие меры следует предусмотреть, чтобы избежать проблем? Совершенный проект системы — основа будущего совершенного продукта. Эта задача не должна возлагаться ни на дилетантов, ни на далеких от практики экспертов. Справиться с ней в состоянии только зрелые разработчики, которые знают, что проект системы будет эволюционировать по мере создания продукта, и которые знают, как обеспечить, чтобы эта эволюция принималась во внимание.

Пробная версия

Компания Google начала с системы классификации веб-страниц, основанной на подсчете ссылок на эти страницы. Как только эта система заработала, ученые приступили к сбору данных обо всем: какие результаты были использованы и какие нет, какие фразы были заданы для поиска данных результатов, какие опечатки наиболее типичны, как быстро предоставлялись результаты поиска и т.д. Компания Google непрестанно совершенствовала свою систему классификации страниц, а также аппаратные средства и архитектуру программного обеспечения, основываясь на анализе этих обширных данных. Были разработаны средства, облегчающие разработку приложений для доступа к “безбрежным” информационным структурам компании. Создавалось множество пробных приложений, которые передавались в Google Labs, где пользователи могли проголосовать за тот или иной продукт и помочь совершенствовать их через обратную связь.

Никто не ожидает, что на стадии определения осуществимости проекта появится готовый, сформировавшийся продукт. На этом этапе только определяется, будет ли создаваемый продукт работать. В этот момент реальная разработка продукта только начинается. Правильно построенный процесс разработки состоит из серии проб и ошибок, кото-

рые позволяют усовершенствовать дизайн продукта. Целью этого этапа является не завершить создание продукта, а довести его до состояния, когда он (с минимальным набором функциональных возможностей) может быть выпущен в виде пробной версии для оценки пользователями. При этом предполагается, что после первой пробной версии последуют другие, более полные, и таким образом продукт постепенно обретет контуры.

Что касается стадии испытания пробной версии программного продукта, то здесь имеется бо́льшая вероятность выпуска и передачи подобной версии пользователям для испытаний (чем на аналогичной стадии создания аппаратуры), хотя, как будет показано в главе 8, в рамках программы разработки подводных лодок Polaris были созданы несколько функционирующих подлодок в период от трех до восьми лет после начала программы. Однако, даже если пробная версия аппаратуры (или прототип) реально не передается пользователям для испытаний, все равно удачной идеей является планирование создания прототипов, включающих встроенное ПО, так рано в процессе разработки, как это возможно.

Выпуск программного обеспечения (и передача его пользователям для оценки) на стадии создания пробной версии почти всегда оказывает благотворное действие на процесс разработки (если на стадии определения осуществимости проекта были приняты правильные проектные решения для будущей системы). Конечно, выпуск ранней версии программного продукта не всегда целесообразен. Встроенное программное обеспечение, например, не может быть выпущено раньше аппаратуры. Игровые программы также не могут выть выпущены в незаконченном состоянии, поскольку первое впечатление об игре — это почти все, а срок жизни игровых программ очень краток. Однако в большинстве ситуаций все же целесообразно осуществлять ранний выпуск программного обеспечения в виде альфа и бета-версий для его оценки.

Прибыль

Компания Google в конце концов начала зарабатывать много денег — пропорционально ценности, предоставляемой рекламодателям и пользователям. Компании пришлось много экспериментировать и многому научиться, прежде чем было определено, как сделать рекламу неназойливой и такой же уместной, как результаты поиска. Google приходится сталкиваться с конкуренцией, и, чтобы обеспечить постоянные прибыли, компании необходимо неустанно заботиться о том, чтобы пользователи были довольны.

Восхищенные потребители

В 1984 году доктор Нориаки Кано (Noriaki Kano) из Токийского университета опубликовал работу, посвященную “свойству привлекать”, которая явилась вехой в своей области.[10] В этой работе была представлена модель Кано (рис. 3.1) — средство, позволяющее прояснить, как создаются выдающиеся продукты, восхищающие потребителей.

Модель Кано демонстрирует: для того чтобы только “протиснуться в дверь”, необходимо удовлетворить элементарные потребности. После этого открываются две перспективы: 1) увеличивать привлекательность продукта, добавляя в него новые функциональные возможности; и 2) выявить потребности пользователя, о которых он пока сам не до-

[10] Noriaki K., Seraku N., Takahashi F., Tsuji S. *Attractive Quality and Must-Be Quality, Hinshitsu*. The Journal of the Japanese Society for Quality Control, *April 1984*, p. 39–48.

гадывается, и восхитить его, неожиданно удовлетворив эти потребности. Доктор Кано отмечал, что простое улучшение характеристик (или добавление новых функциональных возможностей) обеспечивает лишь линейный рост удовлетворенности продуктом — в этом случае удовлетворение потребителя возрастает пропорционально числу новых функциональных возможностей. Чтобы получить экспоненциальный рост удовлетворенности, необходимо выявить и удовлетворить потребности, на удовлетворение которых потребитель сам пока не рассчитывает. Кано утверждает, что пользующиеся спросом продукты не создаются на основе только пожеланий потребителей. Для этого производителю необходимо глубоко понимать запросы потребителя, обнаружить его скрытые запросы, удовлетворить которые пока еще никто не пытался, и таким образом удивить и восхитить его.

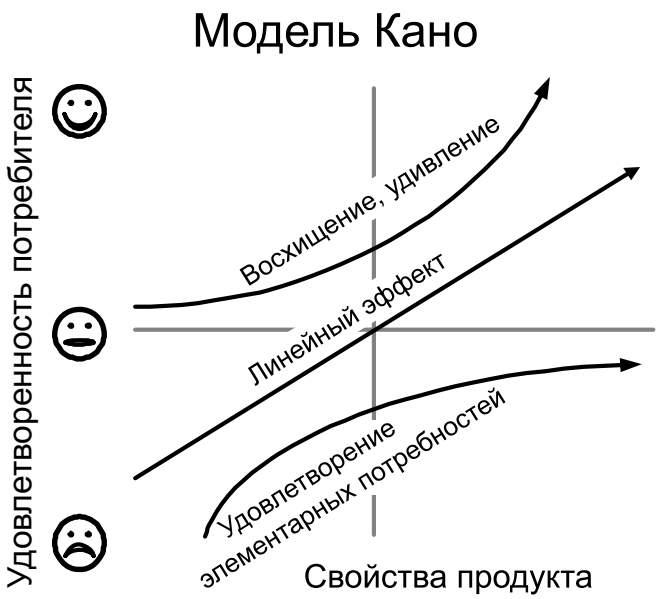


Рис. 3.1. Модель Кано

Компания Google, кажется, очень успешна в том, чтобы восхищать пользователей. Во время путешествий по миру с докладами и курсом лекций мы часто задавали вопрос: “Кто из присутствующих доволен Google?” Почти все поднимали руки, а затем мы узнавали о множестве способов использования продуктов Google. Наш сын регулярно посылает нам ссылки на неизвестные пока новые функциональные возможности. А наша внучка (учащаяся начальной школы) постоянно использует средства Google для выполнения домашних заданий.

Глубокое понимание потребителя

Том[11] однажды сказал интервьюеру: "Великолепное программное обеспечение является результатом объединения усилий тех, кто знает соответствующий бизнес (в котором будущее ПО предполагается использовать), и тех, кто разбирается в соответствующей технологии". Глаза репортера неожиданно зажглись. "Я понял!" — воскликнул он, а затем рассказал следующую историю.

> Это было в конце 90-х годов, когда каждой газете вдруг потребовалось иметь свой веб-сайт. Все другие газеты, для создания своих сайтов, обратились к услугам соответствующих фирм. В отличие от них, наше издание наняло программиста, и работать с ним было поручено мне. Сначала было очень трудно уловить, о чем он говорит, однако через некоторое время я начал понимать, чем свой веб-сайт может быть очень полезен для нас. В этой ситуации я представлял бизнес, а программист, о котором шла речь выше, был техническим специалистом, и вместе мы разобрались, что желательно для нашей газеты и что возможно. В конце концов это сотрудничество позволило создать нечто полезное. Наши конкуренты предоставляли доступ к содержимому своих сайтов совершенно бесплатно, в то время как мы за доступ брали все увеличивающуюся плату. И тем не менее очень многие подписчики были готовы раскошелиться. В настоящее время все газеты, аналогичные нашей, испытывают финансовые трудности, а мы даем существенную прибыль.

В мире имеется много программного обеспечения среднего качества, и только иногда что-то по-настоящему инновационное привлекает наше внимание. И мы хотим знать *Как им это удалось?*, или точнее, *Как бы нам сделать то же?*

Фокусироваться на работе

В 1982 году 30-летний Скотт Кук (Scott Cook) задавал себе тот же вопрос.[12] Только что на рынке появился персональный компьютер IBM PC, и Кук был один из многих предпринимателей, задумавших создать программные продукты для продажи владельцам ПК. Кук имел преимущество. До этого он пять лет работал в компании Procter & Gamble, где приобрел опыт, как следует предлагать продукты, которые изменят жизнь людей. Кук сделал то, что на его месте бы сделал любой хороший работник отдела маркетинга в P&G. Он начал с поиска работы (или задачи), которую необходимо было выполнять и для которой пока не существовало никаких подходящих средств. Он обратил внимание на отчаяние своей жены, когда она приводила в порядок семейные финансы, и решил, что соответствующее программное обеспечение существенно облегчило бы задачу. Кук был не одинок в своем открытии. Ко времени, когда он основал компанию Intuit и представил программный продукт Quicken в1983 году, у него было уже 46 конкурентов.

В P&G Кук усвоил, что для борьбы с конкурентами прежде необходимо до конца понять задачу, стоящую перед потенциальными потребителями, и выяснить, каковы недостатки средств, используемых для ее решения в настоящее время, и что именно означает выполнить данную работу лучше. Он начал с методичного исследования, как люди управляют своими финансами. Он выяснил, что большинство делают при этом только три вещи: они платят по счетам, контролируют свою чековую книжку и периодически суммируют и про-

---

[11] Один из авторов. — *Примеч. ред.*

[12] Taylor S., Schroeder K. *From Inside Intuit*, Harvard Business School Press, *2003*.

сматривают свои расходы. И он обнаружил, что люди находят это занятие неприятным и стремятся освободиться от него как можно скорее. Поэтому программа Quicken с самого начала была создана для выполнения трех (и только трех) задач — быстрее, чем их можно было бы выполнить вручную.

Далее Кук обратил внимание, что конкурирующее программное обеспечение по управлению финансами перегружено излишними функциональными возможностями и установка его на компьютере занимает несколько часов. Кук решил создать такую простую программу, которую даже пользователь-новичок смог бы быстро инсталлировать на ПК и заполнять чеки уже через пятнадцать минут. Затем он создал систему тестирования простоты использования с тем, чтобы программу Quicken 1.0 можно было тестировать и упрощать до тех пор, пока она не будет соответствовать поставленным целям. Тем временем те же тесты показали, что опытному пользователю ПК необходимо около пяти часов, чтобы выполнить установку и создать первый чек на программном обеспечении конкурентов, перегруженном излишними функциональными возможностями.

Поскольку просто иметь лучшее средство недостаточно, компания Intuit занялась исследованиями в области ценообразования и упаковки. Люди, которым необходимо было выполнить некоторую работу, должны были сразу понять, что это лучшее средство для ее выполнения. Кроме того, Intuit занималась постоянным совершенствованием своей программы, чтобы она оставалась наилучшим средством для выполнения данной работы. Кук полагал, что потребительскую ценность можно создать, если глубоко вникнуть в суть работы, которую необходимо выполнить, и усовершенствовать программное обеспечение настолько, чтобы с его помощью можно было бы выполнять эту работу лучше и быстрее, чем при использовании альтернативных средств.[13]

Мои заказчики

Когда я участвовала в создании системы управления процессом, не возникало вопроса, кто является заказчиками (рядовые рабочие, выполняющие реальную работу), и в чем состоит работа (в производстве ленты для видеокассет). Если наша система не окажется полезна при выполнении данной работы, рабочие проигнорируют новую систему и будут производить ленту без нее. Условия были ясны: если новая система не будет использоваться, вина за это ляжет только на нас — разработчиков: значит, мы не поняли до конца работы, облегчить выполнение которой была призвана наша система. (Мэри Поппендик)

Ориентированная на потребителя организация

Как задумываются и создаются выдающиеся продукты? Книга Кларка и Фуджимото *Product Development Performance*[14] (см. главу 1), появившаяся в 1991 году, рассказывает, что выдающиеся продукты являются результатом хорошо организованного потока информации. То, как потребители воспримут продукт, определяется качеством информационного потока между рынком и коллективом разработчиков. Техническая целостность продукта определяется качеством информационного потока между членами этого коллектива.

[13] Christensen C., Cook S., Hall T. *Marketing Malpractice: The Cause and the Cure*, Harvard Business Review, December, *2005*.

[14] Clark K., Fujimoto T. *Product Development Performance*, Harvard Business School, *1991*.

Способствовать этому потоку можно с помощью двух мер: 1) обеспечить руководство и 2) поручить работу многофункциональному коллективу (comletre team).

Руководство

Давно замечено, что создание успешного продукта тесно связано с присутствием *чемпиона* (или лидера) — лица, глубоко понимающего суть проблем (или задач), стоящих перед потребителем, и способного предложить решение, которое бы удивило и восхитило этого потребителя. На чемпионе лежит ответственность в принятии ключевых (для создаваемого продукта) решений, а также за результаты этих решений. Вот как об этом говорит Мартин Кэган (Martin Cagan) из компании Silicon Valley Product Group.

> Создание выдающегося продукта возможно только, если это возложено на лицо с глубоким сочувствием к потребителю, даром видеть возможное, а также способностью понять, что существенно и что случайно. Это лицо глубоко понимает как потребителя, так и возможности своего коллектива разработчиков. Такое лицо может занимать должность менеджера продукта или быть кем-то другим — от рядового инженера до учредителя компании. Главное, чтобы эту роль взял на себя кто-то, кто не боится ответственности, а также обладающий необходимыми качествами.

Когда приходится наблюдать программу разработки некоторого продукта в бедственном положении, обычно всегда оказывается, что роль чемпиона здесь вакантна. Существует несколько моделей реализации этой роли. Для лучших моделей характерна ясность, на ком лежит ответственность за успех продукта.

Главный инженер

В компании Toyota вся ответственность за экономический успех семейства автомобилей ложится на главного инженера, ответственного за данное семейство. Этот руководитель является высококвалифицированным инженером с глубоким знанием конструкции автомобиля. Кроме того (с помощью небольшого штата) он старается глубоко понять потребителя (для которого предназначена та или иная модель) и что этот потребитель ценит. Подобный руководитель знакомится с исследованиями рынка, общается с дилерами, а также регулярно посещает места, где часто можно встретить потенциальных потребителей. Он получает финансирование и общие указания от вышестоящего менеджмента. Однако в конце концов именно главный инженер лично занимается объединением всей этой информации, решает, что понравится потребителям, создает концепцию будущего продукта и руководит разработкой.[15]

Возьмем, например, модель Sienna. Первая версия автомобиля типа minivan (небольшой фургон) компании Toyota продавалась не особенно хорошо. Когда главный инженер Юджи Йокоя (Yuji Yokoya) взялся улучшить модель, он знал, что ему, чтобы справиться с задачей, потребуется не только группа для изучения соответствующего сегмента рынка и информация от потребителей. Поэтому он последовал традиции Toyota "разобраться на месте". Он проехал на Sienna 53000 миль (85000 километров) через каждый штат в США и Мексике, а также каждую провинцию в Канаде. Обычно он путеше-

[15] Информация для данного абзаца взята из главы 7 диссертации (Sobek D. K. II) "Principles That Shape Product Development Systems: A Toyota-Chrysler Comparison," диссертация была представлена на соискание ученой степени доктора философии (специальность Industrial and Operations Engineering) в университете Мичигана, 1997г.

ствовал с одним из ключевых членов коллектива разработчиков, включая Джона Джулэ (John Jula), инженера с выдающейся комплекцией, которому предстояло переделать конструкцию сидений. В ходе путешествия Йокоя пришел к пониманию, что должен оценить потребитель: больше пространства, удобные передние сиденья для родителей и задние — для детей. В результате продажи новой модели Sienna в 2004 году удвоились и данная модель оказалась среди лучших.

Мы (авторы. — *Примеч. ред.*) недавно купили минивэн Sienna. Мы всерьез рассматривали покупку других автомобилей, однако нашли, что пространство для ног водителя и переднего пассажира в других аналогичных моделях слишком тесно. Отпуск мы проводим "на колесах", поэтому удобство передних мест по-настоящему важно для нас. Мы могли согласиться, когда попробовали Sienna, что Джон Джулэ думал о нас при конструировании передних мест этого автомобиля.

Преимущества, связанные с существованием должности главного инженера, состоят в том, что одно лицо берет на себя ответственность за объединение и осмысление множества разнородной информации. Кроме того, дух предпринимательства, возникающий там, где имеется частная собственность, имеет долгую историю создания блестящих инноваций. Люди любят следовать за успешными руководителями, поскольку они помогают делать успешными целые коллективы.

В качестве примера можно также вспомнить, что большинство проектов Open Source[16] имеют "сильного руководителя проекта" (аналог главного инженера).[17] Подобные проекты обычно начинаются с идеи одного индивидуума, и даже если множество добровольцев работают над проектом, весь программный код просматривается и исправляется руководителем или специально отобранными помощниками.

Должность главного инженера — не панацея. Вполне возможна ситуация, когда кто-то, занимающий эту должность, видит свою задачу в том, чтобы "руководить," а не объединять усилия всей команды. Это не проблема для проектов Open Source, поскольку невозможно для руководителя такого проекта привлечь разработчиков-добровольцев, не являющихся энтузиастами своего дела. Однако это может стать проблемой в компаниях, где должность главного инженера проекта не является хорошей школой для профессионального роста, или в компаниях, где полагают, что главный инженер должен сосредоточиться больше на решении конкретной задачи, а не на объединении информации (о возможных вариантах конструкции и о запросах потребителя).

Многие успешные компании, занятые созданием программного обеспечения, были основаны главными инженерами, в которых технические знания объединялись с ясным в*и*дением, в чем нуждается незаполненный рынок. Хорошим примером является eBay[18] (подобные компании, как правило, создают главные инженеры). К сожалению мы выделили множество успешных главных инженеров, которые, основав компанию, по мере ее

---

[16] Открытое программное обеспечение, т.е. ПО с "открытым" исходным кодом (англ. open source software) — способ, при котором исходный код создаваемых программ открыт, т.е. общедоступен для просмотра и изменения. Это позволяет всем желающим использовать уже созданный код для своих нужд и, возможно, помочь в разработке открытой программы. Более подробно об Open Source можно прочесть в главе 9.

[17] Raymond E. S. *The Cathedral and the Bazaar*, `www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/, 2000`.

[18] (читается: "И-Бэй") — компания, предоставляющая услуги интернет-аукционов (основное поле деятельности), интернет-магазинов, мгновенных платежей.

роста становились посредственными генеральными директорами (CEO). Они не потрудились вырастить новых главных инженеров, часто передоверяя задачу определения потребительской ценности (customer value) будущего продукта отделу маркетинга. Через пять или десять лет такие компании часто приходят в упадок, по мере того как появляется все больше конкурентов и не оказывается на месте главного инженера, способного предвидеть будущее и нацелить коллектив на создание конкурентоспособных продуктов.

Команда руководителей

Целью создания нового продукта всегда является удовлетворение некоторого спроса, который до этого оставался неудовлетворенным, — либо из-за отсутствия необходимой технологии, либо из-за того, что этот спрос был не распознан, либо из-за какой-то комбинации первых двух причин. В последнем случае подчас лучший способ соединить две идеи состоит в том, чтобы иметь двух руководителей, один из которых обладает глубоким пониманием рынка, а второй — обширными техническими знаниями. В компании Intuit, например, практика объединения эксперта в области маркетинга с техническим специалистом началась с основателей. Скотт Кук определил в*и*дение рынка для программы Quicken, в то время как Том Проул (Tom Proulx) руководил техническими разработками.[19]

Если в этом уравнении имеется третий важный элемент, — производство, например, — имеет смысл иметь трех руководителей разработки продукта. В компании Honda при создании новых моделей практикуется объединение усилий специалистов по продажам, по производству и в области опытно-конструкторских работ, которые работают как единая команда при принятии ключевых для создаваемого продукта решений. Помимо команды руководителей, Honda также имеет в крупных проектах единого руководителя, роль которого сходна с ролью главного инженера проекта в компании Toyota.

Крайне важно обеспечить качественное руководство в наиболее важных точках процесса разработки нового продукта. Например, в Alias, компании из Торонто (теперь являющейся частью компании Autodesk), которая специализируется на создании трехмерных графических программ, пользовательский опыт максимально учитывается при создании каждого нового продукта. С этой целью в коллективы, занятые созданием новых программ, обычно назначаются по два интерактивных дизайнера (interaction designer), на которых возлагается полная ответственность за учет пользовательского опыта и удобства использования (usability) будущей программы.[20] Большое количество полезной информации приходит через обратную связь с пользователями (разного рода опросы, деловые встречи, анкетирования и проверки удобства использования). При этом ожидается, что объединит эту информацию интерактивный дизайнер, отвечающий за поведение программы и обладающий глубоким пониманием задач, решать которые (наилучшим образом) призвано создаваемое программное обеспечение (см. главу 8).

От команды руководителей ждут выработки общего направления, придерживаться которого в процессе создания нового продукта смог бы коллектив разработчиков.

Совместное руководство

Модель минивэна Chrysler NS (Caravan/Voyager/Town and Country) была представлена в 1996 году и стала “хитом” на рынке. Эта модель (в которой была пересмотрена кон-

---

[19] См. Taylor S., Schroeder K. *Inside Intuit*, Harvard Business School Press, *2003*.

[20] Miller L, *Case Study of Customer Input for a Successful Product*, Experience Report, *Agile 2005*.

цепция минивэна — добавлена дверь для пассажира рядом с водителем) получила премию автомобиля года (Motor Trend's Car of the Year). Мы (авторы) владели минивэном Chrysler (Town and Country) минивэна выпуска 1997 года в течение восьми лет. Этот автомобиль был создан не главным инженером проекта, а командой экспертов, сумевших выработать общее понимание, чего хотят потребители. Этот коллектив провел глубокое исследование рынка, выполнил подробный анализ технологии развертывания функций (quality function deployment — QFD) и даже осуществил серию экспериментов по простоте и удобству использования, прежде чем принять специфические решения. Эта целеустремленная команда разработчиков проводила вместе много времени (не менее одного дня в неделю), оценивая состояние дел, учась друг у друга и решая текущие проблемы.[21]

Хотя не ясно, кто при этом несет за все ответственность, совместное руководство может явиться удачным подходом для проектов (умеренного объема) по разработке программного обеспечения. Удачным примером использования такого подхода в сообществе Open Source является веб-сервер Apache.[22]

Кто за все отвечает

Хорошее маркетинговое и техническое руководство не исключает существования заинтересованного коллектива, который совместно изучает нужды потребителей, рассматривает с разных точек зрения возникающие вопросы и вырабатывает общий взгляд на потребительскую ценность (customer value). Мэри приобрела опыт разработки продукта на 3M[23], где следовали традиционному представлению, что каждый новый продукт нуждается в "чемпионе". На роль такого чемпиона обычно приходит технический специалист, обнаруживший неудовлетворенный спрос на рынке и создавший технологию для выполнения соответствующей задачи. Однако чемпион не будет успешным, если он не работает с многофункциональным коллективом (фактически создаст предприятие мелкого бизнеса для коммерциализации нового продукта). Подобный коллектив совместно определяет запросы рынка и основные характеристики будущего продукта. Такое взаимодействие чемпиона и коллектива, члены которого представляют все важные функции бизнеса, шло на пользу компании 3M десятилетиями, обеспечивая невероятный рост через создание множества новых продуктов ежегодно. На 3M общие ключевые решения обычно принимал коллектив разработчиков, однако в конце концов ответственность за разработку продукта и преодоление всех проблем нес именно чемпион. Аналогично, в компании Toyota основным принципом является: "Коллективный труд — ключ к выполнению высококачественной работы, но ответственность должен нести кто-то один".[24] В своей книге *Who Has the D? How Clear Decision Roles Enhance Organizational Performance* ("Кто принимает решения? Как ясность в том, кто принимает решения, может повысить эффективность организации") авторы Пол Роджерс (Paul Rogers) и Марция Бленко (Marcia Blenko) утверждают, что хорошо отработанный механизм принятия решений яв-

---

[21] Sobek, Там же. Автор лично наблюдал работу данного коллектива и рассказал об этом в своих тезисах.

[22] Fielding R.T. *Shared Leadership in the Apache Project*, Communications of the ACM, *April 1999*.

[23] Предприятие по производству видеокассет.

[24] Morgan J., Liker J. *The Toyota Product Development System: Integrating People*, Process, and Technology, Productivity Press, *2006*, p. 103.

ляется отличительной чертой успешных организаций, и что ясность в том, кто какие решения принимает, является залогом принятия хороших, своевременных решений.[25]

Назначение "принимающего решения"

Я руководила созданием нескольких новых продуктов в компании 3M. По моему мнению, это очень ответственная, но и благодарная роль. На еженедельных собраниях коллектива, занятого разработкой соответствующего продукта, естественно, нередко возникали разногласия, поскольку представители разных специальностей имели различные точки зрения. Подчас не было ясно, собирается ли ответственный за качество когда-либо выпустить продукт на рынок, и понимает ли специалист по маркетингу, что процесс разработки продукта — это тяжкий труд. Я заметила, что разногласия обычно касаются не достоинств и недостатков решения, а того, кому именно следует решать.

Поэтому я сочла полезным каждый раз назначать специальное лицо, ответственное за принятие данного решения. При этом я выбирала специалиста, наиболее подходящего для текущего случая. Ответственный за качество одобрит выпуск окончательной версии, маткетолог решит, какими свойствами должен обладать продукт, разработчик скажет, возможно ли реализовать ту или иную функцию, и т.д.

После назначения принимающего решения тон дискуссии немедленно изменился. Члены коллектива начали пытаться воздействовать на принимающего решения с целью убедить его стать на их точку зрения. При этом ответственный за принятие решения постарался сформировать собственную точку зрения, учитывающую мнения всех сторон. После этого все последующие дискуссии обычно касались достоинств и недостатков того или иного решения, а не того, кому следует решать. (Мэри Поппендик)

Многофункциональные коллективы

Совершенные продукты создаются многофункциональными командами. Когда руководство компании Intuit приняло решение о создании программы Quicken Rental Property Manager, была создан коллектив, включавший всех специалистов, необходимых для создания продукта и продвижения его на рынке, а "не только разработчиков программного обеспечения".[26] Членам нового коллектива приходилось работать вместе и раньше — у них за плечами был опыт 20-летнего совершенствования программы Quicken. Однако новый продукт с маркой Quicken предстояло создать впервые за много лет. Задача коллектива состояла в том, чтобы решить соответствующую проблему потребителей — ни больше, ни меньше, — создав программу с минимальным набором функций и в кратчайшие сроки.

Коллектив разработчиков пришел к выводу, что его основная задача состоит в том, чтобы учиться — узнать все о задачах, стоящих перед будущими пользователями программы с тем, чтобы эта программа включала как раз необходимый набор функциональных возможностей. Коллектив регулярно отчитывался перед руководством компании о состоянии дел, получал общие указания, а затем работа продолжалась до следующей "контрольной точки". На каждой из таких встреч коллектив разработчиков рассказывал, что нового было выяснено о работе будущих пользователей программы и что планируется выяснить до следующей "контрольной точки". За исключением общих указаний, получаемых на таких встречах, решения по будущему продукту принимались в коллективе, создающем программу.

---

[25] Rogers P., Blenko M. *Who Has the D? How Clear Decision Roles Enhance Organizational Performance*, Harvard Business Review, *January 2006*.

[26] Сообщение Soni Meckem на конференции Lean Design & Development 2005, Cgicago, March, 2005.

Команда создателей Quicken Rental Property Manager начала с того, что отправилась опрашивать потенциальных пользователей. При этом было замечено, что вопросы разработчиков отличаются от вопросов маркетологов. В ходе опроса, благодаря различным точкам зрения, все члены коллектива получили более глубокие представления о том, каким следует быть будущему продукту. Выяснилось также, что при таком подходе процесс разработки не требует так много бумажной работы, как это представлялось ранее, поскольку все члены коллектива слышали от пользователей одно и то же.

После этого члены коллектива полностью погрузились в процесс создания нового продукта. Через год программа Quicken Rental Property Manager была выпущена для оценки пользователей, которые были в восторге от ее понятного интерфейса, незамысловатого процесса установки и простоты использования.

Технологичность

Один из наибольших эффектов в процессе производства имеет место, когда производственные инженеры приглашаются в коллектив разработчиков будущего продукта. При этом каждое конструкторское решение, прежде чем быть одобренным, оценивается с точки зрения простоты производства. В результате появляются продукты, которые не только легко и быстро производить, но и просто обслуживать. Это называется технологичностью конструкции (design for manufacturability).

Технологичность изделия

Давным-давно на нашем предприятии видеокассеты собирались вручную. Руководитель предприятия предлагал руководителям всех отделов регулярно тратить некоторое время, работая на сборочной линии, поэтому я время от времени работала в ночную смену, собирая видеокассеты. Там был один винт, доступ к которому был затруднен для обычной отвертки, и поэтому мы использовали для него специально изготовленную согнутую отвертку. Каждый раз, когда мне приходилось тянуться за этим специальным инструментом, а затем мучиться, вкручивая винт, я мечтала о том, чтобы привести конструкторов на сборочную линию и заставить их собрать несколько кассет.

Позже, когда конструкторы стали подчиняться мне, я всегда требовала, чтобы их разработки просматривались представителями производства, прежде чем быть утвержденными. Сначала конструкторы были поставлены в тупик, но очень скоро почувствовали уважение к реалиям производства. Вскоре чуть ли не через каждые два дня они сами несли свои чертежи к контролеру производства, надеясь получить полезные замечания. Не стоит и говорить, что новые изделия было легко собирать. (Мэри Поппендик)

В разработке программного обеспечения эквивалентом технологичности является функциональность. Каждый коллектив разработчиков должен иметь в своем составе специалистов, которые бы побуждали коллектив учитывать, какие проблемы могут возникнуть, когда программное обеспечение будет использоваться в реальных условиях, и что нужно сделать, чтобы избежать их возникновения.

Все, что может случиться, рано или поздно случится

Коллега, бывший разработчик, который теперь производственный менеджер, имел любимую поговорку, которую разработчикам полезно запомнить: ''Все, что может случиться на производстве, рано или поздно случится, и необходимо заранее предусмотреть меры, позволяющие выявить и локализовать проблему, а затем устранить ее последствия''. Недавно он писал, рассказывая о своей последней проблеме: ''Компания ''XZY'' практически еженедельно выпускала новые версии программного обеспечения. Эти версии как правило были вполне работоспособными и не демонстрировали слишком много регрессивных ошибок. Основная проблема состояла в том, что ПО

проявляло нестабильность либо с течением времени, либо с увеличением нагрузки. При прохождении функциональных тестов на качество оно (ПО) часто проявляло несоответствие требованиям производства.''

Я предложила проанализировать источник каждого отказа. Оказалось, что это уже делалось.

''Исчерпывающий анализ неизменно является частью реакции на каждую проблему. В результате была обнаружена не единственная ошибка, а целый набор ошибок, которые все можно в конечном счете отнести на счет небезопасной практики программирования.''

В этом примере коллектив разработчиков не ощущал ответственности за созданное программное обеспечение (после передачи его заказчику) и был не заинтересован в результатах анализа отказов. (Мэри Поппендик)

Разработка на заказ

То обстоятельство, что программное обеспечение разрабатывается только для одного заказчика, не освобождает соответствующую компанию от обязанности создать потребительскую ценность и порадовать потребителя. По-прежнему, целью является создание программного обеспечения, которое явится ценностью для организации, платящей за него. В действительности потребность в хорошем руководстве и многофункциональной команде, вероятно, только возрастает для продуктов, создаваемых на заказ, по сравнению с продуктами, которые затем предлагаются на рынке. Дело в том, что отсутствие конкуренции может ослабить ориентирование на нужды потребителя, которое является отличительной чертой успешных компаний, занятых созданием программного обеспечения.

От проектов к продуктам

Программное обеспечение на заказ часто создается как проект, однако мы полагаем, что разработка его в качестве продукта представляет собой более полезную модель. Одно из интересных обстоятельств, связанных с проектами, состоит в том, что они (проекты) имеют тенденцию быть профинансированными вначале и сразу (рис. 3.2). После выделения средств естественно, возникает вопрос: *что и когда мы получим в обмен на наше инвестирование?* Ответы на подобные вопросы рассматриваются как обязательства (в конце концов, деньги-то были заплачены!). Иными словами, успех проекта определяется на основе того, не вышел ли проект за пределы выделенной суммы, соблюден ли график и обладает ли созданное программное обеспечение (если цель проекта — создание ПО) всеми необходимыми характеристиками.

Рис. 3.2. Типичный график финансирования проекта

С другой стороны, продукты, как правило, финансируются поэтапно (рис. 3.3). Поэтапное финансирование — это четкий сигнал для всех, что потребители ожидают от создаваемого продукта на каждом этапе все большего соответствия его требованиям. Успех продукта обычно измеряется на основе занятой им доли рынка и прибыльности, которой он в конце концов достигает.

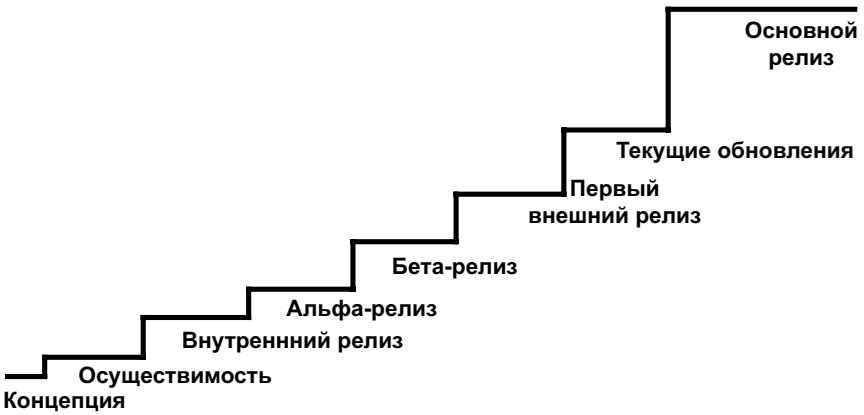


Рис. 3.3. Типичный график финансирования продукта

Имеются и иные отличия между проектами и продуктами. Проекты имеют начало и (очевидно) окончание. Продукты, с другой стороны, имеют начало и (как надеются разработчики) не имеют конца. Программное обеспечение больше похоже на продукт, чем проект, поскольку хорошие программы используются и подвергаются усовершенствованиям в течение долгого времени. Как отмечалось в главе 2, бо́льшая часть всего программного обеспечения разрабатывается после выпуска первой версии.

Созданную на заказ систему нельзя закончить

Когда мы приступали к внедрению системы управления технологическим процессом на предприятии, нас предупредили, что будет множество запросов на изменения. Это показалось хорошим знаком. Если наша система была нужна предприятию, следовало ожидать множества идей, как лучше использовать технологию.

Поскольку изначально не предполагается, что управляющую систему можно "закончить", мы всегда создавали наши системы с возможностью изменения конфигурации. Подобные системы обычно создаются для управления таким оборудованием, как насосы, вальцы, печи и т.п., поэтому некоторые оригинальные идеи часто поступают от операторов, работающих на подобном оборудовании. Так как линия по изготовлению ленты постоянно использовалась для производства различных продуктов, управляющая система должна была позволять оператору очень легко изменять используемые скорость, давление и температуру.

Даже при наличии системы с возможностью изменения конфигурации мы любили уходить из цеха с длинным списком изменений и дополнительных функциональных возможностей, которых хотели операторы линии. Для нас это была высшая оценка. (Мэри Поппендик)

Для выполнения каждого проекта, как правило, создается новый коллектив. С другой стороны, продукты обычно разрабатываются постоянными коллективами, которые продолжают поддерживать их и в дальнейшем. Поддержку программного обеспечения лучше

осуществит постоянный коллектив, поскольку знание запросов заказчика, программный код и т.п. трудно передать другим разработчикам как "наследство". А большинство программного обеспечения обычно используется долго, подвергаясь множеству усовершенствований и улучшений.

Сотрудничество между ОИТ и бизнесом

В отчете When IT's Customers Are External[27], опубликованном в онлайн-журнале *McKinsey Quarterly* Симон Мак-Гиббон (Simon MacGibbon) и его соавторы выражают мысль, что подразделения отделов информационных технологий (ОИТ) крупных корпораций, которые непосредственно взаимодействуют с бизнесом (в котором участвует данная корпорация), должны управляться и быть организованы как компании, производящие программное обеспечение. В отчете указывается, что компании-производители программного обеспечения имеют три отличия от внутренних (для крупных корпораций) ОИТ.

1. Компании-производители программного обеспечения проводят исследования рынка, чтобы по-настоящему понять запросы потребителей с тем, чтобы создавать продукты, которые будут продаваться. Если это будет сделано плохо, соответствующие компании-производители не продержатся долго. Собственно, подобные исследования выполняются регулярно в поиске наилучших ответов на запросы рынка. Поскольку такие компании не могут претендовать на то, чтобы их потребители участвовали в определении проектных решений для будущего продукта, они придумывают разного рода консалтинговые службы, которые призваны узнавать из первых рук, что именно нужно пользователям. Что касается ОИТ, то они часто пренебрегают маркетинговыми исследованиями, предполагая, что это уже сделал кто-то другой, и удивляются, когда созданный ими продукт не отвечает потребностям бизнеса.
2. Компании-производители программного обеспечения действуют на рынке, где имеет место жесткая конкуренция, поэтому они вынуждены поддерживать свои цены на достаточно низком уровне. И они создают достаточно простые продукты, разработка и последующая поддержка которых рентабельны, в то же время заботясь, чтобы предлагаемый набор функциональных возможностей привлекал пользователей и побуждал их покупать их программное обеспечение. В то же время ОИТ часто исходят из того, что перечень функциональных возможностей будущего программного обеспечения должен включать все, в чем может нуждаться бизнес, в том числе и очень дорогие функциональные возможности. У этих отделов мало стимулов поддерживать баланс между затратами и эффектом, что вынуждены делать компании-разработчики, чтобы оставаться на плаву.
3. Компании-производители программного обеспечения понимают, что если пользователи их продуктов не будут успешны в своем бизнесе, они (компании-производители) не будут иметь жизнеспособного бизнеса, поэтому производители пользуются каждой возможностью помочь компаниям-пользователям добиться успеха. В то же время ОИТ очень часто полагают, что успех в использовании созданных ими систем полностью зависит от подразделений корпорации, занимающихся

[27] MacGibbon S. P., Schumacher J. R., Tinaikar R. S. *When IT's Customers Are External*, McKinsey Quarterly, *Q1 2006*.

соответствующим бизнесом. Хотя и верно, что эти корпоративные подразделения должны нести ответственность за успех использования соответствующего программного обеспечения, так же верно, что усилия ОИТ нельзя признать успешными, если созданные им продукты не вносят соответствующий вклад в успех бизнеса.

Многие ОИТ для разработки своего программного обеспечения используют модель проекта. А по нашему мнению, для создания здоровых отношений между таким отделом и соответствующим бизнесом больше подходит модель продукта, поскольку стимулы, связанные с этой моделью, с большей вероятностью смогут создать отношения сотрудничества. Так как ОИТ является частью корпорации, не существует реальных причин для возникновения отношений "мы и они", на которые ориентированы проекты. В этом случае разработчикам нет необходимости делать то, что важно для потребителей программного обеспечения. А чтобы предоставить потребительскую ценность, наиболее необходимую для бизнеса, в кратчайшие сроки и за минимальную цену, помочь в эффективном использовании системы, а затем участвовать в совершенствовании программного продукта, необходимо сотрудничать со своим бизнес-партнером.

Путь к успеху

Однажды на конференции я разговорилась со старшим представителем информационной службы крупной финансовой холдинговой компании. По образованию он был инженером-электриком, которому однажды было поручено "навести порядок" в ОИТ.

"Это было более трех лет назад, и тогда я потерпел неудачу. Я перепробовал несколько методов и спустя два года потерял надежду и вернулся к прежней работе.

"Однако еще через год меня попросили попробовать снова и на этот раз я поступил иначе. Я разделил отдел на две группы: отвечающую за эксплуатацию и занимающуюся разработкой. Затем я разделил группу разработки на шесть небольших коллективов. Каждый коллектив должен был продавать созданное им ПО представителям бизнеса и результаты их деятельности оценивались по полученной от использования этого ПО прибыли в сравнении с его стоимостью.

"Прошло только шесть месяцев, но уже видно, что принятые меры оказались чрезвычайно успешными. Не понимаю, почему это не пришло мне в голову раньше." (Мэри Поппендик)

Ответственность

В крупных корпорациях традиционно имеются отделы информационных технологий (ОИТ), часто независимые от бизнеса, который они обслуживают. Это объясняется тем, что корпорации обычно стремятся иметь стандартную информационную инфраструктуру, а также тем, что технические разработки легче осуществлять, когда соответствующие специалисты являются частью организации. Однако при таком положении дел имеет место неясность, кто является ответственным за результаты деятельности, связанной с разработкой программного обеспечения, а также как оценивать эти результаты.

Вопрос об ответственности не нов для ОИТ. Он возникает каждый раз, когда члены одного коллектива разработчиков выполняют работу для разных подразделений корпорации с различными способами оценки продуктивности. Например, управление разработкой встроенного программного обеспечения может осуществляться отдельно от разработки аппаратных средств.

Не особенно эффективным подходом является распределение ответственности по всем этапам процесса разработки, когда одной части коллектива разработчиков предлагается предъявлять "требования" другой части. Подобный подход имеет тенденцию де-

лать неясной конечную цель деловой активности совместного предприятия и часто ведет к посредственным результатам. Гораздо эффективнее, когда члены многофункционального коллектива делят ответственность за достижение результатов, которые бы оправдывали финансирование их работы.

Однако в конце должна быть единая ответственность по общим результатам инвестирования в ОИТ. Мы полагаем, что эта ответственность должна быть перед бизнесом, который осуществляет финансирование. Когда руководители бизнеса будут относиться к инвестициям в ОИТ с той же строгостью, с какой они управляют своим бизнесом, эти инвестиции наверняка дадут значительные результаты.[28]

Попробуйте это

1. Используя модель Кано, выполните анализ процесса разработки, в котором вы участвуете. Для этого составьте список функциональных возможностей — на высоком уровне, — а затем перенесите пункты этого списка на индексные карточки. На большом листе бумаги создайте модель Кано и прикрепите его к стене. Далее прикрепите карточки к соответствующим областям модели Кано. Как много функциональных возможностей попало в категорию "Восхищение и удивление"? Сколько это процентов?
2. В главе 10 мы описываем показатель, являющийся хорошей мерой удовлетворенности пользователя, который известен как "чистая оценка готовности заказчика рекомендовать организацию, создавшую данный продукт" (net promoter score). Определите оценку готовности рекомендовать вашу организацию для ваших пользователей. Для этого проведите простой опрос, чтобы выяснить у пользователей, насколько вероятно, что они рекомендуют ваш продукт или услугу коллеге. Предложите им оценить это по 10-балльной шкале. Какой процент ответов составляют оценки 9 и 10? Этот процент определяет число "промоутеров" вашей организации. Какой процент ответов составляют оценки от 1 до 6? Этот процент определяет число "очернителей" вашей организации. Вычтите число процентов "очернителей" из числа процентов "промоутеров". Полученный результат положителен или отрицателен?
3. Составьте список программ или проектов в вашей организации, которые в настоящее время открыты. Для каждого пункта списка укажите имя имеющегося в данный момент руководителя (или руководителей). Как много руководителей имеет каждая программа или проект? Есть ли среди них кто-нибудь, способный осуществлять одновременно маркетинговое и техническое руководство? Существует ли какая-либо связь между состоянием проекта и качеством руководства (или отсутствием такового)?
4. Многофункциональные коллективы: имеете ли вы производственных инженеров в вашем коллективе? Проводятся ли регулярные консультации с кем-нибудь из группы технической поддержки при разработке новых функциональных возможностей? Участвуют ли тестировщики в процессе разработки с самого начала? На каком этапе привлекаются создатели инструкций (technical

[28] Lohmeyer D., Pogreb S., Robinson S. *Who's Accountable for IT?*, McKinsey Quarterly, *December 7, 2004.*

writers)? Относятся ли у вас к представителям заказчика как к полноправным членам вашего коллектива?

5. Проекты/Продукты: финансирование разработки у вас осуществляется поэтапно или сразу? Что является критерием успеха разработки? Сохраняются ли коллективы после завершения разработки или люди обычно переводятся в другие коллективы? Сохраняет ли коллектив созданный им программный код? Кто несет ответственность за прибыль от инвестиций в разработку программного обеспечения?

Глава 4

Потери

Меньше программного кода

Если поискать причину непроизводительных затрат при разработке программного обеспечения, наиболее подходящим "кандидатом" окажется излишняя сложность, которая делает код хрупким и неустойчивым. В своей книге *Conquering Complexity in Your Business* ("Как контролировать сложность вашего бизнеса") Майкл Джордж (Michael George) высказывает мысль, что сложность подобна холестерину: она засоряет артерии организации; она "молчаливый убийца прибылей и роста".[1] Рекомендация, как избежать этого, проста: *создавайте менее объемный программный код*[2]!

Давайте присмотримся к компании из Европы с многомиллиардным оборотом, которая поступает именно так.

Zara

Когда Мадонна недавно давала серию концертов в Испании, то, что она надела для своего первого концерта, можно было увидеть на молодежи в зале на ее последнем концерте. Одежду, такую как на Мадонне, можно было купить у Zara — быстро растущей сети магазинов модной одежды. Стратегия Zara состоит в том, чтобы быстро следовать за модой. В этой компании полагают, что куда выгоднее воспроизводить модели, на которые уже есть спрос, чем разрабатывать что-то совершенно новое и продвигать это на рынке. В быстро изменяющемся мире моды следовать за модой необходимо по-настоящему быстро. У Zara путь от концепции к прибыли занимает две недели; возможно, месяц, если для новых моделей требуются совершенно новые производственные линии.

Товары от Zara составляют около 70% всех продаж у Inditex — компании по реализации готовой одежды, расположенной в западной Испании и продающей ежегодно более чем на 5 миллиардов евро. Zara жестко конкурирует с такими компаниями, как H&M из Стокгольма, Benetton из Венеции, а также Gap из Сан-Франциско, и получает одни из крупнейших прибы-

---

[1] George M., Wilson S. *Conquering Complexity in Your Business: How Wal-Mart, Toyota, and Other Top Companies Are Breaking Through the Ceiling on Profits and Growth*, McGraw-Hill, *2004.*

[2] Данная рекомендация (которая буквально применима только для компаний, создающих программное обеспечение) в более широком смысле означает, что чем проще организован бизнес, тем больше у него шансов на успех. — *Примеч. ред.*

лей в своей отрасли. Когда пишутся эти строки, компания имеет около 900 магазинов, около двух третей которых расположены в Европе, и быстро расширяется.

Вместо того чтобы привлекать к себе высококлассных дизайнеров для создания новых моделей на каждый сезон, Zara берет на работу лучших выпускников дизайнерских школ. Эти выпускники работают с менеджерами магазинов для создания предметов одежды, которые спрашивают покупатели. Благодаря этому Zara выпускает 11 тыс. новых моделей ежегодно, по сравнению с от 2 до 4 тыс. у ее конкурентов.[3] Zara не хранит больших запасов товаров. Заказы из магазинов отсылаются в штаб-квартиру компании дважды в неделю, и заказанная одежда прибывает двумя-тремя днями позже, готовая к продаже. Zara избегает массового производства и производит свои товары мелкими партиями, обычно в кооперативах западной Испании. Компания обычно использует только часть своих мощностей, чтобы поддерживать свою способность выполнять заказы дважды в неделю даже в периоды максимального спроса.[4]

Компания Zara получает за свою деятельность больше дохода, чем ее конкуренты. Она продает 85% своих изделий за полную цену, по сравнению со средним показателем по отрасли от 60% до 70%. А непроданные предметы составляют менее 10%, по сравнению со средним показателем по отрасли от 17% до 20%.[5] Zara, по сравнению с конкурентами, также меньше тратит. Родительская (для Zara) компания Inditex расходует на рекламу около 0,3% прибыли, по сравнению со средним показателем по отрасли от 3% до 4%. И она тратит на информационные технологии, возможно, 0,5% прибыли, по сравнению со средним показателем по отрасли около 2%.[6]

Стоп! Давайте задумаемся над последней цифрой. Компания с годовым оборотом в 5 миллиардов евро тратит 0,5% прибыли на информационные технологии? Действительно, штат ОИТ, составляющий около 50 человек, самостоятельно разрабатывает все приложения. В магазинах используются карманные компьютеры (PDA — Personal digital assistant) и модемы для передачи итогов продаж и новых заказов в штаб-квартиру. Вместо использования системы CRM[7], дизайнеры Zara непосредственно общаются с менеджерами магазинов. А использование системы ERP[8] при низких запасах было бы крайностью. Интересно, что генеральный директор (chief executive officer — CEO), который применил такой экономичный подход, является бывшим менеджером отдела информационных технологий.[9]

В своей статье *Do You Have Too Much IT?*[10] Эндрью Мак-Афи (Andrew McAfee) перечисляет пять принципов, определяющих использование технологии компанией Inditex.

1. **Информационные технологии оказывают помощь в принятии решений, но не заменяют при этом людей.**

   Информационные системы помогают менеджерам разобраться в данных, необходимых для принятия решений, но они не принимают решения за людей, и даже не предлагают их.

---

[3] *Inditex: The Future of Fast Fashion*, The Economist, *June 18, 2005*.
[4] Ferdows K., Lewis M.A., Machuca J.A.D. *Rapid-Fire Fulfillment*, Harvard Business Review, *November 2004.*
[5] Из статьи "Rapid Fire Fulfillment," Там же.
[6] McAfee A. *Do You Have Too Much IT?* MIT-Sloan Management Review, *Spring 2004.*
[7] Веб-система для эффективной работы с клиентами.
[8] Веб-сервис управления контактами.
[9] Из статьи "Do You Have Too Much IT?" Там же.
[10] Там же.

2. **Компьютеризация стандартизуется.**

   Магазинам и регионам не позволяется отклоняться от единых корпоративных решений, и разработчики, скорее, *минимизируют* функциональные возможности системы, а не расширяют их.

3. **Технологические инициативы рождаются внутри компании.**

   Формы использования технологии компанией всегда определяются целями бизнеса, а не наоборот. ОИТ работает с менеджерами, чтобы понять, в чем они нуждаются, и только после этого выбираются решения на основе имеющейся технологии.

4. **На первом месте всегда бизнес-процесс.**

   Карманные компьютеры (PDA), имеющиеся в магазинах у менеджеров, не являются для них "устройствами повышения личной продуктивности". Они помогают ежедневно отчитываться о продажах и дважды в неделю делать заказы новых товаров. Эти устройства делают процесс постоянным во всех магазинах.

5. **Взаимопонимание развивается.**

   Именно способность добиваться взаимопонимания помогла стать генеральным директором бывшему менеджеру из ОИТ, однако взаимопонимание между бизнесом и ОИТ подразумевает гораздо большее. Представители бизнеса и представители технологии по-настоящему понимают друг друга. Нет разделения на "мы" и "они" и нет программного обеспечения, "пришедшего извне".

Сложность

Почти каждая начинающая компания, создавшая пока единственный удачный программный продукт, способна быстро реагировать на запросы потребителей и адаптироваться к изменениям. Однако через пару лет успеха такие компании часто замедляют свое развитие и становятся "неповоротливыми". К этому времени они обычно создают обширные объемы программного кода, длинный ряд продуктов и запутанную организационную структуру. Если в этот момент не взять сложность под контроль, она сведет к нулю способность компании реагировать, которая давала ей на первых порах конкурентное преимущество.

Цена, в которую обходится сложность, с течением времени изменяется не линейно, а экспоненциально; в конце концов она превосходит все остальные затраты (рис. 4.1). Сложный программный код (если говорить о влиянии сложности на программное обеспечение) хрупок и склонен к сбоям; в такой код трудно вносить изменения. Организации, занятые созданием программного обеспечения, в которых преобладает мудрый подход, содержат код простым, чистым и небольшим по объему.

Каждая функциональная возможность должна быть оправдана

Первый шаг по контролю за сложностью состоит в том, чтобы агрессивно ограничивать в создаваемом программном обеспечении число функциональных возможностей. Право на существование имеют только те функциональные возможности, стоимость создания которых ниже производимой ими потребительской ценности. Засорение программных продуктов избыточными функциональными возможностями — это ленивый подход к маркетингу: "Мы не знаем, что может понадобиться пользователям, поэтому мы

предоставляем множество функциональных возможностей в надежде, что пользователи сами найдут среди них то, что им нужно". Однако засорение программного обеспечения избыточными функциональными возможностями хуже простой лени; это прямой путь к беде. Программное обеспечение сложно по своей природе, и если этой сложностью не управлять тщательно, она быстро выходит из-под контроля.

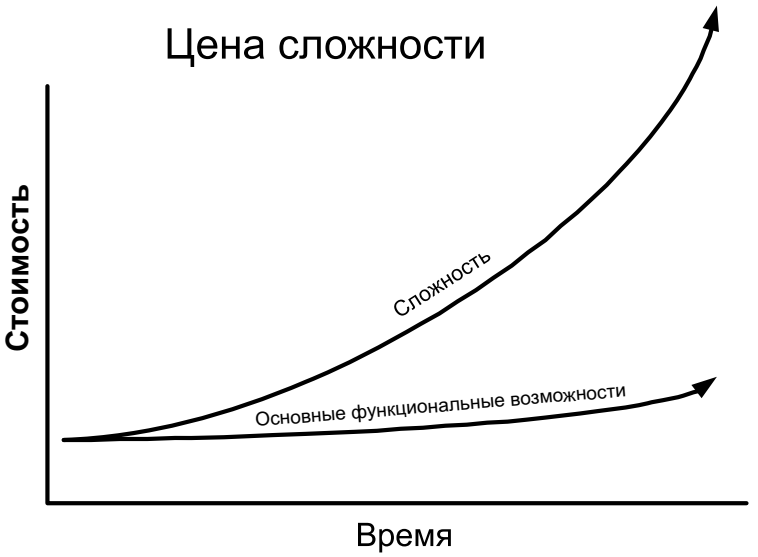


Рис. 4.1. Цена сложности растет экспоненциально

Необходима смелость, чтобы при разработке программного обеспечения ограничивать набор функциональных возможностей в нем, однако такой подход всегда многократно окупается. Выпуск продукта, обладающего минимально необходимым набором функциональных возможностей, ни больше, ни меньше демонстрирует, что компания по-настоящему понимает, чего хотят пользователи. Такой была стратегия компании Intuit, когда она создавала программу QuickBooks. В Intuit понимали, что очень мелкий бизнес не нуждается в программном обеспечении, предназначенном для бухгалтеров, и поэтому был создан продукт, который даже не поддерживал двойной бухгалтерии (double entry bookkeeping), — того, что конкуренты считали обязательным. В Intuit хорошо понимали, какая категория людей является потребителями их продукта, в чем суть их работы, и потому был создан продукт, содержащий только функциональные возможности, необходимые для пользователей.

Джефф Сазерленд (Jeff Sutherland) — технический директор (Chief Technology Officer — CTO) в компании PatientKeeper (о которой речь пойдет в главе 5), подчеркивал, что разработчики не должны создавать ни одной функциональной возможности, пока не будет явно выраженного требования рынка. Вот что он говорит.[11]

[11] Опубликовано в группе новостей `object-technology@yahoogroups.com` March 11, 2003, Message 121. Использовано с разрешения.

Задача в том, чтобы совершенствовать программный код с тем, чтобы он лучше соответствовал нуждам потребителя. Если эти нужны не ясны, программисты не должны создавать ни строчки кода. Каждая строка кода стоит денег, чтобы ее написать, и еще больше денег, чтобы ее впоследствии поддерживать. Лучше пусть разработчики занимаются серфингом, чем создают ненужный программный код. Если создать код, который никогда не будет использоваться, мне придется платить за него до конца жизни системы, которая, как правило, дольше, чем моя профессиональная жизнь. Если они займутся серфингом, им будет весело, а я получу менее дорогую систему и меньше головной боли впоследствии.

Минимально-полезные наборы функций

Независимо, разрабатывается ли программное обеспечение на заказ или создается в качестве продукта для продажи на рынке, идеальный подход состоит в том, чтобы разделить это программное обеспечение на минимально-полезные наборы функций и поставлять их по одному набору за раз, начиная с наиболее нужных (или обеспечивающих наибольшую отдачу). Минимально-полезный набор функций — это тот, который помогает пользователям выполнить лучше некоторую полезную часть их работы. Хотя существуют организации, где подход "все или ничего" воспринимается как единственно-возможный, он редко диктуется необходимостью с технической или пользовательской точки зрения.[12] Мы привыкли думать, что разработка программного обеспечения крупными наборами функциональных возможностей или с использованием подхода "все или ничего" — это хорошо. Наступило время пересмотреть взгляды и прийти к выводу, что в разработке хорошего программного обеспечения этот подход наихудший из возможных.

Ввод в эксплуатацию у заказчика (deployment) минимально-полезных наборов функций (в рамках проекта создания программного обеспечения на заказ) позволяет заказчикам начать использовать это программное обеспечение гораздо раньше. Поскольку эти наборы функций начинают давать прибыль раньше, компания может инвестировать меньше денег и, как правило, получать больше прибыли в течение срока использования системы.[13] С технической точки зрения разработка минимально-полезных наборов функциональных возможностей (когда это делается правильно) целесообразна, поскольку в этом случае программный код создается порциями и упрощается с каждым новым набором. С точки зрения пользователя раннее получение минимально-полезных наборов функций означает более быстрое выполнение им работы, при этом создается обратная связь, позволяющая разработчикам выяснить, как программное обеспечение должно функционировать, когда еще есть много времени для внесения изменений. И с точки зрения устойчивости системы, создаваемые поэтапно, с минимально-полезными наборами функций, обычно легче поддерживать, поскольку поэтапная реализация может быть растянута на срок жизни системы. Иными словами, в реализации системы минимально-полезными наборами функций нет ничего плохого.

Не следует автоматизировать запутанные процессы

Мы не поможем нашим потребителям простой автоматизацией сложного или запутанного процесса; в этом случае запутанность процесса будет просто дополнена сложностью программного обеспечения. Любой процесс, который является "кандидатом" на ав-

---

[12] Игровые программы являются заметным исключением.

[13] Denne M., Cleland-Huang J. *Software by Numbers: Low-Risk, High-Return Development*, Prentice Hall, *2004.*

томатизацию, должен быть прежде очищен и упрощен, возможно, даже с удалением уже имеющихся элементов автоматизации. Только тогда процесс может быть ясно понят и в нем могут быть выявлены некие критические точки, с которых должна начинаться эффективная автоматизация.

"Наша компания — это сама сложность"

На недавнем семинаре одна из карт потока создания ценности (о которой речь еще будет идти далее в этой главе) изображала не процесс разработки программного обеспечения, а процесс обработки заказа на него. Проанализировав карту, мы увидели, что около половины всех заказов приходится пересматривать, даже после того как они прошли через сложный процесс одобрения заказчиками. Задержки и изменения, вызванные чреватым ошибками процессом, были причиной большого недовольства заказчиков и медленными поставками.

"Почему же вы не занялись этими ошибками раньше? Зачем было так долго ждать?" — спросила я.

"Тому имеется множество причин, — объяснил слушатель, представляющий на семинаре карту потока создания ценности. "Часто проблемы проявляются не сразу. С заказом все в порядке в момент, когда он делается, однако с течением времени срок контракта заканчивается и цена или условия изменяются."

При более пристальном рассмотрении стало ясно, что группа обработки заказов прилагает героические усилия, чтобы все было в ажуре, однако структура ценообразования была настолько запутана, что это сводит на нет весь труд.

"Возможно, вам следует обратить внимание на структуру ценообразования, — предложила я. — Она кажется мне крайне сложной."

"Наша компания — это сама сложность", — последовал немедленный ответ. "Мы только что пересмотрели нашу структуру ценообразования; при этом цель была не упростить ее а оптимизировать доход, получаемый нами от каждого заказчика."

Я предположила, что при всех проблемах, которые видны на карте потока создания ценности, дополнительный доход, вероятно, несколько раз "съедается" сложностью.

"Дело не только в этом, — согласился кто-то, а затем уточнил — новая система ценообразования — это компромисс, который никому не нравится: ни заказчикам, ни группе обработки заказов, ни даже отделу сбыта."

Несмотря на то, что проблема, связанная с обработкой заказов была давно заметна, до этого момента никто не пытался выяснить, в чем ее корни, поскольку никто не привык думать о сложности как о чем-то плохом. В действительности в тот момент рассматривалась идея создать компьютерную систему, которая бы смогла справляться со сложностью.

Я высказала мысль, что процесс нуждается в упрощении, а не в автоматизации, поскольку, если они автоматизируют нынешний процесс со всеми его сложностями, вероятно, это не позволит упростить процесс обработки заказов *никогда*. (Мэри Поппендик)

Семь непроизводительных расходов

Каждый, кто изучал бережливое производство, знает о семи непроизводительных расходах производства Сигаэо Синго (Shigeo Shingo).[14] В нашей предыдущей книге мы рассказали о семи непроизводительных расходах разработки программного обеспечения, аналогах семи непроизводительным расходам производства. В этой главе мы вновь приводим эту информацию, которая представлена в табл. 4.1. Обнаружив эти расходы,

[14] Shingo S. *Study of "Toyoda" Production System from an Industrial Engineering Viewpoint*, Productivity Press, *1981,* Chapter 5.

нецелесообразно классифицировать их и распределять по категориям. Категории воспитывают привычку мириться с потерями. Настоящей целью обнаружения и ликвидации непроизводительного расхода (или потери) является сокращение себестоимости и повышение эффективности наших продуктов.

Таблица 4.1. Семь непроизводительных расходов

Производство	Разработка программного обеспечения
Излишние запасы	Частично выполненная работа
Перепроизводство	Избыточные функциональные возможности
Повторное выполнение работы	Повторное приобретение знания
Транспортировка (излишняя — узлов и материалов по предприятию)	Передача работы
Перемещения (излишние — людей или оборудования)	Переключение между задачами
Ожидание	Задержки
Дефекты	Дефекты

Частично выполненная работа

Частично выполненная работа при разработке программного обеспечения является эквивалентом излишних запасов в производстве. Для уменьшения объемов частично выполненной работы следует с начала разработки системы приступить к созданию сразу интегрированного, проверенного, документированного и готового к использованию программного кода. Единственный способ осуществить это состоит в том, чтобы разделить работу на небольшие этапы, или итерации.

Примеры частично выполненной работы

1. **Документация, созданная заранее.** Чем дольше документация о будущем программном обеспечении и требованиях к нему "лежит на полке", тем больше вероятность, что потребуется вносить в нее изменения. У разработчиков часто возникает раздражение из-за необходимости вносить изменения в требования, но с точки зрения заказчика настоящая проблема в том, что эти требования были сформулированы слишком рано.
2. **Несинхронизированный программный код.** Когда система разрабатывается параллельно (над различными элементами системы одновременно работают несколько разработчиков или даже коллективов), эти элементы в конце концов необходимо объединять (или синхронизировать). Так вот, это объединение (или синхронизация) должно осуществляться максимально часто, поскольку, чем дольше элементы системы разделены, тем труднее будет их объединить.
3. **Непротестированный код.** Создание программного кода без немедленного тестирования — это прямой путь к увеличению объемов частично выполненной работы. При определении объема выполненной работы код, созданный не полностью, учитываться не должен. Либо код интегрирован, протестирован и принят, либо его нет вовсе.
4. **Недокументированный код.** Если необходимость документации программного кода существует, документация должна писаться по мере создания программного кода. В идеале код должен быть самодокументированным (self-documenting), однако пользовательская документация, экраны справки и т.п. могут также понадобиться. Признано целесообразным, чтобы в составе коллектива разработчиков были тестировщики программного кода; то же верно и для создателей документации (technical writers). В конце концов именно им пред-

стоит помочь пользователям выполнять их работу. Можно было бы избежать включения в код большого числа ненужных функциональных возможностей, если бы создатели документации постоянно спрашивали остальную часть коллектива разработчиков: "Как именно эта функциональная возможность поможет нашим заказчикам выполнить их работу?"

5. **Редко выпускаемый код.** Не всегда выпуск программного кода возможен так часто, как это желательно, поскольку новое программное обеспечение (из-за большого числа новшеств в нем) может мешать заказчикам в выполнении их текущей работы. Тем не менее следует стремиться выпускать (deploing) код как можно раньше. В действительности для пользователей часто легче воспринимать изменения небольшими порциями: при этом требуется меньше обучения и помехи в работе обычно удается свести к минимуму.

Мы вынуждены иметь дело с большим объемом документации, но не в нашей власти изменить это, поскольку государственные нормы (США — *Примеч. ред.*) предписывают, чтобы код соответствовал Спецификации требований к программному обеспечению (Software Requirements Specification — SRS) и поддерживал трассируемость к программному коду (traceability to code)

Если вам приходится поддерживать SRS и трассируемость, в этом случае, возможно, имеет смысл создать максимальное число SRS в виде выполняемых тестов. Попробуйте использовать структуру интеграционного тестирования (Framework for Integrated Tests — FIT) или какое-либо иное средство приемочного тестирования, чтобы создать "спецификацию на основе примера" (specifications-by-example).[15] Подобное средство может предоставить вам бесплатную трассируемость результатов тестов к программному коду (traceability of tests to the code). Если эти тесты использовать каждый день и сохранять результаты в вашей системе управления конфигурацией, вы будете иметь данные о том, какие тесты выполнились и какие нет в каждый момент времени. Тем, кто призван следить за соблюдением норм и спецификаций, такой подход понравится.

Избыточные функциональные возможности

Тайити Оно (Taiichi Ohno) подчеркивал, что излишнее производство (т.е. создание запасов, в которых нет немедленной необходимости) — это худший из семи непроизводительных расходов в производстве. Аналогично, худший из семи непроизводительных расходов в разработке программного обеспечения состоит в добавлении в ПО функциональных возможностей, которые не нужны заказчикам при выполнении ими своей текущей работы. Если нет явной экономической необходимости, данная функциональная возможность *не должна разрабатываться*.

Но можно ли предвидеть заранее, какие функциональные возможности могут понадобиться?

Нам часто задавали этот вопрос. Вот несколько советов, которые, как нам кажется, помогут определить перечень необходимых функциональных возможностей.

1. Руководствуйтесь здравым смыслом.
2. Вникните в суть работы заказчика программного обеспечения. Не являются избыточными только те функциональные возможности, которые абсолютно необходимы для выполнения заказчиком его работы.

[15] Mugridge R., Cunningham W. *Fit for Developing Software: Framework for Integrated Tests*, Prentice Hall, 2005. См. также Интернет-ресурс `www.fitnesse.org`.

3. Не торопитесь добавлять в создаваемое программное обеспечение новые функциональные возможности. Если в необходимости той или иной функции возникает сомнение, значит, добавлять ее преждевременно.
4. Весьма полезно создать архитектуру программного обеспечения таким образом, чтобы поддерживалось добавление новых функциональных возможностей на максимально позднем этапе. Также неплохой идеей является предварительное создание для данного предприятия перечня повторяющихся работ. А вот попытки создания некоего приложения, которое можно было бы конфигурировать для выполнения чуть ли не любой мыслимой работы, обычно ведут к неудаче.

Повторное приобретение знания

Недавно мы принимали участие в дискуссии о том, как гибкая (agile) методология разработки программного обеспечения влияет на пользователей, когда кто-то спросил: "Есть ли в этой комнате пользователи? Было бы интересно услышать мнение настоящих потребителей." В ответ поднялась единственная рука. Пользователю задали вопрос: "Какие сложности гибкая методология представляет для вас?"

"Мне кажется, — ответил он — что наибольшая проблема, с которой я столкнулся, состояла в том, чтобы помнить, какие решения я уже принимал, и что я уже пробовал, поэтому нередко мне приходилось проделывать то же снова."

Повторное открытие чего-то, что мы раньше знали, но забыли — это, вероятно, наилучший пример "повторного выполнения работы" при разработке. Мы знаем, что должны запомнить то, чему научились. Тем не менее наш подход к накапливанию знания очень часто поверхностный и куда менее строгий, чем ему следовало бы быть. Сложная тема создания и сохранения знания будет рассматриваться далее в этой книге в главе 7.

Еще одним примером пренебрежения знанием является игнорирование опыта членов коллектива. Это еще хуже, чем забыть то, что мы уже знали. Крайне важно использовать знание всех членов коллектива, приобретенное ими с течением времени.

Мы стараемся документировать все наши решения по дизайну программного обеспечения по мере того, как они принимаются. Проблема в том, что в эту документацию никто никогда не заглядывает.

Существует общая проблема, которая является причиной, почему во многих организациях совсем перестали документировать проектные решения, касающиеся будущего программного обеспечения. Знание, почерпнутое из попытки (что-то сделать), которая оказалась неудачной, может быть наиболее существенно для решения вашей проблемы, однако сохранение этого знания таким образом, чтобы к нему было легко обратиться позже, представляет трудность. Начните с вопроса: работает ли то, что вы делаете в настоящий момент? Если нет, не ст*о*ит продолжать делать это. Лучше обратитесь снова к вашим глобальным целям и работайте с вашим коллективом в поиске метода сохранения знания, обеспечивающего достижения этих целей наиболее эффективным образом. Возможно, некоторые идеи на этот счет вы сможете почерпнуть в главе 7.

Передача работы

Ничто не сравнится с обучением ребенка езде на велосипеде. Сначала он должен научиться держать равновесие во время движения, поэтому вам приходится бежать рядом, слегка поддерживая велосипед. Это достаточно утомительно, поэтому, когда он приобретает чувство равновесия, вы тут же начинаете учить его, как ехать. Первый раз, когда он

успешно отъезжает от вас, вы вдруг вспоминаете, что забыли его научить, как останавливаться! Еще несколько неудач, и вот уже ребенок едет самостоятельно. Двумя часами позже вы удивляетесь, как уверенно он мчится по аллее, легко отъезжая и останавливаясь с визгом тормозов.

Передать незаконченную работу коллеге, это примерно то же, что дать велосипед кому-то, кто не знает, как ехать. Можно при этом дать объемистую инструкцию, как управлять велосипедом, однако от этого будет мало пользы. Гораздо лучше, если бы вы оставались рядом и помогли коллегам приобрести чувство равновесия, которое приходит во время движения, затем помогли некоторыми указаниями, когда они практикуются отъезжать, останавливаться, поворачивать, съезжать с горки и въезжать на нее. При таком подходе ваш коллега очень скоро научится ездить на велосипеде, хотя он не сможет объяснить, как он это делает. Подобное знание называется молчаливым знанием (tacit knowledge); его трудно передать другим людям через документацию.

Когда работа передается коллегам, большой объем молчаливого знания остается в уме того, кто начинал работу. Задумайтесь: если каждая передача работы оставляет позади (по очень скромной оценке) 50% знания, тогда:

- после того как работа передавалась дважды остается 25% знания;
- после того как работа передавалась трижды остается 12% знания;
- после того как работа передавалась четыре раза остается 6% знания;
- после того как работа передавалась пять раз остается всего 3% знания.

Поскольку молчаливое знание передать другому лицу трудно, передача работы коллегам всегда ведет к потере знания. Вопрос в том, как свести к минимуму эту потерю.

Некоторые способы сокращения потерь при передаче работы

1. Сократить число случаев, когда работа передается другому лицу (или коллективу).
2. Использовать многофункциональные коллективы, в которых люди могли бы учиться друг у друга.
3. Где возможно, заменить передачу знания через документацию очными обсуждениями, непосредственным наблюдением, а также работой с макетами, прототипами и имитаторами.
4. Выпускать созданные продукты заблаговременно (и частично готовыми) заказчикам для ознакомления и организации с ними обратной связи — так быстро, как это возможно, и так часто, как это практично.

Переключение между задачами

При разработке программного обеспечения, чтобы вникнуть во все сложности задачи и без ошибок добавить в головоломку еще один недостающий элемент, требуется большая сосредоточенность. В этих условиях вынужденное переключение на другую задачу не только отвлекает внимание, но также отнимает время и часто ухудшает результаты для обеих задач. Когда работники умственного труда вынуждены одновременно работать над тремя или четырьмя задачами, они тратят больше времени на вхождение в курс дела при каждом переключении с задачи на задачу, чем на собственно работу. Подобное переключение неизбежно сопровождается потерями.

Более того, попытка выполнять несколько задач одновременно обычно бессмысленна. Предположим, вам необходимо выполнить три задачи: А, Б и В. Предположим также, что для выполнения каждой задачи требуется неделя. Если работать над задачами по очереди, тогда к

концу первой недели будет выполнена задача А, которая тут же начнет предоставлять потребительскую ценность. К концу второй недели будет выполнена задача Б, которая также начнет предоставлять ценность. А к концу третьей недели будут выполнены все три задачи и к этому времени будет произведен большой объем потребительской ценности.

Но, предположим, было принято решение работать над всеми тремя задачами одновременно (возможно, чтобы дать понять заказчику каждой задачи, что его заказ для вас важен). Наиболее благоприятный вариант этого сценария представлен на рис. 4.2. Каждая задача разделена на девять равных частей и на переключение между задачами тратится минимальное время. Даже в этом идеальном случае ни одна из задач не будет выполнена к концу третьей недели. В дополнение к потерянному времени, необходимому для выполнения задач, потенциальная ценность, которая могла бы быть получена, если бы задачи были выполнены раньше, также потеряна.

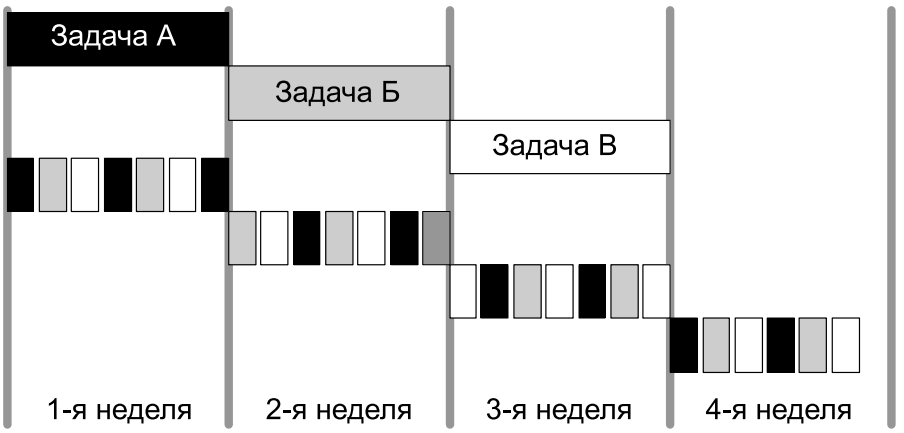


Рис. 4.2. Переключение между тремя задачами (для выполнения каждой из которых требуется неделя) означает, что к концу третьей недели ни одна из них не будет выполнена

Нецелесообразно иметь отдельный коллектив для поддержки каждого программного продукта, поэтому от переключения между задачами никуда не деться.

Каждый программный код, который поставляется заказчикам, нуждается в поддержке, и иногда создаются отдельные коллективы по поддержке соответствующих программных продуктов с тем, чтобы разработчики смогли сосредоточиться на процессе разработки и не отвлекались на задачи поддержки. Однако в целом мы не рекомендуем подобный подход, поскольку верим, что лучше всего, когда коллектив остается со своим продуктом в течение всего периода его жизни. В противном случае разработчики могут начать верить, что код можно "закончить", что, как правило, является мифом. Программный код — это живой организм, который подвергается постоянным изменениям.

В случае, когда коллектив разработчиков поддерживает созданный им код, не избежать переключения между задачами. Естественно, это создает сильную мотивацию создавать свободный от ошибок программный код, чтобы впоследствии не отвлекаться от разработки новых продуктов.

Тем не менее невозможно игнорировать требования со стороны потребителей о поддержке программных продуктов. Вот несколько способов свести к минимуму ущерб, связанный с отвлечением разработчиков для поддержки ранее созданного программного кода.

1. Ежемесячно (или для каждой итерации) назначать двух разработчиков, которые бы решали все вопросы, связанные с поддержкой, в течение данного периода.
2. Выделять два часа каждое утро, в течение которых коллектив совместно решал бы все проблемы, возникшие за последние 24 часа. После этого целесообразно провести дневную планерку для решения вопросов о разработке новых продуктов на остаток дня.
3. Жестко устанавливать очередность для запросов на поддержку и срочно выполнять только неотложные из них. Выделять раз в неделю (или в две недели) время для решения вопросов, связанных с поддержкой. Такой подход помогает выровнять загрузку, связанную с поддержкой.[16]
4. Выпускайте новую версию программного продукта каждую неделю. Устраняйте в этой версии все проблемы, возникшие за последнюю неделю, и требуйте, чтобы все перешли к использованию новой версии.

Задержки

Огромные потери из-за задержек имеют место, когда разработчикам приходится разыскивать людей, которые могут предоставить им необходимую информацию. Дело в том, что разработчики принимают критические решения каждые 15 минут, и не всегда всю необходимую для этого информацию можно найти в прилагаемой документации. Решение может быть принято быстро, если разработчик хорошо понимает, для чего предназначен создаваемый им программный код, и если рядом имеется кто-то, кто может прояснить остающиеся вопросы. В противном случае у разработчика имеется три варианта действий: остановиться и попробовать найти ответ на вопрос самостоятельно, заняться другой задачей или просто попробовать догадаться, каким в данном случае является правильный ответ, а затем продолжить работу. Если с поиском ответа на вопрос возникают трудности, разработчик примет второй или третий вариант. Если задержка не чревата взысканием, разработчики предпочтут долго ждать ответа на возникший вопрос, прежде чем продолжить работу. Ни один из перечисленных подходов нельзя признать приемлемым.

Резко уменьшить задержки (и повысить при этом качество решений) способны многофункциональные, сбалансированные коллективы и краткие итерации с регулярной обратной связью. Это не единственный способ сократить задержки, однако независимо от того, где физически находятся члены коллектива, важно, чтобы соответствующее знание было доступно именно тогда (и там), когда (и где) оно необходимо — не раньше, или оно может успеть измениться, не позже, или оно окажется уже ненужным.

Почему на разработку программного обеспечения уходит так много времени?

С точки зрения заказчика время, которое ему приходится ожидать заказанного ПО, куда более продолжительно, чем задержки, связанные с ожиданием разработчиком ответов на вопросы.

1. Ждут, пока я буду знать, что именно мне нужно, прежде чем они приступят к решению моей проблемы. Как я могу это знать заранее?!
2. Ждут месяцами одобрения проекта!
3. Приходится ждать "вечно", пока для выполнения задачи будут выделены соответствующие специалисты!

[16] Нам следует поблагодарить Бента Йенсена (Bent Jensen) за его разрешение упомянуть этот подход (который он считает очень полезным).

4. Приходится ждать, пока выделенные специалисты освободятся (от предыдущих задач)!
5. Этот досадный процесс утверждения изменений! Они заставляют меня ждать месяцами, при этом, наверное, думая, что в моем бизнесе ничего не меняется!
6. Приходится ждать, пока вся система будет закончена, прежде чем я получу основные функциональные возможности, которые нужны прямо сейчас!
7. Приходится ждать, пока программный код пройдет все тесты — почему на это уходит так много времени?!
8. Приходится ждать, пока будут осуществлены соответствующие настройки, чтобы вновь полученная программа перестала мешать работе моих уже имеющихся программ — или наоборот — кто знает?!

Дефекты

Каждый программный код должен включать набор защищающих от ошибок тестов, которые не позволят проникнуть в него дефектам, как на стадии приемочных испытаний (acceptance tests), так и блочного тестирования (unit tests). Однако подобные тесты только способны подтвердить, что код функционирует так, как мы этого от него ожидаем, и что он не отказывает так, как это предполагается. При этом программное обеспечение каким-то образом находит обходные пути, чтобы отказать, поэтому код должен тестироваться достаточно рано и достаточно часто, чтобы выявить и устранить максимальное число подобных непредвиденных дефектов.

Когда дефект обнаружен, должен быть создан соответствующей тест, чтобы упростить устранение подобных дефектов в будущем. Кроме того, могут понадобиться средства для тестирования "дыр" в безопасности, способности выдерживать нагрузку (load capability) и т.д. Средства комбинаторного тестирования также могут быть очень полезны. Дефекты должны обнаруживаться и устраняться так рано, как это возможно, чтобы, когда работа будет вестись над финальной версией, не пришлось, как это часто бывает, заниматься поиском дефектов. Если привычна ситуация, когда программное обеспечение поступает на заключительную верификацию с дефектами, значит, для его создания использовался несовершенный процесс.

Программный код, разработанный коллективом профессионалов, где практикуется гибкая (agile) методология разработки, содержит чрезвычайно малое число дефектов, поскольку работа строится так, чтобы код был максимально защищен от появления в нем дефектов; при этом сам факт появления дефекта рассматривается как событие экстраординарное. А на втором месте в таком коллективе стоит выявление дефектов на максимально раннем этапе и предотвращение повторного появления этих дефектов в будущем.

Однако настоящая причина переноса тестирования в начало процесса разработки не сводится только к защите от ошибок. Приемочные тесты (acceptance tests) наиболее эффективны, когда они участвуют в определении проектных решений для будущего программного продукта и согласуют эти решения с особенностями бизнеса заказчика. В то же время блочные тесты (unit tests) целесообразно применять при проверке программного кода. Создание блочных тестов до создания кода ведет к получению более простого, понятного и доступного для тестирования кода. Эти тесты продемонстрируют точно и подробно, как будет функционировать код и конечный продукт.

Всестороннее автоматическое тестирование — источник потерь?

Сигаэо Синго (Shigeo Shingo) утверждает, что "контроль с целью предотвращения дефектов" абсолютно необходим в каждом процессе, но "контроль с целью выявления дефектов" — это источник потерь.[17] Коллективы разработчиков рано или поздно приходят к выводу, что предотвращение ошибок и их устранение, прежде чем создавать новый программный код, в конце концов оказывается более быстрым подходом. Более того, использование в полной мере тестов делает простыми и легкими изменения кода в будущем, что еще более усиливает эффект. Наконец, в организациях, где привыкли создавать подробные спецификации с требованиями к будущим продуктам, вдруг обнаруживают, что создание тестов в виде выполняемых спецификаций может отнимать куда меньше времени, чем написание обычных спецификаций, а затем трассирование их к программному коду.

У нас нет возможности выполнить тестирование в уникальной среде заказчика, поэтому поиск дефектов мы всегда осуществляем при инсталляции.

Иногда невозможно выполнить заключительное комплексное тестирование в реальной среде у заказчика, поэтому в таких случаях поиск дефектов осуществляется при инсталляции нового программного обеспечения. Однако и при этом следует воспользоваться программой совершенствования процесса (improvement program), которая выявит наиболее распространенные случаи сбоев при инсталляции, а затем позволит устранить причины, вызвавшие эти случаи (начиная с наиболее приоритетных), пока дефекты, проявляющиеся в среде заказчика, не станут редкостью.

Несколько известных нам компаний делают инсталляцию *частью итерации разработки*. Итерация не считается завершенной до тех пор, пока программное обеспечение не заработает у заказчика в реальных условиях его среды. При таком подходе весь коллектив разработчиков участвует в инсталляции программного обеспечения у заказчика. При этом специально выделяется время, чтобы иметь возможность реагировать на требования заказчика (и в ходе этого не делается никаких попыток различать дефекты и функциональные возможности ПО), которым, если они практичны, всегда идут навстречу.

Картирование потока создания ценности

Тайити Оно (Taiichi Ohno) выразил суть производственной системы Toyota следующим образом: "Все сводится к совершенствованию графика выполнения работ с момента получения заказа и до получения оплаты за выполненный заказ. При этом сроки сокращаются за счет ликвидации всех ненужных затрат."[18] (Об это уже шла речь в главе 2.) График выполнения работ, который упоминает Оно, может быть изображен в виде карты потока создания ценности (value stream map), которая является диагностическим инструментом, часто используемым при реализации бережливых решений. Мы полагаем, что это же средство может быть использовано и для разработки, только в этом случае следует изменить точки начала и останова отсчета времени, чтобы учесть иное взаимодействие заказчика с процессом разработки.

Карты потока создания ценности начинаются с заказчика и заканчиваются заказчиком. В разработке на карте потока создания ценности отсчет времени начинается, когда заказчик делает свой заказ (точный смысл этого действия в разных организациях может различаться). Отсчет времени прекращается, когда предоставляется успешное решение проблемы заказчика. Карта потока создания ценности отображает во времени основные события, происходящие с начала и до останова отсчета времени.

---

[17] Shingo, Там же, p. 288.

[18] Ohno T. *Toyota Production System: Beyond Large Scale Production*, Productivity Press, *1988,* p. 6.

Целью бережливого подхода является сокращение времени разработки посредством ликвидации затрат, которые не добавляют потребительской ценности. Карты потока создания ценности зарекомендовали себя как эффективное средство выявления непроизводительных затрат, поскольку присутствие задержек в потоке почти всегда является признаком значительных потерь. Если присмотреться к продолжительным задержкам (которые являются признаком очередей) и к случаям, когда переделывается уже сделанная работа, возникает ясная картина потерь, имеющих место в процессе. На наших курсах небольшие команды для практики, не заботясь при этом об особой точности, часто строят карты потока создания ценности (на что у них обычно уходит полчаса). Несмотря на то, что эти карты отображают реальность очень приближенно, они очень помогают понять основные недостатки производственных процессов.

Подготовка

Выбор процесса

Первый шаг в построении карты потока создания ценности состоит в том, чтобы решить, что она должна отображать. Идеальным решением является отображение на этой карте всего процесса, а не какого-то одного события или набора событий, однако для разработки программного обеспечения это может быть затруднено. Хорошая альтернатива состоит в том, чтобы построить карту потока создания ценности для одного проекта, который является достаточно типичным. При выборе потока создания ценности для отображения на карте, целесообразно сгруппировать вместе сходные этапы разработки. Например, можно было бы отобразить на этой карте, сколько времени проходит от создания концепции продукта средней сложности до выпуска этого продукта. Или можно подобным образом отобразить графически добавление новой (высокоприоритетной) функциональной возможности в существующее приложение.

Например, в большинстве подразделений, занятых поддержкой программного обеспечения, знают, как можно группировать вместе сходные процессы. Обычно при этом запросы на обслуживание делятся на три категории (по их срочности) и время реакции гарантируется согласно категории, которой принадлежит тот или иной запрос. Например, решение крайне неотложной проблемы может быть гарантировано в течение двух часов, важной проблемы в течение одного дня, а повседневная проблема может быть отложена до выпуска очередной версии (это может иметь место раз в две недели). Организации, занятые поддержкой программного обеспечения, могут дать пару уроков о том, как следует строить карты потока создания ценности.

Выбор начала и останова отсчета времени

Далее необходимо решить, с какого момента следует начинать отсчет времени. При создании нового продукта началом процесса разработки часто полагают принятие концепции продукта. Однако на этот счет существуют различные мнения, поэтому, возможно, имеет смысл начать отсчет несколько раньше, например, когда специалисты по маркетингу распознали рыночный спрос. Однако отсчет не в коем случае не следует начинать после принятия концепции продукта, даже если ваша организация не участвует в процессе с самого начала. Если вы разрабатываете встроенное программное обеспечение, в идеальном случае отсчет времени следует начать с начала разработки основного продукта, а не его программной части.

При разработке программного обеспечения в ответ на заявку со стороны потребителя счет времени обычно следует начинать в момент подачи заявки (если под заявкой подразумевается соответствующим образом оформленный заказ). Чтобы начать отсчет обычно не требуется ожидать, пока будет одобрен проект, поскольку в большинстве случаев *процесс одобрения должен быть включен в поток создания ценности*. На потребительскую ценность следует смотреть с точки зрения *заказчика*, а заказчику не важно, есть ли у вас другие заказы и насколько вы заняты. Заказчика волнует, сколько времени потребуется, чтобы вы выполнили *его* заказ.

Необходим ответственный

Можно отобразить поток ценности на карте и без ответственного (owner), однако от карты будет гораздо больше пользы, если ее создание (а также осуществление соответствующего процесса впоследствии) будет осуществляться под руководством ответственного лица. Мы обнаружили, что наибольшие проблемы всегда возникают на границе между взаимодействующими организациями, где никто не несет ответственности за выполнение заявки заказчика, и люди по обеим сторонам такой границы стараются оптимизировать собственную локальную эффективность. Заявка может неопределенное время содержаться в очереди, прежде чем ее одобрят, бесконечно ожидать, пока ее передадут из одной организации в другую (если в выполнении данного заказа участвуют несколько организаций) или вечно дожидаться выполнения тестирования и передачи заказчику. Если нет ответственного за выполнение заказа на протяжении всего процесса, кажется, никто не в состоянии ликвидировать эти источники потерь.

Не нужно усложнять

Карты потока создания ценности — это диагностические средства, призванные помочь выявить непроизводительные расходы, однако сами по себе они обычно не добавляют сколько-нибудь ощутимой потребительской ценности. Эти карты помогают разработчикам взглянуть глазами заказчика на свой процесс разработки. Карты потоков создания ценности изменяют взгляды разработчиков и побуждают их вести диалог с заказчиками на конструктивной основе. Они — хорошая отправная точка для обнаружения и ликвидации потерь.

Целью карты потока создания ценности является отображение, возможно, десяти или больше этапов — от оформления заявки заказчиком до создания и выпуска продукта, удовлетворяющего заказчика, — на одном или двух листах бумаги. При этом более важно охватить всю информацию, представленную на карте в целом — от концепции до получения оплаты за выполненный заказ, — чем вдаваться в детали отдельных этапов. Создав карту, попробуйте ответить на два вопроса.

1. Сколько времени занимает разработка продукта или выполнение заказа? (Учитывать при этом следует прошедшее время, а не подлежащие оплате часы.)
2. Какой процент этого прошедшего времени действительно был потрачен на увеличение потребительской ценности? (Этот показатель известен как эффективность цикла процесса (process cycle efficiency — PCE).[19])

[19] George M., Wilson S. *Conquering Complexity in Your Business: How Wal-Mart, Toyota, and Other Top Companies Are Breaking Through the Ceiling On Profits and Growth*, McGraw-Hill, *2004,* p. 29.

Мы видели множество различных форматов карт потока создания ценности, каждый из которых приемлем, поскольку все они вызывают много полезных дискуссий о потерях. На карте можно делать разного рода пометки и надписи, если это поможет выявить потери. Только в процессе их создания не следует забывать о назначении карт потока создания ценности: выявлять потери с тем, чтобы их ликвидировать.

Примеры

Вероятно, наилучший способ понять, чем полезны карты потока создания ценности, состоит в том, чтобы попробовать создавать их. На наших курсах учащиеся проявили способность создавать такие карты, руководствуясь минимумом инструкций. Однако польза от карт потока создания ценности заключается не в самом их создании а в возможности анализировать содержащуюся на них информацию. Поэтому мы приведем примеры нескольких карт, достаточно типичных среди тех, которые мы видели на наших курсах, а затем обсудим их смысл[20].

Пример 1

Пример 1 (рис. 4.3) демонстрирует карту потока создания ценности (из созданных на наших курсах) изменения небольшой высокоприоритетной функциональной возможности (некоторой программы). Заявка от заказчика поступает по электронной почте диспетчеру, которой санкционирует ее (в среднем) через два часа. Еще два часа занимает краткая техническая оценка, после чего эта заявка может быть передана разработчику. Слушатель наших курсов, составивший эту карту, заметил: “Конечно, разработчик был быстро найден, поскольку это была высокоприоритетная работа”. Поэтому разработчик приступил к выполнению заказа через час, и на выполнение всей работы ушло два часа. После этого немедленно была проведена финальная (или заключительная) верификация, а затем готовый продукт тут же был передан заказчику для ввода в эксплуатацию. Вывод: на выполнение небольшого высокоприоритетного заказа ушло восемь часов (от принятия заявки до передачи заказчику продукта). И всего два часа и сорок минут из них были потрачены на реальную работу над выполнением заказа, что составляет треть от восьми часов общего времени и делает эффективность цикла процесса равной 33%.

Многие организации, занятые поддержкой программного обеспечения, имеют дело с процессами, аналогичными этому. Это очень эффективный процесс (к такому выводу мы сможем прийти вскоре, когда сравним его со следующим примером). Однако и в этом случае имеются возможности для совершенствования (например, заявка ожидала технической оценки два часа!). Если бы эту оценку возложить на разработчика, в этом случае экономились бы в среднем два часа на заявку!

---

[20] Следует иметь в виду, что карты потоков ценности обычно создаются от руки. Для этой книги мы использовали текстовый процессор, однако читателям рекомендуем пользоваться карандашами и миллиметровой бумагой.

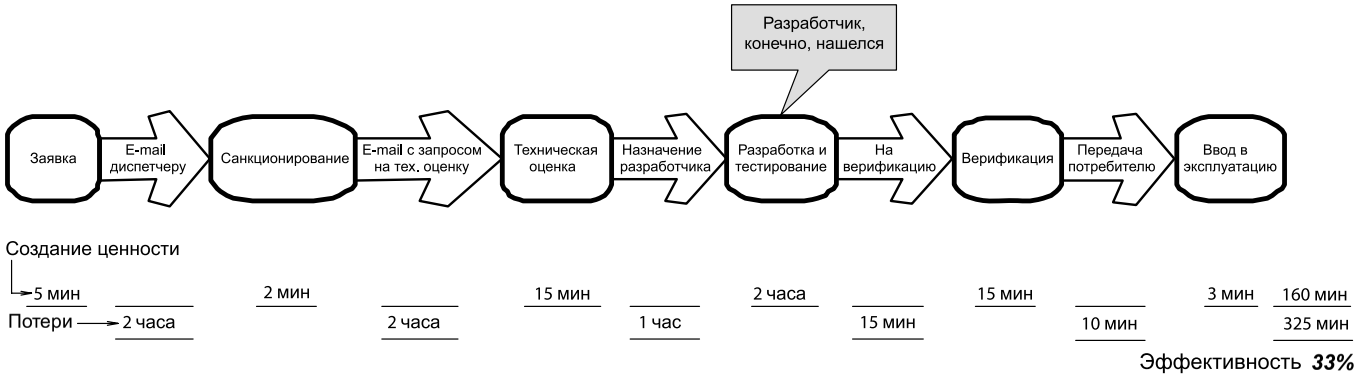


Рис. 4.3. Карта потока создания ценности для изменения небольшой высокоприоритетной функциональной возможности — Организация А

Пример 2

Пример 2 (рис. 4.4) демонстрирует карту потока создания ценности для заявки примерно такой же трудоемкости, как в примере 1. Нужно внести изменение в небольшую функциональную возможность (некоторой программы); на изменение программного кода и тестирование требуется около двух часов. Однако на выполнение этой работы ушло более шести недель! С точки зрения заказчика для оформления заявки (по сравнению с предыдущим примером, где заявка была осуществлена по e-mail) дополнительно требуется всего 15 минут, поскольку в этом случае должна быть использована стандартная форма, для заполнения которой требуется значительно больше информации. Поскольку заявки рассматриваются один раз в неделю, заявка ожидает в среднем половину недели, чтобы получить санкцию. Затем заявка ожидает в среднем две недели, пока будет выполнена техническая оценка, а затем еще в среднем две недели, пока освободится разработчик. После двух часов работы по внесению изменений в программный код и тестирования, заявка ожидает в среднем неделю, поскольку релизы (новых версий) по плану проводятся только раз в две недели. Перед релизом должна выть выполнена финальная верификация. Несмотря на то, что весь программный код был тщательно протестирован при его создании, одна из добавленных в релиз за последнюю неделю порций программного кода внесла в него дефект, который остался необнаруженным вплоть до финальной верификации. Поэтому потребовалось четыре часа, чтобы устранить дефект и выполнить тестирование (вдвое больше времени, чем потребовалось для первичного внесения изменений и тестирования программного кода). (Поскольку верификация в предыдущем примере заняла всего 15 минут, остальные три часа и сорок пять минут являются потерями, навязанными процессом.) Наконец, все готово для ввода созданного (или вернее модифицированного) программного обеспечения в эксплуатацию в среде заказчика (deployment), однако еще уходит в среднем половина недели на то, чтобы заказчик освоился с новой версией ПО (содержащего новую функциональную возможность) и начал использовать ее в своем производстве.

Примеры 1 и 2 близки к реальным картам потока создания ценности, которые создавались на наших курсах, поэтому мы смогли убедиться, что представленные здесь окружающая среда и проблемы достаточно типичны. Организация Б согласилась, что если бы использовался процесс, принятый в организации А, заявка, о которой идет речь в примере 2, могла бы быть выполнена в течение дня.

Из этих двух примеров следует извлечь два урока. Во-первых, несмотря на то, что разработчики в обеих организациях одинаково загружены, в организации А работа организована таким образом, что всегда найдутся разработчики, которые оставят низкоприоритетную работу для выполнения высокоприоритетной заявки. С другой стороны, организация Б настолько сосредоточена на максимальном использовании ресурсов (full resource utilization), что заявке из примера 2 пришлось ждать в очереди дважды по две недели. Как будет показано в главе 5, попытка добиться полного использования ресурсов (full resource utilization) ведет к появлению длинных очередей, на поддержание порядка в которых уходит гораздо больше усилий, чем они того ст*о*ят и которые в действительности только уменьшают реальное использование ресурсов. В организации А смогли избавиться от очередей путем разделения задач на низкоприоритетные, которые могут подождать, и высокоприоритетные, выполнение которых максимально ускоряется.

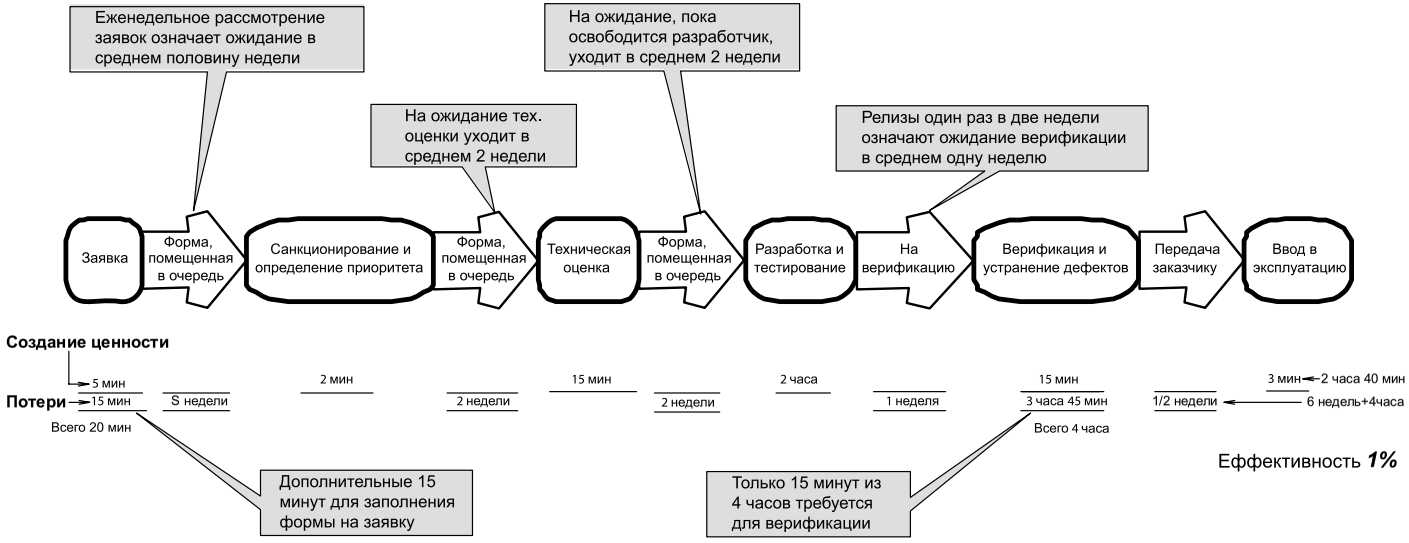


Рис. 4.4. Карта потока создания ценности для изменения небольшой высокоприоритетной функциональной возможности — Организация Б

Второй урок состоит в том, что периодические выпуски новых версий (релизы) заставляют аккумулировать существенные объемы программного обеспечения, ожидающего ввода в эксплуатацию у заказчика (undeployed). Выпуски новых версий раз в две недели в организации Б вынуждают коллектив разработчиков накапливать программный код с изменениями в течение двух недель, прежде чем могут быть выполнено интеграционное тестирование (integration testing). Такой подход — ошибка. Даже если выпуски новых версий (или релизы) производятся периодически, интеграционное тестирование должно выполняться гораздо чаще. Цель в том, что во время заключительного тестирования (final testing) дефекты выявляться не должны; они должны обнаруживаться и устраняться ранее. Если дефекты постоянно выявляются во время финального тестирования, значит, тестирование проводится слишком поздно.

Совершенно очевидно, что множество очередей, поддерживаемых в организации Б, так же, как и ведущая к потерям, скороспелая интеграция в конце, полностью съедают всякое преимущество мифического полного использования ресурсов (full resource utilization), получаемого с помощью применяемого подхода (предполагающего существование очередей и выполнение работ большими партиями).

Пример 3

В примере 3 (рис. 4.5) мы рассмотрим карту потока создания ценности для ускоренного проекта; с аналогичным проектом мы имели дело на наших курсах. Из-за давления со стороны конкурентов, компания торопилась с разработкой некоторого программного обеспечения. Поэтому процесс разработки был разделен на небольшие проекты, которые были переданы нескольким коллективам для одновременного выполнения. Аналитики, разработчики и тестировщики работали в тесном сотрудничестве, благодаря чему каждый коллектив быстро справился со своей работой и передал созданный им хорошо протестированный программный код в отдел контроля качества (quality assurance — QA) для финальной верификации. После этого освободившиеся коллективы были загружены новой работой и никто из их членов не интересовался, что было с этим созданным кодом дальше. Но, поискав, мы нашли человека, который знал, что было дальше. Отдел качества потратил два *месяца*, объединяя множество небольших проектов и устраняя несовместимости между ними[1]. Подобные потери остались более или менее незаметны, поскольку соответствующий программный код создавался в географически разделенных организациях и, казалось, не было никого ответственного за этот код после передачи его в отдел контроля качества.

Позже на наших курсах мы выполняли упражнение, где оценивали основные параметры компании (см. упражнение 4 в конце главы 8). Все четыре группы оценили дисциплину управления конфигурацией в компании на 2 или ниже по шкале 0–5. Мы никогда не видели таких низких оценок управления конфигурацией. Оказалось, что большинство разработчиков знали, что существующая система управления конфигурацией не в состоянии объединить проекты корректно, однако в этой организации параллельное выполнение работы использовалось более широко, чем в какой-либо иной знакомой нам компании. Как и следовало предполагать, на объединение проектов уходило куда больше времени, чем создание самого программного кода.

---

[1] По всей видимости, в этой организации отдел контроля качества помимо своей прямой функции (контроль качества) занимался также сборкой программного обеспечения — *Примеч. ред.*).

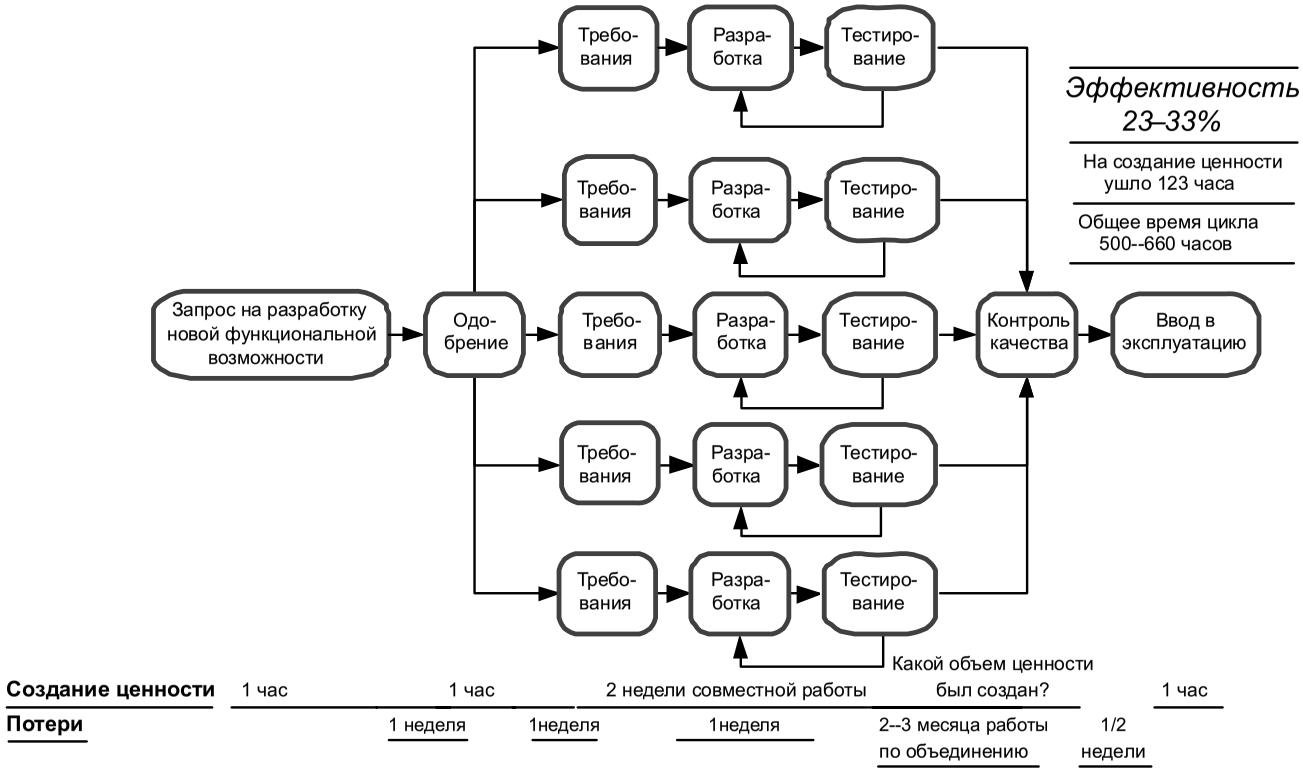


Рис. 4.5. Карта потока создания ценности ускоренного проекта, разделенного на множество более мелких проектов, которые выполнялись параллельно

В этом случае ответственный за поток создания ценности присутствовал на курсах, и когда мы создавали карту потока создания ценности, он подал идею, как можно улучшить положение вещей. Поскольку параллельное выполнение работы порождало так много проблем, следовало отказаться от него в пользу использования средств непрерывной интеграции (continuous integration tools), которые уже существовали. Он предполагал, что два-три месяца интеграционного тестирования (integration testing) могут быть сокращены до одного-двух дней.

Пример 4

Очень часто на наших курсах мы видим карту потока создания ценности, аналогичную представленной в примере 4 (рис. 4.6). На этой карте заявка наконец попадает (после двух месяцев анализа и ожидания) в комиссию по рассмотрению (которая собирается ежемесячно) только для того, чтобы быть дважды отвергнутой, прежде чем наконец получить одобрение. Такое положение вещей достаточно типично для организаций, где существуют длинные очереди ожидающей выполнения работы, поэтому мы обычно спрашиваем, как длинны в действительности эти очереди. В конце концов, каждый проект в очереди был оценен, и цифры суммировать несложно. Обычно обнаруживается, что очереди невыполненной еще работы могут быть длиной в *годы*. В подобных случаях мы неизменно выдвигаем идею, что б*о*льшая часть такой работы должна быть исключена из очереди, и размер этих очередей должен быть ограничен объемом работы, которую данная организация могла бы выполнить в ближайшем будущем. В конце концов, не существует опасности, что работа закончится. Создание очередей работы на несколько лет вперед не служит какой-либо иной цели, кроме растранжиривания времени тех служащих этих организаций, которые рассматривают заявки, и поддержания неоправданных ожиданий со стороны заказчиков.

В примере 4 детализированные требования (к заказываемому программному продукту) не формулируются, пока проект не будет одобрен, однако, когда эти требования определены и одобрены, придется еще два месяца ожидать, прежде чем будет назначен коллектив разработчиков для выполнения данного проекта. Эти разработчики не будут выделены для работы только над этим одним проектом, поэтому потребуется шесть месяцев, чтобы создать программный код (на создание которого в действительности требуется меньше двух месяцев). В дополнение ко всему, тестирование не начинается до тех пор, пока код не будет создан полностью, поэтому имеют место три цикла тестирования и устранения дефектов. После завершения тестирования код должен ожидать ближайшего релиза, которые осуществляются каждые шесть месяцев, что делает среднее время ожидания этого равным трем месяцам. Когда протестированные модули наконец объединяются, требуется еще два месяца, чтобы устранить все дефекты, проявляющиеся при взаимодействии этих модулей, поскольку в организации не практикуются непрерывная интеграция (continuous integration) и непрерывное тестирование, по мере того как новые функциональные возможности добавляются в релиз.

Наконец, программное обеспечение готово к демонстрации пользователям, через два года (без нескольких месяцев) после того, как оно было заказано. Не удивительно, что многие требования к нему изменились. В действительности ко времени, когда программное обеспечение готово к релизу, около четверти высокоуровневых требований к нему стали неактуальны и около половины требований, сформулированных перед началом проекта, изменились.

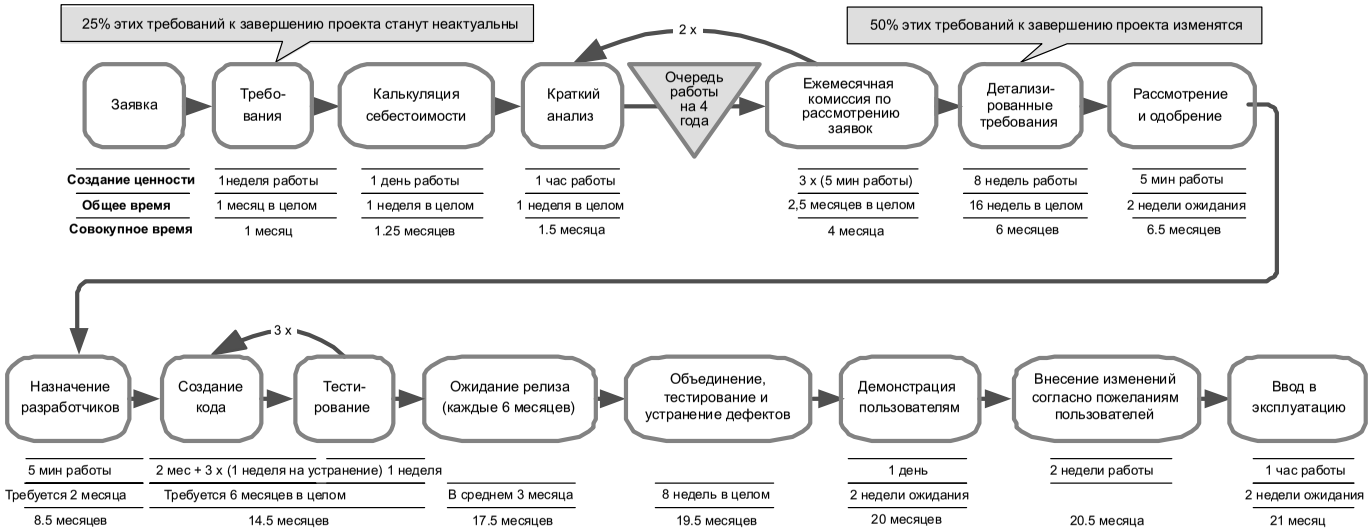


Рис. 4.6. Карта потока создания ценности для проекта средней трудоемкости в перегруженной работой организации

Для решения проблемы люди, занимавшиеся разработкой и тестированием, оставляют свою текущую работу для изменения программного кода, чтобы он делал то, чего в данный момент хотят заказчики, на что обычно уходит всего пара недель. Однако, когда сделано все, что необходимо, сотрудники отдела ввода в эксплуатацию не готовы "развернуть" вновь созданное ПО в среде заказчика, поэтому уходит еще около двух недель, прежде чем созданное программное обеспечение будет введено в эксплуатацию. Со времени подачи заявки прошел двадцать один месяц, из которых на создание потребительской ценности было потрачено не более четырех, что делает эффективность цикла процесса (process cycle efficiency —PCE) равной 19%. (При более точном подсчете эффективность цикла процесса окажется еще меньше.)

К сожалению, подобные карты потока создания ценности приходится наблюдать все время. Часто приходится слышать, что организация перегружена работой, однако, если присмотреться к соответствующей карте потока создания ценности, можно заметить широкие возможности увеличить производительность этой организации, просто устранив непроизводительные расходы. В описанном примере можно увидеть, что изменились 50% детализированных требований к создаваемому программному обеспечению (что вызывает необходимость переделывать уже сделанную работу). Это признак, что требования были сформулированы слишком рано. Требования должны предоставляться понемногу (небольшими порциями) и гораздо ближе к моменту, когда они начнут воплощаться в программный код, и желательно в форме выполняемых тестов (executable tests). В этом примере также можно заметить, что имеет место слишком позднее тестирование (и устранение дефектов — как на этапе разработки, так и интеграции программного кода). Код должен тестироваться немедленно, а затем постоянно находиться под тестовой нагрузкой (plugged into a test harness), чтобы гарантировать, что дефекты в нем не появились по мере внесения изменений в другие части программного кода.

Наконец, можно заметить, что порядок, когда релизы осуществляются каждые шесть месяцев, который был призван консолидировать усилия и уменьшить непроизводительные расходы, в действительности служит увеличению этих расходов. Код должен ожидать в среднем три месяца после создания, прежде чем попасть к заказчикам, в течение которых он может стать неактуальным, или не синхронизированным с добавленными позже в релиз новыми функциональными возможностями. Тем временем разработчики заняты другими проектами, поэтому, когда им приходится устранять возникшие проблемы, требуется некоторое время, чтобы вникнуть в программный код. Две недели перед вводом в эксплуатацию у заказчика (deployment) также могут быть сокращены, если люди из соответствующего подразделения будут подключены к процессу раньше.

Итоги

Карта потока создания ценности представляет собой график (с указанием основных этапов) осуществления проекта от концепции до выпуска готового продукта (или от запроса до ввода готового программного кода в эксплуатацию в среде заказчика). Подобные карты должны отображать среднее время для каждого типичного этапа в процессе. После создания карты первое, на что следует обратить внимание, — это задержки и случаи переделок уже выполненной работы. Если что-то приходится переделывать, это указывает, что имеют место проблемы с синхронизацией — например, детализированные требования были сформулированы слишком рано или тесты были разработаны и тестирование было проведено слишком поздно.

Задержки обычно являются следствиями долгих очередей, существование которых свидетельствует, что данная организация перегружена работой. Как будет показано в главе 5, система функционирует куда быстрее, если длина очереди, ожидающей выполнения работы, ограничена способностью организации выполнить ее. Задержка также может свидетельствовать, что достигнут этап, когда работа должна быть передана в другую организацию. Задержка, например, имеет место, когда работа передается в организацию, которая пока не готова приступить к ее выполнению. Лучшее средство против задержек — привлечь людей из организации, где эта работа должна выполняться (чтобы они вникли в суть дела заранее), до того как наступит время передать работу.

Существуют и другие источники потерь, которые выявляются картами потока создания ценности: неудачная синхронизация, запутанный процесс одобрения, недостаток участия ответственных за ввод в эксплуатацию и поддержку созданного ПО. Однако эти явления станут заметны только, если отобразить на карте весь процесс — от концепции до получения оплаты за выполненный заказ.

Перспективные карты потока создания ценности

Каждая организация, с которой мы имели дело, располагала бо́льшим числом заказов, чем в состоянии была выполнить. Однако, как правило, обнаруживалось, что можно выполнять больше работы, быстрее и с более высоким качеством, просто ликвидировав обширные потери, которые выявляют большинство карт потока создания ценности. К окончанию наших курсов мы просили каждую группу создать перспективную карту потока создания ценности, используя принципы бережливости для усовершенствования их процесса. Мы ставили условие, что карты должны отображать процесс, который организация в состоянии реализовать в течение не более чем трех-шести месяцев. Независимо от того, как выглядела карта потока создания ценности, отображающая текущее положение в этих организациях, перспективные карты неизменно показывали улучшение в эффективности цикла процесса (process cycle efficiency) и общей продолжительности цикла (overall cycle time) на от 50 до 500%.

Карты потока создания ценности для текущего процесса окажутся сравнительно бесполезны, если их не использовать для выявления и устранения потерь. Создание перспективных карт потока создания ценности — хороший способ составить план ликвидации наибольших непроизводительных расходов. Однако перспективная карта не должна быть идеальной картой: она должна указывать путь к немедленному улучшению. Выберите наибольшие задержки, или самые длинные очереди, или худшие случаи повторного выполнения уже сделанной работы и обратитесь к этому в первую очередь. Создайте новую карту, которая бы показывала, каких реальных успехов ваша организация могла бы достичь через три-шесть месяцев при одном-трех ключевых изменениях. После внесения этих изменений наступит время создания новой карты, отображающей текущее положение дел, которая бы помогла выявить и устранить следующие наиболее заметные недостатки, и т.д.

Попробуйте это

1. Сколько вашей организации потребуется чтобы выполнить (и ввести в эксплуатацию в среде заказчика) изменение одной строки в программном коде? Ваша организация производит изменения ПО с такой скоростью постоянно?

2. Оценка сложности: оцените сложность продуктов, которые производит ваша организация, по пятибалльной шкале (1–5), где оценка 1 соответствует минимальной сложности (как у компании Inditex), а 5 — когда создаваемые продукты имеют максимальную сложность. Какая единственная мера смогла бы уменьшить эту оценку на один балл? (Примечание: более сложный процесс одобрения заявки нельзя признать хорошей мерой, позволяющей снизить сложность. Подход компании Inditex к рассмотрению и одобрению заявок был таким же минималистским, как и сама созданная ими система.)
3. Семь непроизводительных расходов: выберите один из этих расходов, который наносит наибольший ущерб вашей организации:

   а) частично выполненная работа;

   б) избыточные функциональные возможности;

   в) повторное приобретение знания;

   г) передача работы;

   д) переключение между задачами;

   е) задержки;

   ж) дефекты.

   Какая единственная мера смогла бы ощутимо уменьшить этот непроизводительный расход?
4. Еженедельно устраивайте “санитарный час” по сокращению потерь в вашей организации. В первую неделю начните этот час с обсуждения семи непроизводительных расходов. Сначала выясните, в чем в действительности состоят потери в ваших условиях. Затем создайте список примеров соответствующих потерь в вашем процессе разработки. Выберите в этом списке пять пунктов и в течение следующей недели определите, сколько потерь исходит от каждого из них. На следующей еженедельной встрече обсудите ваши оценки и определите, какой из этих расходов наносит вашей организации наибольший ущерб, и составьте план его сокращения (или ликвидации). Выполните этот план, а затем в течение следующей недели оцените полученные результаты. На следующей еженедельной встрече перейдите к следующему непроизводительному расходу и повторите процесс.
5. Карта потока создания ценности: если вы решили создать подобную карту для процесса, используемого в вашей организации, выполните это в три этапа.
   - Соберите всех, кто согласен потратить час (не больше) на создание этой карты, а затем полчаса на ее обсуждение.
   - Потратьте полдня на то, чтобы собрать информацию, которой недоставало во время первого упражнения, а также попытайтесь привлечь нескольких ключевых людей, отсутствие которых на первом собрании было заметно. На этот раз потратьте два часа на создание карты, а затем час на обсуждение выводов, к которым данная карта заставила прийти.
   - Достаточно ли вы выяснили во время первых двух экспериментов, чтобы знать, что представляет собой ваш наибольший непроизводительный расход? Если да, быстро создайте одну-две перспективных карты потока создания ценности, а затем приступите к ликвидации (или хотя бы сокращению) вашего наибольшего непроизводительного расхода.

Глава 5

Скорость

Поставлять быстро

Скорость поставок свидетельствует об отсутствии непроизводительных расходов.

Если вы усердно работаете над тем, чтобы ликвидировать потери, то увеличиваете процент времени, потраченного на увеличение потребительской ценности во время каждого производственного цикла (process cycle). И при этом вы будете доставлять (готовый продукт потребителю) быстрее — возможно, гораздо быстрее.

Бережливый подход работает в производстве. Он работает в логистике. Он работает в офисной деятельности. Следующий раздел познакомит вас с хорошим примером, как это работает в создании программного обеспечения.

PatientKeeper

Пять лет назад в здравоохранении появилось потрясающее приложение для карманного компьютера (PDA), предоставляющее докторам доступ к информации о пациенте. Сегодня[1] компания PatientKeeper (создавшая приложение, о котором идет речь), похоже, выигрывает соревнование за доминирование на этом быстро развивающемся рынке. Она обезоруживает конкурентов тем, что на рынке появляются (чуть ли не еженедельно) новые продукты и новые версии продуктов этой компании. Около шестидесяти технических специалистов производят больше программного обеспечения, чем многие другие организации, в несколько раз крупнее, и программное обеспечение этой компании сертифицировано для работы с информацией, от которой может зависеть жизнь. И не наблюдается ни малейших признаков, что возрастающая сложность деятельности этой компании приводит к замедлению ее функционирования, несмотря на то, что они продают свои продукты множеству очень крупных организаций здравоохранения, поддерживая при этом несколько платформ.

Основа стратегии, позволяющая компании PatientKeeper держаться впереди, заключается в способности поставлять свои продукты невероятно быстро. За последние три года PatientKeeper осуществляла по 45 релизов программного обеспечения ежегодно, предназначенных для крупных организаций здравоохранения, используя при этом одновременные параллельные итерации (известные как Sprint) (рис. 5.1). Каждая итерация завершается очередным релизом, каждый из которых был осуществлен своевременно.

[1] В 2006 году, когда вышла эта книга.

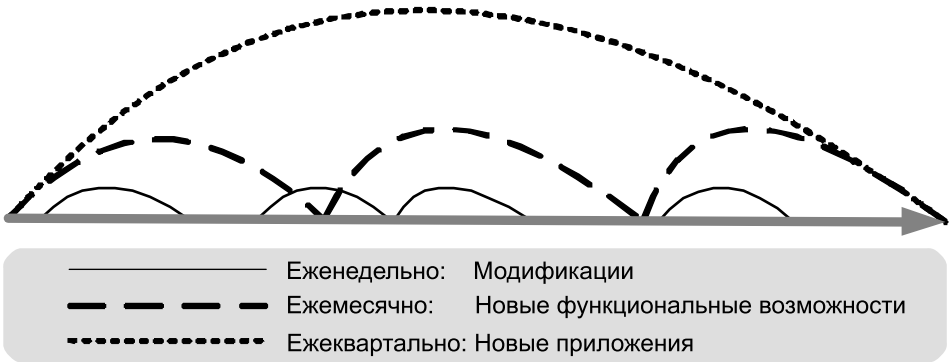


Рис. 5.1. Одновременные параллельные итерации, реализуемые несколькими коллективами разработчиков[2]

Технический директор (CTO) компании PatientKeeper, Джефф Сазерленд (Jeff Sutherland), объясняет, как это работает[3] (см. предисловие Джеффа Сазерленда).

- Результатом всех итераций Sprint является релиз (или выпуск) программного обеспечения, который тиражируется и поставляется потребителям.
- Отдел контроля качества (QA department) приступает к тестированию сразу же, как только разработчики предоставят первый программный код. Разработчики могут независимо выполнять процесс сборки программного обеспечения и направлять его на любой сервер отдела контроля качества.
- Посреди итерации Sprint группа, отвечающая за установку и наладку нового ПО в среде заказчика (install team), "разворачивает" версию Release Candidate 1 у заказчика для тестирования. Теперь вместе с отделом контроля качества тестированием занимается также заказчик.
- По мере того как заказчики высказывают свои замечания, они направляются (в качестве обратной связи) разработчикам, так же, как и замечания от отдела контроля качества.
- Как правило, требуются две или три версии Release Candidate, чтобы завершить все задачи по разработке данного ПО, а также удовлетворить все претензии со стороны отдела контроля качества и заказчика. Отдел контроля качества выполняет регрессивное тестирование всей системы.
- К концу итерации Sprint все потребители удовлетворены. Результат может представлять собой систему, охватывающую несколько больниц, с сотнями докторов (пользователей PDA) и тысячами пользователей веб.
- Завершение означает, что потребители не имеют существенных критических замечаний. Обратите внимание, что здесь имеет место тесное взаимодействие между

[2] Из отчета (автор Jeff Sutherland) Future of Scrum: Parallel Pipelining of Sprints in Complex Projects, Research Report, Agile 2005. Использовано с разрешения.

[3] Опубликовано в `Scrumdevelopment@yahoogroups.com` (September 25, 2005). Использовано с разрешения.

заказчиками и группами, инсталлирующими программное обеспечение в их среде, а также между разработчиками и отделом контроля качества.

В компании PatientKeeper за решение, чего именно хотят заказчики, а также подробное определение, какими именно функциональными возможностями должен обладать будущий программный продукт, ответственны менеджеры продуктов. Это означает, например, что менеджер продукта определяет (с помощью прототипов), какими свойствами должен обладать интерфейс пользователя, и принимает окончательные (и детализированные) решения, как данный интерфейс должен работать. При этом как заявки на элементы создаваемого программного обеспечения, так и выявленные дефекты заносятся в автоматизированный журнал предстоящей работы (backlog). Менеджер продукта несет ответственность за выбор работ в этом журнале для включения в очередной релиз.

Для каждого такого релиза (для которого отобрана работа из журнала предстоящей работы) назначается коллектив разработчиков. Разработчик берет одну из назначенных для релиза работ, делит ее на задачи и делает оценку каждой задачи. Эти оценки вводятся в систему и автоматически ассоциируются с соответствующей работой в журнале. В конце каждого дня каждому разработчику требуется не более минуты, чтобы ввести в следящую систему данные о времени, потраченном на каждую задачу, а также предполагаемый процент ее завершения. С этими данными в следящей системе кто угодно в компании может получить достоверную информацию о том, сколько времени еще требуется, чтобы завершить тот или иной релиз.

Является правилом, что все релизы должны осуществляться вовремя, поэтому, если следящая система покажет, что в каком-то из них имеется слишком много невыполненной работы, что-то может быть удалено из этого релиза, чтобы объем работы соответствовал возможностям коллектива разработчиков. Поскольку следящая система предоставляет точные и актуальные данные о времени, необходимом для завершения любого набора задач из журнала предстоящей работы, легко могут быть определены подходящие варианты перераспределения работы, и может быть принято эффективное решение. Приоритеты определяются на еженедельных совещаниях, на которых присутствуют все лица, принимающие решения, включая генерального директора (CEO). Менеджеры продуктов выполняют принятые решения, изменяя набор работ, назначенных в релиз из журнала ожидающей своей очереди работы, а возникающие при этом проблемы коллективы разработчиков решают сами.

Компания PatientKeeper функционирует быстро: она может создать любое приложение в течение 90 дней или раньше. Никого, понимающего смысл бережливого подхода, не удивляет, что компания должна обеспечивать высшее качество, чтобы стали возможны такие быстрые поставки. Джефф Сазерленд отмечает[4], что такое быстрое время цикла:

- чрезвычайно интенсифицирует обучение;
- заставляет максимально упростить процесс инсталляции создаваемого ПО, потому что вы не сможете установить 45 релизов за год, если этот процесс не будет простым;
- улучшает процесс модернизации (upgrade) программного обеспечения, поскольку в этом случае имеет место постоянный поток новых версий, использование которых является обязательным.

---

[4] Опубликовано в `Scrumdevelopment@yahoogroups.com` (November 21, 2004, message 5439). Использовано с разрешения.

Хотя такая организация работы кажется естественной в PatientKeeper, она крайне удивляет посторонних. Один из факторов, делающих это возможным, состоит в том, что все в компании работают вместе в духе доверия, уважения и постоянного стремления к совершенствованию. Коллективы, создающие программное обеспечение, включают менеджеров продуктов, разработчиков, а также специалистов из отделов проверки качества и поддержки.

Время — универсальная валюта

Все, недостатки, которые возникают в процессе создания ПО, в конце концов проявляются в виде задержек. Речь идет о дефектах, излишней сложности, низкой производительности, сопротивлении персонала необходимым переменам и создании ненужных функциональных возможностей. Перегруженность работой приводит к длинным очередям, которые также причина задержек.

Время цикла (cycle time) — это универсальный показатель бережливости, который заставляет нас насторожиться, когда что-то не в порядке. Жизнеспособный процесс разработки преобразует выявленный потребительский спрос в доставленную потребительскую ценность (delivered customer value) в постоянном ритме, который определяется временем цикла. Именно этот показатель заставляет организацию функционировать, поддерживать поток создания ценности и внедрять качество в продукт. Именно время цикла позволяет оценить потенциал организации.

Предприятие, где внедрены бережливые методы, организовано таким образом, что процесс всегда *свободен*, когда появляется работа, и всегда *способен* справиться с этой работой[5]. Если процесс *свободен* и *способен*, в нем отсутствует явление, известное как "проталкивание работы" (expediting). Это явление может возникать, когда поступает работа, которая начинает выполняться параллельно с другими работами, однако кто-то думает, что эта работа настолько важна, что ему следует лично заняться ее "проталкиванием". Если требования от "толкача" поступают регулярно, значит, ваш процесс не лишен недостатков. Либо он не *свободен*, когда появляется работа, либо он не *способен* выполнить данную работу.

Что это значит для разработки программного обеспечения? Предположим, имеется отдел поддержки программного обеспечения, который гарантирует реакцию в течение двух часов для обращений высокой срочности, одного дня при обычной проблеме и двух недель для каких-то низкоприоритетных изменений. Когда имеет место срочное обращение, отдел может обещать выполнить соответствующую работу не более чем через два часа, возможно, гораздо скорее. Когда поступают иные запросы, отдел также может обещать надежную реакцию в течение времени, определяемого приоритетом запроса. Поскольку отдел работает в надежном, постоянно повторяющемся ритме, установился предсказуемый предел работы, которая может быть выполнена. Когда достигается этот предел, низкоприоритетные обращения должны отвергаться либо подключаться дополнительные мощности.

Организации, где внедрены бережливые методы, оценивают свою эффективность, измеряя время цикла основного технологического процесса от начала и до конца. Карты потока создания ценности в главе 4 отображают процессы (от начала и до конца), которые начинаются и заканчиваются заказчиком. На этих картах изображены этапы процесса и указано время для каждого этапа, а также суммарное время — от концепции до поставки (для продуктов) или от заказа до ввода программного обеспечения в эксплуата-

[5] См. www.lean.org.

цию (для заказанного ПО). Высокая эффективность имеет место, когда этот цикл завершается с минимальными непроизводительными затратами времени и усилий.

Наилучший способ определить эффективность процесса разработки программного обеспечения — измерить среднее время цикла (от начала и до конца) процесса. В частности, сколько времени (в среднем) потребуется, чтобы неоднократно и надежно пройти все этапы процесса от концепции до оплаты (или от заказа до ввода программного обеспечения в эксплуатацию)? Идея не в том, чтобы измерить один раз это время цикла; определить следует среднее время, требующееся вашей организации на весь процесс, — от выявления потребительского спроса до его удовлетворения.

Однако на карте потока создания ценности время цикла постоянно изменяется. Что в этом случае означает понятие "среднее"?

Если целью вашей работы является создание продукта, лучший подход заключается в том, чтобы установить постоянный цикл релизов — каждые две недели, каждые шесть недель, каждые шесть месяцев — то, что наиболее подходит для вас. Сделайте его настолько коротким периодом, насколько вам это удастся. Затем определите, как много работы может быть выполнено в одном релизе, и не пытайтесь это количество увеличить. Никогда не откладывайте релиз из-за того, что что-то не готово — удалите из него что-то, но при этом не пытайтесь соответственно увеличить количество работы в следующем релизе. Очень скоро вы будете знать, сколько работы приемлемо для одного релиза.

Было бы полезно разделить работу внутри цикла на более мелкие элементы таким же образом, как это делается в компании PatientKeeper. В главе 8 мы обсудим, как можно разделить работу на "истории" (stories), для выполнения каждой из которых требуется от одного до трех дней. С течением времени коллектив разработчиков будет выполнять эти "истории" с надежной быстротой.

Если релизы бессмысленны в вашей ситуации, попытайтесь установить верхний лимит на продолжительность проекта — например, шесть месяцев — но не более двенадцати. Уже в течение многих лет компании Wal-Mart и Dell ограничивают свои проекты в области информационных технологий девятью месяцами. Без сомнения, и вы сможете то же.

Определив верхний лимит продолжительности проекта, сгруппируйте ваши проекты в две или три категории — с высокой и низкой сложностью. Затем попытайтесь определить "уровень обслуживания" (service level) для каждой категории.

Наши проекты слишком трудоемки и уникальны, чтобы говорить о циклах. На работу над ними уходят годы.

Независимо, насколько трудоемок проект, работа всегда осуществляется небольшими этапами. Мы привыкли воспринимать эти этапы как непрерывную последовательность и создавать большие объемы частично выполненной работы на каждом этапе. Подумайте, как иначе можно разделить эту большую работу. Разделите ее на "порции" (или инкременты), каждая из которых представляет собой минимально полезный набор функциональных возможностей. В главе 8 будет показано, как упрощенный вариант подводной лодки Polaris в рамках девятилетней программы создания был спущен на воду через три года после начала программы. Без сомнения, найдется вариант (упрощенный) и вашей системы, который может быть продемонстрирован по истечении трети периода разработки.

Далее, попытайтесь установить более мелкие циклы — от шести недель до трех месяцев, — где все готовое к этому сроку уже интегрировано, каждую существующую функциональную возможность можно продемонстрировать и оценить и соответствующие решения приняты. После этого вы будете иметь циклы продолжительностью не более трех месяцев, и целью является надежно, в каждом цикле, поставлять повторяющиеся объемы работающего программного обеспечения.

Теория очередей

Эта теория изучает все, связанное с очередями. А в разработке программного обеспечение очереди, несомненно, имеют место — имеются списки заявок от заказчиков и списки дефектов, ожидающих устранения. Теория очередей может многое предложить, что помогло бы управлять этими списками.

Закон Литтла

Этот закон гласит, что в устойчивой системе среднее время, затрачиваемое процессом на одно изделие, равно числу изделий, находящихся в производстве, деленному на среднюю производительность системы (рис. 5.2).

$$\text{Время цикла} = \frac{\text{Кол-во изделий в производстве}}{\text{Средняя производительность}}$$

Рис. 5.2. Закон Литтла

В предыдущем разделе отмечалось, что целью бережливой разработки является сокращение времени цикла. Данное уравнение дает нам ясное представление, как этого достичь. Один из способов сократить время цикла состоит в том, чтобы ускорить процесс — или увеличить производительность. Однако это обычно связано с дополнительными расходами. Если лишних денег у вас нет, иной способ уменьшить время цикла — сократить число объектов, которые одновременно обрабатываются процессом. Для этого требуются значительные умственные способности, однако такой подход не связан с большими дополнительными расходами.[6]

Вариация и загрузка

Закон Литтла применим к устойчивым системам, однако существует пара факторов, которые делают систему неустойчивой. Первый из них известен как вариация (или изменчивость — variation). Хорошей мерой против вариации является уменьшение партий изделий, проходящих через систему. Предположим, вам нужно интегрировать в систему некоторый программный код. Если этот код создавался шесть недель, можно быть уверенным, что он содержит множество проблем. Однако если код создавался всего 60 минут, разнообразие возможностей, что что-то может пойти не так, ограничена. В крупных проектах вариация производственных графиков (schedule variation) будет очень существенна. Однако в малых проектах подобная вариация будет неизмеримо меньше.

Высокая загрузка оборудования (или максимальное использование ресурсов) — еще один фактор, делающий систему неустойчивой. Это особенно хорошо понимают те, кому приходилось застревать в автомобильных пробках. Как только загрузка (или использование) дороги достигает 80%, скорость движения по ней начинает снижаться. Если сюда добавить еще несколько автомобилей, очень скоро вы будете двигаться с черепашьей скоростью. Еще пример:

---

[6] George M., Wilson S. *Conquering Complexity in Your Business: How Wal-Mart, Toyota, and Other Top Companies Are Breaking Through the Ceiling on Profits and Growth*, McGraw-Hill, *2004*, p. 37.

когда загрузка серверов в часы пик достигает 80%, с этого момента время ответа начинает увеличиваться и в подобных случаях обычно подключают больше серверов.

Поскольку компания Google была создана группой ученых, исследующих проблемы поиска данных, неудивительно, что серверная структура этой компании отражает глубокое понимание теории очередей. Прежде всего, данные хранятся небольшими порциями. Вместо огромных серверов с большими объемами данных на каждом из них, Google содержит тысячи и тысячи небольших и недорогих серверов, рассеянных по всему миру, объединенных в очень сложную сеть. Никто не ожидает от этих серверов стопроцентной надежности; разного рода сбои ожидаются и когда возникают, выявляются немедленно. Не беда, если сервер неисправен, поскольку все данные разделены на мелкие порции и содержатся во множестве мест. Поэтому, когда серверы выходят из строя и автоматически отключаются от сети, содержавшиеся на них данные берутся в некотором хранилище и их копия быстро помещается на работающий сервер. При этом пользователи даже не замечают, что что-то случилось; они по-прежнему получают почти мгновенные ответы.

Если вас удивляет, почему в Google полагают необходимым, чтобы 20% рабочего времени ученых и инженеров, работающих в этой компании, было зарезервировано, присмотритесь к графику на рис. 5.3. Здесь показано, что время цикла начинает увеличиваться при загрузке ресурсов чуть выше 80% и что этот эффект только усиливается при использовании крупных партий работы (и при высоком уровне вариации). Представьте себе группу ученых, которые профессионально занимаются теорией очередей. Предположим, они организовали компанию и управляют ею. Доставка на рынок новых продуктов для компании считается задачей высшего приоритета. Для этих ученых создание в их компании 20% избыточных мощностей будет наиболее логичным решением. Поэтому странно, что обозреватели аплодируют Google за избыточные серверы, но не понимают концепцию избыточных мощностей в организации, занятой разработкой.

Многие администраторы были бы уволены за попытки добиться максимальной загрузки для каждого сервера, поскольку общепризнанно, что высокая загрузка замедляет работу сервера. Почему же тогда занятые в разработке менеджеры, когда видят отчет, в котором говорится о 90%-ном использовании рабочего времени в прошлом месяце, тут же говорят: “Глядите! Здесь есть время для еще одного проекта!” Определенно эти менеджеры не собираются применять теорию очередей к движению в их организации, где явно назревает пробка.

Законов математики не избежать, даже в проектной организации. Если вы нацелены на повышение загрузки мощностей, не следует удивляться, если производительность начнет снижаться. Если вы полагаете, что крупные партии — это путь к повышению загрузки, вы еще больше все замедлите и при этом также снизите загрузку. Однако если работа будет выполняться небольшими партиями и приоритет будет отдан организации бесперебойного потока, в этом случае загрузка мощностей будет очень высокой, — но загрузка никогда не должна быть главной целью!

Сокращение времени цикла

Давайте согласимся, хотя бы на данный момент, что нашей целью является сокращение среднего времени цикла (average cycle time) от концепции до оплаты за выполненную работу или от поступления заказа от потребителя до введенного в эксплуатацию программного обеспечения. Как достичь этой цели? Теория очередей указывает несколько способов сокращения времени цикла.

1. Добиться ритмичности поступления работы.
2. Свести к минимуму количество одновременно выполняющейся работы.
3. Минимизировать размеры объектов, одновременно находящихся в производстве.
4. Создать постоянный ритм работы.
5. Объем работы не должен превышать возможности организации.
6. Использование планирования с “вытягиванием”.

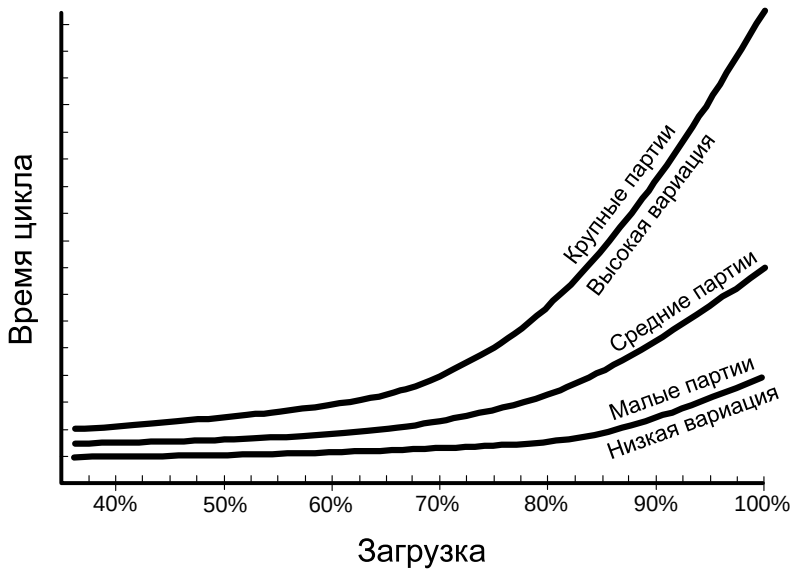


Рис. 5.3. Теория очередей применима к процессу разработки так же, как к уличному движению

Добиться ритмичности поступления работы

В основе каждого бережливого процесса лежит постоянство объема работы, которую предстоит сделать. Например, фабрике “спущен” план производить 10000 единиц продукции ежемесячно; чтобы выполнить план, фабрика должна выпускать одну единицу ежеминутно. После того как это определено, фабрика начинает работать в постоянным ритме, производя единицу продукции каждую минуту.

Со стороны подразделений, ответственных за утверждения бюджета и одобрения проекта, вероятно, исходит “наибольшее сопротивление”, когда дело касается организации постоянного ритма разработки. Заявки держатся в очередях месяцами, а крупные проекты могут ждать ежегодного бюджетного цикла для одобрения. Некоторые, вероятно, думают, что, рассматривая все предложения одновременно, организация сможет сделать лучший выбор, как распорядиться своим бюджетом. Однако подобная практика создает долгие очереди ожидающей выполнения работы, а если всю работу запустить в производство одновременно, это создаст хаос в проектной организации. Более того, это значит, что решения принимаются не синхронно с возникновением спроса, и к началу работы над проектом реальные потребности заказчика, вероятно, претерпят существенные из-

менения. Привязывание одобрения проекта к бюджетному циклу, как правило, лишено какой-либо необходимости и обычно является нереалистичным кроме случаев, когда скорость выполнения проектов не важна.

Вся наша работа поступает в первом полугодии

Недавно я обедала с менеджером высокого ранга очень крупной организации, принадлежащей отрасли информационных технологий. В этой организации практиковалось постоянное перемещение разработчиков в иные проекты, и я предложила, что, возможно, им целесообразно назначать для различных компаний-заказчиков постоянные коллективы разработчиков с тем, чтобы они лучше узнали особенности бизнеса своих заказчиков.

"Мы не можем пойти на это, — ответил он — все руководители бизнеса, с которыми мы имеем дело, хотят, чтобы их заказы выполнялись в первой половине года, с января по июнь, поэтому мы стараемся всю несрочную работу, в которой они не заинтересованы, отложить на второе полугодие."

"Почему им нужно, чтобы вся работа была выполнена в первой половине года? — спросила я — ваш бизнес сезонный?"

"Нет", — ответил он. А затем, немного подумав, добавил: "Они получают свои бюджеты в конце декабря и поэтому, как мне кажется, им хочется, чтобы вся работа была выполнена после этого максимально быстро."

"Я понимаю, вы хотите быть предупредительным по отношению к руководителям компаний-заказчиков, — сказала я, — но вам не следует позволять им такое. Вы должны разъяснить, что подобный график работы не может не влиять на вашу способность выполнять их заказы. Что если назначить постоянный коллектив разработчиков для обслуживания каждой компании, предоставив руководителям соответствующих компаний самим решать, как этому коллективу следует расходовать свое рабочее время в течение года?"

"Это слишком радиальное решение, — он усмехнулся — это потребует очень больших изменений..." И пообещал об этом подумать. (Мэри Поппендик)

Создание очереди работы в начале процесса разработки может восприниматься как удачное решение, обеспечивающее равномерную загрузку организации-разработчика. Однако такие очереди должны быть не больше, чем необходимо, чтобы выровнять неравномерность поступления работы. А там, где работа всегда поступает равномерно, длинные очереди абсолютно излишни.

Визит к доктору

В Соединенных Штатах, на список пациентов, записавшихся на прием к доктору, существует ограничение. Во многих клиниках эти списки составлены на два месяца вперед — им не позволяется быть длиннее, однако, похоже, и короче они тоже не становятся. С точки зрения теории очередей это можно рассматривать как устойчивую систему.

В одной из клиник Миннесоты познакомились с идеями бережливости и решили посмотреть, что произойдет, если сократить время ожидания. На первых порах большинство докторов еженедельно работали дополнительно половину дня, в то время как планировщики обеспечивали, чтобы загрузка докторов была постоянной. В течение полугода клиника сократила время ожидания для пациентов до двух дней. В клинике обнаружили, что доктора принимают те же количества пациентов с тем же набором проблем. Некоторые доктора были удивлены, когда оказалось, что нет никакой необходимости заниматься управлением очередью на шестьдесят дней вперед, чтобы обеспечить им постоянную загрузку; в действительности они не заметили каких-либо изменений загрузки.

А с точки зрения пациента произошло невероятное изменение — вдруг появилась возможность позвонить и получить назначение к врачу в течение одного-двух дней. Для пациентов это было поистине "бережливое решение". (Мэри Поппендик)

Свести к минимуму количество одновременно выполняющейся работы

В производстве давно заметили, что излишние запасы (материалов, деталей и т.п.) мешают производственным процессам. Однако в разработке программного обеспечения эта истина почему-то пока не стала очевидной. Мы часто стараемся увеличить периоды времени между релизами и позволяем программному коду накапливаться, прежде чем очередной релиз попадет к заказчикам. Существуют процедуры одобрения проектов, которые служат накоплению работы до объемов, которые исключают всякую надежду, что она будет когда-либо выполнена. Широко используются последовательные процессы, где количество одновременно выполняющейся работы непомерно велико. Создаются и поддерживаются длинные списки дефектов; иногда мы даже испытываем гордость от того, как много дефектов было выявлено. Вся эта частично выполненная работа в разработке программного обеспечения то же, что запасы (материалов, деталей и т.п.) в производстве — она замедляет ритм работы, скрывает проблемы с качеством и очень быстро ведет к тому, что уже созданный программный код становится устаревшим.

Один из наиболее очевидных факторов замедления процесса разработки — длинный список заявок от заказчиков, для которых у разработчиков пока не нашлось времени. Каждая известная нам (авторам) организация, работающая в сфере разработки программного обеспечения, имеет очередь работы больше, чем она способна "переварить", однако мудрость в этом случае диктует не принимать работу, для которой вряд ли найдется время. Почему список заявок должен быть кратким? С точки зрения заказчика как только что-то было принято к исполнению, заявка считается попавшей в процесс разработки, и начинается отсчет времени ответа (или времени реакции). Ожидающие одобрения проекты, содержащиеся в очереди работы, отнимают рабочее время каждый раз, когда они оцениваются, обсуждаются на совещаниях или пересматриваются их приоритеты. Очереди не начатой работы часто служат в качестве буферов, изолирующих разработчиков от заказчиков; они могут быть использованы для сокрытия реальности и они часто вызывают нереалистичные ожидания.

У нас такой длинный список работы, как нам его сократить?

Как правило, длинные очереди ожидающей выполнения работы являются нереалистичными и излишними. Ниже приведены несколько советов, как их изжить.

1. Начните с вопроса себе: ''Сколько в этой очереди работы, которую мы вряд ли когда-то сможем выполнить?'' Удаляйте немедленно все, что вы не собираетесь выполнять. Будьте при этом честны. И не стесняйтесь пользоваться клавишей <Delete>.
2. На сколько уменьшилась очередь после первого упражнения? Наполовину? Теперь выполните анализ Парето (Pareto) для оставшихся элементов списка. Оцените приоритет каждой работы в списке по пятибалльной шкале (от 1 до 5). Для маловажных элементов списка укажите оценку 1. Для высокоприоритетной работы укажите оценку 5. Теперь удалите из списка все, что не имеет оценок 4 и 5. (Не стесняйтесь пользоваться клавишей <Delete>!) Не волнуйтесь, если какая-то удаленная работа окажется важной; она сама напомнит вам о себе.
3. Теперь присмотритесь к оставшимся элементам списка и подсчитайте, сколько дней, месяцев или лет работы они представляют. Опираясь на эту информацию, определите, способна ли ваша организация справиться с оставшейся в списке работой в ближайшем будущем? Если нет, не следует ли добавить дополнительные мощности?
4. Если ваш список по-прежнему нереалистично длинный, вероятно, существует какая-то иная цель, которой он служит, помимо принятия решений, какую работу выполнять и какую нет. Например, длинный список может отвлекать чье-то неуместное внимание от чего-то в этом списке или служить "пристанищем" для каких-то несерьезных работ (которые со-

держатся в этом списке только для отвода чьих-то глаз). Если это так, создайте два списка. Один для внешнего использования, а второй (настоящий, который следует содержать кратким) для принятия решений.

Семь лет?

"Мы пересматриваем приоритеты элементов этого списка каждую неделю", — объяснил менеджер.

"Сколько всего заявок в этом списке?" — спросила я.

"Здесь их около 750", — сказал он.

"А сколько заказов вы выполняете в среднем, например, в месяц?" — спросила я.

"Каждый месяц приблизительно девять, — ответил он — компания уделяет большое внимание статистике."

"Ну и ну! — воскликнул кто-то — здесь работы на семь лет!"

"Семь лет! — менеджер был удивлен — я никогда не смотрел на это таким образом."

"Зачем же вы содержите все это в списке, если знаете, что никогда не приступите к этой работе?" — спросила я.

"Кое-кто у нас полагает, что мы ничего не должны терять из виду. И мы не тратим на этот список много времени еженедельно", — ответил менеджер.

"Итак, вы никогда не идете к заказчикам, чтобы сказать, что не сможете выполнить их заявку. И они, вероятно, продолжают питать какие-то надежды, — сказала я — не следует ли вам быть честнее с заказчиками?"

"Да, в прошлом мы говорили им *нет* все время и очень настойчиво. Мы тогда были маленькой компанией и я более серьезно воспринимал свои обязанности. Заказчикам, кажется, нравится честность. Возможно, нам следует начать так делать снова..." (Мэри Поппендик)

Минимизировать размеры объектов, одновременно находящихся в производстве

Объем незавершенной работы в организации зависит либо от продолжительности ее релиз-цикла (т.е. промежутка времени между двумя релизами), либо от размера партий работы (work packages). Поддерживать релиз-цикл кратким и максимальный предел партии небольшим достаточно сложно. Естественная тенденция состоит в том, чтобы растягивать релиз-цикл или продолжительность проекта, поскольку действия, связанные со сдачей работы, кажется, требуют так много труда. Однако увеличение периода между релизами является движением в направлении, противоположном бережливости. Если релизы отнимают много времени, не откладывайте их. Выясните, с чем связаны эти временны*е* потери, и постарайтесь уменьшить их. Если что-то дается с трудом, делайте это чаще, и постепенно вы научитесь делать это лучше.

Релизы осуществляются редко

После состоявшегося в компании разговора, менеджер из отдела контроля качества пришел ко мне и сказал: "Я не вижу, как мы можем последовать вашему совету. Поскольку релизы так редки, мы вынуждены помещать максимальное число функциональных возможностей в каждый из них."

"Почему бы не осуществлять рилизы чаще?" — спросила я.

"Мы не можем осуществлять рилизы чаще, поскольку очень много времени занимает процесс верификации", — последовал немедленный ответ.

"Почему же верификация занимает так много времени?" — удивилась я.

“Во время верификации обнаруживается множество проблем, нуждающихся в устранении”, — ответил он. Я уже начала видеть порочный круг.

“А нельзя ли большинство этих проблем устранять до верификации?” — спросила я. В конце концов он был менеджером из отдела контроля качества.

“Верификация должна быть независимым процессом, — ответил менеджер — если мы будет осуществлять проверку программного кода в то же время, когда разработчики создают его, это нарушит нашу независимость.”

Я была по-настоящему удивлена. Существует множество веских причин отложить заключительную верификацию. Во встроенных системах аппаратура обычно не готова до самого конца. Если по условиям заказа вы должны ввести новое ПО в эксплуатацию у заказчика, вы обычно не имеете доступа к его среде до самого последнего момента. Однако это был не тот случай. “Я не понимаю, что такое независимая верификация, — сказала я — но как это связано с продолжительными периодами между релизами? Не может ли верификация оставаться независимым процессом и при этом выполнять тестирование меньших объемов программного кода?”

“Мне необходимо обдумать это, но мы таким образом можем несколько утратить нашу независимость от разработчиков” — ответил он.

Его голос звучал неохотно. Я догадалась, что в этой организации в ближайшее время релизы вряд ли будут осуществляться чаще. (Мэри Поппендик)

ТЕПЕРЬ я понял!

“Мы проводили занятия на наших курсах, где учащиеся создавали сначала текущие, а затем перспективные карты потока создания ценности. Последняя группа представляла свою перспективную карту. Вдруг кто-то из другой группы воскликнул: “ТЕПЕРЬ я понял!”. Рассказывающий о карте остановился и повернулся посмотреть, в чем дело.

“Когда я создавал свою перспективную карту, о которой я только что рассказывал, — сказал воскликнувший, — я сократил релиз-цикл на треть от его первоначальной продолжительности и был по-настоящему расстроен, что я по-прежнему имел достаточно низкую эффективность цикла процесса (process cycle efficiency). И только сейчас я понял, что таким образом пытался оптимизировать использование ресурсов; вот что снижало мой показатель эффективности. Если осуществлять релиз сразу, как только готова соответствующая функциональная возможность (например, некоторая “заплата” (patch)), а не ждать “официального” релиза, когда он предусмотрен по графику, эффективность цикла процесса была бы куда выше!”

Говоривший явно испытывал гордость от своей проницательности, но, оглянувшись, заметил, что на большинство окружающих он не произвел особого впечатления. Затем он объяснил несколько застенчиво: “Я догадался, что именно это вы и пытались сказать все утро. Так много времени потребовалось, чтобы эта идея наконец дошла до меня.” (Том и Мэри Поппендик)

Создать постоянный ритм работы

Ритм работы организации, занятой разработкой программного обеспечения, определяют итерации. Каждые пару недель что-то изготавливается и передается заказчикам; через некоторое время заказчики начинают рассчитывать на это. Они могут строить планы, основываясь на регулярности и своевременности предыдущих поставок. Объем работы, который может быть выполнен в рамках итерации, быстро становится очевиден; через некоторое время все споры по этому поводу стихают. Имеет место устойчивый ритм, который движет все через систему с постоянной скоростью.

Насколько быстрым должен быть этот ритм? Один наш знакомый отдавал предпочтение недельным итерациям. Он полагал, что это достаточно долго, чтобы заказчики со срочной работой успели убедиться, что проблема действительно реальна, прежде чем

разработчики смогут ее разрешить, и в то же время достаточно быстро для выполнения обычных работ. Другой знакомый говорил, что 30 дней лучше, поскольку такой долгий срок дает возможность коллективу разработчиков все обдумать, прежде чем приступать к созданию программного кода, и в то же время это достаточно быстро, чтобы заказчики могли подождать ближайшей итерации.

Ритм выбран правильно, когда процесс функционирует равномерно и стабильно. Если в конце итерации имеет место "вспышка" активности, вероятно, продолжительность итераций *слишком велика*; в этом случае более короткие итерации смогут помочь "сгладить" рабочую нагрузку. Итерации должны быть достаточно краткими, чтобы заказчики смогли подождать до конца текущей итерации, и в то же время достаточно продолжительными, чтобы позволить системе стабилизироваться. Это легче всего понять на примере бытового термостата. Если бы термостат выключал отопительный котел в момент, когда температура поднялась выше заданной, и включал его снова при падении температуры ниже заданной, в этом случае котел бы выключался и включался слишком часто и, вероятно, скоро вышел бы из строя. Поэтому в термостатах имеется встроенное устройство, обеспечивающее запаздывание регулирующего воздействия. Термостат ждет, пока температура упадет на один-два градуса ниже заданной, прежде чем включить котел, и он ждет повышения температуры на те же один-два градуса выше заданной, прежде чем его выключить. Эта задержка достаточно мала, чтобы люди не ощущали особой разницы, и в то же время достаточно велика, чтобы котел не функционировал в колебательном режиме. Используйте эту же концепцию при поиске подходящего периода для вашего процесса.

Асинхронная работа

В одном известном нам (авторам) отделе, специализирующемся на встроенном программном обеспечении, сочли, что их график релизов становится слишком запутанным, когда увеличилось количество моделей соответствующей аппаратуры. Поэтому было решено создать единую версию программного обеспечения (точнее, что-то наподобие полуфабриката такой версии), которая бы поддерживала все модели аппаратуры. После уточнения, выпуск какой модели аппаратуры предстоит, можно было бы вносить (с трехнедельными интервалами) в программное обеспечение соответствующие изменения. Новая модель может дождаться очередного релиза программного обеспечения (не более трех недель) или можно убедить отдел программного обеспечения изменить дату этого релиза (не в ущерб другим моделям аппаратуры, естественно).

Сделав разработку программного обеспечения независимой от аппаратуры, отдел программного обеспечения смог установить собственный ритм работы, и не потребовалось много времени чтобы обнаружить, что новый порядок куда более эффективен. (Мэри Поппендик)

Объем работы не должен превышать возможности организации

Слишком часто можно услышать, что программный продукт "должен быть выпущен к такой-то дате". Возможности разработчиков в таких случаях никто не учитывает. При этом не только проявляется неуважение к людям, разрабатывающим продукт; такой подход также существенно замедляет процесс разработки. Мы знаем, что случается с компьютерной системой, когда нагрузка превосходит ее возможности — тогда имеет место явление, известное как "зависание".

Долгая суббота в аэропорту

Мы попали в аэропорт Мельбурна в прекрасное субботнее утро в 7:30, имея в запасе более чем достаточно времени, чтобы пересесть на рейс до Окленда (в 10:00). Регистрация билетов только что началась, поэтому мы положили наш багаж на весы и попробовали передать билеты служащей. "Нужно подождать, — сказала она — зависли компьютеры." Оглянувшись, мы наконец заметили длинные очереди вокруг.

"Как давно они зависли?" — спросили мы.

"Около часа назад", — она ответила.

В большинстве аэропортов США представительство каждой авиакомпании, вероятно, имеет доступ к нескольким компьютерным сетям, но в то же время во многих аэропортах (подобных Мельбурну) имеется доступ к единственной сети, которой пользуются все. Таким образом, целый аэропорт — как местные терминалы, так и международные — не функционировал в течение часа.

Вскоре зазвонил телефон. Кто-то сообщал, что компьютеры сейчас снова будут работать. Мы могли видеть, как по всему аэропорту десятки людей застыли над клавиатурами. А затем каждый из них начал яростно стучать по клавишам — все одновременно. Потребовалось не более пятнадцати секунд, чтобы система зависла снова. "Такое продолжается уже полчаса, — сообщила нам служащая — каждые десять минут сообщают, что сейчас система будет работать, а затем она зависает снова."

Не трудно было догадаться почему: вероятно, система не была рассчитана на то, чтобы сотни людей начинали нажимать клавиши одновременно.

Система наконец заработала тремя часами спустя. На оклендский рейс мы опоздали. (Мэри и Том Поппендик)

Время в организации, разрабатывающей программное обеспечение, может показаться резиновым. Люди часто работают сверхурочно, и когда такое случается, в течение короткого периода они даже могут таким образом выполнить больше работы. Однако продолжительная сверхурочная работа не приносит выгод. Люди устают, становятся невнимательны в конце долгого дня, и подобная "штурмовщина", скорее, замедляет выполнение работы. Иногда организация пытается работать с таким превышением своих возможностей, что начинает "зависать".

Проблема с обслуживанием заказчиков

Меня пригласили в компанию ознакомиться с процессом обслуживания заказчиков. Рисуя карту потока создания ценности на доске, мы дошли до места, где коллектив, занятый обслуживанием заказчиков, инсталлирует в среде заказчика вновь созданное программное обеспечение. И здесь мне сообщили, что на этом этапе у них имеется проблема. "Как часто эта проблема возникает?" — спросила я.

"*Каждый раз*", — ответил кто-то, и с этим согласились все.

"Тогда как эта проблема решается?" — спросила я.

Никак. Опять общее согласие.

Никак? Я не была уверена, что услышала правильно.

"Да, заявка на разработку поступает и помещается в очередь. И она там так и остается, если только она не от одной из трех (Топ 3) наиболее приоритетных для нас компаний. Дело в том, что разработчики слишком перегружены", — последовал ответ.

Чем? возник у меня невольный вопрос, но я спросила: "Что же в таком случае делается у заказчика?"

"Там остаются люди из отдела обслуживания заказчиков. Они занимаются иной работой. Могут пройти недели, прежде чем они получат какую-либо помощь", — был ответ.

"Но из сказанного выше следует, что они могут получить помощь только, если данная компания принадлежит к Топ 3", — сказала я.

"Ну, заказчик в конце концов начинает так сильно жаловаться, что его перемещают в Топ 3, а кого-то оттуда удаляют", — ответили мне.

"У меня складывается впечатление — сказала я, — что здесь проблема не столько с обслуживанием заказчиков, сколько с созданием программного кода, от которого можно было бы ожидать, что он будет работать в среде заказчика." Из последующего обсуждения мне стало ясно, что это больная тема.

"Почему отдел разработки не может сосредоточиться на том, чтобы выяснить, что именно вызывает появление этой проблемы?" — спросила я.

"Наши инвесторы очень требовательны, — последовал ответ — у нас есть график выпуска новых продуктов. Мы должны разрабатывать новые системы для новых заказчиков. Мы должны увеличивать число заказчиков."

"Но зачем вам нужны новые заказчики, если вы не будете в состоянии выполнить их заказы? — спросила я. — Сдается мне, что вы близки к зависанию."

"Да, это очень подходящее слово", — кто-то согласился. Однако остальные, похоже, не полагали, что данная проблема — это что-то экстраординарное. Что, в конце концов, вероятно, и было главным источником проблемы. (Мэри Поппендик)

Использование планирования с "вытягиванием"

Когда коллектив разработчиков отбирает работу (из очереди) для предстоящей итерации, правилом является, что члены коллектива должны брать столько работы, сколько они смогут выполнить. В продолжение одной-двух первых итераций они могут ошибаться и брать слишком много работы. Однако скоро определится производительность данного коллектива, и это позволит им отбирать как раз нужные объемы. Этот механизм "вытягивания" служит ограничению объема взятой работы возможностями коллектива. Если работа будет выполнена досрочно, что маловероятно, больше работы всегда может быть взято из очереди. Описанная система "вытягивания" работы из очереди имеет предохранительный механизм. В случае чего-то непредвиденного или когда что-то идет не так, как планировалось, коллектив может адаптироваться к новым обстоятельствам, либо завершив (раньше времени) текущую итерацию, либо официально переместив невыполненную работу в следующую итерацию.

Пример планирования "с вытягиванием"

Я посетила среднего размера отдел в крупном финансовом учреждении. Свен (ненастоящее имя) был ответственным за предоставление экономических результатов, и ему были необходимы какие-то изменения в программном обеспечении, чтобы их предоставить. Он был очень расстроен, поскольку было похоже, что его заявки никогда не будут выполнены. Расстройство Свена могло сравниться только с расстройством Карла (также ненастоящее имя), старшего менеджера в отделе информационных технологий, который считал, что Свен не понимает, насколько сложны его требования и насколько неприятны проблемы, создаваемые постоянно изменяющимися требованиями.

Карл настаивал, что каждая заявка должна подвергаться приблизительному анализу "стоимость/выгода", и в случае положительных результатов должна проводиться более подробная экспертиза (касающаяся архитектуры будущего программного обеспечения), прежде чем данная заявка будет помещена в очередь на выполнение. Свен против этого не возражал. Все, чего он хотел, это иметь некоторое влияние на то, что делается, и понимание, когда его заявка будет выполнена.

Как для управления заявками можно воспользоваться очередью, показано на рис. 5.4.

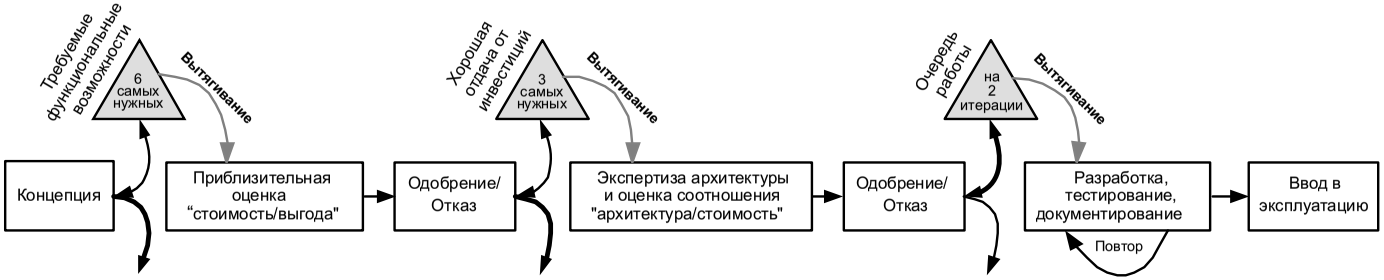


Рис. 5.4. Система с “вытягиванием” для управления процессом

Свен согласился ограничиться максимумом в шесть заявок одновременно. Карл согласился, что приблизительный анализ "стоимость/выгода" (для которого требовалось четыре часа) будет проводиться для каждой заявки в течение недели после ее подачи. На этом этапе Свен может либо передать ее на экспертизу (касающуюся архитектуры будущего программного обеспечения), либо не передать, но он (Карл) согласился иметь в очереди не более трех таких заявок на экспертизу одновременно. Если очередь уже заполнена, новая заявка либо заменит одну из содержащихся в очереди, либо не будет принята. Свен согласился, что когда очередь заполнена, он не будет подавать ни одной заявки, менее важной, чем те три, что в очереди.

Карл согласился, что архитектурная экспертиза и более детальная оценка стоимости должны выполняться в течение не более чем двух недель. Затем Свен может либо принять, либо не принять результаты. Принятые заявки будут направляться в журнал ждущей выполнения работы, и каждые две недели разработчики будут "вытягивать" работу из журнала для предстоящей итерации. Свен согласился, что объем работы в журнале должен быть не более чем на две итерации.

Важным является то, что "владеть" очередями будет Свен. При их краткости он с единого взгляда сможет определить, сколько приблизительно времени потребуется на выполнение той или иной работы. Свен сможет реорганизовывать очереди, добавлять или удалять из нее элементы в любое время, пока тот или иной элемент не был отобран (или "вытянут") из очереди разработчиками Карла. При таком подходе организация Карла будет всегда загружена работой, но не перегружена ею, и разработчики всегда будут работать именно над тем, что нужно Свену.

У Карла было несколько других заказчиков, но от Свена поступало 65% его работы. Карл полагал, что ему удастся интегрировать других заказчиков в эту систему управления очередью либо иметь отдельные коллективы для работы над их заявками. (Мэри Поппендик)

Соединение очередей в каскады (как показано на рис. 5.4) возможно и часто используется там, где в выполнении заказов участвуют несколько организаций. Очереди — это полезное средство управления, поскольку они предоставляют менеджерам возможность менять приоритеты и управлять временем цикла, в то же время позволяя коллективам разработчиков самим выбирать себе работу. Однако очереди это не идеальное решение. При их использовании следует руководствоваться несколькими общими правилами.

1. Очереди должны быть короткими, — возможно, два рабочих цикла. Именно длина очередей определяет среднее время цикла заявки в процессе разработки.
2. Менеджеры могут реорганизовывать или заменять работу в очереди в любое время. Однако, как только разработчики приступили к выполнению заявки, никто не должен вмешиваться в разработку.
3. Разработчики "вытягивают" работу из очереди и работают с постоянным ритмом до тех пор, пока работа не будет выполнена. Именно эта система отбора работы из очереди поддерживает их постоянную занятость, в то же время ограничивая количество одновременно выполняемой работы возможностями коллектива разработчиков ее выполнить.
4. Очереди не должны использоваться для введения заказчиков в заблуждение, что их заявки будут когда-то выполнены, в то время как соответствующий коллектив разработчиков не в состоянии выполнить всю имеющуюся работу.

Резюме

Мерилом зрелости организации является скорость, с которой она в состоянии гарантировано выполнять свои базовые процессы. Базовый процесс при разработке программного обеспечения направлен на преобразование потребительского спроса в соот-

ветствующий продукт. Таким образом, мы оцениваем нашу зрелость по скорости, с которой мы может надежно и неоднократно, руководствуясь потребительским спросом, создавать высококачественное работающее программное обеспечение, которое является продуктом, призванным разрешить проблему заказчика.

Попробуйте это

1. Сколько пунктов в вашей очереди дефектов? Насколько быстро они попадают в очередь? Как быстро они устраняются? При существующей скорости, сколько дней, недель, месяцев или лет работы потребуется, чтобы устранить все дефекты? Как много дефектов из очереди имеют хорошие шансы на устранение? Сколько времени тратится на просмотр и управление очередью? Стоит ли тратить на это время?
2. Сколько заявок имеется в вашей очереди невыполненной работы? С какой быстротой пополняется очередь? Сколько в среднем времени тратилось до сих пор на выполнение одной заявки? Сколько времени тратится на управление этой очередью? Сколько всего (в днях, неделях, месяцах или годах) работы в очереди? Содержится ли в этой очереди работа, которая вряд ли когда-нибудь будет выполнена? Почему? Какой процент от всего объема очереди это составляет?
3. Является ли высокая загрузка рабочего времени причиной перебоев в вашей организации? Измеряют ли у вас “загрузку людей”? Если да, какое влияние оказывает этот показатель: воспринимается ли он серьезно? Определяет ли он ритм работы? Положительное ли это влияние?
4. Что определяет в вашей организации размер партии (batch size): график релизов? Средний объем проекта? Можете ли вы уменьшить время между релизами или объемы проектов?
5. На собрании коллектива обсудите список способов сократить время цикла.
   - Добиться ритмичности поступления работы.
   - Свести к минимуму количество одновременно выполняющейся работы.
   - Минимизировать размеры объектов, одновременно находящихся в производстве.
   - Создать постоянный ритм работы.
   - Объем работы не должен превышать возможности организации.
   - Использование планирования с “вытягиванием”.

Какая из перечисленных мер кажется вам наиболее подходящей? Поэкспериментируйте, применив наиболее эффективный подход и оценивая, как изменится время цикла.

Глава 6

Люди

Система менеджмента

Как утверждает Гари Хэмель (Gary Hamel), "компания Toyota оказалась способна получать от своих служащих бо́льшую отдачу, чем ее конкуренты".[1] Детройту потребовалось двадцать лет, чтобы выявить принцип менеджмента, лежащий в основе способности Toyota бесконечно совершенствоваться. Только после того, как американские автомобилестроители исчерпали все другие объяснения, как компания Toyota добивается таких успехов (заниженный курс иены, дисциплинированная рабочая сила, особенности японской культуры, лучшая автоматизация), они наконец смогли признать, что истинное преимущество этой компании заключается в способности использовать интеллект "простых" служащих.

Бережливой можно назвать только ту систему менеджмента, которая создает увлеченных, думающих людей на каждом уровне организации, и особенно на высшем. Если реализовать все принципы бережливости за исключением одного, — уважения к людям, — удастся извлечь только тень от потенциальных преимуществ, которые способны предоставить бережливый подход. И если реализовать только один принцип, — уважение к людям, — это будет означать, что в вашей организации созданы условия, чтобы служащие реализовали оставшиеся принципы бережливости.

Boeing 777

В 1988 году компания Boeing сообщила о своих планах создать более крупную версию своей модели 767, которая была бы способна перевозить одновременно около 350 пассажиров. Однако, кажется, в тот момент никто в Boeing по-настоящему не понимал, что именно нужно заказчикам, поэтому предлагаемый дизайн будущей модели вызвал лишь коллективное "позевывание". Итак, компания Boeing приступила к проектированию новой модели, теперь стараясь прислушиваться к мнениям клиентов. Двумя годами позже конструкция Boeing 777 участвовала в конкурсе с McDonnell Douglas MD-11 и Airbus A 330 — двумя уже существующими воздушными лайнерами — на крупный заказ от компании-авиаперевозчика United Airlines. Boeing выиграла заказ, однако при этом ей были поставлены два условия: во-первых, новый лайнер должен быть создан и сертифицирован для работы на дальних линиях в течение пяти лет и, во-вторых, компании Boeing и United Airlines будут работать совместно, чтобы обеспечить соответствие нового лайнера запросам заказчика.

---

[1] Hamel G. *Management Innovation* Harvard Business Review, *February, 2006*, p. 74.

Это была сделка, от которой зависело будущее компании: невероятно жесткие сроки и суровые санкции за их несоблюдение, в сочетании с возможностью (которая предоставляется однажды в десятилетие) создать воздушный лайнер будущего поколения. Ответственность была возложена на Алана Мюлелли (Alan Mulally), который начал с того, что создал совершенно новую систему менеджмента для проекта Boeing 777. Находились люди, которые возражали, что сейчас не время для новых систем менеджмента — старая функционировала неплохо в течение многих лет. Однако аналитики, занимавшиеся выявлением причин всех имевших место проблем, которые затягивали выполнение предыдущих проектов, приходили к единодушному выводу: задержки возникали потому, что люди недостаточно сотрудничали, не решали возникающие проблемы вовремя, просто не реагировали на возникающие проблемы, не обменивались информацией быстро и эффективно. Поэтому Мюлелли сконцентрировался на программе "Работать вместе" (Working Together), которая явилась чрезвычайно удачным примером организации работы и которая была вознаграждена экстраординарным успехом проекта Boeing 777. Были организованы более 200 коллективов, членами которых являлись люди из конструкторского бюро, производства, снабжения и авиакомпании — кто угодно, от пилотов до тех, кто занимался обработкой багажа. Эти люди смогли видеть будущее воздушное судно глазами тех, кому предстояло его использовать. Например, благодаря этому подходу обнаружилось, что отверстие для приема топлива на крыле будет так высоко расположено, что автозаправщикам будет проблематично до него доставать — наверняка при старых подходах этот недостаток не был бы выявлен вплоть до поставки.[2]

Мюлелли настаивал, чтобы все "обменивались опытом и обменивались часто" и что в этой совместной работе секретов быть не должно. Каждая проблема должна открыто обсуждаться до тех пор, пока не появится ее решение. Поставщики работали на основе доверия, и заказчикам был предоставлен "беспрецедентный" доступ к процессу разработки, а также право голоса при принятии решений.

Мюлелли исповедовал философию "тестировать рано и устранять проблемы быстро". Технологическая операция тестирования была перемещена в процессе создания нового изделия на всех уровнях на максимально ранние стадии. Впервые конструкция лайнера целиком создавалась с использованием системы трехмерного моделирования. Трехмерное моделирование явилось чем-то наподобие блочного тестирования для каждого узла, давая возможность обеспечить соответствие данного узла всей конструкции. Это позволило разрешить проблему, которая имелась на ранних стадиях разработки всех предыдущих моделей.

Тестирование было всесторонним на каждой стадии. Новый двигатель компании Pratt & Whitney считался настолько проверенным, что его полетные испытания, казалось, можно было бы не проводить. Однако коллективная мудрость инженеров склонялась в пользу его тестирования несмотря ни на что, поэтому новые двигатели были установлены на Boeing 747 (ценой огромных затрат) и испытаны в реальном полете. В результате выяснилось, что двигатель в некоторых режимах функционирует неудовлетворительно и что некоторое важные его узлы должны быть изменены. Реакция Мюлелли на происшедшее была: "Великолепно! Мы выявили проблему заранее."

Все это стало известно благодаря тому, что компания Boeing позволила вещательной корпорации Public Broadcasting System (PBS) заснять на пленку все пятилетние усилия по

---

[2] Carrison D. *Deadline! How Premier Organizations Win the Race Against Time* AMACOM (American Management Association — Американская ассоциация менеджмента), 2003, Chapter 5.

созданию новой модели, без какого-либо редакторского контроля. Получившийся в результате пятичасовый документальный фильм[3] дает возможность понять, как программа "Работать вместе" позволяет быстро воспринимать и реализовывать идеи всех — идея сборщика крыльев на производственной линии была воплощена в жизнь через час, а не через несколько недель. Фильм демонстрирует, как маленькой компании из Австралии, которая производила рулевую систему, приходилось вносить одно существенное изменение за другим, однако, работая на основе доверия, они сумели все сделать вовремя.

Компании Boeing пришлось работать творчески с Федеральной авиационной администрацией (Federal Aviation Administration — FAA), чтобы получить сертификат для нового лайнера на право дальних рейсов, до поставки самого изделия авиакомпании United Airlines. (Обычно машины со спаренными двигателями должны были испытываться в реальных условиях в течение около двух лет, прежде чем им позволялось сколько-нибудь удаляться от аэропорта.) Компания Boeing предложила серию испытаний, настолько жестких, что как пилоты, так и представители FAA были заинтригованы: сможет ли только что созданный лайнер выдержать столь жесткий режим испытаний? Однако стратегия "тестировать рано и устранять проблемы быстро" принесла свои плоды. Новая машина с легкостью прошла экстенсивные испытания и явилась первым лайнером со спаренными двигателями, так быстро получившим сертификат.

Модель 777 явилась первым лайнером компании Boeing, управляемым программным обеспечением и электрическими силовыми приводами, вместо традиционных гидравлических систем и тросов. В рамках программы "Работать вместе" компания Boeing настаивала, чтобы все участвующие в проекте, использовали один язык программирования — Ada. Хотя поставщики поначалу сопротивлялись, в конце концов такой подход им понравился. Крупнейшим компонентом программного обеспечения была "Система управления информацией воздушного лайнера" (Airplane Information Management System), созданная отделом авиатранспортных систем компании Honeywell. Эта компания использовала параллельную архитектуру, позволяющую одновременную и независимую разработку семи главных функций коллективами, состоящими из 60–100 инженеров. В конце концов все программное обеспечение модели 777 было поставлено вовремя, за исключением телефонной системы для пассажиров.

Я участвовал в создании Boeing 777

Я работал в отделе авиатранспортных систем компании Honeywell в Миннеаполисе во время, когда разрабатывалась модель 777. Мы создавали инерционную навигационную систему с тройным резервированием — лазерные гироскопы были призваны точно определять текущее местоположение лайнера, даже при отсутствии информации GPS. Существование программы "Работать вместе" компании Boeing чувствовалось во всем. Дело было до того, как Интернет позволил всему миру пользоваться вездесущей электронной почтой, поэтому мы соединили наши внутренние системы e-mail вместе, что было новинкой в то время. Мы использовали FTP для электронного обмена текущими версиями документов и устраивали еженедельные телеконференции.

Это были трудные времена для авиастроительной отрасли, которая переживала в середине 90-х годов один из циклических спадов. Массовые увольнения приводили к тому, что многие были вынуждены сменить профессию. Однако каждый раз, когда я встречаю кого-то, кто работал над моделью 777, их воспоминания об этом времени неизменно позитивны. (Том Поппендик)

[3] *21st Century Jet: The Building of the 777*, Produced by Skyscraper Productions for KCTS/Seattle and Chanel 4 London, 1995.

Наконец, 7 июня 1995 года три Boeing 777 поднялись в воздух для первых коммерческих рейсов; тем самым сроки, установленные пятью годами ранее, были соблюдены. В течение более чем десяти лет компания Boeing поставляла свою модель 777 ускоренными темпами по 50 машин в год, и сейчас, когда пишутся эти строки, компания имеет заказов на пять лет вперед.

Компания Boeing в течение этих десяти лет приступила к созданию новой модели, 787 Dreamliner. В рамках этой программа появилась еще одна инновация: беспрецедентная степень сотрудничества между Boeing и партнерами этой компании по всему миру в процессе разработки нового лайнера. Boeing создала всемирную инновационную сеть и адресует проблемы, возникающие в процессе конструирования, туда, где существуют наилучшие условия для их решения. Разработка модели 787 поддерживается значительно лучшими электронными средствами организации совместной работы, чем это было при работе над моделью 777, однако именно культура программы "Работать вместе" компании Boeing является тем фактором, который делает это сотрудничество эффективным.

У. Эдвардс Деминг

В июле 1950 года специалист по статистике из США по имени У. Эдвардс Деминг (W. Edwards Deming), который несколькими годами ранее посещал Японию, чтобы помочь с организацией переписи населения, был снова приглашен в эту страну, чтобы прочитать серию лекций группе наиболее влиятельных промышленных лидеров. У Деминга спросили, как изменить сложившееся в мире мнение, что Япония производит низкосортную продукцию. Он ответил, что производство — это система, включающая также поставщиков и потребителей, а потому руководители бизнеса должны постоянно уделять внимание улучшению всей системы. Один из способов сделать это — использовать средства статистического контроля производства (statistical process control) для "встраивания качества в продукцию", вместо того чтобы контролировать ее после изготовления. Записи лекций Деминга были переведены на японский язык и прибыли от их продажи используются Союзом японских ученых и инженеров (Union of Japanese Scientists and Engineers — JUSE) для финансирования престижной премии Деминга.

Деминг посещал Японию несколько раз в последующие 15 лет, однако у себя дома, в США, он был не очень известен. Затем, в 1980 году, NBC показала передачу "If Japan Can...Why Can't We?" (Если может Япония... Почему не можем мы?), в которой Деминг критикует компании США за низкое качество их продукции. В свои 80 лет Деминг наконец-то был "открыт" в собственной стране. Телевизионное шоу, о котором идет речь, транслировалось только однажды, но оно шокировало страну и заставило корпорации обратить внимание на качество. Деминг играл видную роль в движении за качество в Соединенных Штатах, вплоть до своей смерти в 1993 году.

Деминг сформулировал "Систему глубокого знания" (System of Profound Knowledge), которая включает четыре основных положения.

1. **Система прежде всего.** Деминг учил, что взаимопонимание и взаимодействие между составляющими любой системы — это ключ к успеху системы в целом. Поскольку он рассматривал производство как систему, он подчеркивал, что крайне важно управлять взаимодействием ее составных частей — взаимодействием между отделами компании, взаимодействием между поставщиками, производителями, заказчиками и т.п. Деминг был твердо убежден, что интересы системы при-

оритетны; оптимизация отдельных составляющих системы вовсе необязательно идет на пользу системе в целом.

2. **Знание о вариации.** Деминг приходил в ужас от того, как рабочих обвиняли в проблемах, которые были присущи системе, в которой они работали. Его лекции были направлены на то, чтобы менеджеры поняли, что большинство случаев вариации (variation) системы являются присущими системе. Попытки исключить вариацию из системы не только тщетны, но и ведут к ухудшению дел. Деминг подчеркивал, что большинство причин низкого качества и низкой производительности труда являются неотъемлемыми для системы, а потому отдельный рабочий не в силах здесь что-то изменить. В этих условиях жесткие сроки и призывы никак не помогут решить проблемы системы. Что необходимо взамен, настаивал Деминг, — это изменение способов функционирования системы.
3. **Теория знания.** Будучи физиком по образованию, Деминг отдавал предпочтение научному методу в развитии понимания системы (основанному на информации) и принятия решений. Элементы научного метода — создание гипотезы, экспериментирование, приобретение знания, использование полученного знания — образуют хорошо известный цикл Деминга[4]: планировать (plan), делать (do), проверять (check), действовать (act) — PDCA. Подобный цикл можно обнаружить в любой программе совершенствования качества или процесса. Более подробно об этом речь пойдет в главе 7.
4. **Психология.** Деминг предостерегал менеджеров, что цифры не всегда отображают полную картину, и там, где дело касается людей, имеют значение мастерство, гордость, опыт, уверенность и сотрудничество.

Деминг суммировал свои многомерные представления о качестве в 14 заповедях для менеджмента (см. ниже).

Четырнадцать заповедей Деминга[5]

1. Заботьтесь о долговременных интересах компании; не следует сосредотачиваться на кратковременной выгоде. Целью является оставаться в бизнесе и предоставлять работу.
2. Мир меняется, и менеджеры должны адаптироваться к новому мышлению. Задержки, ошибки, брак и плохое обслуживание больше неприемлемы.
3. Прекратите полагаться на контроль в выявлении дефектов и начните "встраивать качество в продукцию" в процессе ее производства. Используйте статистическое управление процессом.
4. Не выбирайте поставщиков только на основе предложения низкой цены. Уменьшайте общую стоимость за счет создания долговременных связей с поставщиками, основывающихся на лояльности и доверии.
5. Постоянно работайте над совершенствованием системы производства и обслуживания. Совершенствование — это не разовое действие, а постоянный процесс; функционирование системы должно подвергаться постоянному совершенствованию, чтобы сократить расходы и улучшить качество.

[4] Деминг называл это *циклом Шуорта*, поскольку первоначально этому учил Уолтер Шуорт (Walter Shewhart), который в 30-х годах (XX в.) занимался разработками в области управления статистическими процессами в Bell Laboratories в США.

[5] Верный своей философии непрерывного совершенствования, Деминг постоянно занимался перефразированием своих 14 пунктов. Список в виде, в котором он представлен в данной книге, можно найти в его книге *Out of the Crisis*, MIT Press, 2000; originally published in 1982.

6. Поощряйте учебу. Менеджеры должны знать, как делать работу, которой они управляют, и быть способны обучать рабочих. Менеджеры также должны учиться, чтобы понимать производство.
7. Будьте лидером. Работа менеджера состоит в том, чтобы помочь людям лучше выполнять свою работу и устранять помехи в системе, которые мешают им выполнять работу с гордостью. Наибольшая проблема Америки — недостаточное использование способностей людей.
8. Изживайте страх. Чтобы хорошо выполнять свою работу, люди должны чувствовать себя в безопасности. Не должно существовать конфликта между интересами компании и интересами человека на данном рабочем месте.
9. Устраняйте барьеры между подразделениями компании. Создавайте совместные коллективы, в которых каждый способен взглянуть на вещи глазами коллег. Не подрывайте сотрудничество в таком коллективе, поощряя индивидуальные достижения отдельных его членов.
10. Прекратите использовать лозунги и призывы. В дефектах и низкой эффективности виновата система, а не рабочие. Призывы не меняют систему; это ответственность менеджмента.
11. Откажитесь от цифровых показателей для рабочих и менеджмента. (Мы (авторы) бы добавили: откажитесь от зачастую произвольно устанавливаемых сроков для коллективов разработчиков.) Это управление через страх.
12. Ликвидируйте помехи, которые не позволяют людям испытывать гордость за свое мастерство. Перестаньте воспринимать рабочих с повременной оплатой как придаток производства. Откажитесь от использования для таких работников такого показателя, как ежегодный показатель эффективности (annual performance ratings).
13. Поощряйте самообразование и самосовершенствование. Образованная рабочая сила — это ключ к будущему.
14. Делайте что-то, чтобы достичь нужных изменений. Команда высшего менеджмента должна подавать пример действием, а не просто поддерживать чьи-то усилия.

Алан Мюлелли создал систему менеджмента для разработки модели Boeing 777, основываясь на идеях Деминга во время, когда многие компании в Соединенных Штатах открывали для себя мощь этих идей. Коллективы, объединяющие разработчиков и производственников, еженедельные собрания, на которых обсуждалось, как выявлять, диагностировать и решать проблемы, позволили вызвать профессиональную гордость у всех, участвующих в создании нового лайнера. Хотя сроки поджимали, работа всегда продолжалась в постоянном ритме.

Деминг говорил, что истинной целью существования любой коммерческой компании является не “деланье денег”, а создание потребителей, настолько удовлетворенных, что они будут продолжать покупать продукцию данной компании. Мюлелли позаботился о том, чтобы коллектив создателей модели 777 не концентрировался на соблюдении сроков при наименьших затратах; взамен целью являлось создание машины, которую было бы легко производить, выгодно (для авиакомпаний) эксплуатировать, просто (для летчиков) пилотировать, несложно (для наземного персонала) обслуживать и в которой было бы удобно путешествовать (для пассажиров).

Деминг подчеркивал, что поставщики должны выбираться на основе их способности работать в тесном сотрудничестве с производителем, чтобы свести к минимуму общую себестоимость системы. Компания Boeing не только следовала этому принципу, но шла дальше и увеличивала ценность системы в долгосрочной перспективе. Например, в компании Boeing полагали, если данную модель предполагалось продавать по всему миру, то узлы для нового лайнера должны производиться в разных странах мира. Так, 20% узлов модели

Boeing 777 предполагалось производить в Японии, стране, которая является одним из крупнейших покупателей широкофюзеляжных реактивных лайнеров компании Boeing.

То новое в управленческом мышлении, что Деминг предложил миру, заключается в следующем: когда что-то не ладится, причина почти всегда заключается в недостатках системы, а потому это проблема менеджмента. Менеджеры, которые оказывают давление на людей с целью заставить их уложиться в сроки или устранить дефекты, игнорируют недостатки системы и обычно только ухудшают положение дел. Взамен руководители должны видеть недостатки и проводить коренные изменения, направленные на их устранение.

Почему хорошие начинания оказываются неудачны

В течение ряда лет мы могли наблюдать множество корпоративных инициатив — от программ повышения качества Деминга (и других) до (появившихся позже) программ, связанных с бережливым подходом в начале 90-х годов и программ "Шесть Сигм" (Six Sigma). Сейчас мы видим возврат к бережливому подходу. Мы также наблюдали модели систем управления качеством CMM и ISO 9000. Все эти инициативы начинались с самыми лучшими намерениями и они принесли свои плоды. Однако некоторое время спустя при реализации часто утрачивается их исходная суть. Каждая из упомянутых инициатив принесла ощутимые и устойчивые улучшения в одних компаниях и от посредственных до разочаровывающих результатов во многих других. Почему это так?

Гари Хэмель (Gary Hamel) полагает, что крупнейшие усовершенствования в компаниях происходят от инноваций в менеджменте — заметный отход от привычных управленческих принципов, процессов, практик или организационных форм, — которые обеспечивают такой компании преимущество над конкурентами.[6] Если инновация в менеджменте создает преимущество над конкурентами, значит, это крупная инновация, которую очень трудно копировать, поскольку инновации в менеджменте означают существенный отход от общепринятых норм.

Однако это не удерживает компании от попыток копирования лучшего опыта конкурента и подражания той или иной управленческой новинке. Почти всегда в подобных случаях фундаментальный принцип инновации оказывается незамечен "заимствующей" компанией, поскольку, как говорит Гари Хэмель: "ортодоксы от менеджмента часто настолько пропитаны мышлением исполнителя и настолько старательно стараются воплотить задачу в жизнь, что от этого страдает конечный результат. Чем более "неудобен" принцип, лежащий в основе управленческой инновации, тем больше времени требуется конкурентам для реакции. Иногда "чесание затылка" продолжается десятилетиями."

Иными словами, суть инновации в менеджменте часто оказывается невидима для большинства управленческих команд, пытающихся ее скопировать. Например, многие компании пытались копировать бережливый подход, однако они часто не замечали главное. Как заметил Гэри Хэмель, истинная инновация компании Toyota состоит в ее способности использовать интеллект "простых" служащих. Успешные бережливые инициативы должны основываться прежде всего на глубоком уважении к каждому работнику компании, особенно к этим "простым" людям, которые производят продукцию или разрабатывают программы.

---

[6] Hamel G. *Management Innovation*, Harvard Business Review, *February, 2006*, p. 74.

Вы полагаете, что мы хотим похитить ваши плечики?[7]

Когда Деминг лежал в больнице Сибли (Sibley Hospital) в начале 90-х годов, он пригласил администратора больницы Джерри Прайса (Jerry Price) в свою палату и спросил: "Вы не доверяете вашим пациентам?" Прайс был озадачен.

"Загляните в шкаф!" — заревел Деминг. Ему приходилось много кричать, особенно на управленцев высокого ранга. Прайс заглянул в шкаф, который оказался заполнен плечиками для одежды, типа тех, которые используются в дешевых гостиницах.

"Как бы вы себя чувствовали, если бы вам было 92 года, вы были больны и при этом не имели возможности даже повесить одежду? — поинтересовался Деминг. — Вы полагаете, что ваши пациенты хотят похитить эти плечики?"

Деминг послал больнице чек на 25 тыс. долл. с просьбой приобрести новые плечики для одежды для всех больничных палат.

Прайс так и сделал. Затем дал указание часть из них почистить наждачной бумагой, а затем послал их Демингу с тем, чтобы он подписал их. Сегодня, когда кому-либо присуждается премия за качество в больнице Сибли, ему одновременно вручаются также прочные деревянные плечики с автографом Деминга. (Клер Кроуфорд-Мейсон, продюсер передачи (на NBC) 1980 года "Если может Япония... Почему не можем мы?")

Задумайтесь о тех сигналах, которые посылают "простому" человеку (а также поставщикам) политика и практика в вашей компании. Доверяете ли вы своим служащим или запираете кладовые? Говорите ли вы о важности качества, или вас заботят только сроки? Предлагаете ли вы людям работать коллективно, но в то же время противопоставляете их друг другу, вручая премии за индивидуальную эффективность? Рассуждаете ли вы о доверии, когда общаетесь с поставщиками, в то же время настаивая на контрактах с фиксированными ценами? Если вы делаете эти вещи, значит, вы упустили главное в принципе бережливости: уважение к людям.

Коллективы

Наши дети были неплохими пловцами, когда учились в высшей школе (high school). Почти в течение десяти лет фамилия нашего сына была на доске почета в школьном бассейне, пока кто-то в конце концов не проплыл на спине быстрее. Мы также любили заниматься плаваньем, поскольку наши сын и дочь (разница в возрасте три года) были в том же плавательном клубе. Они практиковались вместе и пришли к взаимопониманию. Спорт требовал дисциплины и помог им приобрести полезные привычки. Однако занятия плаванием в действительности не научили их работе в коллективе.

Разница между индивидуальными и командными видами спорта совершенно ясна. В индивидуальных видах соперники побеждают или проигрывают в зависимости от своих возможностей. А в командных видах побеждает или проигрывает вся команда, в зависимости от того, насколько члены команды способны взаимодействовать. Во время учебы в колледже наш сын стал рулевым восьмерки в академической гребле. Он сидел сзади в конце длинной узкой лодки, синхронизируя усилия восьми дюжих гребцов. Академическая гребля — это определенно командный вид спорта.

[7] Crawford-Mason C. Special to *The Washington Post*, *Tuesday, April 23, 2002*, p. HE01, Использовано с разрешения.

Прыжки в воду и гимнастику также можно было бы отнести к командным видам спорта, поскольку здесь индивидуальные очки суммируются для командного зачета. Но это не то же самое, что баскетбол или футбол, где члены команды должны синхронизировать свои усилия, чтобы выиграть. То же касается и коллективного труда. Люди, работающие бок о бок, не обязательно являются коллективом, даже если коллективный результат получается путем суммирования индивидуальных результатов. Группа рабочих становится коллективом, когда его члены чувствуют взаимную ответственность при создании "продукта коллективного труда".[8]

Нет ничего плохого в индивидуальных видах спорта, просто они отличаются от командных видов. И нет ничего плохого, когда рабочие работают группой. Просто они не являются коллективом. Если собрать вместе группу людей и назвать их коллективом, это еще ничего не значит. Группу людей в коллектив превращает взаимное стремление его членов объединить их навыки и работать вместе для достижения общей цели.

Что такое коллектив?

Что поднимало нашего сына каждое утро и заставляло его мчаться с холма к клубу гребцов Корнельского университета (Cornell University)? В конце концов езду на велосипеде назад под гору можно было счесть наказанием. Но предстояли соревнования, и для выигрыша на них гребцам приходилось прилежно работать каждый день, чтобы научиться синхронизировать свои усилия с тем, чтобы быть в состоянии грести быстрее и интенсивнее, чем команды других учебных заведений. Если кто-то будет грести неправильно или не будет выкладываться до конца, команда не добьется прогресса, необходимого для победы на соревнованиях.

Процесс создания программного обеспечения больше похож на греблю, чем на плавание. Разработка ПО связана с постоянным решением сложных проблем. Разработчикам приходится много раз на день принимать сложные решения, последствия которых часто могут лежать за пределами их собственной работы. В этих условиях, чтобы получить хорошие результаты, очень важно объединить вместе различные навыки и знания.

Коллективу нужны общая цель и взаимное обязательство работать вместе для достижения этой цели. Члены коллектива зависят друг от друга и должны помогать друг другу. В организации, где преобладает мудрость, постараются создать среду, в которой поставленная задача воспринимается как вызов, и где члены коллектива стараются работать с использованием всех своих возможностей, чтобы не подвести коллег.

Группа или коллектив?

1. *На утренней планерке каждый отчитывается о сделанном за предыдущий день. В конце менеджер проекта говорит: "Хорошо, Джо, сегодня ты займешься..." И так для каждого работника. Получив задание на день, все расходятся.*

   Это группа. Назначая работу, менеджер проекта берет на себя ответственность, что все будет выполнено. Люди, входящие в такую группу, не испытывают взаимной ответственности за выполнение проекта.

2. *На утренней планерке каждый отчитывается о сделанном за предыдущий день, рассказывает, что он собирается сделать сегодня и нужна ли ему помощь. Джон напоминает всем, что сложная функция (о которой, наверное, уже шла речь не раз) все еще не разработана,*

[8] Katzenbach J.R., Smith D.K. *The Wisdom of Teams*, Harvard Business School Press, *1992*.

и добавляет: ''Если никто не возьмет на себя это, боюсь, данная часть работы не будет сделана вовремя. Я бы сам это сделал, но мне обязательно нужно разрешить другую проблему, над которой я сейчас работаю.''

В ответ Майк говорит: ''Эта функция, вероятно, более важна, чем то, чем я занят сейчас, но это не та область, где я чувствую себя как рыба в воде.''

Джон предлагает, что он может потратить пару часов, чтобы помочь Майку.

Далее ScrumMaster[9] замечает, что разработка интерфейса к встроенной программе (firmware) никак не сдвинется с мертвой точки; не поможет ли делу, если еще кто-то подключится к этой работе. Трое разработчиков тут же соглашаются.

Это коллектив. Собравшиеся обсуждают, как достичь общей цели. ScrumMaster уделяет особое внимание возникшему ''узкому месту''.

Это не так легко...

1. *Я пробовал коллективный подход, но мне не удавалось сделать так, чтобы задачи, которые следовало решить в ходе текущей итерации, были восприняты серьезно. Люди отмечались и уходили домой в конце рабочего дня. Когда итерация подошла к концу, работа, на которую коллектив, как предполагалось, должен был быть нацелен, осталась не выполнена никому до этого не было дела.*

   Вы получили то, чего можно было ожидать. Ожидали ли непосредственные руководители, что члены коллектива проявят приверженность коллективным целям и будут действовать соответственно? Какова была их реакция, когда обнаружилось равнодушие вместо приверженности? Что вознаграждалось при подведении итогов — индивидуальные достижения или вклад каждого, способствовавший достижению общего успеха? Когда в организации индивидуализм ставится выше сотрудничества в коллективе, почти невозможно поддерживать в течение долгого времени здоровый коллективистский дух.

2. *Мы используем компьютерную систему, которая назначает людей в тот или иной коллектив. Сначала в течение нескольких месяцев приходится работать в одном коллективе, затем в другом, затем в третьем. В этих условиях очень трудно сделать так, чтобы люди были привержены общей цели.*

   Действительно, очень трудно создавать коллективы в среде, где люди постоянно переводятся с одного места на другое. Возможно, в каком-то случае группа становится коллективом, однако после расформирования члены такого коллектива редко приносят коллективистский дух в другую группу.

3. *В моей организации существует система оценки индивидуальных достижений. Чтобы получить высшую оценку, приходится выкладываться. В этих условиях помощь коллегам становится невыгодна; более того, другой сотрудник может за счет моих идей получить более высокую оценку, чем я.*

   До тех пор, пока существует система индивидуальных оценок или система индивидуальных премий, можно напрочь забыть о коллективе.

4. *Люди были бы рады быть привержены коллективной цели, однако они состоят в пяти коллективах одновременно. В этих условиях трудно предвидеть, когда и в каком коллективе появится срочная работа. Поэтому через некоторое время людей просто перестают интересовать коллективные цели.*

   Наш сын не смог бы стать высококлассным пловцом в школе или высококлассным рулевым в колледже, если бы он занимался одновременно несколькими видами спорта. Мы имели два приоритета — высокие оценки в учебе на первом месте и спортивные успехи на втором. Это почти максимальные цели, которые можно ставить одновременно. Люди могут одновременно состоять в пяти рабочих группах, но только в одном, максимум в двух, коллективах.

Компетентность

Лодка в академической гребле очень узкая, быстро движется, поворачивает медленно и переворачивается легко. Просто чтобы оставаться сухими, команда лодки должна прилагать большие усилия для слаженных действий. Академическая гребля очень зависит от течения, ветра, совместных усилий и стратегии. Синхронные усилия очень важны для равномерного движения лодки и экономии сил для финального рывка. Чтобы выиграть гонку, гребцы прежде всего должны быть хорошо развиты физически и уметь синхронизировать свои усилия с усилиями своих товарищей по команде при каждом гребке. Требуется много часов тренировок, прежде чем отличные атлеты превратятся в отличных гребцов.

В компании Toyota полагают, что устойчивое совершенство в производстве продукции начинается с "накапливания технической компетентности всеми инженерами".[10] Речь при этом идет не о знаниях, которые приобретаются в колледже; имеется в виду компетентность, приходящая со временем, для которой требуются годы опыта и самообразования. Путь новоиспеченных инженеров в компании Toyota начинается на предприятиях, где они участвуют в производстве автомобилей как рядовые рабочие. После этого несколько месяцев они работают в системе сбыта, продавая автомобили и общаясь с потребителями. Еще через полгода молодые инженеры назначаются в конструкторский отдел, работать под руководством старшего инженера. Первое, чем они здесь занимаются, — это реализуют "проект новичка" — проект по совершенствованию уже имеющегося изделия, который достаточно сложен и воспитывает стремление (общее для инженеров компании Toyota) к постоянным улучшениям. Приблизительно через год новые инженеры приступают к напряженной двухгодичной программе обучения на практике, с помощью которой они должны приобрести стандартный набор навыков. Только после этого они становятся инженерами начального уровня и способны к работе в коллективе. После всех перечисленных подготовительных этапов инженеру требуется провести пять или шесть лет, работая по специальности, прежде чем его будут считать первоклассным инженером.

Компетентность может быть разной

У нас сотни разработчиков. При таком количестве работников их компетентность очень различается --- у одних отличная, у других средняя, у третьих ниже средней.

Мы полагаем, если в организации создан соответствующий климат, вполне возможно из "обычных людей" сделать "коллективистов". И начинать нужно с того, чтобы сделать задачей высшего приоритета для каждого менеджера развивать соответствующие черты характера людей, за которых они несут ответственность. В компании Toyota "воспитание людей является фундаментальной задачей менеджера. Все менеджеры рассматривают эффективность своих коллективов как непосредственное отражение их собственных способностей. Это становится личным делом".[11]

Совершенные продукты программного обеспечения создаются высококомпетентными техническими экспертами во многих областях: архитектуре, объектно-ориентированных

---

[9] Лицо, ответственное за правильное применение методологии Scrum.

[10] Morgan J., Liker J. *The Toyota Product Development System: Inte-grating People, Process, and Technology*, Productivity Press, *2006*. Chapter 9.

[11] Morgan J., Liker J. *The Toyota Product Development System: Inte-grating People, Process, and Technology*, Productivity Press, *2006*, p. 169.

технологиях, стратегиях создания программного кода, информационных структурах, автоматизации тестирования. Это не зависит от методологии разработки; за каждым совершенным программным продуктом, независимо от того, как он разрабатывался, вы найдете совершенных (технически) людей. Это также актуально для бережливого подхода в разработке, и мы полагаем, что политика организации должна быть направлена на привлечение и воспитание таких экспертов.

Целесообразно, чтобы коллектив разработчиков включал специалистов из разных областей, которые необходимы для достижения поставленных целей. Члены коллектива должны быть привержены специфическим целям коллектива и обеспечивать надежное их достижение путем совместной работы, объединения опыта и взаимопомощи там, где это необходимо. Взаимопомощь наиболее действенна в среде, где бок о бок работают представители различных профессий. В процессе разработки возможна бо́льшая потребность в тестировании, чем в создании нового программного кода, или в проектировании пользовательского интерфейса, чем в работе над базой данных. В этих условиях разработчики могут помочь тестировщикам, создатели интерфейса могут тестировать взаимодействие с пользователем, члены коллектива с невысокой квалификацией могут сотрудничать с опытными, помогая им и учась у них. При этом каждый участник процесса является прежде всего членом коллектива, а уже пото́м специалистом в какой-то области. Никто в коллективе не получит признания за то, что он сделал до тех пор, пока общая цель не достигнута. Поэтому все трудятся одинаково напряженно, независимо от квалификации каждого.

Специалисты

1. *Наш коллектив постоянно сталкивается с задержками, поскольку создатели документации (technical writers) вынуждены работать над несколькими проектами одновременно. Документация пользователя является важной частью нашей продукции, однако создатели документации не в состоянии поспевать за коллективом из-за своей загрузки.*

   Создатели документации, так же, как и другие специалисты, должны быть постоянными членами коллектива. Почти невозможно полноценно работать одновременно более чем в двух коллективах. А еще лучше, если это один коллектив. В то же время возможны ситуации, когда может не оказаться достаточно работы, чтобы загрузить создателей документации на всех этапах разработки проекта.

   Существует несколько способов решения этой проблемы — какой из них выбрать, зависит от особенностей вашего случая. Один подход состоит в том, чтобы расширить роль создателей документации и вовлечь их в процесс разработки продукта, — например, они могли бы помочь с созданием пользовательского интерфейса, поскольку эти специалисты по определению имеют хорошее представление о мышлении пользователя. Они также могут помочь с тестированием или созданием любой технической документации, а не только пользовательской.

   Второй подход может состоять в том, чтобы переосмыслить методы создания документации. Новые методы, например, с использованием стандарта DITA (Darwin Information Typing Architecture), которые в основном предполагают постепенное создание документации (в процессе разработки соответствующего продукта) и повторное использование готового текста, могут позволить создателям документации распределить их работу более равномерно в течение цикла разработки.[12]

   Или, возможно, начать создание пользовательской документации следует кому-то другому, кто сможет справиться с данной задачей. Например, специалисты из отдела технической

[12] OASIS, организация управления стандартами, одобрила метод DITA в июне 2005 года..

поддержки лучше всех знают, как пользователи "борются" с системой и ее документацией. Поэтому именно у них, скорее всего, появятся хорошие идеи насчет того, какая документация будет наиболее полезна для пользователей.

2. *В нашей компании личные или групповые интересы нередко ставятся выше коллективных. Например, за создание большого экрана дизайнеры графики получат куда бо**о**льшую оценку, чем за то, чтобы этот экран взаимодействовал с остальной системой. А люди, отвечающие за безопасность, имеют собственные интересы, и им не нравится, когда мы вообще создаем какой-либо программный код.*

   Подобная разобщенность интересов — первый признак, что используются показатели, которые поощряют оптимизацию составляющих за счет всей системы. Это особенно типично для крупных организаций, где трудно определить, как лучше оптимизировать целое, поэтому взамен часто оптимизируют частности (или функции). Возможно, решение проблемы состоит в том, чтобы создать карту потока создания ценности для демонстрации, как показатели, поощряющие оптимизацию отдельных функций, противоречат главной цели создания потребительской ценности. Использование времени цикла потока создания ценности вместо упомянутых выше показателей может очень помочь развитию сотрудничества между подразделениями крупной организации.

3. *Нам не удается сделать так, чтобы молодые люди работали так же быстро, как и остальные члены коллектива. Поэтому у нас над наиболее критичными проектами работают более опытные люди, а молодежи поручается все остальное.*

   Коллектив должен быть "сочетанием" опыта и молодости. В коллективах, состоящих исключительно из опытных работников, нет никого, кому можно было бы поручить второстепенную работу (которая всегда найдется), нет никого, кто бы предложил что-то новое, и при этом опытным людям некому передать свой опыт. В коллективах, состоящих исключительно из молодежи, часто "изобретается много велосипедов" и, как правило, отсутствует общее видение проблемы.

Руководство

Во время соревнований команда из восьми гребцов нуждается в рулевом, который бы учитывал гидродинамику и аэродинамику, а также по ходу формировал и воплощал в жизнь стратегию гонки. Рулевой должен быть умным, уверенным, быстро думающим и при этом весить как можно меньше. Рулевой — это лидер команды, заставляющий гребцов грести синхронно с тем, чтобы лодка гладко скользила по воде и воздуху. Во время гонки задача рулевого — учитывать погоду, курс и положение соперников, принимать быстрые решения, направлять лодку и следить, чтобы гребцы экономно расходовали силы с тем, чтобы у них осталась энергия для финального рывка.

Крупный коллектив без руководства — это то же, что команда лодки без рулевого. Вероятно, команда смогла бы плыть быстро в определенном направлении, но трудно представить, что гребцам бы удалось самим добиться синхронности в гребле и что они смогли бы делать те мелкие коррекции курса, которые способствуют созданию продукта-победителя. В компании Toyota исповедывается базовый принцип, что коллективная работа важна, но кто-то должен отвечать за все.[13] Это не вызывает возражений. Слишком часто обнаруживалось, что "работа, которую должны делать все, нередко оказывается невыполненной". В главе 3 обсуждалась важность существования чемпиона (или лидера), который бы обладал глубоким пониманием потребностей рынка, а также мог бы выбрать технологию, которая способна удовлетворить этот спрос. Вопрос, который ставится сей-

[13] Там же, p. 103.

час, звучит так: "Нуждается ли коллектив разработчиков также и в административном руководителе (process leader)?". Сколько руководителей в действительности нужны коллективу и какой должна быть их роль?

Не следует ошибочно думать, что административный руководитель может быть заменой для технического руководителя (product и technical leadership). Об этом речи нет. Роль чемпиона (который одновременно знает технологию и рынок) остается незаменимой: в коллективе должны быть один-два человека, глубоко понимающих потребителя и глубоко знающих технологию. Если это два человека, они должны "говорить в один голос" и иметь общую ответственность за успех продукта. А когда организация создает и вводит в действие новый процесс, неплохой идеей может быть присутствие административного руководителя. Однако роль административного руководителя должна быть хорошо синхронизирована с ролью чемпиона — либо через тесное сотрудничество, либо путем объединения этих ролей в одном лице. Если уже имеются два человека, играющих роль чемпиона, три может оказаться уже слишком много. Если чемпион один, ему бы могло очень помочь содействие административного руководителя.

Руководители

1. *В коллективе, где используется методология гибкой разработки Scrum, имеется ScrumMaster, однако его роль состоит в административном управлении процессом. Не предполагается, чтобы он принимал решения, касающиеся продукции.*

   Мы не понимаем, почему функции ScrumMaster должны быть ограничены только административным руководством, поскольку техническое руководство (это понятие включат как знание технологии, так и рынка) так важно для успеха любого крупного проекта. Если не предполагается, чтобы ScrumMaster выполнял функции технического руководителя, следует возложить соответствующие обязанности на кого-то другого. После того как коллектив внедрит у себя гибкую (agile) методологию разработки ПО, роль административного руководителя снизится и соответствующему лицу вполне можно поручить выполнение также функций технического руководителя.

2. *В коллективе, где используется методология гибкой разработки Scrum, имеется Scrum Product Owner (менеджер продукта), однако он не очень вовлечен в процесс и не понимает по-настоящему технологию.*

   Чемпион или менеджер продукта должен быть полноценным членом коллектива, глубоко вникающим в его повседневные дела, который также глубоко понимает потребителя и технологию. Без сочетания всех этих качеств вы получите посредственную продукцию. Плохо, когда менеджер продукта (понимающий потребителя) не разбирается в технологии, не является членом коллектива и не заинтересован в достижении общих целей.

3. *А какова роль менеджера продукта?*

   Если присмотреться, какую роль играли менеджеры весьма успешных продуктов, обнаружится, что они обладали одновременно обширными техническими знаниями и пониманием рынка — качествами, присущими создателям всех продуктов, имевших громкий успех. Руководство Алана Мюлелли созданием модели Boeing 777 часто называли менеджментом продукта в лучшем смысле. Он был главным создателем авиалайнера, инновационных совместных коллективов конструкторов и производственников, а также неустанно стремился восхитить потребителей.

   С другой стороны, менеджер продукта, роль которого сводится в основном к обновлению графиков Ганта (Gantt) и раздаче указаний что делать, вероятно, будет не особенно ценным приобретением для коллектива.

Основанные на ответственности планирование и управление

У команд гребцов имеются тренеры. Тренер обычно плывет рядом на моторной лодке с мегафоном, отдавая указания и критикуя. Однако, когда начинаются соревнования, все, что остается делать тренеру, это сидеть и наблюдать. Основанные на ответственности планирование и управление функционируют по тому же принципу: наступает время, когда следует прекратить указывать хорошо обученным работникам, что им делать, и предоставить им решать это самостоятельно.

Замечательно, как это работает в компании Toyota. Процесс разработки изделия имеет некоторые точки синхронизации (готовность, например, эскизов будущего автомобиля, моделей из глины, конструкторского проекта, прототипа и т.д.). Расписание таких событий устанавливается главным инженером проекта. Каждый причастный к данному проекту инженер и поставщик понимают, что соответствующие даты никогда не переносятся. Они знают точно, что от них требуется в каждой точке синхронизации, и всегда выполняют требуемое к этой дате. Никакие отговорки во внимание не принимаются. При этом предполагается, что инженеры сами найдут для себя все необходимое (например, соответствующую информацию), чтобы быть готовыми вовремя. В компании Toyota инженерные задачи, которые должны быть решены в каждой точке синхронизации, хорошо известны и стандартизированы. Никто ничего не контролирует. Повсеместно считается, что графики и планы никогда не нарушаются. Остальное — ответственность инженеров.

Это может показаться нереальным подходом, однако примеры успешных его применений можно найти в каждом крупном городе по всему миру. В каждом крупном городе имеется ежедневная газета, которую выпускает группа очень опытных людей, создающих новый продукт каждый день. Они (почти) никогда не опаздывают, даже если новости изменяются ежечасно, даже если компьютеры зависают, даже если приходится освещать местные бедствия. Поэтому, чтобы увидеть процесс создания продукта к определенному часу, посетите редакцию вашей местной газеты.

Когда имеется необходимость жесткого соблюдения производственных графиков, важно помнить, что люди всегда должны иметь время для выполнения работы к сроку, предписываемому графиком. Как отмечалось в главе 5, если необходимо, чтобы коллектив уложился в жесткие сроки, *следует ограничить объем работы способностью коллектива ее выполнить*.

Существует несколько способов осуществления этого. В компании Toyota прекрасно научились определять, сколько труда требует тот или иной цикл разработки, и стараются сделать так, чтобы нагрузка для всех подразделений была более или менее равной. В компании Toyota не делается попыток создания детализированных производственных графиков, но ожидается, что каждое подразделение или поставщик предоставит дополнительную помощь членам соответствующего коллектива, когда она потребуется.

В компании PatientKeeper (о которой шла речь в предыдущей главе) трудоемкость создания элементов разрабатываемого программного обеспечения оцениваются ежедневно и скрупулезно. Эти свежие оценки позволяют вовремя заметить, если объем работы превосходит возможности ее выполнить. Если такое имеет место, будут внесены соответствующие коррективы с тем, чтобы для текущей итерации было запланировано не больше работы, чем можно было бы выполнить к сроку.

Организации, занятые техническим обслуживанием, выполняют низкоприоритетные заявки только тогда, когда нет срочной работы. А когда такая работа появляется, она делается в первую очередь. Какие бы подходы не использовались, всегда следует помнить,

что невозможно осуществить планирование и управление (основанные на ответственности), если нет четкого представления о возможностях коллектива и объем работы не ограничивается этими возможностями.

Производственные графики и планы

1. *Нужно ли создавать графики и планы для задач?*

   Большинство графиков для решения задач (или производственных графиков) лишены гибкости. Если создавать подобные графики с использованием данных, которые имеют свойство со временем изменяться (например, даты завершения задач), реальность *всегда* будет в противоречии с таким графиком. А попытки исключить вариацию (variation) из сроков завершения (т.е. исключить возможность изменения этих сроков) является как раз тем неудачным подходом, против которого предупреждал Деминг, поскольку вариация является присущей системе.

   Вам придется изменить систему, и для этого имеются три возможности:

   А. можно предусмотреть в графике возможные варианты с моделированием по методу Монте-Карло; однако лучше всего это работает на уровне элементов системы;

   Б. можно использовать производственный график с буфером проекта и с применением метода критической цепи (см. главу 10);

   В. можно упростить себе жизнь, не занимаясь вовсе составлением графиков для задач; взамен используйте основанные на ответственности планирование и управление.

2. *Не нуждаемся ли мы в структурной декомпозиции работ (work breakdown structure)?*

   Проектные решения, касающиеся будущего программного обеспечения, играют ключевую роль. Иными словами, обязательно должны быть определены "контуры" основных подсистем и их интерфейсов. Однако в данном случае целесообразнее говорить о структурной декомпозиции функциональных возможностей (feature breakdown structure), а не структурной декомпозиции работ. Вместо того чтобы думать о стоящих задачах, определите, какие именно функциональные возможности должны быть продемонстрированы и когда. Затем разработку этих возможностей следует возложить на соответствующие коллективы (при условии, что они (коллективы) обладают необходимыми квалификацией и средствами для выполнения данной работы) и предоставить этим коллективам самим организовать ее выполнение.

3. *А как с зависимостями?*

   Прежде всего не следует предполагать, что все зависимости реальны. Задача управления зависимостями должна быть выполнена в рамках определения проектных решений для системы, а не планирования.

   На высоком уровне зависимости нейтрализуются путем разработки крупных подсистем в логической последовательности. На более низком уровне зависимости между подсистемами должны быть нейтрализованы путем обмена информацией между разработчиками и коллективами; кроме того, должны быть предусмотрены частые точки синхронизации. Предполагается, что коллективы разработчиков и индивидуумы таким образом организуют свою работу, чтобы взаимозависимые задачи решались с учетом общих целей. Простой и эффективный обмен информацией в сочетании с прозорливыми проектными решениями для системы и "множественным" подходом к проектированию (см. соответствующий раздел в главе 7) являются лучшими методами для управления зависимостями.

Визуальная рабочая среда

Аэропорт — это удивительное место, особенно узловой аэропорт. Вы можете приземлиться в одном самолете и двумя часами позже улететь в другом, при этом ни с кем не

разговаривая, не пользуясь компьютером и не звоня по телефону. Это ваша задача пересесть с одного рейса на другой вовремя, не пользуясь без необходимости какой-либо помощью от людей, технологиями или средствами коммуникации. Вместо того чтобы содержать кого-то (или что-то), кто подсказал бы вам, как пересесть с одного рейса на другой, аэропорт просто максимально облегчает для вас (и для всех прочих пассажиров, которых всегда множество в аэропорту) задачу — определите это сами.

Прежде всего вы посм*о*трите на свой билет, где должно быть указано, какой рейс вам нужен. Кроме того, билет сообщит вам массу других подробностей: какая авиакомпания осуществляет нужный вам рейс, номер рейса, а также планируемое время вылета. (В производстве мы бы назвали такой билет канбаном (kanban — карточка с инструкциями для следующей работы)). Итак, вам необходимо выяснить, откуда состоится вылет нужного рейса и как попасть на него вовремя. Ответить на эти вопросы вам поможет большое табло, на котором указано, к какому терминалу вам следует пройти и на какое время в действительности назначен нужный рейс. В более крупных аэропортах можно найти еще одно табло, на котором будет указано, сколько минут вам потребуется, чтобы дойти до нужного терминала (чтобы помочь вам решить, идти ли пешком, или воспользоваться электрокаром). Далее вы руководствуетесь хорошо заметными указателями, указывающими путь к вашему терминалу, откуда предполагается вылет нужного рейса. Наконец, вы см*о*трите на табло у терминала, чтобы убедиться, что здесь производится посадка именно на тот рейс, который указан в вашем билете.

Билеты, табло и указатели используются в аэропортах широко. Но чего нет в большинстве аэропортов, так это некой информационной системы с экраном, чего-то, что позволило бы получить общее представление о том, как функционирует аэропорт. Это не мешает, пока все идет хорошо. Однако, если случится что-то непредвиденное, вы можете оказаться в сложном положении.

Альтернативный путь домой

Однажды утром, несколько лет назад, я прибыла в аэропорт Санта-Анна (Santa Ana) и тут же обнаружила, что лайнер, выполнявший один из рейсов, несколькими минутами ранее съехал со взлетно-посадочной полосы. Никто не пострадал, но все взлеты и посадки были временно отменены. В результате моя задача изменилась с пересадки на иной рейс на поиск способа попасть домой до полуночи. Первый вопрос, на который следовало ответить, был: насколько вероятно, что в конце концов объявят посадку на мой рейс? В отличие от большинства аэропортов, в аэропорту Санта-Анна *была* общая информационная система (dashboard), которая показывала время и терминалы для всех посадок и вылетов. Кроме того, я скоро выяснила, что мой рейс, вместо того, чтобы приземлиться здесь, был перенаправлен в Лос-Анжелес, до которого ехать 45 минут. Итак, я немедленно перерегистрировала билет на лос-анжелесский рейс, села в автобус, предоставленный авиакомпанией, успела на нужный мне рейс и смогла попасть домой даже немного ранее первоначально планировавшегося времени. К сожалению, многие пассажиры не сориентировались так быстро, как я, и им пришлось провести ночь в Санта-Анна. (Мэри Поппендик)

Самоуправляющийся производственный процесс

Когда быстро происходит множество событий, управленческая стратегия должна с диспетчеризации (dispatching) перейти на разблокирование (или снятие запретов — enabling). Работа такси, например, управляется диспетчером. Я звоню из дому; приезжает такси и везет в аэропорт. И так во многих городах, включая и мой родной Миннеаполис, такси предоставляет быстрый способ добраться до аэропорта. Теперь предположим, кто-

то вышел из гостиницы в центре Миннеаполиса, сел на метро и быстро доехал до аэропорта (вместе со множеством других пассажиров). Это делает для многих людей излишним звонить диспетчеру, ожидать такси, затем (одному) ехать, надеясь, что машина не застрянет в пробке по дороге к аэропорту.

Диспетчеризация предполагает планирование каждого этапа чьей-то работы. Сделать производственный процесс самоуправляющимся (self-directing) означает организовать среду, в которой люди смогли бы сами понять, как следует выполнять их работу, не ожидая указаний. Когда быстро происходит множество событий, самоуправляющийся процесс — это единственный подход, который жизнеспособен и эффективен. В запруженном автотранспортом центре города часто проще "поймать" такси на улице, чем звонить диспетчеру, а затем ожидать у гостиницы. Если менеджеры действительно хотят, чтобы все делалось быстро, им следует сосредоточиться на организации работы таким образом, чтобы производственный процесс был самоуправляющейся.

Когда производственный процесс самоуправляющийся, каждый, пришедший на работу утром, может сам быстро определить, что ему следует делать, не ожидая, пока кто-то это скажет. По завершении текущей работы немедленно становится ясно, что следует делать дальше. Если в течение дня у людей возникают вопросы, рабочая среда организована таким образом, что можно найти ответы, просто осмотревшись. В подобной среде задачу можно представить в виде пакета, отправленного по почте. Вы присоединяете к пакету инструкции (в виде адреса получателя) и с этого момента каждый человек и каждая машина, которые имеют дело с вашим пакетом, могут просто взглянуть на инструкции и тут же выяснить точно, что с ним следует делать.

Если вы менеджер, который в недоумении, чем ему заняться, когда планирование и управление (основанные на ответственности) освободят его от большинства имеющейся на руках в данный момент работы, вот для вас новое задание: организуйте рабочую среду так, чтобы производственный процесс стал самоуправляющимся. Имеется три ключевых уровня информации, необходимой при организации самоуправляющейся рабочей среды: канбан, андон и наглядное информирование.

Канбан

Кан — это японское слово, которое означает *карточка*, а *бан* — это также японское слово, означающее *сигнал*. Поэтому *канбан* — это *сигнальная карточка*. В аэропорту ваш билет является сигнальной карточкой, которая сообщает (вам персонально), на какой рейс вам следует сесть. Индексные карточки (index cards) — это также хороший пример сигнальных карточек. Каждая карточка содержит указание на небольшой объем работы (например, разработка "истории" (см. главу 8)), а также какие-то объясняющие суть примеры, доступные для контроля). Можно легко менять последовательность карточек с тем, чтобы кто-то, когда ему нужно знал, что делать дальше, мог выбрать карточку вверху стопки и сразу приступить к работе, которая на ней указана. Карточки могут быть бумажными, которые удобно прикреплять к физической доске объявлений (это удобно, когда все члены коллектива работают в одном месте), или они могут быть электронными (этот вариант полезен, когда члены коллектива географически рассредоточены). По количеству карточек можно судить о длине очереди дожидающейся выполнения работы. Когда карточек становится слишком много, вероятно, очередь работы должна быть сокращена.

Трудность, связанная с использованием *канбана* не в том, чтобы понять, каким образом люди будут выбирать, что им делать дальше. Это самое простое. Трудность в том,

чтобы обеспечить корректное содержимое каждой карточки и ее местоположение, чтобы, когда карточка будет выбрана, она бы содержала достаточное описание работы (которая была бы приемлемого объема и как раз тем, что следует делать дальше). Карточка — это не спецификация работы; это сигнал (или уведомление), что предстоит следующая работа, которая должна объединить вместе соответствующих специалистов.

Какой механизм использовать, чтобы сделать производственный процесс самоуправляющимся, — вопрос непростой. Каждая работа должна быть достаточно описана, набор работ должен включать все работы, расположенные в нужной последовательности, а правила предложения и выбора работ должны обеспечивать, что эти работы будут выполняться людьми, обладающими соответствующей квалификацией.

Организация работы

Как это работает на практике?

1. Предположим, некоторая компания собирается через четыре месяца создать дочернюю компанию. В качестве первого шага определяется набор сравнительно независимых задач, которые придется решать в ходе предстоящей реорганизации. Одна из этих задач состоит в том, чтобы разделить серверы и содержащиеся на них данные с тем, чтобы каждая новая компания имела собственные серверы и собственную информацию. Организуется коллектив из восьми человек, представляющих все специальности, необходимые для выполнения данной задачи.
2. Первое, что делает вновь созданный коллектив, — это на "клеящихся" бумажках на стене указывает все возможные действия, необходимые для раздела серверов и данных. Целью на этом этапе является создание общего представления о характере и объеме работы (в этот момент речь не идет о подробном описании с указанием сроков). При этом коллектив должен включать людей всех нужных специальностей, которых объединяет общая цель выполнить работу вовремя, и процесс организуется таким образом, чтобы выявить все задачи, необходимые для выполнения работы качественно.

   Когда все уверены, что заметки на стене представляют все, что должно быть сделано, коллектив организует их (заметки) в логичное расписание (или производственный план) — устраняя при этом зависимости, где только можно, и выявляя и учитывая реальные зависимости.

   Это всего лишь "эскизный" план работы; следующий шаг состоит в том, чтобы определить, что должно быть сделано в следующем месяце, и все включенные сюда задачи представить более подробно. (В беспокойной среде реорганизации компаний подробное планирование более чем на месяц вперед обычно является пустой тратой времени.) Наконец, коллектив приступает к решению стоящих задач. Разработчики должны создать (и протестировать) сценарии упорядочения данных и приложений; те, кто отвечает за аппаратуру, должны приобрести соответствующие серверы (однако прежде кто-то должен экспериментальным путем определить подходящую конфигурацию); те, кто отвечает за безопасность, должны разработать несколько методов доступа к системе с различными правами, и это целесообразно сделать сразу, чтобы осталось время для тестирования.

   Выполнение этих задач означает, что каждая работа должна иметь поддающийся проверке результат; что все согласны в том, что должно быть сделано и протестировано к концу месяца и что коллектив собирается ежедневно, чтобы выяснить, в каком состоянии различные задачи и кто нуждается в помощи. В конце месяца все собираются вместе (к этому моменту месячный план должен быть выполнен) и рассматривают общее состояние дел, а также вносят коррективы в планы, если необходимо. Затем коллектив обращается к следующему набору задач (и критериев проверки).

Андон

Андон — это переносной японский фонарь, сделанный из бумаги, растянутой на каркасе из бамбука. В компании Toyota слово *андон* используется для обозначения шнура, за который рабочие могут потянуть, чтобы "остановить линию" (останов линии обычно сопровождается миганием лампочек, призванных привлечь внимание к возникшей проблеме). Идея использования этого шнура в том, чтобы делать проблемы видимыми для немедленного их устранения.

С течением времени слово *андон* начало также использоваться для обозначения любой доски объявлений (или иного предмета для публичной демонстрации сообщений), содержимое которой может легко изменяться. (Так же иногда называют табло на железнодорожных станциях, где указывается время прибытия и отправления.) Андон призван привлекать внимание к любой аномалии, оперативно отображать сообщения об изменившейся ситуации и указывать расположение объектов, которые часто его меняют (например, какой сервер в настоящий момент используется для тестирования).

Визуальные сигналы

Андон в разработке программного обеспечения может представлять собой две лампы (красная и зеленая), которые переключаются в зависимости от результатов сборки и тестирования. Если сборка модулей оказалась успешна и прошла тестирование, в этом случае весело подмигивает зеленая лампа. А если вдруг замигала красная лампа, в этом случае разработчики знают, что обнаружена проблема. Конечно, система *андон* может быть реализована на основе электронной почты, однако в этом нет ничего интересного.

Наглядное информирование

Людям нравится работать в успешных коллективах. Приятно добиваться успеха при выполнении локальных задач (например, в разработке электронной управляющей системы для модели 777). Однако по-настоящему почетно сделать вклад в успех всей программы — например, наблюдать за первым взлетом Bouing 777 и знать, что его управляющую систему разработал ваш коллектив. Алан Мюлелли часто пытался собрать всех, причастных к созданию модели 777, вместе, чтобы они могли увидеть, как продвигается создание нового авиалайнера. Самая знаменательная встреча состоялась за два года до первого полета по расписанию, когда первая машина сошла со сборочной линии (точнее, выкатилась на собственных колесах). Мюлелли пригласил на знаменательное событие 10000 (или около того) рабочих и членов их семей, чтобы поздравить каждого с их общими успехами и отдать должное их постоянной приверженности общей цели.

Для коллективов важно видеть общий прогресс их работы в своем собственном контексте, а также в более широком контексте целей всей компании. Для этого мы используем разного рода средства наглядного информирования. В комнате каждого коллектива на стенах должны быть большие, наглядные диаграммы, демонстрирующие всем (членам коллектива, а также случайно зашедшим в комнату), насколько успешна (или неуспешна) деятельность данного коллектива. А диаграммы, отображающие общее положение дел и результаты усилий всех коллективов, должны находиться в главном "командном пункте" (как называют соответствующую комнату в компании Boeing), где их смогут увидеть все. "Географически рассеянные" коллективы также должны иметь доступ к такой информации, и здесь очень полезными будут электронные средства подобного "наглядного инфор-

мирования". Электронные средства также полезны для отображения ключевых показателей, которые легче генерировать электронным способом, чем вручную.

На наших курсах мы часто задаем вопрос: "Как в вашей организации можно узнать, каков прогресс в данный момент в выполнении общих планов?" (см. упражнение 4 в конце данной главы.) Мы были удивлены большим числом ответов: "В действительности, подобная информация не очень-то доступна." Организации, где людей не заинтересовывают в конечных целях, упускают прекрасную возможность сделать так, чтобы отдельные работники и коллективы в целом с энтузиазмом начали прилагать усилия для их достижения.

Побудительные мотивы

В своей книге *The Living Company*[14] ("Компания-долгожитель") Эри де Джуз (Arie de Geus) обсуждает, почему одни компании живут долго (иногда столетие и более), а другие нет. Он пишет: "Компании умирают потому, что их менеджеры сосредотачиваются на производстве товаров и услуг и при этом забывают, что истинная природа их организаций та же, что и природа сообществ людей.[15] То, насколько люди вовлечены, испытывают доверие и выкладываются в работе, оказывает прямое влияние не только на результаты труда, но и на продолжительность жизни компании."[16]

Согласно де Джузу, существует два вида компаний: стремящиеся разбогатеть и плывущие по течению. Целью первых является получение максимальных результатов при минимуме затрат и достижение богатства для группы менеджеров и инвесторов. А цель вторых состоит в том, чтобы продолжать плыть, т.е. оставаться в бизнесе и предоставлять работу в долгосрочной перспективе. Например, Сакичи Тойода и Кийитиро Тойода (см. главу 1) в качестве главной цели рассматривали предоставление работы японским рабочим (для этого им приходилось определять, как производить сложную продукцию, которую в противном случае пришлось бы импортировать).

В стремящейся разбогатеть компании, утверждает де Джуз, имеется неписаный контракт между компанией и индивидуумом: индивидуум предоставляет свое умение в обмен на оплату. А в компании, плывущей по течению, неписаный контракт совсем другой: индивидуум предоставляет свою приверженность целям компании в обмен на то, что компания сделает все, чтобы *максимально развить потенциал каждого индивидуума*.[17] Приверженность — это улица с двусторонним движением: люди дорожат компанией, если они ощущают, что компания дорожит ними.

Оценка эффективности

Вероятно, наиболее игнорируемым советом Деминга является следующий: ликвидировать ежегодные оценки индивидуальной эффективности (annual performance ratings) для работников, которые на окладе; не следует подрывать сотрудничество в коллективе, награждая за индивидуальную эффективность. Деминг был непреклонен в своей вере, что

[14] Geus A. *The Living Company*, Harvard Business School Press, *1997.*
[15] Там же, p. 3.
[16] Там же, p. 10.
[17] Там же, p. 118.

персональные годовые оценки заслуг скорее создают соперничество, чем сотрудничество, и подавляют профессиональную гордость. Однако в зависимости от неписаного контракта между индивидуумом и компанией, оценка индивидуальной эффективности служит различным целям. В компании, стремящейся разбогатеть, где упомянутый контракт предусматривает обмен квалификации на оплату, целью оценки индивидуальной эффективности является определение размеров оплаты, причитающейся индивидууму. Действительно, такая цель имеет явную тенденцию воспитывать, скорее, соперничество, чем сотрудничество.

С другой стороны, когда компания нацелена на максимальное развитие потенциала каждого индивидуума, оценка эффективности может служить для размышления, каков этот потенциал в данный момент, и что нужно сделать (компании и индивидууму) для его развития. Таким образом, система оценок помогает определить, следует ли данному индивидууму развиваться как техническому специалисту или он будет эффективнее как менеджер; какие тренинги и виды работ лучше всего подходят для его развития в предстоящем году; что в первую очередь нуждается в улучшении (например, умение говорить публично) для дальнейшего карьерного роста.

Ежегодные оценки индивидуальной эффективности не должны удивлять работодателей неожиданной обратной связью. Оценка эффективности должна производиться чаще одного раза в год или даже одного раза в квартал. Если ежегодная оценка — единственная возможность для работника узнать, каковы его успехи, значит, с системой оценок что-то неладно. Критерии оценок должны отдавать предпочтение коллективной работе, делая вклад в коллективный успех таким же важным (если не важнее), как и индивидуальные достижения. Вы получаете то, что поощряете; если вы найдете эффективные способы поощрения сотрудничества, сотрудничество будет развиваться.

Персональные оценки навязывают впечатление, что улучшать работу необходимо только оцениваемому работнику, однако Деминг настаивал, что большинство проблем с эффективностью исходят от менеджмента. Если служащий показывает низкую эффективность, первый вопрос, который менеджер должен задать себе, следующий: “Что я делаю не так?” Менеджеры должны нести личную ответственность за эффективность их организации и сотрудничать с людьми, чтобы повысить эффективность системы.

Система оценок

Представьте коллектив, старающийся устранить ошибку, которая создала сложную проблему. В бережливой среде члены такого коллектива работают вместе, обсуждая идеи, экспериментируя, пока не будет найдено решение. А затем коллектив сделает все возможное, чтобы не искать виноватого. В конце концов, каждый из них мог допустить подобную ошибку. Вместо этого они будут продолжать работать, чтобы защитить систему от подобных ошибок с тем, чтобы данная проблема никогда не возникла вновь.

Теперь представим тот же коллектив, только теперь его члены знают, что один раз в год руководство собирается на засекреченное совещание, на котором оценивается деятельность каждого работника по его вкладу в успехи компании. От этих оценок зависят повышения зарплаты и продвижения по службе. В этих условиях логика диктует, что каждый член коллектива должен каким-то образом показать руководству, что он лучше своих коллег. Например, найти виновного в ошибке или приписать себе заслугу в ее устранении; иными словами, подобная система оценок очень вредит сотрудничеству. Если аналогичная система оценок существует в вашей компании, все бережливые инициативы

окажутся здесь по меньшей мере бессодержательными. Итак, система оценок поощряет соперничество, сокрытие негативной информации (чтобы выглядеть лучше) и сокрытие проблем (чтобы не выглядеть плохо).

Вознаграждение[18]

Ничто с большей вероятностью не вызовет разногласий и помещает сотрудничеству, чем система, используемая для определения вознаграждения, причем в данном случае речь идет о любой используемой системе, поскольку не существует совершенной системы определения вознаграждения. Мы предлагаем следующие советы, которые призваны помочь вам сориентироваться.

Совет 1: обеспечить, чтобы обоснованность повышений была неоспорима

В большинстве организаций значительные повышения зарплаты ассоциируются не с ежегодными регулярными ее повышениями, а с продвижениями по службе (когда переведенный на более высокую должность работник переводится в более высокую категорию тарифной сетки). Там, где продвижения по службе исключены (например, для большинства учителей), ежегодные повышения зарплаты приводят к напряженности в коллективе, поскольку они — единственный способ получать больше денег. А когда повышения в должности реальны, служащие склонны придавать меньше значения ежегодным повышениям зарплаты и все свои ожидания связывать с повышением в должности. Подобная система поощряет людей переходить в менеджмент, когда они исчерпывают возможности повышения, работая как специалисты. Компании часто решают эту проблему, предлагая техническим гуру зарплату на уровне менеджмента.

Основой любой системы поощрений является тарифная сетка, соответствующая стандартам отрасли и данного региона. Люди должны правильно "оцениваться" согласно этой сетке, чтобы их квалификация и ответственность соответствовали требованиям данной профессии и данной должности. Исходное определение на должность, а также решения о повышении должны приниматься командой менеджеров после тщательного рассмотрения.

Как правило, должности имеют оригинальные названия и о переводе на новую должность становится широко известно. Должность того или иного работника обычно рассматривается как открытая информация. Если служащие правильно "размещены" по должностям, и повышение происходит только когда они определенно соответствуют новой должности, в этом случае повышение зарплаты, соответствующее новой должности, как правило, считается обоснованным. Таким образом, коллектив может включать одновременно сотрудников, работающих как на "старших", так и на "младших" должностях, руководителей и высококвалифицированных специалистов, зарплаты которых различаются. До тех пор пока система назначений на должности и повышений прозрачна и считается справедливой, различный уровень оплаты редко является источником проблем.

---

[18] Poppendieck M. *Unjust Deserts*. Better Software Magazine, *august 2004*.(Материал данного раздела частично взят из указанной статьи.)

Совет 2: не придавать особого значения ежегодным повышениям зарплаты

Когда основным средством для существенных повышений зарплаты является продвижение по службе, важно уделить максимальное внимание тому, чтобы порядок повышений был справедлив. А что касается системы оценок, определяющих ежегодные повышения зарплаты, лучше всего не прилагать особых усилий для какой-либо классификации людей. Целесообразно, в основном, использовать эту систему, чтобы обеспечить, что каждый работник находится на соответствующем уровне, согласно тарифной сетке. Оценки могут выявить тех, кто созрел для повышения, и отстающих; соответственно, эти оценки должны служить основанием для повышения или поводом для работы с отстающими. Четырех уровней для оценки обычно достаточно, и компетентный руководитель, располагающий хорошими оценочными критериями и информацией из соответствующих источников, может дать справедливые оценки, которые бы служили соответствующим целям.

Даже если ежегодные повышения зарплаты не связаны жестко с заслугами, оценки всегда имеют большое значение для служащих, поэтому следует, чтобы они были справедливыми и сбалансированными. В течение последнего десятилетия сбалансированная система показателей (ССП) (balanced scorecards) стала широко использоваться для оценки менеджмента (по крайней мере так считается). Эта система обеспечивает оценку работы менеджера по множеству аспектов. Простейший вариант ССП может также использоваться для оценок, которые влияют на ежегодные повышения зарплаты (чтобы подчеркнуть, что люди должны хорошо выполнять свои обязанности по многим показателям, чтобы их работа была эффективной). Информация для ССП должна поступать отовсюду: от коллег, заказчиков, руководителей. Важно, чтобы служащие поняли, что информация, используемая для ССП, обоснованна и беспристрастно освещает множество аспектов их работы. Также важно избегать излишней сложности, поскольку сложность привлечет неоправданное внимание к системе оплаты (которая работает лучше, когда она понятна).

Совет 3: награждать в зависимости от вклада в общий результат, а не за личные достижения

Не существует фактора, оказывающего большее отрицательное воздействие на мотивацию, чем система вознаграждения, которая воспринимается как несправедливая. Причем неважно, справедлива или нет система в действительности. Если возникло ощущение несправедливости, тогда те, кто считает, что их обошли, быстро теряют мотивацию. Люди усматривают несправедливость, когда они не получают премий, которые, как они полагают, ими заслужены. Совершенные продукты создаются совместными усилиями многих людей, поэтому в процессе разработки при распределении индивидуальных премий неизбежно имеются недовольные люди, которые полагают, что с ними поступили несправедливо. Более того, если индивидуальные премии поощряют соперничество в среде, где для успеха важно сотрудничество, даже те, кто получает премии, вряд ли ценят это.

Нам не нужны премии!

В одной компании руководство планировало выплату поощрительных премий разработчикам, которые соблюли индивидуальные сроки и выполнили производственные задания. Поскольку руководители, планировавшие премию, были из отдела маркетинга, они были наиболее привычны к вознаграждению, зависящему от итогов индивидуальной деятельности (commission-based compensation) и пытались создать некоторое подобие этого для инженеров.

Разработчики были шокированы, услышав об этом. "Если сейчас кто-то приходит ко мне и задает вопрос, я рад помочь. — сказал один из них. — А при подобной поощрительной системе с моей стороны было бы глупо тратить время, помогая кому-то. При этом я не только заработаю меньше денег, поскольку в это время я не только не буду занят моей работой, но и помогу другим выглядеть лучше и заработать больше."

К счастью, на собрании был менеджер коллектива разработчиков, который слышал все эти замечания. Позже он настоял, чтобы предлагаемая поощрительная система была пересмотрена. (Мэри Поппендик)

Общепринятая мудрость гласит, что люди должны вознаграждаться на основе зависящих от них результатов. Однако в условиях, когда важны совместное использование информации и сотрудничество, вознаграждение должно зависеть от вклада в общий результат, а не от личных достижений. Например, тестировщики не должны вознаграждаться за количество дефектов, которые они выявили; взамен разработчики и тестировщики должны вознаграждаться вместе за способность коллектива создавать свободный от дефектов программный код. Разработчики не должны вознаграждаться за технический успех их усилий; взамен весь коллектив должен быть вознагражден за экономический успех результатов их работы.

Совет 4: найти лучшую мотивацию, чем деньги

Хотя денежные вознаграждения могут быть мощным стимулом, мотивации, которые они создают, недолговечны. Как только люди достигают адекватного заработка, ими движут более мощные мотивации: успех, профессиональный рост, контроль над собственной работой, признание, повышения и дружественная рабочая среда. Как ни была бы хороша система оценок и вознаграждений, не следует ожидать от нее выдающейся эффективности в долгосрочной перспективе.

Сомнительная премия

Вице-президент одной машиностроительной компании рассказал мне о том, как компания выдала премию 100 тыс. долл. коллективу из 14 человек. Членам коллектива было предложено разделить эти деньги так, как они считают нужным. Казалось бы, задумано неплохо, но для коллектива оказалось невозможно прийти к общему мнению. В итоге деньги оказались поделены поровну между всеми членами коллектива, но многие были недовольны, так как полагали, что это несправедливо. (Мэри Поппендик)

В своей книге *Hidden Value*[19] ("Скрытая ценность") Чарльз О'Рейли (Charles O'Reilly) и Джеффри Пфеффер (Jeffrey Pfeffer) рассказывают о нескольких компаниях, которые добивались великолепной эффективности от обыкновенных людей. В этих компаниях на первое место ставились (и определяли все действия на всех уровнях) ценности, ориентированные на человека. Здесь инвестировали в людей, предоставляли широкий доступ к информации, полагались на коллективы и больше ценили лидерство, а не администрирование. Наконец, в этих компаниях не использовали деньги как основное средство мотивации; здесь старались вызвать интерес к работе, стремление расти и добиваться успехов.

Рассматривайте денежные вознаграждения как взрывчатые вещества, поскольку они имеют мощное воздействие независимо, нужно это вам или нет. Поэтому используйте их

[19] O'Reilly C., Pfeffer J. *Hidden Value: How Great Companies Achieve Extraordinary Results with Ordinary People*, Harvard Business School Press, *2000*.

умеренно и с осторожностью. Они, скорее, доставят вам неприятности, чем разрешат ваши проблемы. Как только вы пойдете по пути денежных вознаграждений, не исключено, вы не сможете повернуть назад, даже когда эти вознаграждения потеряют эффективность, как это неизменно и случается. Старайтесь, чтобы люди получали справедливое и адекватное вознаграждение, а затем используйте иные, более эффективные способы повышения эффективности.

Попробуйте это

1. Вспомните *систему глубокого знания* Деминга, которую мы перефразировали следующим образом:
   - **а)** имеет значение продукт в целом, коллектив в целом или система в целом;
   - **б)** когда что-то плохо, скорее всего, это вызвано системой, а значит, это проблема менеджмента;
   - **в)** используйте научный метод для изменений и улучшений;
   - **г)** там, где дело касается людей, имеют значение мастерство, гордость, опыт, уверенность и сотрудничество.

   Представьте, что Деминг собирается посетить вашу организацию на следующей неделе. Подготовьте для него презентацию, объясняющую, как в вашей организации применяется каждый из этих принципов. Как вы думаете, каков будет его совет? Будете ли вы готовы воспользоваться этим советом?
2. Обсудите в вашем коллективе четырнадцать заповедей Деминга. Для каждой из них определите: актуальна ли эта заповедь сегодня для вашего коллектива? Если заповедь актуальна, что вы должны сделать иначе? Что потребуется для изменения?
3. Как в вашей организации формируются и управляются коллективы? Это действительно коллективы или просто рабочие группы? К какому количеству коллективов одновременно в вашей организации обычно принадлежат отдельные индивидуумы?
4. Заполните таблицу оценки визуальной рабочей среды (Visual Workspace Assessment):
   - **а)** во втором столбце запишите ответы (истинные для вашей организации) на вопросы, содержащиеся в первом столбце;
   - **б)** в третьем столбце оцените, насколько в вашей организации производственный процесс самоуправляющийся (используйте систему оценок 0–5, где 0 означает, что людям нужно постоянно говорить, что им делать, и 5 — люди всегда сами определяют, что им делать дальше);

Вопрос	Текущая практика	Оценка
1. Откуда люди знают, чего в действительности хотят потребители?		
2. Как люди получают ответы на технические вопросы?		

3. Как люди выясняют, над чем именно им работать дальше?		
4. Как люди выясняют, над какими дефектами им следует работать дальше?		
5. Как люди узнают, успешно ли прошли испытания?		
6. Как люди узнают о своем продвижении к цели работы?		

5. В следующий раз, когда ваша компания будет осуществлять оценку служащих, добавьте следующие вопросы:

а) считаете ли вы систему вознаграждений справедливой;

б) считаете ли вы систему продвижений по службе справедливой;

в) оцените, как вознаграждение влияет на вашу приверженность работе (воспользуйтесь пятибалльной шкалой оценок).

1. Существующая система вознаграждений мешает мне работать.

2. Она иногда раздражает и время от времени мешает работать

3. Она никак не влияет ни на меня, ни на мою работу.

4. Она время от времени мотивирует меня работать лучше.

5. Она мотивирует меня усердно работать каждый день.

Глава 7

Знание

Создание знания

Раньше самым важным ресурсом была земля. Тот, кто владел землей, контролировал все. В какой-то момент на первый план вышло умение заниматься коммерческой деятельностью, и купцы стали богаче землевладельцев. Позже промышленная революция привела к тому, что главным ресурсом стал капитал, и власть стала зависеть от капитала. Сегодня на первое место выходит знание: техническое, управленческое, административное и маркетинговое. Значительная часть этого знания представлена в виде программного обеспечения.

Наша нынешняя система взглядов и понятий по-прежнему ориентируется на промышленность и финансы. Однако доминировать в текущем столетии будут организации, где больше всего ценится знание.

В чем ваша проблема?

В 1950 году компания Toyota почти обанкротилась и была вынуждена уволить треть своих служащих. Решимость компании никогда не допускать подобной болезненной ситуации в будущем привела к появлению производственной системы Toyota. Эта система по определению была и остается ориентированной на сокращение расходов. Спрос определяется рынком, полагают в компании, и рынок устанавливает цену. Прибыль — это остаток от вычитания расходов из цены, поэтому, чем ниже расходы, тем выше прибыли, и, следовательно, выше шансы оставаться в бизнесе и сохранять рабочие места.

Компания Toyota не располагала большими деньгами или иными ресурсами, когда была создана производственная система Toyota. Эта система имела четыре основные цели.

1. Предоставлять потребителю товары и услуги максимально высокого качества.
2. Развивать потенциал каждого служащего на основе взаимного уважения и сотрудничества.
3. Сокращать расходы посредством ликвидации потерь в каждом процессе.
4. Создавать гибкие производства, способные реагировать на изменения рынка.

Производственная система Toyota — это подход, с помощью которого компания Toyota планировала достичь перечисленных целей.

Арт Смайли (Art Smalley) — один из немногих американцев, который изучал производственную систему Toyota, работая в этой компании в Японии в 80-х годах. Он заме-

тил, что бережливые инициативы не нашли широкого применения в этой компании, и старался выяснить, с чем это связано. В результате он выделил две причины: 1) имеющиеся бережливые реализации, кажется, больше уделяли внимания средствам и методам, чем решению основных проблем; и 2) обучение бережливому методу, похоже, было больше предназначено для специалистов высокого ранга, чем для улучшения навыков решения проблем в трудовых коллективах.

Смайли полагал, что любая попытка улучшить дело должна начинаться с двух основных вопросов.

1. Как именно вы намереваетесь получать прибыль и удовлетворять ваших заказчиков?
2. В чем ваша главная проблема?

Бережливые инициативы должны всегда начинаться с ясного в*и*дения, как вы "делаете деньги", и четкого понимания наиболее критичной проблемы, которая мешает этому. При создании ПО этой проблемой может быть излишняя сложность программного кода. Проблема может заключаться в том, что не каждая новая видеоигра достаточно забавна. Это может быть слишком медленная реакция системы. Это может быть недостаточная интеграция аппаратуры и встроенного программного обеспечения. Какова ни была бы проблема, бережливость начинается с четко определенной главной проблемы, которая мешает успеху бизнеса.

Следующий шаг состоит в том, чтобы использовать упорядоченный (disciplined) метод ее разрешения. Следует обучить всех участников процесса, как это делается, с учетом каждого аспекта их работы. Создайте среду, в которой коллективы на всех уровнях компании продолжат использование упорядоченного метода для решения крупнейшей из имеющихся в данный момент проблемы, на что, возможно, уйдет не один год. Это нелегкая работа, требующая перспективного мышления. Тем не менее это путь, который проторила компания Toyota и который привел ее к многолетнему успеху.

Научный метод мышления

Том имеет ученую степень доктора философии (Ph.D). Наш сын Дастин имеет степень доктора в области инженерии окружающей среды. Каждый из них провел несколько лет, используя научный метод (под руководством известного профессора) для создания и публикации знания. Как известно каждому ученому, научный метод состоит из следующих этапов.

1. Наблюдать и описать явление или группу явлений.
2. Сформулировать гипотезу, объясняющую явление.
3. Использовать гипотезу для предсказания чего-то — существования иного явления или результатов новых наблюдений.
4. Провести серию экспериментов, чтобы проверить истинность предсказаний.
5. Если гипотеза подтверждается экспериментами, она может рассматриваться в качестве теории или правила.
6. Если гипотеза не подтверждается экспериментами, она должна быть отвергнута или изменена.

Научный метод является основой производственной системы Toyota. Каждого рабочего компании специально обучают использованию основных методов решения проблем в качестве главного подхода к выполнению своих обязанностей. Рабочие компании Toyota действуют как сообщество ученых, экспериментируя, постоянно учась и фиксируя приобретенные знания (для будущего использования) таким же образом, как это уже давно делают ученые.

На рис. 7.1 представлен (по пунктам) метод разрешения проблем, применяемый в компании Toyota (слева), и показано сходство этого метода с циклом PDCA (планировать (plan), делать (do), проверять (check), действовать (act)), о котором Деминг читал лекции в Японии в 50-х годах. В компании Toyota научный метод используется в качестве дополнения концепции останова производственной линии Сакичи Тойода и идеи поставок "точно вовремя" (Just-in-Time) Кийитиро Тойода. Результатом стала производственная система Toyota — упорядоченное разрешение проблем на низшем уровне (но в масштабах всей компании) с фокусированием в каждый момент на устранении наибольшей проблемы, мешающей получать прибыли.

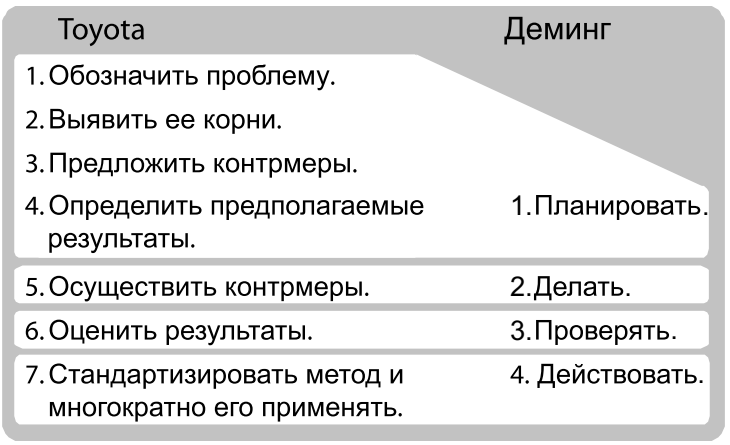


Рис. 7.1. Метод решения проблем и цикл PDCA

При этом важно помнить, что данный метод решения проблем никогда не используется в вакууме. Он используется последовательно и многократно для решения наибольшей в данный момент проблемы, с которой сталкивается коллектив. Как в любой дисциплинированной научной среде, предполагается, что коллектив фиксирует все свои наблюдения, гипотезы, результаты экспериментов. При этом считается, что неудачные эксперименты предоставляют по меньшей мере столько же знания, как и эксперименты, позволившие решить проблему.

Сохранение знания

"Знаете, какая наибольшая проблема с итеративной разработкой? — спросил нас менеджер. — Мне нравится экспериментировать, но очень трудно запомнить все перепробованные варианты. И потому возможна ситуация, когда одно и то же делается дважды и, может быть, большее число раз." Если знание не сохраняется так, чтобы им можно было в лю-

бой момент воспользоваться, тогда действительно люди забывают то, что они узнали. И даже если отдельные работники что-то помнят, организация не извлечет пользы из полученного знания, если оно не сохранено в виде, позволяющем воспользоваться им в будущем.

Но Сакичи Тойода и Кийитиро Тойода были изобретателями по своей сути. У них имелись патенты, выданные по всему миру. Продажа патентных прав на автоматизированный ткацкий станок предоставила Кийитиро начальный капитал, чтобы основать автомобильную компанию. Изобретатели, которые регистрируют патенты, хорошо научились сохранять все, чему они научились, поскольку процесс получения патента очень зависит от постоянного фиксирования информации о том, кто какими знаниями обладает, когда те или иные знания были приобретены и как удалось придти к данной идее. Поскольку предполагается, что блокноты изобретателей будут использоваться регулярно (самими изобретателями, а также их коллегами), все записи в них делаются тщательно и дополнены таблицами и графиками. Поскольку эти записи имеют юридическую силу и может так случиться, что они сыграют роль доказательства в патентном споре, изобретатели постоянно следят, чтобы они были корректны, полны и актуальны.

Мы все имели блокноты

Я имею два патента: US Patent 5,995,690 и US Patent 6,052,135.

На 3M (предприятии, по производству видеокассет) большие инвестиции направлялись на создание новых изделий, и эти инвестиции тщательно защищались путем поэтапного фиксирования всех разработок с целью получения патента впоследствии. Все технические служащие-новички, когда они приступали к работе, получали лабораторный блокнот, и обязанностью менеджера было научить их пользоваться им.

На практике мы учились заполнять эти блокноты, читая записи коллег. (Каждая запись в блокноте коллеги должна была быть прочитана, понята и подписана, предпочтительно в течение дня.) В этих блокнотах можно было увидеть изображенные вручную графики, эскизы экспериментальных устройств, таблицы с данными (часто в виде кратких компьютерных распечаток, приклеенных к странице блокнота), а также краткие выводы.

Мы быстро поняли, что когда мы делаем что-то, что генерирует знание, это должно фиксироваться в нашем блокноте. Когда блокнот был заполнен, мы регистрировали его в библиотеке и получали новый блокнот. К этому моменту важная информация уже была извлечена из блокнота и использована в дальнейших экспериментах, поэтому обычно не было необходимости обращаться к старым лабораторным блокнотам снова. Однако если бы такая необходимость возникла, найти нужную информацию можно было бы очень быстро. (Мэри Поппендик)

В разработке программного обеспечения тесты и программный код часто являются соответствующей комбинацией строгости и лаконичности, необходимых для документирования знания, содержащегося в программном обеспечении. Однако проведенные эксперименты и исследованные варианты, необходимые для принятия решений, касающихся находящегося в разработке программного продукта, легко забываются, поскольку подобная информация никогда не попадает в программное обеспечение. Просто записать что-то еще не означает превратить это в знание, и информация может так же легко потеряться в море излишней документации, как она теряется из-за отсутствия таковой.

The Knowledge-Creating Company[1] ("Компания, создающая знание") авторов Икуджиро Нонака (Ikujiro Nonaka) и Хиротака Такеучи (Hirotaka Takeuchi) — это книга о механизмах

[1] Nonaka I., Takeuchi H. The *Knowledge-Creating Company: How Japanese Companies Create the Dynamics of Innovation*, Oxford University Press, *1995*.

и процессах, с помощью которых знание создается в коммерческой компании. В книге отмечается, что в западных компаниях знание воспринимается как то, что записано. А в японских компаниях записанное знание считается только вершиной айсберга: бо́льшая часть знания содержится в субъективных представлениях, интуициях и интеллектуальных моделях. Это подразумеваемое знание приобретается не учебой, а через опыт. Его не так-то просто обработать с помощью компьютера или содержать в базе данных. Его трудно формализовать и передать другим.

Некоторые полагают, что если мы "документируем" то, что узнали в процессе итеративной разработки (т.е. записываем принятые решения; объясняем, почему они приняты, и перечисляем, что мы узнали нового), то создаем материал, который может быть использован в данной организации для обучения коллег. Однако, скорее всего, эти кипы документации будут просто собирать пыль (или занимать место на диске), совершенно бесполезные в качестве инструмента для учебы. (Точно так же мы часто ведем дневники, неосознанно ассоциируя глубину излагаемых в них идей с толщиной папки.)

Но для тех, кому придется пользоваться подобной документацией, толщина папки обычно обратно пропорциональна ее полезности в передаче знания. Для создания знания необходимо думать не о процессе его сохранения, а о людях, которые будут использовать то, что записано. Создатели документации (technical writers) являются специалистами в этом деле, поэтому логично было бы поручать им ключевую роль в сохранении ключевого знания.

Отчеты A3

В начале своей карьеры в компании Toyota инженеры учатся дисциплине отображения идей на листе формата A3[2]. Это приучает людей фильтровать и упорядочивать их мысли с тем, чтобы кто-то, кто будет читать такой отчет, смог получить ответы на все свои вопросы. Разные отчеты A3 имеют различные цели, но все они содержат важные знания, представленные в виде, облегчающем их хранение в базе данных, работу с ними в рабочей обстановке, передачу их менеджеру, а также использование их в будущих экспериментах.

Документация A3

Расположите рядом два листа бумаги формата A4, и вы получите формат A3. Одного листа бумаги такого размера должно быть достаточно для описания сути работы заказчика, различных сбоев системы или суммирования всех данных, необходимых для принятия решения. Пространные отчеты ведут к пустой трате времени и являются неэффективными, особенно когда они предназначены для передачи сложных идей.

Советы

1. Используйте минимум слов.
2. Одно изображение ст*о*ит тысячи слов. Старайтесь максимально использовать рисунки, графики, таблицы.
3. Вся информация должна помещаться на одной стороне листа формата A3.

---

[2] Morgan J., Liker J. *Toyota Product Development System* Productivity Press *2006* p. 269–276. Здесь можно найти множество образцов подобных отчетов; Liker J. *Toyota Way* McGraw Hill *2004* (p. 244–248). В этой книге также имеется один образец.

4. Отчеты А3 — это динамичные документы. Их использование и изменение основывается на обратной связи.
5. Если отчет не помещается на листе А3, сократите его до формата А4!
6. Содержимое и структура имеющейся информации должны соответствовать ее целям.
   - Отчет А3, содержащий решение проблемы, может включать:
     а) резюме проблемы;
     б) анализ причин ее возникновения;
     в) предлагаемую контрмеру;
     г) планируемые эксперименты;
     д) данные об измерениях и информацию обратной связи.
   - Отчет А3, предназначенный для распространения знания, может, например, включать:
     а) идентификацию определенной проблемы зависания базы данных;
     б) примеры зависаний вместе с описанием (подозреваемых) причин, вызвавших их;
     в) перечень известных причин, вызывающих зависание;
     г) обсуждение дополнительных причин, вызывающих зависание, (существование которых возможно).
   - Отчет А3, описывающий минимально полезный набор функциональных возможностей, может включать:
     а) описание работы заказчика, которую предполагается выполнять с помощью создаваемого программного обеспечения;
     б) данные об экономической ценности для заказчиков и разработчиков;
     в) оценку, как изменится работа заказчиков, когда они начнут применять этот новый набор функциональных возможностей;
     г) диаграмму, показывающую взаимодействие с другими системами;
     д) желательные сроки.

Век Интернета

Теперь, в век Интернета, в нашем распоряжении разнообразные методы создания и поиска знания. Мощные поисковые средства изменили способ индексации данных. Блоги предоставляют авторам возможность выхода к огромному числу людей, связанных с техникой. Википедия является моделью для быстрого и эффективного накапливания очень крупных объемов информации. В то же время какая-либо “система управления знанием” даже не упоминается в недавнем обзоре средств массовой информации, используемых людьми, которым приходится работать со знанием.[3] Средства поиска и публикации данных в век Интернета не освобождают нас от необходимости суммировать наше знание некоторым дисциплинированным и полезным образом. Но, без сомнения, они оказывают влияние на наши подходы к содержанию того, что мы знаем, в упорядоченном виде.

[3] См. McAffe A. *Enterprise 2.0: The Dawn of Emergent Collaboration*, MIT Sloan Management Review, *Spring, 2006*.

“Точно вовремя ”

Наш друг работает в коммерческой авиакомпании и живет в Инсбруке. Недавно он повышал свою квалификацию в Соединенных Штатах. Он рассказал нам, что основное внимание при этом уделялось принятию решений, поскольку исследования показывают, что причиной большинства аварий с самолетами, пилотируемыми одним пилотом, является принятие им неверных решений. Вот его рассказ.

> Когда вам приходится принимать решение (предположим, вы летите в тумане и необходимо решить, идти на посадку или нет), первое, что необходимо сделать, это определить, *когда именно* следует принять данное решение. Затем необходимо ждать до этого момента, прежде чем принять решение, поскольку именно тогда вы будете располагать максимальной информацией о ситуации. Но в таких случаях очень опасно *пропустить* нужный момент, так как тогда возникает большая опасность (например, столкновения со скалами, которых не видно из-за тумана)[4].

Разработка программного продукта — это процесс, связанный с приобретением знания, и чем дольше мы откладываем принятие решения, тем больше сможем узнать. Но мы не хотим столкнуться со скалами, поэтому главное не пропустить нужный момент. В организациях, занятых разработкой программного обеспечения, где применяется бережливый подход, решения целесообразно принимать с использованием принципа “точно вовремя” (Just-in-Time),

Программное обеспечение — это уникальный продукт (по самой своей природе предполагающий изменения), поэтому, как правило, мы можем откладывать решения достаточно долго. К сожалению, это свойство программного обеспечения поощряет некоторых разработчиков мешкать в начале процесса. Если не двигаться к конечной цели шаг за шагом, цель никогда не будет достигнута. Если мы не проводим эксперименты и не имеем обратной связи, вероятно, мы многому не научимся, а, в конце концов, целью любого процесса разработки программного обеспечения является создание знания (которое воплощено в ПО).

Параллельная разработка альтернатив

В мире разработки программного обеспечения обозначились два различных подхода к приобретению знания, основанные на генерации знания через итеративные эксперименты. В первом случае эксперименты производятся в ходе создания допускающей изменения системы (по мере приобретения знания, оно добавляется в систему с помощью метода, который известен как *рефакторинг*). Во втором случае эксперименты имеют вид хорошо согласованных и тщательно изученных вариантов (options) с тем, чтобы в последний момент можно было выбрать наилучшее решение.

Второй метод известен как параллельная разработка альтернатив (или множественное проектирование — set-based design), и он очень подходит для случаев, когда требуется принять чрезвычайно важное, необратимое решение. Большое число подобных решений приходится принимать в процессе разработки аппаратуры (hardware), поэтому множест-

[4] Благодарим Вернера Вильда (Werner Wild) за эту историю. Использовано с разрешения.

венное проектирование особенно подходит для аппаратуры, а также очень часто для элементов встроенного программного обеспечения, предназначенных взаимодействовать с аппаратурой. Что касается разработки обычного (не встроенного) программного обеспечения, то важных и необратимых решений здесь принимается меньше, но нельзя сказать, что они отсутствуют полностью. Например, подходы, использованные при создании интерфейса, может быть трудно изменять, после того как пользователи к ним привыкли. Программные продукты сезонного характера, имеющие один релиз (например, игры и программное обеспечение для подготовки налоговых документов), являются подходящими "кандидатами" для разработки с использованием множественного проектирования. Множественное проектирование также является предпочтительным подходом для принятия ключевых решений, касающихся архитектуры программного обеспечения, изменение которых обходится дорого.

Как выглядит применение множественного проектирования на практике? Предположим, с вашей компанией приключилось какое-то несчастье. Необходимо принять немедленное решение и две вещи ясны: 1) имеются сроки, которые ни при каких обстоятельствах не могут быть продлены и 2) вся работа должна быть выполнена в эти сроки. Что следует предпринять? Если вы хотите быть абсолютно уверены в успехе, следует организовать три коллектива, которые бы разработали три варианта решения проблемы. Коллектив А займется разработкой очень простого решения, которое при любых обстоятельствах гарантированно будет готово к заданному сроку. Это решение будет далеким от идеального, но по крайней мере оно будет готово вовремя. Коллективу Б следует поручить разработку предпочтительного решения, которое, возможно, будет готово к заданному сроку, если все пойдет хорошо. Наконец, коллектив В будет разрабатывать идеальное решение, но вероятность того, что оно будет готово к сроку, крайне мала.

Коллектив А разрабатывает свое простейшее решение, но члены коллектива надеются, что коллектив Б успеет в срок, так как они знают, что решение коллектива Б лучше. Коллектив Б прилагает все усилия, чтобы успеть в срок, в то же время полагаясь на коллектив А, поскольку это их резервный вариант, а также надеясь (надеждам вопреки), что коллектив В все-таки успеет вовремя. Коллектив В усердно работает над своим решением, делая все возможное, чтобы все сделать вовремя, но члены коллектива очень довольны, что коллективы А и Б подстраховывают их.

Когда наступит срок, будет готово решение, лучшее из возможных на данный момент. Более того, даже если сроки будут сокращены, функциональное решение все равно может быть готово вовремя, а если сроки были бы продлены, в этом случае могло бы быть готово вовремя и оптимальное решение. Также, при таком подходе компания могла бы начать с использования решения А, затем перейти к использованию решения Б (когда оно будет готово) и затем, в конце концов, начать использовать решение В.

Если абсолютно необходимо предоставить лучшее из возможных (в максимально короткий срок) решение, множественное проектирование кажется наиболее подходящим методом, и, тем не менее, он воспринимается многими как противоречащий здравому смыслу. Наиболее охотно воспринимают множественное проектирование люди, работающие в среде, где продукты выпускаются регулярно и где не укладываться в сроки считается недопустимым. Хорошим примером здесь является любая ежедневная газета, так же, как и большинство журналов. Кроме того, спектакли в театрах всегда начинаются вечером в определенный час, выпуски новостей осуществляются по расписанию каждый день и сезонная одежда всегда готова к поставкам, когда наступает соответствующий сезон.

Создание продукции в подобных средах имеет вид подготовки различных вариантов, а затем сборки наилучшей комбинации непосредственно перед соответствующей датой. Когда срок по-настоящему важен, к нему не относятся как к независимой переменной; это константа, которую все понимают и уважают.

Поэтому не должно удивлять, что компания Toyota никогда не запаздывает с поставками своей продукции. Так же не должно удивлять, что в Toyota используют множественное проектирование. Это не просто способ уложиться в сроки, это метод узнать как можно больше, чтобы принять лучшее решение.

Как это работает при создании программного обеспечения? На рис. 7.2 можно видеть девятимесячный график, где в запланированных точках синхронизации должны приниматься критически важные решения. Каждое из этих решений запланировано на самый последний момент; чуть позже уже считалось бы опозданием. Для каждого из этих решений разрабатываются несколько полноценных вариантов. Когда приближается момент синхронизации, система собирается и тестируется с использованием каждого варианта. После этого выбирается вариант, предоставляющий наилучшее решение для всей системы в целом. Обратите внимание, что в каждом случае не обязательно выбирается вариант, предоставляющий наилучшее решение для данной подсистемы. Каждый раз выбирается наилучшее решение для всей системы в целом. Именно поэтому набор вариантов должен быть полностью разработан для тестирования в точке синхронизации.

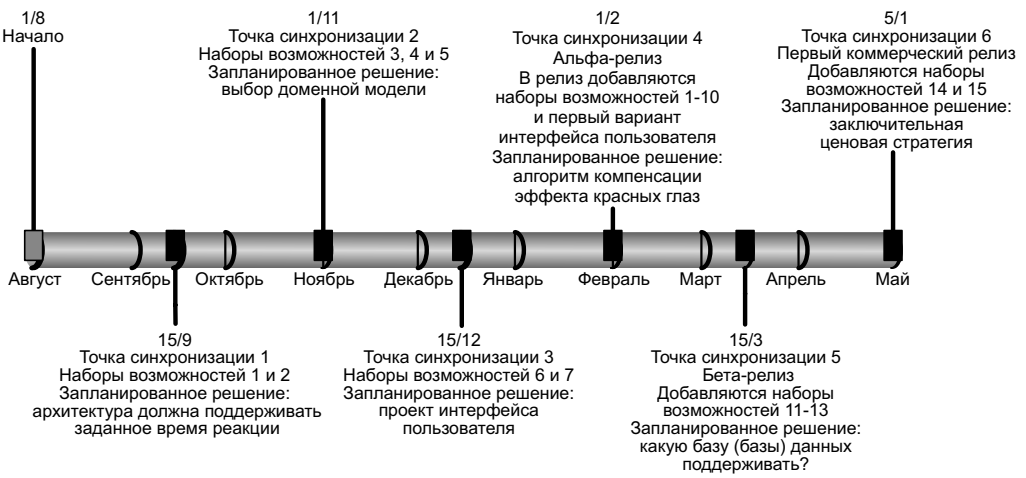


Рис. 7.2. График разработки продукта с ключевыми решениями, запланированными в точках синхронизации

Пример 1: интерфейс для медицинского устройства

Один менеджер, с которым мы разговаривали, использовал множественное проектирование в разработке интерфейса для некоторого устройства, предназначенного для использования в медицине. Его отдел отвечал за создание экранных интерфейсов, которые технические специалисты использовали для управления медицинскими устройствами. Когда чемпион предложил ему новую идею продукта, менеджер понял, что чемпион очень смутно представляет, каково в действительности назначение этого интерфейса. Поэтому менеджер назначил три или четыре коллектива для разработки различных вариантов в течение шес-

тинедельной итерации. По истечении шести недель коллективы показали свои разработки чемпиону, которому понравились одни функциональные возможности из одного варианта и другие функциональные возможности из другого варианта. После этого менеджер назначил два коллектива для более полной разработки двух вариантов в течение второй шестинедельной итерации. В конце этой итерации обнаружилось, что чемпион может указать с уверенностью, какой интерфейс ему нужен. "К этому моменту — рассказал нам менеджер — программное обеспечение было уже наполовину готово, и я знал, что чемпион будет полностью удовлетворен продуктом."

Пример 2: компенсация эффекта красных глаз

В ходе разговора некто, заведовавший разработкой программного обеспечения для принтеров, предложил Мэри дилемму. Он рассказал, что у него есть 12 недель, чтобы создать программное обеспечение для нового цветного принтера, и что сроки просто не могут быть нарушены. В отделе маркетинга ему сказали, что смогут продать дополнительно очень много принтеров, если эти принтеры будут поддерживать автоматическую функцию компенсации эффекта красных глаз. К этому времени проблема ручной компенсации этого эффекта была решена (оператору достаточно было только указать красный глаз на экране компьютера и щелкнуть). Однако в данном случае требовалось, чтобы компьютер сам определял, где на фотографии имеются красные глаза, и автоматически это исправлял.

Дилемма была следующей: имелась идея простого алгоритма, который (менеджер знал это) будет готов вовремя, но это будет не лучший алгоритм из возможных. И имелась идея гораздо лучшего алгоритма, но менеджер не был уверен, что удастся завершить его разработку в течение 12 недель. Итак, как ему поступить в данной ситуации — пойти на риск и создавать лучший алгоритм или выбрать безопасный вариант и отдать предпочтение простому алгоритму?

"Совершенно очевидно, вы должны разрабатывать оба алгоритма", — сказала Мэри.

"Но где я возьму необходимые для этого ресурсы?" — спросил менеджер.

"Вы сказали, что сможете продать очень большие дополнительные количества принтеров, если сумеете решить проблему, — ответила Мэри. — И вы спрашиваете, где брать ресурсы?"

"Это логично, — ответил менеджер, — но как объяснить руководству, что я собираюсь разрабатывать два варианта, один из которых затем придется "выбросить"?"

"Скажите им следующее, — ответила Мэри. — Не волнуйтесь. Я гарантирую, что вы будете иметь средство компенсации эффекта красных глаз вовремя. И, более того, мы создадим эффектный алгоритм, который, возможно, будет готов к первому релизу, но если мы не успеем, тогда он будет готов к следующему релизу."

Пример 3: сменные интерфейсы

Некоторые решения не ясны до конца процесса разработки. Возможно, не выбраны внешние интерфейсы, или в случае встроенного программного обеспечения, еще не создана аппаратура. Часто точная роль различных пользователей не определена бизнесом, или технологии, такие как промежуточное программное обеспечение (middleware), еще не выбраны. Некоторые решения для данного релиза могут не быть приняты вовсе. Например, в первоначальном релизе оператор может осуществлять ручной ввод с клавиатуры, а создание системы распознавания голоса будет отложено на потом. Одно из реше-

ний для подобных случаев заключается в создании сменных интерфейсов, выбираемых с помощью флажков. При этом взамен решений, которые отложены, или для вариантов (options), которые будут выбраны позже, в данный момент используются некоторые объекты-имитаторы.

Встроенное программное обеспечение должно разрабатываться таким образом, чтобы оно работало в двух средах: в среде разработки с использованием имитатора и в реальной среде. Обычно в разных средах процессы регистрации пользователя и реакции на ошибки отличаются; в подобных случаях выбор можно сделать с помощью флажков. Использование сменных интерфейсов для баз данных, промежуточного программного обеспечения (middleware) и интерфейсов пользователя является довольно обычным делом, но создание множественных вариантов для ролей пользователя менее привычно. На наших курсах один из разработчиков описал свою стратегию определения размера файла: "Было неясно, большим будет файл или маленьким, поэтому я выполнил разработку для обоих случаев с тем, чтобы можно было решить в последнюю минуту, какой из вариантов использовать", — рассказал он.

"Но не стали же вы тратить время на разработку двух вариантов, зная, что один из них не будет использован?" — спросил кто-то.

"Стал! — ответил он. — Для этого не потребовалось больших дополнительных усилий, и когда пришло время сдачи программного обеспечения заказчику, я был к этому готов."

Почему это не потери

Что считать и что не считать потерями, почти полностью зависит от того, как мы воспринимаем проблему, которую пытаемся разрешить. Вспомним, как в аварийных обстоятельствах были созданы коллективы А, Б и В. Нас спрашивали, почему бы в этом случае не перевести людей из коллектива А в коллектив Б, увеличив таким образом шансы получить лучшее решение раньше? Вспомните, однако, что это была аварийная ситуация и *невыполнение задачи было абсолютно неприемлемо*. Если бы было сказано, что *это вопрос жизни и смерти и что невыполнение задачи означает, что кто-то умрет*, покажется ли в этом случае существование коллектива А излишеством?

В примерах выше несоблюдение диктуемых рынком сроков вело бы к куда бо́льшим потерям (включая потраченное время членов всех коллективов), чем сравнительно небольшие затраты, связанные с созданием нескольких технических вариантов. Разработка технических вариантов для гарантии, что будут приняты оптимальные решения, ведет к более высокому качеству продукции (что в свою очередь ведет к снижению расходов на будущее обслуживание).

Рефакторинг

Как отмечалось в главе 2, в большинстве случаев целью разработки программного обеспечения является создание допускающего преобразования программного кода, который можно адаптировать к внешним изменениям. Как будет показано в главе 8, это осуществляется с использованием итеративного процесса (а также дисциплины), который допускает возможность изменений кода. (Отличительными особенностями такого процесса являются автоматизированное тестирование, небольшие партии работы (small batches), быстрая обратная связь.) Этой стратегии сопутствуют две опасности. Во-

первых, можно принять важное необратимое решение неправильно, если решения принимаются слишком рано. Как мы уже видели, эту опасность можно выявить и нейтрализовать с помощью множественного проектирования.

Вторая опасность инкрементной разработки состоит в том, что по мере внесения в него изменений мы увеличиваем сложность программного кода и он быстро становится неуправляемым. Это немалая и очень реальная опасность, если вовремя не принять адекватных мер. Действенный метод, позволяющий смягчить этот риск и свести к минимуму отрицательные последствия сложности программного кода, известен как рефакторинг. Рефакторинг позволяет разработчикам откладывать добавление функциональных возможностей в код до тех пор, пока они (возможности) не станут необходимы, а затем добавлять их без возникновения каких-либо проблем, как будто они были в программном коде с самого начала.

Логические обоснования использования рефакторинга можно сформулировать следующим образом.

1. Добавление функциональных возможностей в код до того, как они станут необходимы, неоправданно увеличивает сложность программного кода, которая в разработке программного обеспечения является худшей формой потерь. Поэтому большинство функциональных возможностей должны добавляться в код постепенно (incrementally).
2. При добавлении в существующий программный код новых функциональных возможностей и внесения других изменений важно не увеличивать сложность программного кода.
3. Рефакторинг позволяет уменьшить сложность кода путем упрощения его структуры. Это позволяет добавлять в код новые функциональные возможности с минимальным увеличением его сложности.

Не поддерживайте ненужных функциональных возможностей!

Подход YAGNI[5] можно сформулировать так: "Не поддерживайте функциональных возможностей, которые вам пока не нужны".

Гордон Белл (Gordon Bell) утверждает, что "самые дешевые быстродействующие и наиболее надежные компоненты компьютерной системы это те, которых в ней пока нет".[6]

Все решения по структуре программного обеспечения следует принимать на временной основе. Затем, когда будете знать больше, вы сможете добавить в программный код лучшее решение. (Билл Уэйк[7])

Вероятно, величайшая ошибка, которую мы можем совершить, состоит в том, чтобы рассматривать рефакторинг как "неудачную попытку сделать правильно с первого раза". В действительности рефакторинг — это фундаментальная возможность ограничить сложность программного кода, расширить возможность вносить в него изменения, повысить потребительскую ценность и долговечность кода. Рефакторинг подобен физическим упражнениям, которые необходимы для поддержания "здоровья" программного

[5] YAGNI означает You Aren't Going to Need It. (Вам это не понадобится.) Иными словами: "Никаких лишних возможностей", или "Пишите меньше кода".

[6] Bentley J. *More Programming Pearls*, Addison-Wesley, *1988*, p. 62.

[7] Мы благодарим Билла Уэйка (Bill Wake) за сотрудничество. Использовано с разрешения.

кода. Даже самые занятые в мире люди (те, кто принадлежат к исполнительным структурам власти, а также президенты) находят время для физических упражнений, чтобы поддерживать свою способность отвечать на постоянно возникающие вызовы. Аналогично, даже наиболее занятые в мире разработчики обязаны найти время для рефакторинга, чтобы поддерживать способность программного кода к изменениям.

Рефакторинг предполагает широкое использование автоматизированного тестирования. По мере внесения в программный код изменений с целью уменьшения его сложности, разработчики постоянно должны использовать соответствующие средства в качестве тестовой нагрузки, чтобы быть уверенными, что изменения в коде не вызывают каких-либо нежелательных последствий. Постоянная тестовая нагрузка (test harness) немедленно выявит любые дефекты, по небрежности внесенные в программный код при упрощении его структуры. Эта тестовая нагрузка также должна поддерживать непрерывную интеграцию (continuous integration) и культуру "останова линии" в момент обнаружения дефекта.

Время и усилия, потраченные на автоматизированное тестирование и непрерывное совершенствование программного кода, а также связанных с этим тестов, это время, потраченное на устранение потерь и сокращения сложности. Стратегии бережливости, направленные на устранение потерь и сокращение сложности — это самый эффективный способ из известных придать коду новые свойства за самую низкую цену. Рефакторинг похож на рекламу: он ничего не ст*о*ит, наоборот, он приносит прибыль.

Унаследованные системы

Имеется два типа программного обеспечения — толерантное к изменениям и унаследованное (legacy software). Некоторые программы сравнительно легко адаптировать к изменениям в бизнесе и технологии, а другие поддаются изменениям трудно. Эти программы, которые затруднительно изменять, известны в мире разработки программного обеспечения как "унаследованные". Для толерантного к изменениям программного обеспечения характерны ограниченные зависимости (limited dependencies) и комплексная тестовая нагрузка (test harness), которая призвана выявлять непреднамеренные последствия изменений. Итак, унаследованные системы можно определить как системы, которые не защищены набором тестов. Иными словами, вы создаете унаследованный программный код каждый раз, когда ПО создается без ассоциированных с ним тестов.

Унаследованные системы можно найти всюду, и поэтому значительная часть разработки программного обеспечения связана с обслуживанием унаследованного программного кода. Брайан Морик (Brian Marick) идентифицировал три подхода к решению проблем, связанных с унаследованными системами.

1. "Переписать и выбросить".
2. Подвергнуть рефакторингу.
3. "Задушить".

Переделать (или "переписать") программное обеспечение — это привлекательный подход для менеджеров, которые мечтают однажды прийти в офис и обнаружить, что все их проблемы исчезли как по мановению волшебной палочки и что сейчас компьютер делает все то, чего не удавалось сделать раньше со старой системой. Проблема в том, что данный подход имеет печальную историю неудач. Попытки копировать (или "переделать") маловразумительный программный код и конвертировать обширную базу

данных (которая, вероятно, испорчена) очень рискованны. Более того, как уже отмечалось, вероятно две трети функциональных возможностей унаследованных систем не используются и не нужны. Тем не менее слишком много попыток преобразования унаследованных систем были неудачны: при этом сложность систем не уменьшалась и вносить в них изменения не становилось проще.

Применение рефакторинга к унаследованному программному обеспечению, вероятно, является лучшим подходом. Как отмечалось ранее, рефакторинг нуждается в тестовой нагрузке, однако унаследованный код ее не имеет по определению. Поэтому первый шаг состоит в том, чтобы устранить зависимости (break dependencies), а затем подключить тесты к отдельным частям программного кода, чтобы появилась возможность подвергнуть их рефакторингу. Как правило, непрактичным является добавление в унаследованный код блочных тестов, поэтому речь может идти о создании приемочных тестов (acceptance tests) (которые также называют тестами "историй" (story tests)), основанных на примерах функционирования, которого ожидают от данного кода. Более подробно об этом можно узнать из книги *Working Effective with Legacy Code*[8] ("Эффективная работа с унаследованным кодом", ИД "Вильямс", 2009), которая считается библией по рефакторингу унаследованного программного кода.

Третий подход к исправлению унаследованного кода состоит в том, чтобы "задушить" его. Здесь использована метафора Мартина Фаулера[9] (Martin Fowler) о вьющемся растении-паразите, которое обвивает большое дерево, постепенно принимая его форму, и в конце концов душит его. Постепенно дерево засыхает и остается только растение-паразит. Для того чтобы подобным образом "задушить" унаследованный программный код, вы добавляете "душащий" код (содержащий помимо прочего также автоматизированные тесты) каждый раз, когда вам приходится "вторгаться" в структуру унаследованного кода. При этом любое функционирование старого кода, которое необходимо изменить, блокируется и соответствующие "обязанности" передаются новому программному коду, который заменяет старый код. Постепенно вокруг старой системы строится новый код и в конце концов наступает момент, когда все элементы старой системы оказываются заменены новым кодом.

Решение проблем

Наши счета за гостиницу часто оплачивают компании, которые мы консультируем. Интересная ситуация возникает, когда мы прибываем в какую-нибудь гостиницу и у нас спрашивают кредитную карточку. В подобных случаях Мэри говорит: "Кажется, наше проживание здесь оплачивает компания XYZ?" В половине случаев следует ответ: "Да, конечно; кредитная карточка нам нужна просто на всякий случай". А иногда нам отвечают: "Нет, у нас нет никакой информации о том, что компания XYZ платит за ваше комнату". После этого нам приходится убеждать гостиницу отправить счет в компанию XYZ, до того как мы выедем. При этом наиболее разочаровывает, когда соответствующий служащий говорит: "Эта проблема возникает у нас постоянно". И Мэри в таких случаях неизменно отвечает: "Если проблема возникает постоянно, почему вы не принимаете никаких мер?"

---

[8] Feathers M. *Working Effectively with Legacy Code*, Prentice Hall, *2005.*

[9] Fowler M. *StranglerApplication,* `www.martinfowler.com/bliki/StranglerApplication.html`.

Отличительная черта эффективной организации состоит не в том, что там отсутствуют проблемы, а в том, что там нет систематических проблем. В эффективной организации все служащие на всех уровнях имеют постоянный и надежный метод решения проблем, с которыми приходится сталкиваться в повседневной работе. Неоднократно в первоклассных гостиницах нам присылали счет по ошибке — неоспоримое свидетельство, что, несмотря на высокую репутацию гостиницы, там отсутствует процесс систематического совершенствования.

Плохо, когда процесс разработки программного обеспечения не включает какие-либо методы самосовершенствования. Каждый коллектив должен регулярно располагать временем для систематического поиска и устранения проблем, усложняющих ему жизнь. В некоторых компаниях, например, выделяется специальное время в конце каждой итерации для обсуждения проблем и определения, какие изменения целесообразно было бы сделать.

Первое правило при организации процесса улучшений состоит в том, чтобы не пытаться делать все сразу. Когда этот процесс имеет нерегулярный, спорадический характер, люди пытаются решать все проблемы одновременно. Здесь уместно вспомнить, что в крупных проектах и при продолжительных релиз-циклах (release cycles) имеет место аналогичная тенденция создавать крупные партии (batches) работы. Идея в том, чтобы организовать постоянный процесс совершенствования.

Частые регулярные встречи помогут коллективу выявить наибольшую проблему и избавиться от нее, а затем перейти к следующей проблеме. Обычно наибольшая проблема может быть охарактеризована очень неопределенно, например: “Наша жизнь была бы намного легче, если бы у нас было достаточно времени, чтобы выполнить всю работу, которую от нас ждут”. Но такая расплывчатая формулировка отбивает всякую охоту что-то предпринимать для решения проблемы

Упорядоченный подход

Упорядоченный подход к решению проблем позволяет коллективу перейти от размышления о желательном к новому знанию, конкретным мерам и долговременным результатам. Научный метод, обсуждавшийся ранее в этой главе, предоставляет хороший план действий.

1. Обозначить проблему.
2. Проанализировать ситуацию.
3. Создать гипотезу.
4. Провести эксперименты.
5. Оценить результаты.
6. Стандартизировать метод и многократно его применять.

1. Обозначить проблему

Легко для коллектива заявить: “Мы хотим быть более внимательными к заказчикам”. Но что именно планируется предпринять, чтобы осуществить это? Давайте присмотримся к соответствующему процессу, призванному решить проблему. Предположим, наибольшая проблема данного коллектива заключается в длинном списке жалоб от заказчиков, которые утверждают, что их заявки игнорируются. Поэтому коллектив решил пристально присмотреться к проблеме.

2. Проанализировать ситуацию

Прежде всего был составлен список всех функциональных возможностей (features), созданных и переданных заказчикам в последних четырех релизах, и создана диаграмма Парето с временем цикла (cycle time) для этих функциональных возможностей. Данные на диаграмме (рис. 7.3) выглядели неплохо — более трех четвертей заказанных функциональных возможностей были переданы заказчикам в течение шести недель по получении заявки.

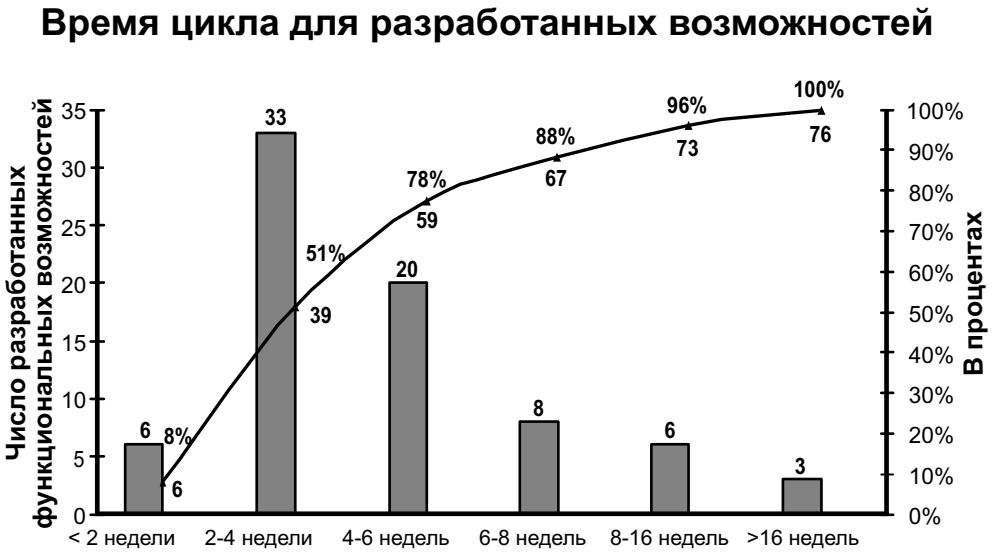


Рис. 7.3. Диаграмма Парето: время цикла для разработанных и введенных в эксплуатацию у заказчика функциональных возможностей

Однако члены коллектива знали, что в очереди несделанной работы было множество проигнорированных заявок; это усугублялось тем, что новые заявки продолжали добавляться все время. Поэтому была создана еще одна диаграмма Парето, отображающая данные о времени пребывания в очереди заявок, находящихся там в настоящее время (рис. 7.4).

Данные свидетельствуют, что более 60% заявок в очереди имеют более чем восьминедельную давность. Однако предыдущая диаграмма демонстрирует, что только девять заявок из выполненных состояли в очереди более чем восемь недель. Иными словами, похоже, что в основном выполнялись наиболее недавние заявки.

Данные также свидетельствуют, что коллектив выполнил в целом 76 заявок в четырех последних релизах, что составляет в среднем на релиз 19 заявок. Релизы имеют место каждые две недели, поэтому средняя производительность составляет около десяти заявок в неделю. Поскольку в очереди имеется 568 заявок, при существующих темпах потребуется более года (55–60 недель) для выполнения всех заявок (если предположить, что в течение этого периода новые заявки поступать не будут).

Далее была создана диаграмма, отображающая, как поступали заявки в течение последних 12 недель (по категориям приоритета) (рис. 7.5).

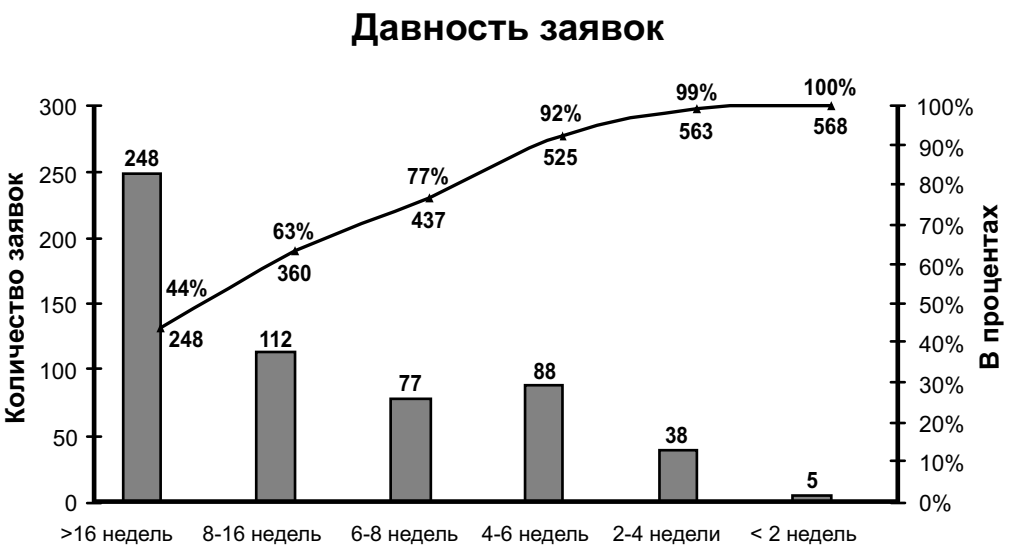


Рис. 7.4. Диаграмма Парето: "возраст" заявок, содержащихся в очереди

Рис. 7.5 Поступление заявок

3. Создать гипотезу

Важной частью научного метода и такой, на которую часто не обращают внимания коллективы, спешащие решать проблемы, является необходимость создания гипотезы, которая бы поддавалась проверке. В нашем примере члены коллектива обратили внимание, что заявки трех наиболее высших категорий срочности (важные, срочные и аварийные) поступают в среднем по десять в неделю, что приблизительно соответствует их ны-

нешней производительности. Поэтому их гипотеза состояла в том, что, если бы они сократили очередь до 40–50 заявок, плюс перестали принимать "обычные" и "несрочные" заявки и при этом слегка увеличили свою производительность, они смогли бы в 90% случаев выполнять новые заявки в течение четырех недель.

4. Провести эксперименты

Члены коллектива начали с того, что решили выяснить, не удастся ли сократить задолженность; для этого были разосланы e-mail всем заказчикам, чьи заявки содержались в очереди более восьми недель. В сообщении говорилось, что задолженность пересматривается и желательно получить подтверждение, что заказчик хотел бы, чтобы его заявка оставалась в очереди. Если подтверждение не будет получено в течение недели, будет считаться, что заявку можно аннулировать. Статистика ответов на это сообщение представлена в табл. 7.1.

Таблица 7.1 Ответы на запрос о подтверждении важности заявки

"Возрастная" категория	Всего	Аннулировано		Подтверждено		Без ответа	
8–16 недель	112	1	1%	26	23%	85	76%
>16 недель	248	20	8%	3	1%	225	91%

Члены коллектива обратили внимание, что 99% заявок старше 16 недель оказались больше не нужны; и то же касается 77% заявок старше 8 недель. Они присмотрелись к 29 заявкам, необходимость которых была подтверждена, и заметили, что все они имеют приоритет "Важная" или более высокий. Далее они взялись за оставшиеся в очереди заявки и разослали аналогичные e-mail, однако на этот раз только тем заказчикам, чьи заявки имели приоритет "Обычная" или "Несрочная". Это сократило очередь до 100 заявок, однако заказчики продолжали подавать заявки с приоритетами "Обычная" и "Несрочная", и было ясно, что если такие заявки перестать принимать, заказчики просто начнут указывать для своих заявок более высокий приоритет. Поэтому были проведены эксперименты с различными способами решения проблемы низкоприоритетных заявок. Наиболее рациональным оказалось приглашать представителя заказчика и работать с ним, чтобы определить, нет ли иного способа решить данную проблему. В 65% случаев находилось сравнительно быстрое альтернативное решение. А в 10% случаев заказчика удавалось убедить, что его заявка нецелесообразна, учитывая объем работы, необходимый для ее выполнения. Таким образом, три четверти низкоприоритетных заявок удавалось аннулировать с помощью такого личного контакта.

5. Оценить результаты

Коллектив смог увеличить свою среднюю производительность до 25 заявок в одном релизе (каждые две недели). В течение последующих нескольких недель очередь была постепенно уменьшена до 50 заявок с помощью как аннулирования старых заявок, так и выполнения высокоприоритетных заявок несколько быстрее, чем они успевали поступать. Коллектив попытался оценить свою способность предоставлять гарантию сроков на удовлетворение заявки. Данные показывали, что длина очереди — это решающий фактор для времени реакции на заявку, поэтому члены коллектива были уверены, что при очереди в 50 заявок они могут взять на себя обязательство в 90% случаев выполнять заявки в течение четырех недель.

6. Стандартизировать метод и многократно его применять

Члены коллектива объявили заказчикам, что они будут выполнять большинство заказов в течение четырех недель, но оказалось, что это поощряет новых заказчиков подавать свои заявки. Поэтому был выделен второй сотрудник коллектива для работы с представителем заказчика для ограничения темпов поступления новых заявок и поиска альтернативных решений для низкоприоритетных заявок. Кроме того, коллектив смог увеличить количество выполняемых (на релиз) заявок до 28. В течение месяца очередь стабилизировалась на четырехнедельном объеме работы. Коллектив подготовил отчет о своем опыте и предложил его другим коллективам на семинаре внутри компании.

Кайдзен-мероприятия

Мы только что говорили о типичном процессе решения проблемы — деятельность, которая должна быть включена в повседневную работу любого производственного коллектива. Иногда коллективу полезно прервать повседневную деятельность и сосредоточиться на совершенствовании ключевых аспектов своего процесса. Это может быть особенно полезно, если предполагаемые изменения затрагивают множество подразделений компании; в этом случае члены различных коллективов должны объединиться для решения стоящей проблемы.

Представим себе организацию, где определили, что основной проблемой является ввод созданного ПО в эксплуатацию у заказчика, и поэтому было решено, что самой важной проблемой в данный момент является автоматизация ввода создаваемого программного обеспечения в эксплуатацию. Внедрение подобной автоматизации повлечет за собой изменения на различных рабочих местах и в разных процессах, в сложившемся взаимодействии между подразделениями компании, повлияет на работу с заказчиками и т.д. Это подходящий случай для проведения кайдзен-мероприятия.

Японское слово кайдзен (или кайзен) означает “изменение к лучшему”. Кайдзен-мероприятия — это хорошо известное средство, применяемое при бережливых подходах, которое собирает вместе (в небольшую группу) на несколько дней представителей различных подразделений компании для решения хорошо обозначенной критичной проблемы. Во время этого мероприятия во все процессы, от которых зависит решение проблемы, Кайдзен-коллективом вносятся изменения. После решения проблемы временный коллектив расформировывается и его члены возвращаются к своим повседневным обязанностям. При этом результатом Кайдзен-мероприятия является усовершенствованный процесс.

Решение проблем с участием больших коллективов

Иногда требуемое изменение слишком “грандиозно”, чтобы его могла осуществить в течение нескольких дней только что сформированная небольшая группа. Например, карта потока создания ценности может показывать, что существенный эффект может быть получен, если тестирование в производственном процессе переместить вперед. Но это может означать существенное изменение функций многих работников и некоторых организационных структур. Это очень серьезная проблема. Для ее решения может потребоваться участие большого коллектива.

Решение проблем с участием больших коллективов часто моделируются на основе GE Workout[10] (программы компании General Electric (GE), которую иногда определяют для данной компании как "культурная революция"). Данная программа, вероятно, является одной из основных причин замечательных успехов GE в 90-х годах. Реализованная Джеком Уэлчем (Jack Welch) задолго до "шести сигм" (Six Sigma), программа GE Workout была предназначена для борьбы с бюрократизмом и решения организационных проблем, до которых, кажется, ни у кого не доходили руки в недавно сократившей штаты компании. Аналогичные методы широко использовались в других организациях для создания условий, когда рядовые работники выступают инициаторами и реализуют как небольшие, так и значительные усовершенствования.

В случае, когда требуется значительное усовершенствование или решение серьезной проблемы, создается большой коллектив, включающая представителей различных подразделений компании на разных уровнях, перед которой ставится задача предложить решение. Это решение должно основываться на принципах бережливости и быть достаточно практичным, чтобы его можно было реализовать в ближайшем будущем — например, в течение 90 дней. Подобное мероприятие обычно выглядит как совместная работа, которая может продолжаться около трех дней. После определения мелких проблем (образующих крупную проблему) создаются малые группы, в каждой из которых организуется мозговой штурм соответствующей мелкой проблемы и вырабатываются рекомендации для ее решения. (В ходе такой работы члены групп встречаются друг с другом для генерации совместных идей и собирают информацию, необходимую для выработки рекомендаций.) К концу мероприятия его участники предоставляют свои рекомендации (вместе со вспомогательной информацией). (У каждой рекомендации имеется свой ответственный, которому будет поручена немедленная реализация данной рекомендации, в случае ее одобрения.) Представленные рекомендации обсуждаются, одобряются или отвергаются. Ответственным за одобренные рекомендации поручается провести их в жизнь; для этого предоставляется соответствующие время и поддержка.

Это теория. И достаточно часто это работает на практике. Но нередки случаи, когда мероприятие по совершенствованию (чего-то) с участием большого коллектива терпит неудачу по целому ряду причин.

1. Недостаточная сосредоточенность на основных процессах ведет к:
   - случайным улучшениям;
   - оптимизации, скорее локальной, чем глобальной.

   Это также может быть проблемой для кайдзен-мероприятий с участием небольших групп. Улучшения должны быть направлены на наиболее критичные и общие (для системы) недостатки.
2. Много идей предлагается и одобряется, но мало реализуется, потому что:
   - никто не несет ответственности за реализацию каждой идеи;
   - нет полномочий для осуществления изменений, затрагивающих несколько подразделений организации;
   - не выделяется время для внесения изменений;

[10] Ulrich D., Kerr S., Ashkenas R. *The GE Workout*, McGraw-Hill, *2002*.

- отсутствует контроль (и стремление довести дело до конца) со стороны вышестоящей инстанции.

Отсутствие решимости довести дело до конца особенно является проблемой при решении проблем с участием крупных групп и часто служит источником глубокого разочарования. Лучше совсем не затевать мероприятия, чем позволить ему потерпеть неудачу. Не доведенное до конца подобное мероприятие — это проявление неуважения к участвовавшем в нем людям.

Кайдзен-мероприятия и мероприятия с участием крупных групп — прекрасная школа, где можно приобрести навыки решения проблем (как внутри одного подразделения, так и общих для нескольких подразделений). Однако, как правило, усовершенствования должны быть ответственностью естественных (или постоянных) коллективов, которые должны постоянно и организованно выявлять и ликвидировать текущую наибольшую проблему.

Попробуйте это

1. На следующем собрании коллектива обсудите самую серьезную вашу проблему и методы ее решения.
2. Обсудите на собрании коллектива метод множественного проектирования. Выясните, нет ли примеров, когда кто-нибудь использовал этот подход при разработке программного обеспечения. (В коллективе из более чем семи человек такие примеры непременно найдутся. Если нет, попробуйте для опыта множественное проектирование, но только не в реальной работе.) Пусть коллега, имеющий такой опыт, о нем расскажет.
3. Какой в вашем коллективе имеется опыт применения рефакторинга? Прежде чем добавить в программный код новую функциональную возможность, вн*о*сите ли вы прежде упрощающие изменения в ее структуру, не меняя при этом функционирования данной возможности, а затем тестируете ли новую структуру, чтобы убедиться, что функционирование не изменилось? После того как новая функциональная возможность заработала, применяете ли вы немедленно рефакторинг для ее упрощения?
4. Возьмите созданный в вашем коллективе какой-либо документ и на собрании общими усилиями попробуйте преобразовать его в отчет формата А3. Удалось ли представить в новом формате всю существенную информацию из документа? Когда этот документ будет распространяться, присоедините к нему отчет формата А3. После того как это будет сделано несколько раз (и докажет свою практичность), попробуйте распространять только отчет формата А3 (без старого документа). Проследите, каковы будут результаты.
5. Если регулярно возникает какая-то мелкая проблема, имеют ли люди, которым она досаждает, возможность ее устранить? Проводит ли ваш коллектив регулярные собрания для обсуждения таких проблем (на таком собрании было бы полезно выяснить, как выполняется работа сейчас и как ее можно было бы выполнить лучше)? Как часто проводятся подобные собрания? Использует ли коллектив упорядоченный подход к решению проблем? Решается ли в каждый момент только одна проблема?

Глава 8

Качество

Обратная связь

Заказчикам, нуждающимся в новой системе, обычно говорят, что для разработки высококачественного программного обеспечения требуется много времени. Однако для некоторых заказчиков это просто неприемлемо. Быстро получить нужное программное обеспечение необходимо просто для выживания — и низкое качество при этом недопустимо. Не видя иного выхода, они часто вынуждены выбирать между качеством и скоростью. В подобной ситуации очень немногие компании обнаруживают, что способ быстро получить великолепное программное обеспечение все-таки существует (и при этом они еще приобретают устойчивое конкурентное преимущество). Когда вопрос касается выживания, старые подходы часто становятся неадекватны и лучшие способы создания новых продуктов изобретаются по необходимости. Однако в каждом случае эти новые, более эффективные процессы разработки имеют две общие черты.[1]

1. Отличная быстрая обратная связь.
2. Великолепная дисциплина.

Программа Polaris[2]

4 октября 1957 года кризис поразил министерство обороны Соединенных Штатов. Советский Союз успешно запустил на околоземную орбиту первый спутник Земли. Удивленные американцы рассуждали, что поскольку Советы уже запускали ракеты среднего радиуса действия со всплывших подводных лодок, а теперь они располагают технологией для создания ракет дальнего радиуса действия, у них появляется возможность создать оружие, на которое у США нет адекватного ответа. Кризис еще более усилился месяцем позже, когда был запущен второй спутник Земли, который был гораздо крупнее и даже с собакой Лайкой на борту.

ВМФ США тогда только начал свою программу Polaris (создание подводных лодок, способных запускать ракеты из-под воды). Согласно этой программе первая подводная лодка Polaris должна была приступить к боевому дежурству в 1965 году. Очевидно, в но-

1 См. Clark K.B. Fujimoto T. *Product Development Performance*, Harvard Business School Press, *1991.*

2 У программ Aegis и Atlas аналогичные истории; однако программа Polaris имела великолепную независимую и объективную документацию.

вых обстоятельствах ожидать восемь лет было невозможно. Две недели спустя после запуска первого спутника сроки программы Polaris были перенесены на 1959 год. Двумя годами позже, 9 июня 1959 года, первая ракета Polaris была запущена с подводной лодки, а к концу 1960 года две подводные лодки Polaris уже патрулировали океан.[3]

Как такая технически сложная задача, для которой, как предполагалось, было необходимо восемь лет, могла быть выполнена в такие сжаты сроки? Прежде всего, когда сроки изменились, технический директор программы, вице-адмирал Леверинг Смит (Levering Smith) поставил перед своим коллективом простую цель: создать новую подводную лодку в предельно краткие сроки. Это не допускало никаких потерь времени, никаких излишних функциональных возможностей и никаких задержек — короче говоря, каждая минута на вес золота.

В течение нескольких недель две строящиеся подводные лодки (изначально не предназначенные нести ракеты) были увеличены в длину на 40 метров, чтобы появилось место для ракет.[4] В этих удлиненных подводных лодках удалось разместить 16 ракет, и это стало стандартом, который соблюдается по сей день. Две подводные лодки понадобились адмиралу Смиту немедленно потому, что цели программы были изменены с создания конечной системы в течение восьми лет на создание ряда систем: A1, A2, и A3.[5] Версия системы A1 будет содержать технологию, которая может быть создана приблизительно за три года. Версия A2 должна была создаваться параллельно, но более медленно, чтобы в ней можно было использовать более адекватные технологии, а версия A3 вберет в себя весь опыт, накопленный при создании предыдущих версий.

Процесс создания каждой версии адмирал Смит разделил на небольшие этапы, для каждого из которых была определена конкретная, узкая цель. Он жестко контролировал разработку конструкции системы, особое внимание уделяя совместимости ее компонентов и лично подписывая чертежи. Однако вместо особой тщательности при создании всех подсистем, была организована разработка нескольких конкурирующих подсистем будущей системы, для которых существовала техническая неопределенность. Это позволило выбрать наилучший вариант, когда технология была создана. Наконец, адмирал Смит требовал высокой надежности, поэтому с самого начала подводные лодки Polaris подвергались основательному тестированию и имели встроенную избыточность.

Историк Харви Сапольски (Harvey Sapolsky) отмечал.[6]

> Программа Polaris рассматривается как выдающийся успех. Ракета для этой программы была создана в рекордные сроки. Не было никаких признаков перерасхода средств. Частые испытания показывали, что ракета функциональна. Строительство подводной лодки было быстро завершено. Не удивительно, что ведомство, отвечающее за спецпроекты (Special Projects Office), считается одним из наиболее эффективных государственных учреждений.

---

[3] Polmar N. *The Polaris: A Revolutionary Missile System and Concept*, Naval Historical Colloquium on Contemporary History Project. `www.history.navy.mil/colloquia/cch9d.html`, *January 17, 2006*.

[4] Там же.

[5] Sapolsky H. *The Polaris System Development: Bureaucratic and Programmatic Success in Government*, Harvard University Press, *1972*.

[6] Там же, p. 11.

В рамках программы Polaris была создана "Методика оценки и проверки программы" (Program Evaluation and Review Technique — PERT), инновационная система планирования, созданная для управления программой. PERT рассматривается как основная причина замечательного успеха программы Polaris, однако Харви Сапольски называет это мифом.[7] Сапольски утверждает, что система PERT, по крайней мере на ранних стадиях проекта Polaris, только предназначена была обеспечить дальнейшее финансирование программы. В те ранние годы PERT как система планирования была крайне ненадежна, поскольку технические цели изменялись так быстро, что актуальность диаграмм PERT поддерживать было просто невозможно. Позже программа Polaris стала жертвой собственной популярности; в конце концов от ее менеджеров стали требовать обязательного использования системы PERT, которую они в основном игнорировали и предпочли бы забыть совсем.

Историк Сапольски относит успех программы Polaris не за счет PERT; он полагает, что успех полностью зависел от технического руководства адмирала Смита, благодаря которому велась одновременная разработка нескольких вариантов компонентов, уделялось особое внимание надежности, а также имело место глубокое понимание важности порученного дела среди всех участников.[8]

Планирование релизов

Когда политические события заставили пересмотреть сроки программы Polaris, первое, что сделал адмирал Смит, это переключился с восьмилетнего плана создания совершенной системы на инкрементный план[9] для системы с минимальными функциональными возможностями, которую в будущем предполагалось совершенствовать. Адмирал планировал продемонстрировать наиболее важный элемент системы (запускаемую с подводной лодки ракету) через четверть срока, отведенного для всей программы. Затем предполагалось создать простейшую систему, которая, безусловно, будет работать, и поставить ее как можно быстрее (вариант A1). В то же время параллельно будет осуществляться разработка лучшей версии (вариант A2), поставка которой также не будет задержана. После поставки первой версии (в это время создание второй версии будет идти полным ходом) предполагалось приступить к разработке "максимальной" версии (вариант A3). Такой подход обеспечил больше возможностей, более быструю поставку и более низкие расходы, по сравнению с оригинальным планом (рис. 8.1).

Если описанный подход оказался хорош для технически сложной аппаратуры (или "железа"), он еще больше подходит для программного обеспечения. Давайте попробуем создать процесс разработки ПО, основываясь на этом подходе.[10] Любой подобный процесс должен начинаться с выявления рыночного спроса, создания концепции продукта, определения целевой цены и сроков. Это устанавливает для данного процесса общие ограничения. Разработчики, знакомые со средой заказчика и существующей технологией, оценивают (приблизительно и быстро), какими реальными функциональными возможностями может обладать создаваемое программное обеспечение с учетом этих ограничений.

---

[7] Там же, Chapter 4.

[8] Там же, Chapter 5.

[9] Иными словами, план, предусматривающий постепенное наращивание возможностей.

[10] Cohn M. *Agile Estimating and Planning*, Addison-Wesley, *2005.* Здесь читатель сможет узнать о планировании более подробно.

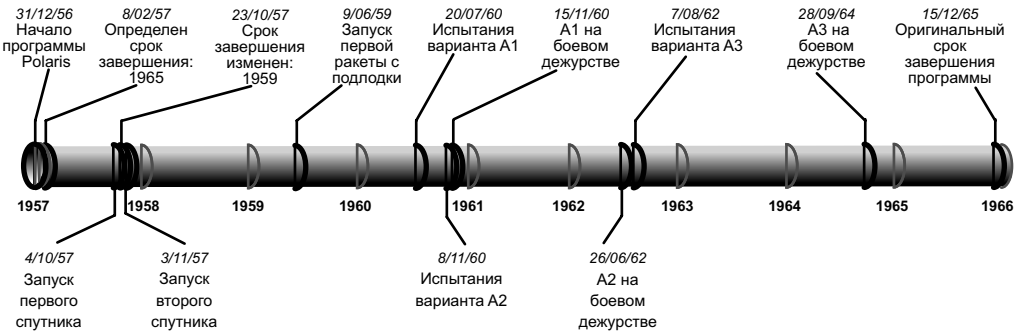


Рис. 8.1. График программы Polaris

Вместо того чтобы тратить выделенное для разработки время на создание подробного долговременного плана, быстро создается план релизов (в котором каждый последующий релиз будет содержать больше функциональных возможностей по сравнению с предыдущим). Планируется разработка на ближайшую перспективу с целью проверить наиболее критические функциональные возможности и создать основы архитектуры. Также планируется ранний выпуск минимальной конфигурации, чтобы начала поступать выручка (и началась окупаемость). После этого периодически организуются дополнительные релизы, содержащие все больше функциональных возможностей.

На рис. 8.2 представлен девятимесячный план организации шести релизов, каждый из которых содержит некоторый (постоянно увеличивающийся) набор функциональных возможностей. Кроме того, каждый релиз разделен на три двухнедельные итерации.

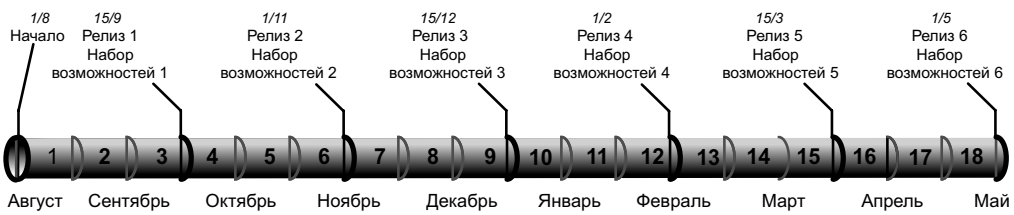


Рис. 8.2. Девятимесячный план шести релизов, разделенных на 18 двухнедельных итераций

Целью первого релиза является разработка набора функциональных возможностей, которые бы доказали осуществимость проекта и создали предварительную архитектуру. После этого новые наборы функциональных возможностей должны выбираться с учетом следующих соображений.[11]

1. Более ценные функциональные возможности должны предшествовать менее ценным.
2. Более рискованные (и ценные) функциональные возможности должны предшествовать менее рискованным.
3. Инновационные функциональные возможности должны предшествовать уже очевидным возможностям.

[11] Там же, p. 80–87.

4. Функциональные возможности, разработка и поддержка которых обходится дешевле, должны предшествовать более дорогим функциональным возможностям.

Представленный выше план релизов создает основу для проверки технических и маркетинговых предположений, прежде чем инвестировать крупные суммы денег в разработку детальных планов. Начав реализацию и оценивая ее прогресс, можно получить раннее представление об осуществимости плана. Ко времени первого релиза вы будете иметь реальную информацию, позволяющую прогнозировать трудоемкость всего процесса куда более точно, чем если бы эти шесть недель были потрачены на разработку подробных планов и обоснований.

Если после первого релиза становится очевидно отставание, это значит, что план слишком агрессивен. Предпочтительный подход в этом случае состоит в том, чтобы упростить бизнес-процесс или создаваемый продукт, упростить реализацию функциональных возможностей или отказаться от некоторых из них полностью. Как правило, лучше строго придерживаться сроков, определенных для итераций и релизов, и ограничить набор функциональных возможностей только теми, которые могут быть полностью разработаны в заданные сроки.

Конечно, существуют среды (речь идет, например, об игровых программах), в которых публичные релизы с постепенным наращиванием конфигурации не реальны. Однако для большинства подобных случаев инкрементная разработка по-прежнему подходящая идея. Просто в данном случае имеет место внутренний "выпуск" программного продукта (в среду, которая максимально близка к реальной среде). Подготовка продукта к выпуску каждые три месяца предоставляет хорошие возможности контроля над процессом. Это особенно хороший способ снизить риск при создании программного обеспечения на заказ.

Архитектура

В течение первого месяца существования ускоренной программы Polaris адмирал Смит предпринял решительные действия, чтобы определить общие контуры и параметры системы. Ракеты предстояло разместить в подводной лодке; подобную подлодку реально было удлинить на 40 метров; на этом пространстве требовалось разместить 16 ракет; две подлодки строились в настоящее время. Таким образом без излишней суеты были определены высокоуровневые ограничения для архитектуры будущей системы. После этого не потребовалось много времени на создание высокоуровневой конструкции (или архитектуры) системы. Данная архитектура в значительной мере была продиктована ограниченным пространством подлодки и существующей ракетной технологией. Лежащий в основе конструкции системы принцип состоял в том, чтобы ясно определить и жестко контролировать сопряжения между элементами системы, в то же время обеспечивая, чтобы эти элементы (или подсистемы) могли разрабатываться и испытываться независимо различными подрядчиками при минимальном обмене информацией. Это было необходимо, поскольку небольшой штат адмирала Смита не мог вникать в подробности каждой подсистемы, а также потому, что было необходимо предоставить подрядчикам свободу творчества (чтобы быть функциональной, система должна была содержать массу инноваций).

Таким образом, отправной точкой разработки крупной и сложной системы должно быть создание ее архитектуры (поддающейся разделению на подсистемы), которая бы позволила творческим коллективам работать (параллельно и независимо) над подсистемами, предоставляющими критические наборы функциональных возможностей. Объем

каждой подсистемы должен быть таким, чтобы ее разработку смог осуществить один коллектив или несколько тесно связанных коллективов.

По мере того как все больше становится очевидно, что толерантность к изменениям является главным ценным качеством для большинства систем программного обеспечения, поддерживающие инкрементную разработку архитектуры (такие, как ориентированные на сервис (service-oriented) и основанные на компонентах (component-based)), быстро заменяют монолитные архитектуры. Однако даже при использовании таких архитектур все равно существуют некоторые ограничения, особенно связанные с нефункциональными требованиями (безопасность, расширяемость и т.п.), которые необходимо учесть до начала процесса разработки. Целью хорошей архитектуры является возможность свести к минимуму такие необратимые решения и предоставить структуру, поддерживающую итеративную разработку.

Сама архитектура может и обычно должна создаваться (или наращиваться) постепенно (т.е. с использованием инкрементного подхода). В своей книге *Software by Numbers* ("Программное обеспечение в числах") авторы Марк Денн (Mark Denne) и Джейн Клилэнд-Хуанг (Jane Cleland-Huang) указывают, что архитектура программного обеспечения, по сути, состоит из некоторых элементов, которые должны добавляться, когда они будут востребованы соответствующими наборами функциональных возможностей, находящимися в данный момент в разработке. Для успешных программных продуктов, срок жизни которых продолжается много лет, крупные архитектурные улучшения можно ожидать каждые три года (или около того), по мере того как для них находятся новые сферы применения или возникает потребность в функциональных возможностях, которые невозможно было предусмотреть в оригинальной архитектуре. Пора отказаться от мифа, что архитектура программного обеспечения должна быть полностью определена до начала процесса разработки.

Итерации

Итеративный процесс создания программного обеспечения — это стиль разработки, результатом которого являются элементы архитектуры этого ПО, создаваемые ритмично и синхронно. Типичный итеративный процесс создания программного обеспечения изображен на рис. 8.3.

Начало изображенного здесь процесса можно найти в нижнем левом углу рисунка. Мы здесь видим журнал предназначенных к разработке функциональных возможностей, описанных без излишних подробностей. Непосредственно перед началом разработки некоторой функциональной возможности она анализируется специалистами, которые знают среду заказчика и соответствующую технологию. Они делят эту функциональную возможность на "истории"[12] (stories), — единицы разработки, трудоемкость которых может быть достаточно надежно оценена и которые могут быть реализованы в течение нескольких дней. На общей планерке коллектив определяет (основываясь на предыдущем опыте), сколько таких "историй" может быть реализовано в следующей итерации, и берет на себя соответствующие обязательства. В течение итерации весь коллектив ежедневно собирается на краткие совещания, чтобы обсудить, как продвигается выполнение планов и не нуждается ли кто-нибудь в помощи. К концу итерации все "истории" должны быть *созданы* — интегрированы, протестированы, задокументированы и готовы к использованию. После этого собирается "оценочное" собрание (review meeting), на котором подводятся итоги, оценивается объем выполненной работы и рассматривается информация, полученная через обратную связь (результатом чего мо-

[12] Cohn M. *User Stories Applied*, Addison-Wesley, *2004*.

жет, например, стать еще одна "история" или добавление еще одной функциональной возможности в журнал предстоящей работы). После нескольких итераций будет создан полезный набор функциональных возможностей (useful feature set), готовый к передаче заказчику и вводу в эксплуатацию в его среде.

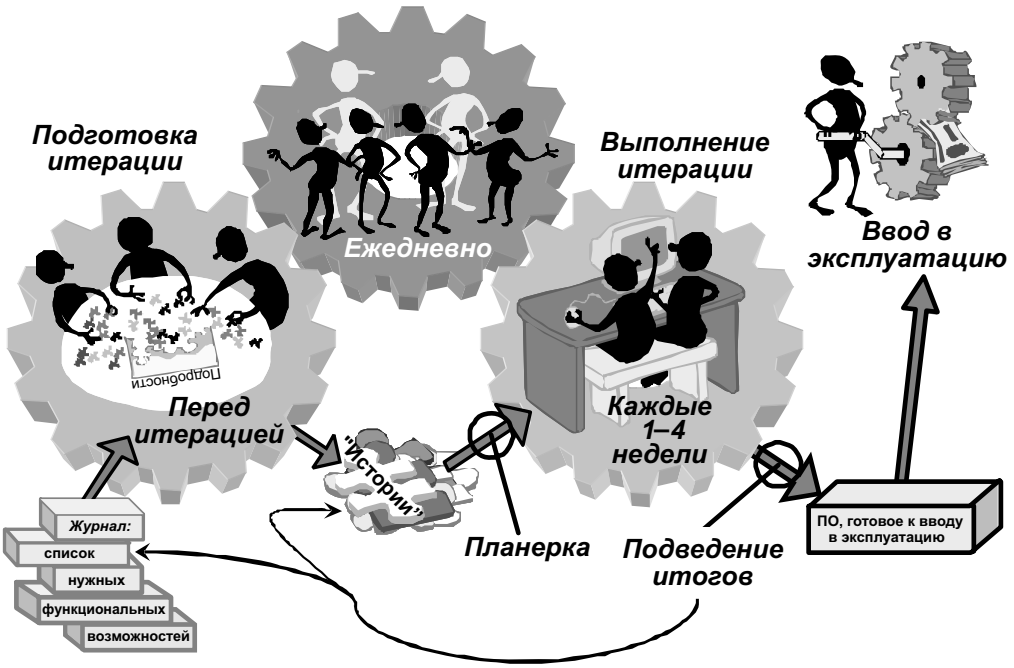


Рис. 8.3. Итеративная разработка[13]

Теперь рассмотрим типичный процесс итеративной разработки ПО более подробно (рис. 8.4).

Подготовка

На рис. 8.4 показан журнал предстоящей работы, который создается в самом начале процесса разработки. В этом журнале представлены требуемые функциональные возможности, ограничения, необходимые средства и т.д. Он представляет собой скорее краткую дорожную карту, чем длинный перечень дел, которые необходимо выполнить. Этот журнал динамический — элементы могут добавляться в него или удаляться из него произвольно (на основе приобретаемого коллективом опыта). Каждый элемент, представленный в журнале работы, имеет приблизительную оценку трудоемкости, и сумма всех оценок позволяет судить о времени, которое может понадобиться для завершения процесса разработки. Имеется чемпион (или в Scrum менеджер продукта (Product Owner)), следящий за тем, чтобы элементы журнала, более нужные заказчику (для которого создается данное ПО), были расположены ближе к его (журнала) началу.

[13] Изображение использовано с разрешения A Bit Better Corporation.

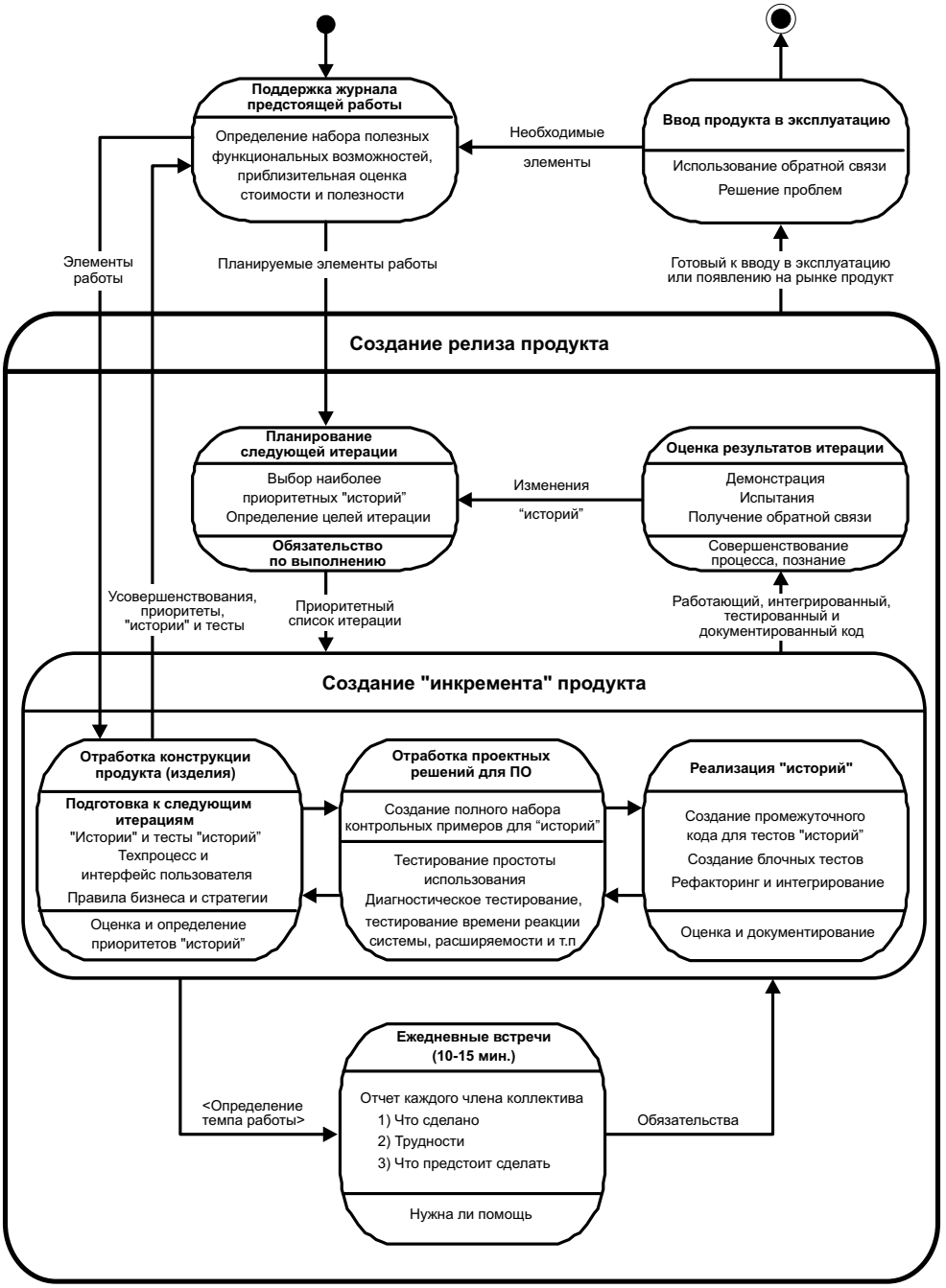


Рис. 8.4. Пример процесса итеративной разработки

Поскольку бережливый подход заключается в том, чтобы откладывать подробный анализ до последнего момента, вначале элементы журнала предстоящей работы обозначены в общем (и очень неопределенно). Однако с приближением времени, когда начнется выполнение той или иной работы из журнала, она (работа) должна быть разделена на более мелкие части (или "истории"), которые легче оценить и которыми легче управлять. Элемент журнала предстоящей работы не готов к разработке, пока он не представлен в виде одной или нескольких "историй". Разработчики, которым предстоит реализовать данную "историю", должны достаточно ясно представлять суть соответствующей работы, чтобы надежно оценить необходимые трудозатраты (надежные поставки требуют надежных оценок).[14] Каждая "история" должна иметь некоторую определенную потребительскую ценность (с точки зрения бизнеса, для которого предназначено создаваемое программное обеспечение). Критерием для "истории" является ее трудоемкость (как правило, для реализации одной "истории" требуется от половины дня до четырех дней). Разработчики, создающие программный продукт, должны решить, соответствует ли предполагаемая ценность данной "истории" затратам на ее реализацию.

Элементы журнала предстоящей работы обычно представляют собой функциональные возможности, описанные в терминах, обозначающих цели бизнеса, для которого создается данное ПО. (Описания целей бизнеса часто больше напоминают эпос, чем "историю".) С приближением времени, когда начнется выполнение той или иной работы из журнала, члены коллектива, хорошо понимающие суть работы заказчика (менеджеры продуктов, аналитики и т.д.), осуществляют разбиение "эпоса" на "истории".

Цель подготовки итерации состоит в том, чтобы определить часть "целого" продукта, которая будет разрабатываться следующей. При этом принимаются решения, касающиеся правил бизнеса, стратегии, технологического процесса, функциональных возможностей и интерфейса пользователя. Хорошая "история" представляет собой тщательно определенную порцию работы для разработчика, достаточно мелкую, что позволяет надежно ее оценить и выполнить в течение следующей итерации. При этом *не является* целью создание пространных аналитических документов для всего продукта. Цель в том, чтобы предоставить достаточно тестов для осуществления выборочного контроля (sample tests), которые бы сделали понятным для разработчика назначение данной "истории" (применительно к бизнесу заказчика).

Планирование

В начале итерации имеет место планерка (planning meeting). Весь коллектив в сотрудничестве с чемпионом или менеджером продукта оценивает, сколько времени потребуется на разработку, тестирование, документирование и ввод в эксплуатацию у заказчика наиболее приоритетных "историй". Члены коллектива выбирают одну из "историй" и обязуются ее создать. Затем выбирается вторая "история" и определяется, удастся ли успеть разработать ее также в ходе итерации. Этот процесс продолжается, пока члены коллектива не начнут сомневаться в том, что успеют создать очередную "историю".[15] Коллектив берет на себя обязательство выполнить цели итерации, которые заключаются в разработке набора функциональных возможностей, выбранных для данной итерации. Никто заранее не сообщает коллективу, каких трудозатрат потребует данный проект, од-

[14] Cohn M. Agile *Estimating and Planning*, Addison-Wesley, *2005*.
[15] Там же.

нако после нескольких итераций коллектив устанавливает собственный темп работы, который позволяет достаточно точно определить, какой объем работы можно сделать за итерацию.

Как уже отмечалось, члены коллектива берут общее обязательство выполнить определенный объем работы и работать вместе как единый коллектив для достижения поставленных целей. Может так случиться, что будут взяты завышенные обязательства или могут возникнуть непредвиденные технические трудности. Если такое произойдет, коллектив должен адаптироваться к ситуации соответствующим образом, однако при этом должны быть выявлены причины взятия излишних обязательств и приняты меры, чтобы это не повторилось.

Реализация

В ходе итерации весь коллектив работает вместе, чтобы выполнить задачи этой итерации. Все собираются на ежедневные встречи (10–15 минут) для обсуждения, что сделал каждый член коллектива после последней встречи и что планируется сделать до следующей, какие возникают проблемы и нужна ли кому-нибудь помощь. Совместные обсуждения на таких встречах предоставляют достаточно информации отдельным членам коллектива, чтобы знать, как следует действовать для выполнения задач, стоящих перед коллективом в целом. (При этом отпадает необходимость постоянно указывать каждому работнику, что ему следует делать.)

Предпочитаемый подход — разработка через приемочное тестирование (acceptance-test driven development). (Приемочные тесты так же известны, как тесты "историй".) Работая в каждый момент над одной "историей", члены коллектива сосредотачиваются на разработке функциональных деталей, технологическом процессе и пользовательском интерфейсе, чтобы выразить точно (в контрольных примерах (test cases)), как должен функционировать создаваемый продукт. Такая совместная работа позволяет определить соответствующие переменные и подходящие стратегии, а также учесть особенности бизнеса заказчика. Подобным образом определяются специфическая функциональность (в зависимости от информации на входе или последовательности действий (sequences of steps)), а также ожидаемые результаты.[16] В то время как некоторые члены коллектива занимаются созданием необходимого количества контрольных примеров (test cases), другие разработчики создают промежуточный программный код, позволяющий использовать эти контрольные примеры с разрабатываемым приложением, а также создают программный код, удовлетворяющий этим контрольным примерам. Когда разработчики нуждаются в прояснении какой-либо дополнительной детали, сотрудники, понимающие особенности работы заказчика, всегда на месте, чтобы обсудить вопрос и помочь создать соответствующие дополнительные тесты, которые бы проверяли достигнутые соглашения. Полный набор таких тестов фактически является очень подробной спецификацией, способной к тому же выполнить автоматическую проверку создаваемого программного продукта.

Для реализации функциональности, определенной тестом "истории", разработчики применяют разработку через блочное тестирование (unit-test driven development). Они начинают с выбора простой функциональности для создаваемого программного кода, которая бы явилась частью функциональности, для проверки которой предназначен данный тест "истории". Для реализации такой функциональности разработчики применяют

[16] Evans E. Domain *Driven Design*, Addison-Wesley, *2003*.

объекты, методы и параметры методов с соответствующими именами. Если возможно, выбираются имена, согласующиеся с именами, используемыми в тесте "истории". В качестве документирования своего проектного решения они создают блочный тест (unit test), который будет выполняться неудачно до тех пор, пока новый программный код не будет функционировать должным образом. Затем они создают простой программный код, с которым бы данный тест успешно выполнялся (при этом новый код не должен мешать выполнению каких-либо предыдущих тестов). Последнее действие в цикле состоит в том, чтобы обратиться к полученной библиотеке кода (code base) и усовершенствовать структуру программного кода (путем рефакторинга) с целью удалить повторения, сделать его простым и легким в понимании. Разработчики повторяют этот цикл, каждый раз добавляя новую дополнительную функциональность до тех пор, пока выбранный тест "истории" не выполнится успешно. Когда успешно выполнится весь набор тестов для данной "истории", наступает время перейти к следующей "истории".

Как работает Fit

В своей книге *Fit for Developing Software*[17] ("Fit для разработки программного обеспечения") авторы Рик Магридж (Rick Mugridge) и Уард Кэннингхем (Ward Cunningham) представляют средство Fit (принадлежащее Open Source). Данное средство призвано помочь коллективам обмениваться информацией путем создания точных примеров того, что должен делать разрабатываемый программный код. Эти примеры представлены в табличном формате. Когда между таблицами добавляется текст, он играет роль доступной для чтения спецификации. Разработчики создают связи между кодом приложения и таблицами. При выполнении приложения в интегрированной среде Fit осуществляется проверка, что каждая заданная функциональность, определенная в таблицах, корректно реализована. Когда связь (между программным кодом и таблицами) установлена, аналитики и тестировщики могут создать любое количество контрольных примеров (test cases), чтобы обеспечить должную работу приложения по мере расширения его функциональности. Средство Fit генерирует отчеты, представляющие собой таблицы с входной информацией, отмеченные зеленым цветом для успешно выполнившихся тестов и красным — для не выполнившихся. Данные отчеты позволяют оценить, насколько имеющееся функционирование программного кода близко к заданному. (Том Поппендик)

"История" не считается созданной, пока коллектив не приведет в порядок все разделы документации для пользователей и заказчиков, а также касающейся соответствия техническим условиям. При этом существование частично созданных "историй" не допускается. Цель в том, чтобы к концу итерации иметь полностью завершенный, готовый к релизу набор функциональных возможностей. Некоторые коллективы идут еще дальше и в конце каждой итерации "запускают вновь созданное ПО в производство". (Иными словами, ПО тиражируется и его копии отправляются потребителям.)

Оценка

В конце итерации созывается "оценочное собрание", призванное продемонстрировать всем имеющим отношение, какой объем потребительской ценности был создан. Если при этом обнаружится что-то, что требуется переделать, это может быть "оформлено" в виде еще одной "истории" (если это какая-то мелочь) или помещено (в виде элемента с соответствующим приоритетом) в журнал предстоящей работы (если это что-то покрупнее).

[17] Mugridge R., Cunningham W. *Fit for Developing Software: Framework for Integrated Tests*, Prentice Hall, *2005.*

На рис. 8.5 изображены три вложенных цикла приобретения знания, один для целой итерации, один для каждой "истории" и один для каждой минимальной "порции" программного кода. Постоянно растущие наборы тестов "историй" и блочных тестов соответствуют текущему знанию о том, как создаваемое программное обеспечение должно функционировать, чтобы быть полезным для заказчика при выполнении им своей работы. В ходе разработки ПО эти наборы тестов постоянно нуждаются в преобразовании согласно вновь приобретенному знанию. Удобство такого подхода в том, что применение этих наборов тестов очень снижает стоимость внесения изменений, поскольку, если случайно будет нарушено принятое ранее проектное решение для программного кода, тест немедленно это выявит. В результате "производственная линия будет остановлена" и коллектив разработчиков либо устранит внесенный в код дефект, либо нужным образом преобразует тест.

Вариация: интерфейс пользователя

На практике большинство организаций модифицируют итеративный процесс в соответствии со своими особенностями. В частности, это касается разработки пользовательского интерфейса. В главе 3 речь шла о компании Alias (которая в настоящее время является частью компании Autodesk), специализирующейся на трехмерной графике, в программных продуктах которой особое внимание уделяется пользовательскому интерфейсу. Поскольку удачный пользовательский интерфейс является основой для получения компанией конкурентного преимущества, в большинство коллективов разработчиков назначаются два проектировщика, которые занимаются только интерфейсом. Во время каждой итерации перед этими проектировщиками стоит несколько задач:

- сбор информации от заказчика для будущих итераций;
- тестирование элементов пользовательского интерфейса, созданных в предыдущей итерации;
- ответы на вопросы, возникшие в процессе создания программного кода в текущей итерации;
- определение подробностей, как код для пользовательского интерфейса должен создаваться в следующей итерации.

Этот процесс представлен на рис. 8.6. Прежде чем приступить к созданию программного кода для пользовательского интерфейса, собирается соответствующая информация от заказчика, испытываются различные варианты на прототипах и таким образом определяются детали различных элементов интерфейса, необходимые при создании соответствующего программного кода. Тестирование удобства и простоты использования (usability) готового кода осуществляется на следующей итерации.

Дисциплина

Мы проводили наши курсы в компании, где для процесса разработки программного обеспечения была характерна хаотичность. Все шло хорошо, пока не была затронута тема качества, вызвавшая некоторое разочарование у аудитории. "Вы полагаете, что нам следует все это делать? Мы думали, что бережливость — это когда все делается быстро, но материал о качестве наводит на мысль о большом объеме работы!"

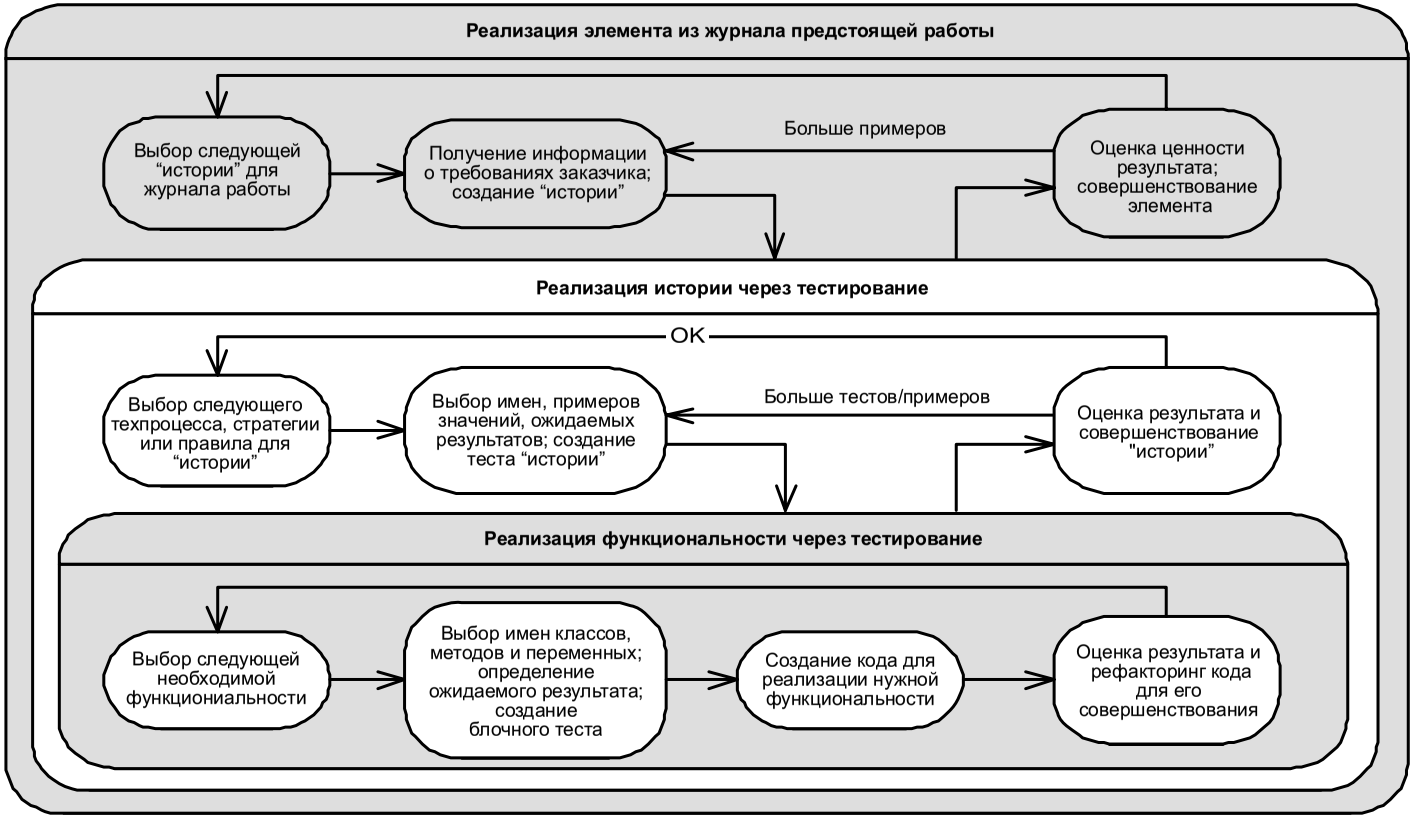


Рис. 8.5. Вложенные циклы создания знания

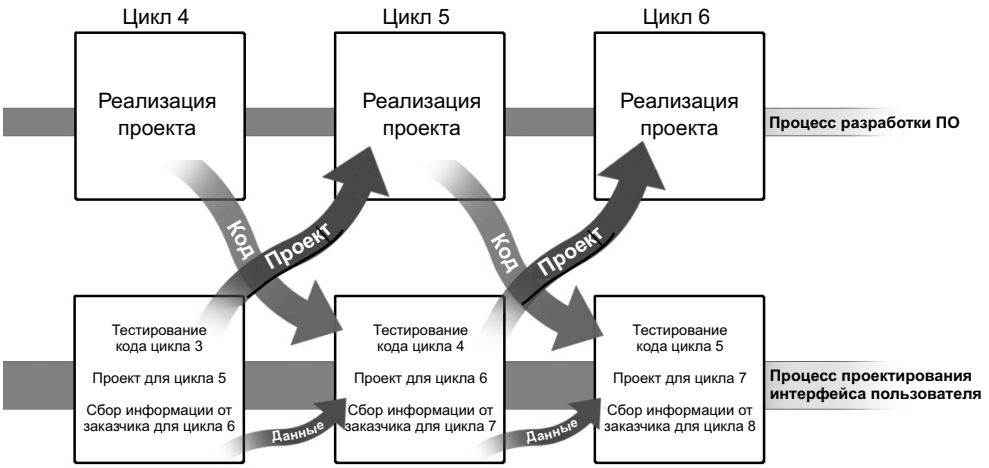


Рис. 8.6. Итеративный процесс создания пользовательского интерфейса[1]

Вскоре после этого мы проводили семинар в одной компании, специализирующейся на создании аэрокосмической техники. Когда речь зашла о качестве, мы услышали: "Это мы уже делаем многие годы." И это действительно было так. Люди в этой группе знали об автоматизированном тестировании, вероятно, не меньше, чем кто бы то ни было.

Нельзя быстро двигаться вперед, не "встраивая качество в продукт", а это требует высокой дисциплины. Мы обнаружили, что такая дисциплина становится второй натурой для людей, работающих с системами, от которых может зависеть жизнь; то же можно сказать о компаниях, где реализуется программа "Шесть Сигм" и аналогичных. Однако для небольшой, но быстро растущей компании, где качеству уделяется не очень много внимания, быть бережливым означает быть очень дисциплинированным при создании программного обеспечения.

Пять "S"

Когда заходишь в комнату, где работает коллектив разработчиков, просто оглянувшись, можно немедленно оценить уровень существующей здесь дисциплины. Если в комнате беспорядок, это, вероятно, свидетельствует о беззаботности коллектива, а если коллектив беззаботен, можно не сомневаться, что это отразится на качестве создаваемого здесь программного кода. В то же время в организации, в которой дела идут хорошо, люди знают, где что следует искать, поскольку здесь есть место для всего, и все здесь всегда на своем месте. Например, на предприятии по выпуску видеокассет, где работала Мэри, проверки чистоты и порядка проводились еженедельно, — как в помещениях, где работали программисты, так и в производственных помещениях.

---

[1] От Линн Миллер (Lynn Miller), ответственной за разработку интерфейса пользователя, Autodesk, Toronto, Использовано с разрешения.

"Пять S" — это классическая программа поддержания бережливости, предназначенная для организации рабочего пространства, где все оказывается под рукой и где нет ничего лишнего. Под "пятью S" подразумеваются пять понятий, обозначенных японскими словами, начинающимися на эту букву. Пять соответствующих английских слов начинаются на ту же букву: sort (упорядочить), systematize (систематизировать), shine (чистить), standardize (стандартизировать) и sustain (поддерживать).

Мы недавно реконструировали нашу кухню, и когда мы осваивали новую обстановку, обнаружили, что подсознательно обустроили наше рабочее пространство в соответствии с этими пятью "S".

1. **Упорядочить (Sort).** Прежде всего мы занялись упорядочением наших кухонных принадлежностей и отложили в сторону все, что не использовалось в течение года. Только то, что мы действительно использовали, "вернулось" в новую кухню.
2. **Систематизировать (Systematize).** Важно было все разместить таким образом, чтобы просто было найти и все иметь под рукой. Мы перемещали полки, приобрели новые шкафчики с выдвижными ящиками и установили крючки в удобных местах на стенах. Мы несколько раз все переделывали, пока не нашли подходящее место для каждого предмета.
3. **Чистить (Shine).** Наконец, разместив все по своим местам, мы навели порядок в кухне и после этого были готовы к приготовлению пищи.
4. **Стандартизировать (Standardize).** Затем мы договорились установить два (новых!) правила: мы будем каждый вечер загружать и использовать посудомоечную машину и утром мы будем все возвращать на свои места.
5. **Поддерживать (Sustain).** После этого все, что оставалось, это придерживаться установленных правил дисциплины.

Рабочая среда, где осуществляется разработка программного обеспечения, — это не только "физическая комната", но также рабочий стол на экране компьютера, обустройство сервера коллектива, а также программный код, над которым все работают. Поэтому, после применения правил (пять "S") к комнате, подумайте о применении тех же правил к логической рабочей среде. И поскольку качество программного обеспечения является отражением порядка в создающей его организации, присмотритесь также к самому коду.

1. **Упорядочить (Sort).** Упорядочьте все содержимое рабочих станций и серверов коллектива, выявите старые версии программного обеспечения, а также старые файлы и отчеты, которые больше никогда не будут использоваться. Если нужно, создайте их резервную копию, а затем удалите их.
2. **Систематизировать (Systematize).** Компоновка рабочего стола и организация файлов в компьютере важны. Они должны быть такими, чтобы все было логически организовано и все было легко найти. Любая рабочая среда, используемая более чем одним человеком, должна быть такой, чтобы люди могли найти то, что им нужно на любом компьютере, с которого они вошли в систему.
3. **Чистить (Shine).** Это предполагает много работы. Наступило время выбросить все эти лишние банки и кофейные стаканчики, стереть следы пальцев на экранах мониторов и убрать всю лишнюю бумагу. Также пора очистить все доски (на которых пишут и рисуют мелом), предварительно скопировав все имеющиеся на них нужные эскизы.

4. **Стандартизировать (Standardize).** Примените к имеющимся рабочим станциям единые стандарты. Для этого убедитесь, что на них на всех установлены последние версии всех программ, резервное копирование осуществляется регулярно и разнообразный компьютерный мусор в них не накапливается.
5. **Поддерживать (Sustain).** Теперь вам остается только поддерживать наведенный порядок.

Стандарты

Когда путешествуешь по миру, начинаешь по-настоящему ценить важность стандартов. Мы ценим сеть железных дорог и метро, которые имеются почти в каждом крупном европейском городе, но мы особенно дорожим этим в Соединенном Королевстве, поскольку там мы делаем все от нас зависящее, чтобы не садиться за руль. Недавно в Австралии нам все-таки пришлось водить машину, но и после двух недель мы были вынуждены проявлять крайнюю бдительность перед каждым поворотом, чтобы не попасть на встречную полосу (рис. 8.7, *слева*). Каждый наш электроприбор может питаться от любой сети с любым напряжением и мы возим с собой множество переходников (рис. 8.7, *справа*). Также мы приучились измерять температуру по Цельсию, а расстояние — в километрах, но мы никак не привыкнем к электрическим выключателям, где положение "вверх" — это выключено.

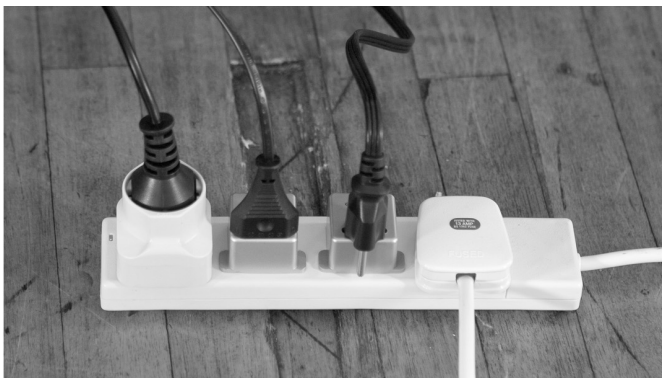


Рис. 8.7. Стандарты?

Стандарты делают возможным действовать рефлексивно (т.е. не задумываясь) и осуществлять обмен информацией без затрат на преобразование. Стандартизованная инфраструктура с общей архитектурой уменьшает сложность и, следовательно, снижает затраты. Ни одна организация, которая стремится динамично развиваться, не сможет обойтись без стандартов. Вот несколько стандартов, которые организации, занятой разработкой программного обеспечения, целесообразно использовать.

1. Соглашения об именах.
2. Стандарты создания программного кода.
3. Соглашения о взаимодействии пользователей.
4. Файловые структуры.

5. Практика управления конфигурациями.
6. Программные средства.
7. Стандарты регистрации ошибок.
8. Стандарты безопасности.

Конечно, стандарты бесполезны, если они остаются на бумаге; чтобы от них была польза, они должны использоваться. В среде, где реализованы бережливые принципы, стандарты рассматриваются как наилучший (в настоящее время) способ выполнения работы, и потому всегда соблюдаются. Однако предполагается, что всегда есть лучший способ. Каждый раз, когда обнаруживается лучший способ, он становится новым стандартом. Дисциплина соблюдения (и постоянного изменения) стандартов должна стать девизом организации.

К стандартам нужно относиться серьезно!

Мне известен случай, когда имелись две программы, обе с именем "SYNC". Они содержались в одних и тех же папках, но в двух разных системах. В системе 1 программа SYNC сохраняла локальные файлы путем слияния их в единый мастер-файл, в то время как в системе 2 программа SYNC восстанавливала локальные файлы, заменяя их мастер-файлом. Оператору, которая обычно имела дело с системой 1, пришлось однажды работать с системой 2. Она нашла программу SYNC в той же папке, где она привыкла ее всегда находить, и потому предположила, что это все та же SYNC. Она запустила программу с целью объединить информацию о большом количестве недавних сделок (transactions) в один мастер-файл. Однако, к ее ужасу, состоялось копирование информации мастер-файла (со старыми данными) поверх ее локальных файлов; при этом вся информация, которую она пыталась сохранить, пропала. Это была очень дорогостоящая "ошибка", причина которой заключалась в отсутствии стандартизации имен. (Мэри Поппендик)

Экспертиза программного кода

На наших семинарах нам часто задают вопрос: "Не является ли источником потерь экспертиза программного кода?" Хороший вопрос. Мы полагаем, что использование экспертиз для укрепления роли стандартов или даже для поиска дефектов, безусловно, ведет к потерям. Для укрепления роли большинства стандартов целесообразно использовать анализаторы кода (code analyzers) и IDE-проверки (IDE checks), а практика автоматизированного тестирования является популярным способом избежать большинства дефектов. Экспертиза программного кода должна быть нацелена на совсем иную группу проблем. Например, код, созданный разработчиками с недостаточным опытом, может на таких экспертизах проверяться на отсутствие излишней сложности, толерантность к изменениям, отсутствие повторений и использование иных хороших практик объектно-ориентированного программирования. В одной известной нам организации для каждого созданного программного кода вычисляется цикломатический индекс сложности Мак-Кэйба[2] (McCabe Cyclomatic Complexity Index), и когда его значение достигает значения 10, это служит основанием для проведения экспертизы данного программного кода.

[2] Цикломатический индекс сложности Мак-Кэйба соответствует числу путей выполнения (execution paths) в программе. McCabe T.J. *A Complexity Measure*, IEEE Transactions on Software Engineering, Vol. Se-2, No..4, *December 1976*.

Формальная экспертиза программного кода

В нашей организации группа разработчиков, создававших до этого программное обеспечение на Cobol, были переведены на использование Java. Они прошли курс обучения синтаксису Java, но объектно-ориентированное мышление оказалось непривычным для людей, много лет занимавшихся процедурным программированием. Мы использовали экспертизы программного кода, чтобы продемонстрировать разработчикам, как при создании и рефакторинге объектно-ориентированного кода, в зависимости от ситуации, использовать стандартные подходы, методы и приемы. Когда разработчик завершал создание раздела программного кода и заявлял о желании организовать экспертизу, это осуществлялось при первой возможности. Разработчик представлял код двум техническим экспертам, которые затем обсуждали, как его можно улучшить. Эксперты не всегда приходили к единому мнению, а имеющая место дискуссия, скорее, указывала разработчикам направление поиска, чем давала готовые ответы. Атмосфера таких экспертиз была настолько открыта и поучительна, что они стали очень популярны. Все больше и больше разработчиков присутствовали на этих экспертизах, поскольку здесь многому можно было научиться. Благодаря этому профессиональные качества новоиспеченных Java-программистов значительно улучшились. (Джилл Аден[3])

Некоторые правила, требующие экспертизы программного кода до его формального принятия, ведут к созданию значительных объемов незавершенной работы (ожидающей экспертизы). В среде, где реализованы принципы бережливости, это неприемлемая ситуация. Один из способов осуществления экспертизы программного кода без задержек — использование парной разработки кода.

Парное программирование

Парное программирование (pair programming или pairing) — это практика объединения двух людей для работы над одной задачей. При этом (помимо прочего) осуществляется также непрерывная экспертиза программного кода. Разумное использование парного программирования может оказаться очень ценным. Разработка программного кода вдвоем ведет к повышению квалификации, улучшению навыков совместной работы, а также совершенствованию умения решать проблемы. Более того, работа вдвоем часто увеличивает производительность, поскольку, когда код просматривают две пары глаз, улучшаются как его качество, так и "живучесть". Кроме того, работа вдвоем не позволяет отвлекаться и заставляет партнеров сосредотачиваться на задаче. Каждый, кому случалось заниматься программированием, скажет, что разработка программного обеспечения — это продолжительный процесс решения проблем, а совместное решение проблем является подходом, использование которого можно обнаружить во многих сферах деятельности. Помимо прочего, работа в паре облегчает "вливание" новых людей в коллектив, поскольку она подразумевает также наставничество.

Работа в паре подходит не для всех людей и годится не для любой ситуации. Однако совместная работа часто создает кумулятивный эффект: двое людей нередко создадут более интегрированный, надежный, свободный от дефектов программный код, работая совместно, чем если бы они работали по отдельности. И работа в паре — это один из лучших способов воспользоваться преимуществами экспертиз программного кода, не накапливая при этом незавершенную работу.

[3] Материал взят из презентации на встрече Twin Cities OTUG, 17 июня 2003 года, и последующих бесед. Использовано с разрешения.

Экспертизы в Open Source

В сообществах Open Source руководители проектов (committers) проверяют весь программный код, прежде чем он будет помещен в библиотеку кода. Роль руководителей проектов выполняют доверенные пользователи, продемонстрировавшие свою компетентность и приверженность проекту. Обычно на одного руководителя проекта приходится десяток ассистентов (contributors), и программный код всегда представляется небольшими порциями. Это позволяет такой код очень быстро проверить и добавить в библиотеку. После этого программный код становится доступен для всего сообщества, где он немедленно проверяется и сопровождается множеством рекомендаций. Считается, что если вы хотите по-настоящему научиться программировать, попрактикуйтесь в Open Source, и вы получите быструю обратную связь. (Мэри Поппендик)

Защита от ошибок

Когда вы подсоединяете проектор к ноутбуку с помощью видеокабеля, это трудно сделать неправильно, поскольку данный кабель защищен от ошибок. Одна сторона его разъема шире другой и содержит больше контактов. Присмотревшись, большинство людей быстро ориентируются что к чему. С другой стороны, кабель USB не так прост. Как часто вы пытались его подсоединить, а затем понимали, что разъем нужно перевернуть? Хотя практически невозможно подключить USB-кабель неправильно, не всегда сразу это очевидно. По нашему мнению, этот кабель недостаточно защищен от ошибок.

Однако бывают и худшие случаи. IDE-кабель появился в компьютерах IBM AT в 1984 году, где он использовался для подключения жесткого диска к системной плате. Много лет этот кабель имел 40 гнезд, которые подсоединялись к 40 штырькам на диске. Имелась красная полоска на боковой поверхности разъема кабеля, указывающая, где расположен контакт 1, но обычно трудно было определить, где контакт 1 на диске — либо красную полоску было трудно заметить, либо вы не проявляли достаточно внимания. В результате нередко кабель подсоединялся неправильно и немало вышедших из строя дисков и контроллеров могут быть отнесены на счет того, что кабель не был защищен от ошибок. Чтобы облегчить ситуацию, на разъемы добавлялись разного рода метки, но они часто были неудачно расположены и, как правило, были неэффективны. Годы потребовались отрасли, прежде чем из разъема каждого диска был удален штырек 20, а также соответствующее гнездо из разъема на кабеле, что сделало неправильное подсоединение кабеля невозможным (рис. 8.8).

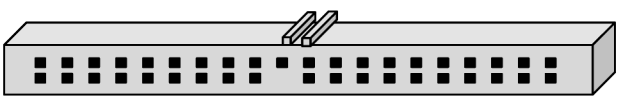


Рис. 8.8. Разъем IDE невозможно подсоединить неправильно

Когда на наших курсах мы спрашиваем присутствующих, приходилось ли им собирать компьютер с непомеченным IDE-кабелем, обычно поднималось много рук. После этого мы предлагали тем, кому случалось подключить этот кабель неправильно, опустить руки. Неизменно все руки опускались. Нам (авторам) также случалось совершать подобные ошибки, и мы выяснили для себя две вещи: каждый, кому приходилось подключать непомеченный IDE-кабель, считал себя экспертом, и каждый знал, что следует проявлять осторожность. И тем не менее почти всем случалось в то или иное время ошибаться при подключении этого кабеля.

Ошибки не вина лиц их допускающих; в ошибках виновата система, не защитившая от них места, где ошибки вероятны. В том, что касается программного обеспечения, все возможные неприятности здесь рано или поздно случаются. Поэтому не тратьте время, подсчитывая случаи, когда разработчики виноваты в появлении дефектов, и оказывая на них давление с тем, чтобы они проявляли больше осторожности. При каждом обнаружении дефекта прекратите работу, найдите источник возникновения этого дефекта и примите меры, чтобы подобные дефекты не появлялись в будущем.

Автоматизация

Один из наиболее эффективных способов защищенной от ошибок разработки — автоматизация рутинных задач.[4] Даже в небольших коллективах следует автоматизировать все, что только можно, и даже единичные задачи должны рассматриваться как кандидаты на автоматизацию. Автоматизация не только исключает ошибки, которые часто допускают люди, ее можно рассматривать как проявление уважения к их интеллекту. Повторяющиеся задачи не только чреваты ошибками; это сигнал, что никто не возражает против того, чтобы на людей смотрели как на роботов. Люди не должны выполнять работу механически; они должны искать эффективные способы выполнения своей работы и решения проблем.

Примеры автоматизации

1. **Сборка в результате одного щелчка.** Автоматизация сборки — это только первый шаг. После сборки следует воспользоваться набором автоматизированных тестов, чтобы проверить, не пришло ли что-нибудь в негодность. Часто для этого используют Ant (инструмент, который призван автоматизировать процесс сборки).
2. **Сборки по расписанию.** Как только появляется возможность осуществить сборку за одно действие, становится легко планировать их (сборки). Сборка может быть инициирована с помощью часов или по завершении создания программного кода. Часто для этого используют Cruise Control. (Cruise Control — инструмент, написанный на Java, для организации процесса постоянной интеграции при разработке любых программных продуктов.)
3. **Уведомление о результатах сборки.** Бессмысленно осуществлять сборку, никому не сообщая о ее результатах. Совсем несложно инициировать уведомление с использованием электронной почты, если сборка окажется неудачна.
4. **Релиз за одно действие.** Вероятно, релиз означает создание ветви (release branch) и упаковку программного кода в один или несколько файлов, которые могут загружаться из Интернета или распространяться на компакт-дисках. Этот процесс непременно следует автоматизировать. Здесь нет места ошибкам.
5. **Надежная инсталляция.** Процесс инсталляции (когда дистрибутив попадает к заказчику) должен быть автоматизирован, даже если у заказчика имеются представители фирмы-создателя программного обеспечения. Дистрибутивный носитель должен также содержать несколько диагностических средств, призванных помочь в случае, когда (ни в коем случае не если) инсталляцию осуществить не удастся.

---

[4] Mike C. *Pragmatic Project Automation: How to Build, Deploy and Monitor Java Applications*, Pragmatic Press, *2004*; Neward T., Clark M. *Pragmatic Project Automation for .NET*, Pragmatic Press, *2007*. В этих книгах можно почерпнуть массу идей о том, что и как автоматизировать.

Разработка через тестирование

Как упоминалось в главе 2, Сигаэо Синго (Shigeo Shingo) утверждал, что возможны две разновидности тестов (или контроля): тесты для выявления уже существующих дефектов и тесты, предназначенные их предотвращать. При этом первую разновидность он рассматривал как не оставляющий сомнений источник потерь. Целью бережливого подхода в разработке программного обеспечения является, прежде всего, предотвращение появления дефектов в программном коде, а осуществить это можно с помощью разработки через тестирование (Test Driven Development — TDD).

"Мы не можем допустить, чтобы тестировщики создавали тесты до того, как будет готов программный код, — говорил нам один тестировщик-менеджер. — Если это позволить, разработчики просто будут создавать код, лишь бы выдерживающий тестирование!" Вот так! Некоторые люди полагают, что задача тестировщиков состоит в том, чтобы интерпретировать спецификации и транслировать это в тесты. Вместо этого почему бы не привлечь тестировщиков к созданию спецификаций в виде выполняемых тестов? Это позволило бы получить защищенный от ошибок процесс трансляции, и если это делать с помощью соответствующих средств, такой подход может обеспечить автоматическую трассируемость от спецификации до программного кода. Чем более регламентирована ваша отрасль, тем более привлекательны могут быть тесты, играющие роль выполняемых спецификаций.

Если задача тестирования — скорее предотвращать появление дефектов, чем выявлять их, тогда необходимо разобраться, что это означает для различных видов тестов, которые обычно используются. Брайан Морик (Brian Marick) полагает, что взгляд на тестирование возможен с четырех точек зрения, которые представлены на рис. 8.9. Этот рисунок демонстрирует, что тестирование имеет две цели: облегчать работу программистов, а также испытывать в различных режимах весь продукт (изделие) в целом, составной частью которого является создаваемое программное обеспечение. Это можно делать как с точки зрения технологии, так и с точки зрения бизнеса. Это дает нам четыре общих категории тестов, которые кратко рассматриваются ниже.

Блочные тесты

Блочные (или модульные) тесты (unit tests) создаются разработчиками для проверки, что созданный программный код действительно соответствует их замыслам. Когда создание программного обеспечения начинается с тестов, в этом случае проектные решения для будущего программного кода создаются на их (этих тестов) основе, и набор блочных тестов становится спецификацией для проекта будущего ПО. Иными словами, предварительное создание тестов обычно обеспечивает более простой программный код. Замечено, что как только разработчики осваивают данный подход (начало разработки с тестов), они уже не хотят создавать программное обеспечение каким-либо иным способом.

Набор блочных тестов используется как тестовая нагрузка (test harness), которая выполняется при сборке. Набор тестов, используемых при сборке, должен выполняться быстро (не более 10 минут), иначе разработчики будут избегать пользоваться им. Поэтому код, тестируемый с помощью такого набора тестов, часто используется не с реальной базой данных, а с различными имитаторами, чтобы ускорить процесс.

Блочные тесты используются не только для проверки небольших элементов программного кода. Эти средства используются также для тестирования на любом уровне: на блочном, на уровне функциональных возможностей или даже всей системы.

Рис. 8.9. Типы тестирования

Тесты “историй”

Тесты “историй” (которые так же известны, как приемочные тесты (acceptance tests)) определяют, насколько данное программное обеспечение соответствует нуждам заказчиков. Когда тесты “историй” создаются до программного кода, это помогает всем лучше вникнуть в суть работы заказчика, а также выяснить, как эту работу может облегчить данное ПО. Коллектив разработчиков изучает примеры того, что система должна делать, и в результате тесты “историй” становятся основанными на примерах спецификациями. Если спецификации решено создавать предварительно, целесообразно создавать их в виде выполняемых тестов.

Тесты “историй” должны быть автоматизированными и запускаться так часто, как это возможно. Как правило, такие тесты не являются частью набора тестов для сборки

(build test suite), поскольку они обычно нуждаются в сервере или базе данных. Тем не менее некоторые тесты "историй" должны использоваться ежедневно, а более полный их набор должен запускаться еженедельно, и все тесты должны успешно выполняться к концу итерации.

Тесты на простоту использования и диагностические тесты

Тесты на простоту использования (usability) и диагностические (exploratory) тесты по определению являются тестами, выполняемыми в ручном режиме. Когда все автоматизированные тесты успешно выполняются, мы знаем, что система делает то, что она должна делать, однако еще не известно, как она будет восприниматься пользователями и не будет ли она в некоторых режимах (для которых пока никто не додумался создать тесты) функционировать непредсказуемо. Во время тестирования на простоту использования систему используют реальные пользователи в реальных условиях. Во время диагностического тестирования опытные специалисты по тестированию выясняют, как система поведет себя при предельных нагрузках или при непредвиденных входных данных. Когда диагностические тесты обнаруживают неустойчивость в программном коде, должен быть создан соответствующий автоматизированный тест, призванный указать разработчикам, что требуется доработать. (И этот тест должен быть добавлен в тестовую нагрузку (test harness).)

Тестирование качества

Тестирование качества (property testing) служит для проверки так называемых нефункциональных свойств системы. Речь идет о времени реакции системы, масштабировании, устойчивости и т.д. Для тестирования системы под нагрузкой, для проверки проблем безопасности и т.п. существуют специальные средства. Также имеются средства, генерирующие комбинаторные тесты, которые проверяют все возможные комбинации входных данных, с которыми может столкнуться система. Если эти средства подходят для вашей среды, не поленитесь освоить их и используйте данные средства максимально широко. Начинайте использовать их в процессе разработки как можно раньше. Старайтесь применять эти средства в среде, которая максимально близка к реальным условиям.

Управление конфигурацией

Управление конфигурацией является основой в любой среде разработки программного обеспечения, а практика гибкой (agile) разработки выдвигает существенные требования к системе управления конфигурацией. Необходимо учитывать следующее.

1. Любой фрагмент программного кода может использоваться несколькими людьми одновременно.
2. Релизы должны осуществляться часто и включать небольшие порции программного кода.
3. Когда один набор функциональных возможностей готов, осуществляется разработка новых функциональных возможностей.
4. Весь код постоянно подвергается рефакторингу (непрерывному совершенствованию).

Система управления конфигурацией играет роль библиотекаря, который "выдает" строки программного кода людям, которые предполагают внести в него изменения; за-

тем, когда новые версии этих строк кода "возвращаются", система аккуратно размещает их по своим местам. Эта система может использоваться не только для управления хранилищами программного кода, но также документацией и результатами тестов. Каждая организация, занимающаяся разработкой программного обеспечения, должна иметь систему управления конфигурацией, поддерживающей сценарии, используемые в данной организации, а также определять правила использования системы.

Вот типичные сценарии, обрабатывать которые призвана система управления конфигурацией.

1. Разработчик "берет" файлы в свое "личное" рабочее пространство. Он вносит в них изменения, а затем, прежде чем вернуть код на место, объединяет изменения с текущим кодом на своей машине и осуществляет свою частную сборку, а затем тестирование. Если все работает, разработчик возвращает программный код в систему управления конфигурацией, которая после этого инициирует процесс официальной сборки и тестирования.
2. Во время релиза создается "ветвь", которая включает весь программный код, предназначенный для включения в релиз. При этом разработчики продолжают добавлять новые функциональные возможности в основную линию программного кода. Любые изменения, которые вносятся в релиз во время тестирования или поддержки, переносятся в основную линию программного кода так быстро, как это возможно. (Когда код направляется в релиз как часть итерации или "ветвь" отдается заказчикам без изменений, данный сценарий излишен.)

Непрерывная интеграция

Когда программный код "забирается" в личное рабочее пространство или создается отдельная линия программного кода (separate code line), между двумя линиями кода неизбежно возникают различия. Чем дольше существуют параллельные линии программного кода, тем больше между ними будет различий. И наоборот, чем чаще осуществляется интеграция этих линий, тем легче выявлять зарождающиеся проблемы и определять их источник. Это настолько очевидно, что регулярное осуществление интеграции, как стандартную практику, можно обнаружить повсюду. Но не все так просто. Прежде всего, частая интеграция с точки зрения разработчиков противоречит логике. Инстинкты подсказывают им, что работа должна быть завершена, прежде чем публиковать ее. Во-вторых, организационные действия требует времени, и это усложняет проблему. Останов работы и устранение проблем, обнаруженных при осуществлении сборки и тестировании, могут очень все замедлить. Поэтому имеется мощный стимул аккумулировать обширные объемы программного кода, прежде чем интегрировать его в библиотеку кода.

Увеличение времени между осуществлением интеграции — это фальшивая эффективность. Бережливый подход состоит в том, чтобы сокращать это время и уменьшить его до значения, когда можно сказать, что имеет место постоянная интеграция, быстрая и безболезненная. Это трудно, частично потому, что развитие способности быстро осуществлять сборки и тестирование требует времени, квалификации и постоянного внимания, а также частично потому, что останов работы для устранения проблем сбивает с ритма. Однако огромные преимущества такого подхода становятся очевидны, когда занятый разработкой ПО коллектив находит способы осуществлять быстрые сборки и непрерывную интеграцию, и каждый работник неизменно останавливает работу для устра-

нения проблем сразу, как только они обнаруживаются. Нам много раз говорили, что это трудно организовать, но, как только подобный подход заработает, никто уже не хочет возвращаться к старым методам работы. Неизменно в подобных случаях коллектив испытывает все усиливающуюся способность справляться со своей работой и это скептиков превращает в энтузиастов.

Вложенная синхронизация

Как отмечалось ранее в этой главе, крупные системы должны иметь поддающуюся делению архитектуру (divisible architecture), которая бы позволяла работать одновременно и независимо над различными подсистемами одной системы. Однако это еще не все. Что происходит, когда независимые коллективы пытаются объединить созданные ими подсистемы? Теоретически сопряжения (или интерфейсы) между подсистемами были заранее и тщательно определены и оставались неизменны, поэтому при объединении подсистем никаких сюрпризов возникнуть не должно. На практике однако, все происходит обычно не так. В действительности возникшие трудности с объединением подсистем в конце разработки являются одной из основных причин невыполнения планов, роста расходов и в итоге неудачных программ.

Если программное обеспечение представить в виде дерева, а подсистемы в качестве ветвей, возможно, это позволит увидеть, в чем заключается проблема. Никто не выращивает ветви отдельно одна от другой, а затем пытается приделать их к стволу дерева. Аналогично, не следует разрабатывать подсистемы совершенно независимо одна от другой, а затем пытаться “привить” их к некоторому общему стволу. Подсистемы должны “расти” от живого ствола — если ствол не существует, следует создать его в первую очередь. Тогда подсистемы органически будут интегрированы в единую систему и нам не придется беспокоиться об их “прививке” в конце.

Правило, которое мы определили для параллельных линий программного кода, подходит также для более крупных подсистем: чем дольше продолжается параллельная (и независимая) их разработка, тем больше несовместимостей будет иметь место. И наоборот, чем чаще осуществляется интеграция подсистем, тем легче выявлять возникающие проблемы и находить их причины. Вместо того чтобы определить интерфейсы в самом начале и ожидать, что они будут работать, в действительности следует сначала создать эти интерфейсы, добиться, чтобы они заработали, а уже затем создавать подсистемы, синхронизируя их одна с другой так часто, как это возможно.

Как это работает на практике? Все начинается с непрерывной интеграции. Несколько раз на день разработчики возвращают новые версии программного кода в библиотеку кода, что инициирует автоматические сборку и тестирование для проверки, что все по-прежнему синхронно на микроуровне. Тестовая нагрузка приемочных тестов (acceptance test harness) запускается на ночь. Различные конфигурации и платформы могут тестироваться в течение уик-энда. В каждой итерации вся база программного кода синхронизируется и приводится в готовность к релизу. Синхронизация может осуществляться с другими подсистемами или, возможно, с аппаратурой. И при каждом релизе код синхронизируется с заказчиками.

С крупными системами и географически рассредоточенными коллективами вложенная синхронизация еще более важна, поскольку она создает основу для постоянных обменов информацией и регулярно объединяет коллективное знание всех коллективов в одном месте. Коллективы, занятые разработкой тесно связанных подсистем, должны

синхронизировать свою работу, используя общую базу программного кода. Синхронизация крупных подсистем должна осуществляться в конце каждой итерации. В крупных проектах вся система должна синхронизироваться как можно чаще. Если в качестве примера вспомнить проект Polaris, то здесь точки синхронизации всей системы имели место при запуске первой ракеты, при тестировании версии A1, при поставке этой версии и т.д. Эти точки синхронизации всей системы никогда не отстояли одна от другой более чем на 18 месяцев, и можно не сомневаться, что кроме того существовали еще точки синхронизации подсистем. Хотя 18 месяцев могут показаться долгим сроком, общесистемная синхронизация все же осуществлялась в шесть раз более часто, чем это было предусмотрено в оригинальном графике для программы Polaris

Общесистемная синхронизация означает приведение системы в состояние готовности к релизу, насколько это осуществимо. Если имеются какие-либо функциональные тесты, точка синхронизации — подходящий повод воспользоваться ими. Все участвующие коллективы и основные заказчики должны участвовать в анализе и принятии важных решений в каждой точке синхронизации. Основные точки синхронизации — это не однодневное событие. Следует выделить несколько дней, чтобы основательно проанализировать готовность системы, абсорбировать все новое, что было сделано, сосредоточиться на ключевых решениях, принять которые пришло время, сделать необходимые коррекции курса, осуществить подробное планирование следующего этапа и отметить текущие успехи.

Попробуйте это

1. Мы использовали термины “план релизов” (release plan), “журнал предстоящей работы” (backlog) и “product road map” (дорожная карта продукта) более или менее взаимозаменяемо. Имеют ли эти термины одно значение в вашей организации? Если нет, в чем и почему их значения различаются? С какой продолжительностью (в неделях работы) ассоциируются у вас понятия, обозначаемые этими терминами?
2. Насколько продолжительны итерации в вашей организации? Почему у вас используется именно такой ритм работы? Сбор информации о проблеме заказчика (решать которую предназначено будущее ПО) осуществляется во время или до итерации?
3. Обсудите на собрании коллектива, в чем разница между разработкой через приемочное тестирование (acceptance-test driven development) и разработкой через блочное (или модульное) тестирование (unit-test driven development)? В чем преимущества и недостатки каждого подхода? Какой из них вы используете? Какой следует использовать?
4. Определите оценки для вашей организации (баллы 0–5), где 0 соответствует низшей оценке и 5 — высшей). Оценивать нужно по следующим параметрам:
   а) стандартизация архитектуры;
   б) стандартизация средств;
   в) соглашения по созданию программного кода;
   г) управление конфигурацией;
   д) автоматизация блочных (модульных) тестов;

е) автоматизация приемочных тестов;

ж) сборка и тестирование по щелчку;

з) непрерывная интеграция;

и) автоматизация релизов;

к) автоматизация инсталляций.

Вычислите общую сумму.

5. На следующей общей встрече предложите коллективу обратить внимание на помещение. Оцените его общий вид (0–5). Затем оцените упорядоченность вашей библиотеки кода по той же шкале. Совпадают ли результаты? Если оценка равна 3 или ниже, предложите коллективу проделать упражнение "Пять S" — сначала для помещения, затем для библиотеки кода.
6. Кто в вашей организации отвечает за определение стандартов? Кто должен отвечать? Насколько стандарты соблюдаются? Как легко они изменяются?

Глава 9

Партнеры

Совместные усилия

"Университетская академическая лодка-восьмерка (плюс рулевой) мирового класса может проплыть 2000 метров по воде приблизительно за 5,5 минут. Однако академическая одиночка (в которой гребец действует двумя веслами) может преодолеть ту же дистанцию приблизительно за 7 минут. Разница во времени объясняется тем, что при синхронной гребле скорость увеличивается, и если бы от быстрой гребли зависело выживание, лучше всего для этого подошли бы люди, умеющие сотрудничать."1

Сотрудничество нужно не для снижения себестоимости и не для снижения риска, и не для увеличения производительности. Основная причина необходимости сотрудничества в самом сотрудничестве. Сотрудничающие люди (и компании) смогут достичь лучших результатов, чем те, которые стремятся действовать в одиночку.

Чрезвычайное происшествие!

Вторник, 2 декабря, 2003 года. Серьезная брешь в безопасности была обнаружена на сервере Gentoo Linux в одном из университетов Италии. Системный администратор проследил проблему настолько, насколько это ему удалось, а затем через цепочку контактов связался с разработчиком из Австралии, который когда-то руководил разработкой соответствующего программного кода. Разработчик оставил все свои дела и работал всю (итальянскую) ночь вместе с системным администратором, чтобы выяснить точное местонахождение бреши. Затем он привлек своего товарища и они работали вместе всю (австралийскую) ночь, создавая соответствующую "заплату" (patch). Они отправили свою "заплату" небольшой группе пользователей-разработчиков сервера Gentoo Linux, которые были предупреждены заранее, и готовы для тестирования. Менее чем через 30 часов после обнаружения атаки объявление о бреши и протестированная "заплата" были распространены среди пользователей Linux по всему миру.

Эта история была рассказана Филиппом Эвансом (Philip Evans) и Бобом Вольфом (Bob Wolf) в их статье "Collaboration Rules"[2] ("Правила сотрудничества") в журнале

---

[1] Corning P.A. Ph.D.*The Synergism Hypothesis: On the Concept of Synergy and Its Role in the Evolution of Complex Systems*, (), Journal of Social and Evolutionary Systems, 21(2), *1998*.

[2] Evans P. Wolf B. *Collaboration Rules*, Harvard Business Review, *June–August, 2005*. Данный раздел представляет собой краткое изложение двух историй, рассказанных в указанной статье.

Harvard Business Review. Эта статья завершается так: "Никто не санкционировал и не руководил этой работой. Никому (ни любителям, ни профессионалам) ничего не платили за участие и никому ничего не угрожало, если бы кто-то уклонился. Ничья основная работа никак не зависела от того, будет ли остановлена атака. Никем не руководил страх или обязательства перед законом. Тем не менее группа приблизительно из 20 человек, которые едва ли когда-либо встречались, работающие в различных компаниях, живущие в разных часовых поясах и абсолютно не побуждаемые своими прямыми обязанностями, совершили за 29 часов (или около того) то, на что у коллег из соседних комнат ушли бы недели или месяцы.

Суббота, 1 февраля, 1997 года.[3] В этот день ранним утром пожар уничтожил главное предприятие компании Aisin — завод, производящий клапаны для тормозной системы (P-Valves), которые используются в автомобилях Toyota. Клапан для тормозной системы — это узел размером с пачку сигарет, который управляет давлением в системе задних тормозов, и у корпорации Toyota было запасов всего, может быть, на два-три дня. Клапан — это высокоточный узел, изготавливаемый с помощью специального оборудования, однако теоретически его производство можно было бы организовать и с использованием стандартного оборудования. Из Aisin немедленно был отправлен призыв о помощи ко всем компаниям, где могло бы быть подходящее оборудование. Шестьдесят две компании откликнулись, и уже через несколько часов были получены копии чертежей клапана и созданы импровизированные технологические линии. Добровольцы сами осуществляли координацию между собой, когда они определяли, как можно производить высокоточные клапаны. (В этом приняла участие даже корпорация Denso — известный поставщик деталей и узлов для автомобилей.) И хотя Toyota пришлось остановить все свои заводы во вторник, небольшая компания-поставщик Kyoritsu Sangyo с гордостью предоставила в этот день первую тысячу только что произведенных клапанов. К четвергу уже 36 поставщиков, которым помогали 150 субподрядчиков, производили небольшие партии P-Valves, и Toyota вновь начала запускать свои заводы. Неделю спустя после пожара Toyota удалось полностью восстановить производство автомобилей (более 16000 ежедневно), каждый из которых содержал два произведенных наполовину вручную клапана для тормозной жидкости. Для восстановления производственной линии после пожара потребовалось шесть недель, и все временные поставщики были должным образом вознаграждены за потраченное ими время.

"Сходство между этими историями поразительно,[4] — утверждают Эванс и Вольф. — В обоих случаях индивидуумы сами нашли друг друга и начали выполнять определенные функции, без какого-либо предварительного плана или созданной заранее контрольно-управленческой структуры. Широкая сеть специалистов возникла на принципах самоорганизации в течение нескольких часов и начала противодействовать возникшей угрозе. Люди, коллективы и компании работали вместе без каких-либо узаконенных контрактов или договоренностей об оплате. И несмотря на отсутствие каких-либо наказаний или поощрений, эти люди работали с полной самоотдачей для решения проблемы."

Люди и компании, откликнувшиеся на эти две чрезвычайные ситуации, были привычны к совместной работе. У них уже имелось понимание, как следует взаимодействовать друг с другом, чтобы сделать то, что требуется. Они научились делать это без види-

---

[3] Там же. См. также Nishiguchi T., Beaudet A. *The Toyota Group and the Aisin Fire*, Sloan Management Review, *1998*.

[4] Там же, p. 99.

мого руководства, выполняя то, что от них требовалось, если они обладали нужными квалификацией и опытом. Ведомые доверием и оплачиваемые аплодисментами, каждое сообщество руководствовалось не контрактами или иерархиями, а правилами совместной работы и взаимодействия.

Open Source

Известно, что превосходное программное обеспечение может разрабатываться и поддерживаться добровольцами, работающими по всему миру, пользующимися несколькими простыми средствами коммуникаций и руководствующимися некоторыми элементарными правилами. В действительности многие люди, создающие программное обеспечение Open Source, помимо этого где-то работают разработчиками ПО на основной работе. Некоторые даже могут получать оплату за работу, которую они позже предоставили как вклад в Open Source. Достаточно часто руководители проектов (committers)[5] в Open Source являются разработчиками, которые модифицируют существующую программу для решения некоторой своей проблемы, а затем делятся результатом с сообществом. Однако все это не объясняет успеха, лежащего в основе модели Open Source.

Большинство проектов Open Source имеют руководителя (или инициатора проекта), по крайней мере на начальном этапе. И все проекты Open Source имеют "вкладчиков" (contributions), или ассистентов руководителя проекта, — внутренний круг пользующихся доверием людей, которые обладают правом добавлять новый программный код в библиотеку кода соответствующего проекта. Первоначально руководитель проекта одновременно является его единственным "вкладчиком", однако по мере развития проекта новые разработчики могут заслужить право присоединиться к этому кругу. Ассистенты руководителя вводят в действие все стандарты, которые могут быть приняты, и играют роль первых критиков представленного программного кода. Это честь для новых членов, чтобы их код был принят в библиотеку кода проекта; в то же время предложение программного кода, который не был всесторонне протестирован, может поставить разработчика в неловкое положение. Активное сообщество в любом проекте подвергнет программный код безжалостной и всесторонней проверке, как только его увидит. Это можно сравнить с азартной командной спортивной игрой, приносящей огромное внутреннее удовлетворение тому, кому удается сделать выдающийся вклад в проект.

Нередко люди имеют навязчивую идею, которая полностью поглощает их внимание за пределами основной работы. С Open Source такой навязчивой идеей является страсть создавать программное обеспечение. В создании ПО есть что-то крайне притягательное. Уместно задаться вопросом, что следует сделать, чтобы создание программного обеспечения было таким же притягательным занятием в вашей организации.

Возвращаясь к 14 пунктам Деминга (см. главу 6), следует признать, что пункт 12, вероятно, является наиболее важным: "Ликвидируйте помехи, которые не позволяют людям испытывать гордость за свое мастерство." Произвольно установленные сроки, менеджеры, не имеющие представления, что такое хороший программный код, "сырые" библиотеки кода и практики его создания, невозможность увидеть, работает ли новый код, никаких сведений, что нужно заказчикам, индивидуальные вознаграждения за коллективный вклад — все это не позволяет любому члену коллектива разработчиков испы-

[5] Разработчики-добровольцы, энтузиасты своего дела, участвующие (бесплатно) в проектах Open Source.

тывать гордость за свое мастерство. Гордость создает приверженность делу, а без общей приверженности группа работников коллективом в действительности не является.

Почему бы ответственному за разработку данного программного обеспечения не создать описание работы, такое же, как у руководителя проекта в Open Source? Чтобы привлечь добровольцев к работе в Open Source, инициатор нового проекта должен предложить привлекательную идею и хороший подход. Для поддержания приверженности проекту его руководитель (и позже сообщество его ассистентов) должен распознавать, проявлять благодарность и "аплодировать" хорошей работе. Хорошая работа в данном случае означает решение проблемы (в большинстве случаев разработчики здесь одновременно являются и заказчиками, которые по-настоящему понимают проблему) с помощью лаконичного, понятного программного кода, добавляемого в систему небольшими "порциями". Разработчики в Open Source не станут тратить время на беспорядочное или нефункциональное программное обеспечение, или, если станут, то начнут с приведения его в порядок. И никто в действительности не указывает разработчикам в Open Source, что им следует делать. Они вполне способны определить это сами.

Для дисциплинированного, компетентного коллектива разработчиков не существует валюты более ценной, чем уважение, доверие и признание.

Глобальные сети

Boeing 787 Dreamliner создавался с помощью всемирной сети, поскольку в компании Boeing полагали, что сотрудничество многих людей, принадлежащих разным культурам и обладающих опытом в различных областях, приведет к появлению совершенного авиалайнера. А в компании Procter & Gamble пришли к выводу, что потребность в новых идеях возникает быстрее, чем эти идеи успевают созреть внутри компании. Поэтому инновационные идеи Procter & Gamble черпает из глобальной сети, а затем использует в новых продуктах.[6] Однако это не единственная причина для существования глобальных сетей. Обе компании поняли, что им необходимо продавать свои изделия по всему миру, поэтому глобальные сети необходимы для того, чтобы помочь им понимать и удовлетворять потребности мировых рынков.

В разработке программного обеспечения имеются свои пользующиеся широким признанием глобальные сети: сообщества Open Source. Из Open Source мы знаем, что людям необязательно находиться в одном часовом поясе, чтобы тесно сотрудничать над созданием программного обеспечения. Вопрос в том, что необходимо, чтобы люди смогли тесно сотрудничать над созданием ПО, особенно если они находятся не в одном месте?

Давайте вернемся к двум историям, о которых шла речь в начале главы. Первое, что предприняла компания Aisin после пожара, это обратилась с призывом о помощи, на который откликнулись 62 компании. Все, кто мог чем-нибудь помочь, делал все, чтобы компания Toyota смогла восстановить свое производство; это стало делом чести и взаимной экономической необходимости. Во время кризиса, последовавшего за пожаром, компания Aisin могла предоставить копии чертежей всем, но не могла помочь поставщикам определить, как наладить ручное производство клапанов для тормозной системы. Компания Aisin использовала для этого автоматизированное оборудование и практически не имела опыта применения инструментов и оборудования для производства изделия

[6] Huston L., Sakkab N. *Connect and Develop: Inside Procter & Gamble's New Model for Innovation*, Harvard Business Review, *March 2006*, p. 58–66.

вручную. Поставщикам пришлось до всего додумываться самим. Организовывались ежедневные совещания, на которых сотрудники обменивались идеями и опытом и таким образом в конце концов сумели решить проблему.

Что касается ликвидации бреши в безопасности Linux-сервера, то решение этой задачи началось с решимости (или приверженности) бывшего руководителя проекта в Австралии устранить ошибку в программном коде, который он в прошлом сам создал. Благополучие многих людей зависело от этого, поэтому он и пришел им на помощь (для него это стало вопросом чести). Он потратил часы, обмениваясь информацией с системным администратором из Италии, прежде чем им удалось обнаружить точное место в коде, где имелась ошибка. Затем он объединил усилия с коллегой, жившим неподалеку, чтобы создать "заплату", поскольку решение проблемы в жестких условиях (особенно если это происходит всю ночь) дается легче, когда есть возможность обмениваться с кем-то идеями.

Нас (авторов) часто спрашивают, как улучшить коммуникационные каналы для рассредоточенных коллективов, однако слишком часто мы обнаруживали, что корень проблемы не в коммуникациях, а в отсутствии коллектива, приверженного выполнению задачи. Глобальные сети и распределенные рабочие группы не становятся коллективами только потому, что их так называют. Организация каналов связи очень важна, когда члены коллектива расположены в различных часовых поясах, но этого вряд ли достаточно для превращения рабочей группы в коллектив.

Задача добиться сотрудничества между людьми, которые географически рассредоточены, аналогична задаче организации сотрудничества людей, находящихся рядом. Прежде всего все члены коллектива должны быть привержены достижению общей цели. В случае рассредоточенного коллектива задача добиться общей приверженности просто несколько усложняется: эффективный обмен между членами коллектива становится более трудным.

От рассредоточенных рабочих групп к рассредоточенным коллективам

Ниже приведены несколько советов, которые, как нам кажется, могут быть полезны чтобы достичь приверженности общей цели и облегчить обмены (информацией и людьми) внутри рабочих групп, которые рассредоточены географически.

1. **Частая интеграция.** Очень часто рабочая группа состоит из нескольких небольших коллективов, расположенных далеко один от другого. Если сферы ответственности этих коллективов соответствуют функциональной архитектуре системы, тогда, как правило, коллективы могут работать над отдельными модулями сравнительно независимо. Это может быть очень эффективным подходом, однако коллективы *обязаны* использовать вложенную (nested) синхронизацию для частой интеграции сделанной ими работы. Регулярные и частые интеграции обеспечивают множество преимуществ — от создания общей приверженности до организации общего хранилища знаний. Чем быстрее знания передаются другим коллективам через библиотеку кода, тем более понятным будет программный код, тем лучше блочные тесты будут соответствовать намерениям каждого локального коллектива и тем лучшими будут общие результаты.
2. **Обмен специалистами.** Очень часто обнаруживается, что коллектив в одной стране обладает всем необходимым техническим опытом для создания системы, однако "запросы" на разработку поступают большими партиями в виде документации, созданной на расстоянии нескольких часовых поясов. Если документация была написана несколько недель или месяцев назад, она уже не отражает того, что заказчикам нужно в данный момент. Большие расстояния между заказчиками или аналитиками и коллективом разработчиков (в сочетании с не очень хорошей связью) редко приводят к хорошим результатам. Лучший выход в подобной ситуации состоит в том, чтобы командировать пару сотрудников одного коллек-

тива в другой коллектив на продолжительный период времени, предпочтительно на основе ротации. Либо пара сотрудников коллектива заказчиков пусть некоторое время поработает с разработчиками, либо, наоборот, два члена коллектива разработчиков пусть побудут у заказчиков. Чередование этих подходов было бы еще более эффективно.

3. **Обмен тестами.** Нам известен случай, когда ''запросы'' от заказчика разработчикам поступали в виде выполняемых тестов. Люди, хорошо понимающие потребности заказчика, разрабатывали подробные тесты, которые должны успешно выполняться с созданным программным кодом. Если тесты успешно выполнились, в этом случае код считается рабочим. Тем не менее обнаружилось, что даже в этих условиях из-за большого расстояния разработчики часто создают не совсем то, что нужно заказчику. Выходом здесь может быть тестирование хотя бы раз в неделю созданного программного кода непосредственно у заказчика.
4. **Посредник.** Нам приходилось видеть очень успешные рассредоточенные коллективы, где за обмен информацией между сотрудниками нес ответственность один сотрудник. В этом случае один из членов базовой группы такого рассредоточенного коллектива берет на себя обязанности посредника между остальными членами этого коллектива. Ежедневно этот сотрудник несет ответственность за большой объем тщательно определенной работы. Каждый день этот сотрудник звонит ''удаленным'' членам коллектива, объясняя им, что требуется сделать, отвечает на вопросы и принимает сделанную работу. Таким образом, ''удаленные'' работники поддерживают тесную связь с представителем базовой (или организующей) группы рассредоточенного коллектива; при этом удаленные сотрудники воспринимаются (руководством) как некое ''продолжение'' посредника, как бы выполняющего работу нескольких человек.
5. **''Путешествующий'' лидер коллектива.** Нам известен случай, когда коллектив был разделен на несколько групп, расположенных далеко одна от другой. В каждом месте имелась комната с графиками и диаграммами на стенах, отражающими состояние выполнения проекта. Эти графики и диаграммы регулярно синхронизировались. Лидер проекта переезжал из одного места в другое, проводя регулярные совещания и решая накопившиеся вопросы. На такие совещания всегда приглашались представители коллектива из других мест, что поддерживало постоянную общую приверженность всех групп совместной задаче.
6. **Нет граждан второго сорта.** Когда часть коллектива вынуждена работать с использованием своего второго (не родного) языка, в то время когда остальные члены коллектива используют первый язык; или когда одна группа является субподрядчиком, а другая является частью компании-подрядчика; или когда какая-то группа явно выше оплачивается или имеет более высокий статус, очень легко может возникнуть представление, что одна группа ''лучше'' другой. Подобные представления быстро разрушают уважение, доверие и приверженность, без которых не обойтись в коллективной работе.

Тесное сотрудничество

Нишиджима-сан (Nishijima-san) решил, что наше светопроводящее волокно прекрасно подойдет для производства в Sumitomo-3M. Мы путешествовали вместе по Соединенным Штатам в течение трех недель в поисках подходящих идей, и Нишиджима-сан вскоре должен был вернуться в Японию. Постоянно последние два с половиной года не менее двух инженеров из его коллектива работали в моем коллективе в Соединенных Штатах. Эти инженеры часто менялись с тем, чтобы каждый японский сотрудник провел несколько месяцев в коллективе в США, помогая нам в нашей работе и одновременно перенимая наш опыт в использовании оптоволоконных технологий. Каждый вечер эти японские инженеры звонили в Японию (где как раз в это время проводилось общее собрание коллектива) и информировали своих коллег о том, что происходит у нас. Затем на нашем собрании они информировали нас о ходе работы в Японии.

Мы пытались воспользоваться видеоконференциями, однако это нам не очень подошло. Впрочем, видеоконференции нам не очень были нужны, поскольку у нас в Соединенных Штатах постоянно были представители из Японии, которых все воспринимали как полноправных членов коллектива, и они регулярно участвовали в наших совещаниях. А когда мы ездили в Японию, то знали всех

членов японского коллектива и чувствовали себя его частью. Мы совместно выбрали имя бренда, которое бы хорошо звучало на любом языке. Если бы мне пришлось делать то же снова, я бы сделала это так же, ну, может быть еще бы отправила двух своих инженеров в Японию в обмен на инженеров, которых Нишиджима-сан присылал нам.

Вероятно, наибольшей проблемой являлась совместная интеллектуальная собственность, несмотря на то, что 3M и Sumitomo-3M были тесно связаны и имели соглашения об интеллектуальной собственности. Однако в ситуации, когда у нас постоянно находились инженеры из Sumitomo-3M, нам трудно было определить, какая идея кому принадлежит. Поэтому вице-президент, которой выступал спонсором программы, предложил нам игнорировать эту проблему и предложил юристам разобраться в ней. Я убеждена, что это решение сыграло решающую роль в том, чтобы оба коллектива чувствовали себя равными. Мы имеем несколько совместных изобретений. Более того, на одном из совместных патентов стоит мое имя. (Мэри Поппендик)

Аутсорсинг

Распределенные коллективы могут существовать в пределах одной компании или их сотрудники могут работать в разных компаниях. В действительности то же верно и для локальных коллективов: члены такого коллектива могут принадлежать одной компании или нескольким компаниям. Аутсорсинг означает передачу одной компанией определенных бизнес-процессов или производственных функций другой компании, специализирующейся в соответствующей области. При этом вторая компания может находиться как на противоположной стороне улицы, так и на другом конце света. Аутсорсинг оказывает куда большее влияние на коллектив, чем то обстоятельство, что он рассредоточен. Там, где имеет место аутсорсинг, неизбежно возникает вопрос лояльности. В какой мере участники должны учитывать интересы собственной компании, компании-партнера и совместного предприятия? И как все это совместить с законным стремлением сохранить собственное рабочее место?

Самые удачные соглашения по аутсорсингу дают очень ясный ответ на эти вопросы: интересы отдельных компаний и их наемных работников лучше всего обеспечиваются, когда на первое место ставятся интересы совместного предприятия. Без подобной ясности работники будут подвержены скрытому конфликту интересов. Как избежать подобного конфликта? Ответ на данный вопрос зависит от того, к какого рода работе был применен аутсорсинг. Давайте рассмотрим три варианта его применения: к инфраструктуре, транзакциям и разработке.

Инфраструктура

Когда мы начинали наш бизнес в 1999 году, мы знали, что нам понадобится веб-сайт и возможность пользоваться электронной почтой, но нам вовсе ни к чему было заводить собственный сервер. Поэтому мы нашли копанию (к слову сказать, из Канады), которая за умеренную плату обеспечила хостинг нашего веб-сайта и предоставила сервис e-mail. С течением времени эта компания постоянно улучшала свой сервис и снижала цены до такого низкого уровня, что мы подчас удивлялись, зачем в принципе малому бизнесу иметь собственный сервер для своего веб-сайта и электронной почты. Нас не волновали проблемы, связанные с недоступностью сервера из-за неисправностей, безопасностью и даже развитием. Сервис был очень прост в использовании и в нескольких случаях, когда мы нуждались в технической помощи, ответ был более чем удовлетворительным.

В описанном примере мы применили аутсорсинг к инфраструктуре — которая, вероятно, проще всего поддается аутсорсингу. Предполагаемые свойства инфраструктуры хорошо определены на рынке, предоставляются многими компаниями и имеют конкурентоспособные цены. Хоты мы были достаточно компетентны технически, чтобы самим создать для себя инфраструктуру, все же более целесообразно было бы потратить наше время иначе. Поэтому в нашем случае прямой смысл имело применить к инфраструктуре аутсорсинг, учитывая, что специализирующаяся на этом организация может предоставить ее (инфраструктуру) более компетентно и дешево, чем если бы этим занялись мы сами.

Транзакции

Помимо инфраструктуры, аутсорсинг может также быть применен к транзакциям. Например, компания, которой необходимо перевести большие объемы бумажных документов в электронный формат, может осуществить их сканирование у себя. Однако, поскольку такое преобразование требует значительных трудозатрат, передача этой работы в страну, где рабочая сила ценится дешево, может быть очень выгодной.

Однако здесь таится опасность. Что если настоящее конкурентное преимущество в отрасти перейдет к компании, которая научилась самостоятельно переводить данные в электронный формат и обходиться без бумажных документов? Компании, которые применят аутсорсинг к переводу документов в электронный формат с целью снизить расходы, потеряют стимул задаваться вопросом, для чего такое преобразование вообще нужно. Прежде чем применять аутсорсинг к транзакциям, следует упростить поток создания ценности и попытаться ликвидировать необходимость в этих транзакциях.

Чаще всего конфликт интересов при применении аутсорсинга к транзакциям связан со сдельной оплатой работы. Когда компания, выполняющая некоторые работы по аутсорсингу, получает оплату за каждую транзакцию, естественным с ее стороны является стремление сократить время, затрачиваемое на транзакцию. Однако если истинная выгода заключается в том, чтобы сократить число транзакций, сдельная оплата создает конфликт между лояльностью работника своему работодателю и его лояльностью по отношению к компании, генерирующей транзакции.

Например, в главе 2 была рассказана история о том, как британская авиакомпания BMI применила аутсорсинг к службе технической поддержки в пользу компании Fujitsu. Как правило, компании, берущие на себя выполнение функций службы технической поддержки, получает оплату от числа поступивших обращений (или транзакций), однако Fujitsu смогла продемонстрировать BMI, что имеется возможность сократить число обращений и одновременно уменьшить издержки BMI. При этом Fujitsu удалось убедить BMI изменить стимул так, чтобы оплата основывалась на *потенциальных* вызовах. Иными словами, Fujitsu получила возможность зарабатывать деньги, одновременно помогая BMI экономить деньги — это то, что называется взаимовыгодным сотрудничеством. Описанный подход к оплате ясно показывает, что забота об интересах совместного предприятия в интересах всех партнеров.

Разработка

Подобно тому, как в высшей степени целесообразно применение аутсорсинга к инфраструктуре, применение аутсорсинга к разработке программного обеспечения для автоматизации некоторых типичных для отрасли процессов (обработки покупок, например) также может быть чрезвычайно выгодно. Однако логично предположить, что с по-

мощью подобного аутсорсинга вы, скорее всего, получите типичное для отрасли решение, поскольку ваш провайдер аутсорсинга вполне вероятно работает и с другими компаниями отрасли. Таким образом, если состояние дел в вашей компании ниже нормы, тогда аутсорсинг — это быстрый способ улучшить их до среднего (для отрасли) уровня. Подобное применение аутсорсинга к разработке принципиально мало чем отличается от применения к инфраструктуре (и при этом редко возникают проблемы с лояльностью).

Некоторые компании применяют аутсорсинг к части проекта, — например, ручное тестирование с использованием интерфейса пользователя часто передается другим компаниям, поскольку оно крайне трудоемко. Однако подлинное преимущество получит та компания, в которой поймут, что тестирование нецелесообразно осуществлять вручную через интерфейс пользователя. Такая компания постарается автоматизировать тестирование "под" интерфейсом пользователя. В действительности в этом случае целесообразно начать работу с разработки тестов, а затем создавать программный код, с которым бы выполнялись данные тесты. Такая компания имела бы ощутимое конкурентное преимущество по сравнению с компанией, применяющей аутсорсинг к ручному тестированию.

Целесообразность применения аутсорсинга к разработке ключевых компонентов вашего продукта (или изделия) или программного обеспечения необходимо тщательно взвесить. Некоторые компании предпочитают самостоятельно осуществлять подобного рода разработку, рассчитывая таким образом создать для себя конкурентное преимущество. Другие компании научились эффективно применять аутсорсинг к подобным критически важным областям разработки. Например, в Соединенных Штатах более 70% деталей и узлов в автомобилях Toyota разрабатываются и производятся поставщиками.[7] А глобальные сети корпораций Boeing и Procter & Gamble, упоминавшиеся ранее в этой главе, предоставляют доступ этим корпорациям к наиболее передовой технической информации в мире, независимо от того, где она возникла. Таким образом, передовые разработки, ранее в основном осуществлявшиеся внутри компаний, сегодня проводятся в сотрудничестве с другими компаниями.

Компании, получающие значительное преимущество, применяя аутсорсинг к критически важным областям разработки, руководствуются несколькими основными принципами.

1. Причина применения аутсорсинга к критическим областям разработки объясняется необходимостью доступа к бо́льшим техническим мощностям, чем те, которыми располагает компания, либо чтобы расширить свое понимание рынка. Возможно одновременное существование обеих причин.
2. Применение аутсорсинга к важным областям всегда должно вести к взаимовыгодному сотрудничеству. При этом не предполагается, что такое партнерство автоматически приведет к синхронности действий. Синхронность достигается дополнительными мерами. Например, система оплаты, правила взаимодействия и ожидания менеджмента способны дать ясные сигналы работникам, что лояльность по отношению к совместному предприятию эквивалентна лояльности их компаниям и что успешное совместное предприятие лучше всего будет отвечать их интересам.
3. Компании проявляют глубокое уважение к своим партнерам, и это уважение является неотъемлемой частью процессов и процедур, используемых при реализации деталей такого партнерства.

---

[7] Dyer J.H. *Collaborative Advantage: Winning Through Extended Enterprise Supplier Networks*, Oxford University Press, *2000*, p. 33.

Организация совместной работы с компаниями-партнерами через аутсорсинг (supplier management) может создать такие же существенные преимущества в разработке, как управление цепью поставок (supply chain management) в производстве. Все начинается с понимания, что цель аутсорсинга не в сокращении издержек и не в уменьшении риска и даже не в повышении производительности. Основная цель — в создании синхронности. Главное понять, что "гребя" вместе, компании способны "придти к финишу" куда быстрее, чем даже самая сильная из них в одиночку.

Контракты

Для чего нужны контракты? На этот счет существуют два очень отличающихся мнения.

- Предназначение контрактов в том, чтобы защитить каждую сторону от беспринципного поведения другой стороны.
- Предназначение контрактов в том, чтобы создать подходящие стимулы совместной (и синхронной) работы компаний.

В зависимости от того, какой вариант вы выберете, контракты могут сделать нереальным бережливый подход к разработке программного обеспечения или заложить основы совместной работы всех сторон с целью достижения наилучших результатов.

Соглашение T5

Компания BAA, которая управляет большинством крупных аэропортов в Соединенном Королевстве, начала в 2002 году строить терминал 5 в аэропорту Хитроу, который планировалось открыть 30 марта 2008 года. Проект был грандиозен; его стоимость оценивалась в 4,2 миллиарда. С самого начала компания BAA осознала, что просто не может себе позволить превышения стоимости или откладывания сроков, которые так типичны для проектов подобного масштаба. Любая существенная задержка по чьей бы то ни было вине будет означать "конец" для BAA.

Поэтому в BAA разобрались, какие недостатки имели место при строительстве других аэропортов, известных большими задержками и перерасходами, а также со всеми иными общественными проектами, отмеченными неудачами. Был сделан вывод, что существенная проблема состояла в том, что контракты пытались переложить риск на подрядчиков. За дополнительный риск подрядчики назначали бо́льшую цену, однако в данном случае в конце концов, при возникновении проблем, нести ответственность пришлось бы BAA, а не подрядчикам. Поэтому в BAA решили взять на себя весь риск по осуществлению данного проекта, поскольку при сложившихся обстоятельствах невозможно было полагаться на контракты и суды.

В BAA разработали то, что стало известно как соглашение T5 — юридический документ, заменивший привычные контракты и изменивший правила игры. BAA согласилась взять на себя весь риск, а взамен подрядчики согласились работать объединенными коллективами, при возможности снижать риск и делать все, чтобы достичь наилучших результатов. Проект был разделен на 150 "подпроектов", каждый из которых имел группу подрядчиков и согласованную смету, а также премиально-рисковый фонд (pool of risk/incentive money). Подрядчики получали оплату за потраченные ими время и материалы. Если что-то потребуется переделать, без каких-либо возражений это также будет

оплачено. Если стоимость подпроекта превзойдет запланированные расходы, деньги для покрытия превышения будут взяты из премиально-рискового фонда для этого подпроекта. По завершении подроекта коллектив, работавший над ним, разделит между собой две трети денег, оставшихся в премиально-рисковом фонде.

Соглашение T5 учитывало как риск, так и стимулы совершенно необычным способом, который, как полагали в BAA, сэкономит значительную сумму денег. Соглашение снижало вероятность конфликтов и стоимость сотрудничества, предоставляя при этом независимым подрядчикам полную свободу сотрудничать друг с другом (при соответствующих стимулах). Благодаря наблюдению BAA, давлению со стороны коллег и профессиональной гордости, коллективы все сделали вовремя и это при том, что несколько недель практически пропали из-за плохой погоды. В настоящее время (т.е., когда пишутся эти строки) есть все основания полагать, что терминал 5 будет готов к дате, на которую назначено его открытие — 30 марта 2008 года[8].

Контракт PS 2000

Этот контракт был разработан Норвежским компьютерным обществом (Norwegian Computer Society) в сотрудничестве с Норвежским университетом науки и технологий (Norwegian University of Science and Technology — NTNU), а также ведущими промышленными и общественными организациями Норвегии. Созданный специально для использования в крупных общественных проектах в области информационных технологий, контракт основывался на наблюдении, что часто трудно составить подробную спецификацию на информационную систему во время процесса закупок и распределения подрядов. В основе контракта лежали следующие предположения.

1. Лучшая практика демонстрирует, что гибкая итеративная модель разработки лучше всего сочетается с неопределенностями и рисками многих крупномасштабных проектов в области информационных технологий.
2. Это требует некоторых механизмов для взаимопонимания между подрядчиком и заказчиком.
3. Сторона, занятая разработкой системы, является наиболее компетентной в определении лучших методов достижения целей и удовлетворения запросов заказчика.[9]

Уникальным в контракте PS 2000 является то, что он регламентирует сотрудничество между заказчиком и подрядчиком; это не спецификация о том, что должно быть поставлено. Содержание контракта предусматривает создание советов (или коллегий), призванных принимать решения, выбор посредников в случае несогласия и установление общих прав и обязательств для всех сторон. Цели проекта и плановые издержки должны определяться в приложениях. Контракт определял метод итеративной разработки системы; по мере согласования ее новых деталей определенные ранее цели проекта будут постепенно уточняться. В приложении, где указываются планируемые из-

[8] В настоящее время (когда *переводятся* эти строки) можно сообщить читателям, что терминал 5 действительно был открыт в заданные сроки. Соответствующие материалы можно найти в Интернете. — *Примеч. ред.*

[9] С веб-сайта `http://dataforeningen.no/?module=Articles;action=ArticleFolder.publicOpenFolder;ID=1044`.

держки, определяется также метод, с помощью которого обе стороны будут нести ответственность за экономию и издержки, а также устанавливаются верхний и нижний пределы общих расходов.

Реляционные контракты

Оба контракта, о которых шла речь выше (соглашение T5 и PS 2000), являются реляционными контрактами (relational contracts)[10], т.е. такими, которые больше определяют взаимоотношения партнеров, чем ожидаемые результаты партнерства. Реляционные соглашения обычно согласуют стимулы и интересы совместного предприятия. Подобные соглашения обычно исходят из того, что люди, естественно, в первую очередь заботятся об интересах своих компаний. Поэтому специально создается реляционное соглашение, призванное разъяснить всем выполняющим работы, что интересы отдельных участвующих компаний лучше всего будут обеспечены, если сосредоточиться на интересах совместного предприятия.

Совместное предприятие

Наши две компании располагали двумя очень оригинальными, дополняющими одна другую технологиями, и мы пришли к выводу, что, объединив их, сможем создать по-настоящему инновационный продукт. Однако по отдельности эти технологии были основой техпроцесса в каждой компании. Невозможно было идти на риск, связанный с совместной работой, до тех пор, пока не будет подписано соглашение о создании совместного предприятия, которое бы защитило интеллектуальную собственность каждой компании.

Крайне сложно было предусмотреть все детали. Мой коллега из другой компании и я знали точно, как именно мы предполагаем сотрудничать. Но необходимо было как-то определить, как совместное предприятие будет организовано, как оно будет управляться, как уже имеющаяся интеллектуальная собственность каждой компании будет защищена, что произойдет, если одна сторона решит выйти из совместного предприятия и т.д. Очень мало времени и сил мы тратили на то, чтобы думать, как будут проводиться совместные исследования (именно для этого было учреждено совместное правление) или как будут поделены издержки (это решить будет просто). Некоторое время мы также уделили подробностям соглашения о разделе прибылей.

По-настоящему трудная задача заключалась в том, чтобы создать соглашение, которое позволило бы ученым и инженерам обеих компаний сидеть в одной комнате и совместно решать, как использовать наши две технологии для создания выдающегося продукта. В этом был вызов, поскольку речь шла о наибольшей ценности каждой компании — ее интеллектуальной собственности.

Разработка документа заняла пару месяцев, и этот документ определял все подробности совместной работы. После того как документ был подписан, мы, наконец, смогли приступить к работе. (Мэри Поппендик)

Питер Друкер (Peter Drucker) полагает, что партнерское сотрудничество способно сэкономить совместному предприятию от 25 до 30% издержек.[11] Определенно, мы видим то же в истории с BAA: снижение вероятности конфликтов и риска для множества подрядчиков привело к существенному сокращению как риска, так и издержек для самой BAA. В этой книге много раз было сказано, что бережливый подход к созданию программного обеспечения способен привести к значительным улучшениям качества ПО, скорости по-

[10] Больше информации о реляционных контрактах можно найти на веб-сайте Lean Construction Institute по адресу `www.leanconstruction.org`.

[11] Drucker P. *Management Challenges for the 21st Century*, Harper Business, *2001*, p. 33.

ставки и оперативности обслуживания системы в будущем. И мы видим, что эти преимущества также проистекают от синхронной работы людей, которые объединяют свои знания и опыт, чтобы совместно создавать знание более эффективно, чем они смогли бы это делать порознь.

Большинство контрактов по созданию программного обеспечения исходит из различных предположений. Они включают спецификацию продукта и предполагают, что эта спецификация в значительной мере описывает ожидаемый результат. Если данное предположение окажется ошибочным, большинство контрактов не содержит на этот случай эффективных "лекарств". Реляционный контракт начинается с предположения, что продукт не может быть эффективно определен в контракте, поэтому основное внимание уделяется совместной работе сторон, чтобы определить, что они собираются создавать.

Реляционные контракты, как правило, определяют плановые издержки и запланированные сроки. Обе стороны соглашаются работать вместе, используя предусмотренные контрактом механизмы, для создания наибольшей потребительской ценности в рамкой заданных издержек и сроков. Контракт определяет стимулы для обеих сторон, которые бы побуждали их к сотрудничеству на благо совместного предприятия и при этом снижает заинтересованность сторон преследовать собственные интересы.

Использование реляционных контрактов для разработки программного обеспечения — новое и быстро развивающееся явление. Мы (авторы) полагаем, что тот, кто научится организовывать синхронное взаимодействие между компаниями, действительно сможет добиться сокращения издержек на 25–30%, как и предсказывал Питер Друкер; впрочем, вполне возможно, результат будет куда лучше.

Попробуйте это

1. На следующем собрании вашего коллектива спросите, случалось ли кому-нибудь вносить вклад в один из проектов Open Source. Если да, задайте каждому такому сотруднику следующие вопросы:
   - **а)** почему вы тратили на это свое время;
   - **б)** как долго вы этим занимались;
   - **в)** сколько времени это у вас отнимало?

   Если сочтете уместныи, задайте также следующие вопросы:
   - **г)** что побуждало вас заниматься этим так долго;
   - **д)** почему вы прекратили это занятие;
   - **е)** что из вашей работы в Open Source понравилось вам больше всего;
   - **ж)** что вызывало наибольшее недовольство?
2. Найдите рассредоточенный коллектив в вашей среде и попытайтесь при помощи членов этого коллектива найти ответы на следующие вопросы:
   - **а)** это единый коллектив или просто объединение коллективов из разных мест;
   - **б)** вносят ли члены коллектива свой вклад в достижение общей цели; что это за вклад; согласятся ли все члены коллектива, что это именно та цель, достижению которой каждый из них лично привержен;

в) как сотрудники, работающие в другом офисе (возможно, на другом континенте), будут знать, что у них значительные достижения? Откуда последуют аплодисменты?

Сравните ответы, поступившие от членов коллектива из разных мест.

3. Какова главная цель применения аутсорсинга в вашей компании? Какую деятельность вашей компании предпочтительнее осуществлять внутри нее, чем применять аутсорсинг? Совпадают ли стимулы, предусмотренные соглашениями по применению аутсорсинга, с интересами вашей компании и ваших заказчиков? Можете ли вы представить ситуацию, когда ваши соглашения по аутсорсингу создали бы конфликт интересов в вашей компании или компании-партнере?
4. Для компаний, заключивших контракт: какого рода контракты вы обычно используете? В чем основное преимущество таких контрактов? В чем главный риск? Если бы вы были на месте ваших партнеров, какие преимущества и риски подобных контрактов вы смогли бы указать?
5. Для партнеров по контракту: в какого рода контрактах вы, как правило, принимаете участие? Определите наиболее подходящий вам формат контракта и рассмотрите его с точки зрения сторон. В чем заключаются для них преимущества и риски?

Глава 10

Путешествие

Куда бы вам хотелось отправиться ?

“Что мне делать дальше? — часто приходится нам слышать. — Моя компания находится в очень трудном положении, которое просто необходимо улучшить. Принципы бережливости кажутся очень разумными, но зачем мне принципы? Что мне делать с ними? Как их реализовать — немедленно — в моей компании?” Это напоминает нам Алису в Стране Чудес[1]:

— Скажите, пожалуйста, куда мне отсюда идти?
— А куда ты хочешь попасть? — ответил Кот.
— Мне все равно... — сказала Алиса.
— Тогда все равно, куда и идти, — заметил Кот.
— ...только бы попасть куда-нибудь, — пояснила Алиса.
— Куда-нибудь ты обязательно попадешь, — сказал Кот.
— Нужно только достаточно идти.[2]

В главе 7 мы говорили, что внедрение бережливого подхода должно начинаться с ответа на два вопроса.

1. Как именно вы намереваетесь получать прибыль и удовлетворять ваших заказчиков?
2. В чем ваша главная проблема?

К этим следует добавить еще два вопроса.

1. Что может помешать продолжительному существованию вашей компании в будущем?
2. Что *в действительности* вы думаете о людях?

---

[1] *Приключения Алисы в Стране Чудес*, написанные преподобным Чальзом Лютвидж Доджсоном (Reverend Charles Lutwidge Dodgson) под псевдонимом Льюис Кэролл (Lewis Carroll), впервые опубликованные 1865 году.
[2] Перевод Нины Демуровой.

Компьютер на колесах

В конце 70-х и в начале 80-х годов прошлого (XX) века дела компании Toyota шли очень хорошо, однако в 1985 году курс иены начал стремительно падать относительно других валют, побудив японских производителей автомобилей переносить производство за границу для лучшего удовлетворения иностранных рынков. В результате Япония обнаружила, что имеет больше мощностей по производству автомобилей, чем мог потребить "домашний" рынок, что привело к снижению цен и самым отрицательным образом отразилось на прибылях. В то время как производители, предлагавшие свою продукцию по низким ценам, такие как Honda и Toyota, были способны поддерживать свои исследования и развитие на высоком уровне, другие японские производители автомобилей оказались вынуждены тормозить свое развитие, что прямо вело к снижению продаж. В конце концов основные японские автопроизводители, за исключением Honda и Toyota, были куплены крупными компаниями, такими как GM, Ford, Daimler-Chrysler и Renault. Неожиданно для себя Honda и Toyota оказались вынуждены конкурировать с этими глобальными корпорациями у себя дома.[3]

В то же время стала широко известной (и широко копируемой) производственная система Toyota. По мере того как другие автопроизводители добивались большей эффективности, ведущая роль Toyota как наиболее эффективного в мире производителя автомобилей неизменно сокращалась. Технические достижения в производстве автомобилей быстро копировались, — например, когда на рынке повысился спрос на такие модели, как минивэны (небольшие фургоны) и SUV (Sport Utility Vehicle — спортивно-утилитарный автомобиль)[4], большинство компаний-автопроизводителей отреагировали быстро. Более того, поскольку многие страны в то время уже были насыщены автомобилями, было очевидно, что будущий рост рынка будет иметь место в Азии.

В этой атмосфере компания Toyota разработала долговременную стратегию создания устойчивого конкурентного преимущества. Данная стратегия, хорошо известная как *Smart Design* (интеллектуальная конструкция), *Smart Production* (интеллектуальное производство), *Smart Car* (интеллектуальный автомобиль), концентрировалась на использовании информационных технологий с целью создания генерирующих прибыль инноваций, которые трудно было бы копировать. В течение 90-х годов Toyota стала очень компетентной компанией в области информационных систем.

Под "знаменем" *интеллектуального производства* компания Toyota заменила традиционную единую сборочную линию несколькими мини-линиями, разделенными некоторым запасом деталей (играющего роль буфера) и синхронизируемыми программным обеспечением, что привело к росту производительности, который трудно было повторить. *Интеллектуальное производство* включало оцифровку производственной документации A3 и имитации производственных процессов на системах CAD в ходе процесса разработки, что позволило существенно сократить период от разработки нового автомобиля до появления его на рынке. Инициатива *интеллектуальный автомобиль* привлекла значительные инвестиции в развитие производства встраиваемой электроники и встраи-

---

[3] Rapp W.V. *Automobiles: Toyota Motor Corporation: Gaining and Sustaining Long-term Advantage Through Information Technology*, Columbia Project: Use of Software to Achieve Competitive Co-Principal Investigator The College of International Relations Ritsumeikan University, Kyoto, Japan, *April 2000*. Информация для данного раздела взята из указанного доклада.

[4] Тип автомобиля, промежуточный между внедорожником и универсалом повышенной проходимости.

ваемого программного обеспечения, что позволило компании Toyota выпустить в 1999 году модель Prius. С электронным управлением газом, тормозами и переключением скоростей, удивительно популярный гибрид был поистине компьютером на колесах.

Успех компании Toyota в использовании информационных технологий для многих явился сюрпризом, поскольку до этого компания была известна своим нежеланием использовать информационные технологии. В то время как конкуренты, чтобы сократить долю ручного труда в процессе производства автомобилей, строили высокоавтоматизированные заводы, Toyota провела эксперименты с автоматизацией и пришла к выводу, что это не будет рентабельно, если учесть общие расходы. Пока конкуренты делали огромные инвестиции в компьютеризированные системы планирования и автоматизацию, в компании Toyota полагали, что в центре производственного процесса должны быть люди и что простые ручные системы будут работать лучше. Менеджмент компании Toyota традиционно оказывал сопротивление использованию информационных систем в управлении потоками материалов и планировании производства.[5] Тем удивительнее, что компания, проявлявшая такую осторожность в использовании информационных систем, вдруг неожиданно стала лидером во внедрении этих систем в свои процессы производства, в разработку, а также в свои изделия.

Долговременная перспектива

Некоторым компаниям удается существовать сотни лет, но таких не много. В своей книге *The Living Company* ("Компания-долгожитель") автор Эри де Джуз (Arie de Geus) указывает, что ожидаемый срок жизни многонациональной корпорации (представленной в списке Fortune 500, или эквивалентной) лежит где-то между 40 и 50 годами.[6] Люди живут в среднем 75 лет, иными словами, средний индивидуум может рассчитывать пережить три четверти компаний, существовавших в момент его рождения. Ожидаемый срок жизни средних, хорошо организованных компаний, на порядок меньше, чем у компаний-долгожителей. Ни один вид живых существ не имеет такой большой разницы между средней и максимальной продолжительностью жизни.

Организации, подобные церкви, армии или университетам, существуют столетиями. Почему же крупные компании умирают или их кто-то приобретает в таком "юном возрасте"? Что еще важнее, как хорошим компаниям пережить два поколения менеджмента? Только для того, чтобы попасть в список компаний, который изучал де Джуз, компания должна быть очень успешной. Нужно не только успешно пройти начальную фазу после создания и выдержать все болезни роста, необходимо стать крупной многонациональной корпорацией. А затем мир меняется, и многим из этих очень успешных корпораций не удается приспособиться к новым реалиям.

Организации, добившиеся успеха, склонны иметь привычки, берущие начало во времена, когда они быстро росли и спрос на рынке был огромный. Однако в конце концов все возможности роста были исчерпаны и прошлые успехи больше не указывают путь к дальнейшему росту. В такие моменты способны выжить только организации, развившие в себе способность учиться и адаптироваться к новым условиям. Де Джуз полагает, что именно способность менеджеров предвидеть будущее позволяет им адаптироваться к

---

[5] Liker J., Meier D. The *Toyota Way Fieldbook*, McGraw Hill, *2006*, p. 208–212.

[6] Geus A. *The Living Company: Habits for Survival in a Turbulent Business Environment*, Harvard Business School Press, *1997*, *2002*, p. 1.

этому будущему до того, как будет слишком поздно. Однако, к сожалению, у менеджеров часто развивается "слепота по отношению к будущему", заставляющая их предпочитать старые методы, которые обеспечивали успех в прошлом.

Это представление о "слепоте по отношению к будущему" разделял Клейтон Кристиансен[7] (Clayton Christensen), который показал, что компании, направляющие взоры на наиболее прибыльны сегменты рынка, чтобы поддержать прибыли на достигнутом уровне, и оставляющие менее прибыльные сегменты конкурентам, часто становятся жертвами "подрывных" технологий. Эти компании настолько поглощены стремлением получать максимальные прибыли в данный момент (иными словами, текучкой), что у них нет времени остановиться и подумать о будущем, полном риска и чреватым падением прибылей. Мы постоянно наблюдаем нашествие "подрывных" технологии в компьютерной отрасли, когда меньшая, имеющая большее быстродействие и меньше потребляющая энергию аппаратура заменяет более крупное и прибыльное оборудование. На рынке программного обеспечения можно видеть, как недорогие программные пакеты приходят на смену предложениям с высокой стоимостью одного рабочего места или рассчитанное на самообслуживание ПО "выталкивает" с рынка бизнес, предлагающий программы, использовать которые без консультаций экспертов невозможно. В такой динамичной отрасли способность предвидеть будущее и вовремя подготовиться к адаптации — это вопрос выживания.

Компания Toyota обладает потрясающей способностью учиться и думать эволюционно. Сакичи Тоеда (Sakichi Toyoda) поучал своего сына Киширо (Kiichiro) и племянника Иджи (Eiji): "Будьте впереди времени." В 1934 году компания Toyoda, помимо производства ткацких станков, начала производить автомобили, хотя потребуется два десятилетия, чтобы это новое производство начало приносить прибыль. В 1958 году, при ежегодных продажах автомобилей едва больше 2000 в месяц, к удивлению дилеров и конкурентов, компания Toyota построила новый завод, способный выпускать в месяц 10000 автомобилей.[8] Однако этот завод позволил занять Toyota лидирующее положение на рынке, когда экономика Японии росла гигантскими темпами в 60-х годах. В 1984 году компания Toyota открыла завод New United Motor Manufacturing Incorporated (NUMMI) во Фримонте (Fremont), Калифорния, который начал выпуск автомобилей в Соединенных Штатах за год *до* резкого падения иены относительно доллара. *Интеллектуальная* стратегия Toyota началась с приобретения завода по производству электроники задолго *до* того, как перепроизводство, подобно чуме, поразило японский рынок. В 1995 году компания Toyota приобрела контрольныйпакет акций в компании Daihatsu, чтобы укрепить свои позиции на азиатском рынке. Сегодня Toyota фокусируется на экологически чистых автомобилях и интеллектуальных системах управления уличным движением, которые рассматриваются как главные направления будущего роста.[9]

Организация, стремящаяся выживать на рынке очень долго, должна развить в себе способность предвидеть будущее и принимать все решения на основе этого предвидения.

---

[7]Christensen C. *The Innovator's Dilemma*, Harvard Business School Press, *1997*; Harper Business, *2000*; Christensen C., Raynor M. *The Innovator's Solution*, Harvard Business School Press, *2003*.

[8] Toyoda E. *Toyota: Fifty Years in Motion*, Kodansha, *1987*.

[9] Rapp W.V. *Automobiles: Toyota Motor Corporation: Gaining and Sustaining Long-term Advantage Through Information Technology*, Columbia Project: Use of Software to Achieve Competitive Co-Principal Investigator The College of International Relations Ritsumeikan University, Kyoto, Japan, *April 2000*.

Люди важнее всего

Ученые скажут, что первый шаг в эффективном решении проблемы состоит в том, чтобы сопоставить проблему с некой концептуальной основой, отражающей ясное понимание, как в действительности все происходит. Например, Фредерик Уинслоу Тейлор (Frederick Winslow Taylor) основал свою работу на неверном понимании психологии рабочих. Он исходил из предположения, что рабочие по своей сути ленивы и, как правило, легко заменимы. Практика менеджмента, основывающаяся на этой посылке, была неспособна использовать интеллектуальный потенциал "простых" рабочих. Если в организации с культурой, основывающейся на мышлении массового производства, попытаться внедрить бережливый подход, вероятно, результаты будут посредственные.

В том, что касается использования технологии на рабочем месте, возможны две концептуальные основы. Первая состоит в том, что нужно автоматизировать имеющиеся рабочие места с тем, чтобы сократить потребность в людях или уровне квалификации, необходимом для выполнения работы. Вторая концептуальная основа поощряет использовать технологию для расширения возможностей рабочих.[10] Как было показано, компания Toyota придерживается второй концепции. Но что же не так с первой? Автоматизация рутинных задач кажется неплохой идеей. В действительности, как отмечалось в главе 8, автоматизация повторяющихся (или рутинных) задач в процессах сборки и ввода в эксплуатацию программного обеспечения в среде заказчика исключает возможность отклонений и обеспечивает более стойкий результат.

Однако отказ от использования людей в процессе (или ожидание, что люди будут выполнять процесс механически) означает, что процесс теряет способность изменяться, адаптироваться и открывать что-то новое. Например, компании, производящие лекарства, потратили миллиарды на автоматизацию рутинных задач в исследовании лекарственных препаратов. Они заменили лабораторных техников роботами, сделав возможным выполнение куда большего объема исследований, чем ранее. Предполагалось, что это приведет к появлению множества потрясающих лекарств, однако результаты оказались разочаровывающими. Многие полагают, что причина этого в том, что "ликвидировав" профессию экспериментатора-фармаколога и отказавшись от присутствия в процессе исследований людей, компании тем самым сделали неявное предположение, в каких направлениях должны вестись исследования, отказавшись при этом воспользоваться человеческой способностью замечать неожиданные результаты и открывать новые области для экспериментов.[11]

Разница между автоматизацией тестирования в производстве лекарств и автоматизацией тестирования в разработке программного обеспечения в том, что в первом случае была "ликвидирована" профессия испытателя-фармаколога (представители этой профессии были заменены роботами), а во втором эффективная автоматизация тестирования ПО освободила испытателей (или тестировщиков) от рутинных задач. В результате у испытателей появилось больше времени, чтобы сосредоточиться на защите процесса разработки от ошибок, упрощении использования будущего ПО, диагностическом тестировании (exploratory testing) и тестировании качества (property testing). В целом, искусст-

---

[10] Там же, и см. также Hunter L.W., Lafkas J.J. *Opening the Box: Information Technology, Work Practices, and Wages*, Working Paper 98-02-C, Wharton, Financial Institutions Center, *June 1999*.

[11] Austin R.D. *Supporting Cheap and Rapid Iteration (with a Human Touch)*, Cutter Consortium Business-IT Strategies Advisory Service, *April 19, 2006.*

венное снижение потребности в квалифицированных рабочих ведет к появлению взаимозаменяемых людей, в то время как повышение потребности в квалификации ведет к появлению думающих людей.

Вопрос, на который следует ответить, прежде чем приступать к реализации бережливой инициативы, следующий: что *в действительности* вы думаете о людях? Подумайте о вашем отношении к процессу. Полагаете ли вы, что хорошо документированный процесс, в котором каждый работник выполняет свою работу без дополнительных вопросов, это путь к совершенству? Или, может быть, вы думаете, что нужно стандартизировать процесс с целью освободить выполняющих работу людей от рутинных задач, чтобы у них появилось время для совершенствования процесса? Принципы бережливости основываются на второй точке зрения.

Задумайтесь о своем отношении к планированию. Нервничаете ли вы, когда кто-то предлагает коллективу разработчиков определить, что они смогут сделать в данные сроки, и сообщить об этом менеджерам, не ожидая, пока менеджеры сами решат это? Задумайтесь также о вашем отношении к тому, как выделяется работа. Кажется ли вам противоестественным, когда людям предлагают самим определить, что им делать дальше, вместо того чтобы кто-то другой решил это?

Чтобы бережливый подход оказался действенным, необходимо понимать, что выполняющие работу люди лучше всех знают, как ее следует делать. Вам должна не нравиться автоматизация, целью которой является отказ от использования в производстве людей или снижение требований к их квалификации. Положение вещей, когда решение, принимаемое на рабочем месте, должно быть санкционировано в нескольких инстанциях (или даже в одной инстанции) должно восприниматься как странное. Необходимо твердо верить, что не существует лучшего способа решения проблем, чем дать людям необходимые инструменты и оборудование, организовать обучение, если нужно, и предоставить им самим решать производственные проблемы и совершенствовать процессы, оказывая им по мере необходимости поддержку.

Чему мы научились ?

Прежде чем что-то менять, неплохо сделать обзор полезных идей, которые уже существуют, и попробовать из них что-то почерпнуть. Ниже мы присмотримся к двум инициативам: Шесть сигм и Теории ограничений.

Шесть сигм

Одна из первоначальных проблем, связанных с инициативой Шесть сигм (Six Sigma, (которая давным-давно разрешена)), заключалась в попытке применить в процессе разработки практику, доказавшую свою пригодность в производстве. Когда стало очевидно, что в разработке такое явление, как вариация (variation)[12] является положительным и что неудачные эксперименты генерируют не меньше знания, чем удачные, компании признали, что DFSS (Design for Six Sigma — Проектирование для Шести сигм) — это разновидность программы Шесть сигм, подходящая для разработки программного обеспечения. Благодаря тому, что DFSS предполагает учет пожеланий заказчика и ре-

[12] См. главы 5 и 6 — *Примеч. ред.*

шение проблем с учетом данных, полученных от заказчика (через обратную связь), эта программа предлагает множество хороших средств, которые будут полезны коллективу разработчиков.

Руководители производства и неофициальные лидеры

В чем программа Шесть сигм отличается от производственной системы Toyota, так это ролью ответственных за внесение изменений в соответствующие процессы (change agents). В программе Шесть сигм часто можно встретить рекомендацию, что около одного процента рабочей силы должно получить максимальное профессиональное обучение с тем, чтобы играть роль официальных лидеров (или руководителей) производства, освобожденных от выполнения рядовых функций в этом процессе. Однако эта программа очень мало говорит о подготовке руководителей низшего звена. Вместо того чтобы фокусироваться на руководителях производства, в компании Toyota основное внимание уделяется соответствующей подготовке (уже имеющихся) неофициальных лидеров трудовых коллективов, поскольку здесь существует мнение, что такие люди должны как осуществлять руководство, так и нести ответственность за результаты работы соответствующих коллективов. Мы (авторы) полагаем, что бережливый подход должен ориентироваться на методы компании Toyota и при этом уделять больше времени подготовке (уже сложившихся) неофициальных лидеров коллективов, чем руководителей производства.

Средства и результаты

Нам известен случай, когда предприятие демонстрировало значительный прогресс по всем основным показателям, используемым в данной компании, однако, когда работа предприятия оценивалась корпоративными производственными экспертами, оно получило очень низкую оценку, потому что результаты были получены без использования санкционированных компанией средств совершенствования производства, применение которых, по мнению экспертов, было обязательно. Это яркий пример того, как проводящие в жизнь хорошее начинание забыли его суть. Каждое средство (включая карты потока создания ценности) было создано для решения определенных проблем. Если имеющуюся проблему нельзя решить с помощью рекомендованного средства, оно не должно навязываться. Трудовой коллектив и его лидер должны быть обучены решению проблем на основе их опыта и должны пользоваться доверием при выборе средства решения той или иной проблемы.

Теория ограничений

Эльяху Голдретт (Eliyahu Goldratt), который изобрел термин *теория ограничений*, начинает свою аудиокнигу *Beyond the Goal*[13] ("За пределами цели") с утверждения, что технология может быть полезна только в случае, если она устраняет ограничения. Если вы согласны с этим утверждением, тогда придется согласиться с очевидным фактом, что люди, как правило, прекрасно обходятся без технологии до ее появления, поскольку в подобных случаях всегда действуют правила привычки к ограничениям (или дефициту). Например, лайнер Boeing 777 был создан до появления широкополосной сети

[13] Goldratt E. *Beyond the Goal, Your coach in a Box Series*, Gildan Autio, *2005*.

(broadband) или World Wide Web. Поэтому, чтобы обойти этот дефицит, в компании Boeing были изобретены FTP-сайты и поддерживалось что-то наподобие ранней версии внутренней электронной почты.

Создание новой технологии само по себе не обязательно ведет к исчезновению ограничений, для устранения которых она изначально была предназначена. Это объясняется тем, что компании настолько приспособились к ограничениям, что перестали их замечать. Например, когда электронная почта уже стала широко доступна от провайдеров Интернета, крупным компаниям потребовались годы, чтобы открыть свои внутренние системы e-mail внешнему миру. Мы (авторы) из дому достаточно долго обменивались сообщениями e-mail со своими студенческого возраста детьми, до того как смогли осуществить то же со своими поставщиками через систему e-mail у нас на работе.

Проблема устранения ограничений в том, что требуется преодолеть не только собственно ограничения, но также избавиться от привычек, выработавшихся за годы совместной (с ограничениями) жизни. Голдретт утверждает, что для того, чтобы успешно перейти к использованию новой технологии, необходимо ответить на четыре вопроса.[14]

1. В чем преимущества новой технологии?
2. От какого ограничения новая технология позволит избавиться?
3. Какие правила помогали нам мириться с ограничением?
4. Какие новые правила нам следует использовать после устранения ограничения.

Обратите внимание на ERP (Enterprise Resource Planning — Планирование ресурсов предприятия) — когда-то новую технологию, которая предоставила менеджерам доступ к обширным объемам информации, призванной помочь в принятии лучших решений. До появления ERP решения принимались на основе правил, созданных локальной оптимизацией, поскольку в то время не существовало данных, позволяющих принимать решения, касающиеся всей системы. Например, производственные менеджеры старались оптимизировать эффективность оборудования, а менеджеры по продажам использовали формулы расчета цены, чтобы определить подходящую цену для продукции. Принятие решений на основе локальной оптимизации было приемлемым подходом, пока не существовало альтернативы. Общие правила принятия решений обеспечили гораздо лучшие результаты, чем произвольные решения. Однако, как только технология ERP предоставила доступ к данным, позволяющим принимать лучшие и общие для всей системы решения, следует отказаться от правил локальной оптимизации, иначе потенциал ERP окажется не реализован. Множество неудачных случаев внедрения ERP может быть отнесено на счет нежелания (или неспособности) отказаться от правил принятия решений, использовавшихся до появления ERP.[15]

Изменение правил

Каждый штат в Соединенных Штатах и каждая провинция в Канаде имеет систему регистрации заложенной собственности. Когда собственность предлагается в качестве залога, дающая кредит организация с помощью упомянутой системы может проверить, не находится ли она уже под залогом, и если нет, то зарегистрировать ее как вновь заложенную (и выдать кредит). Почти все аспекты этих систем определяются законодательством штата или провинции.

[14] Там же.
[15] Там же.

В конце 90-х годов я сотрудничала с компанией, занимавшейся автоматизацией системы регистрации заложенной собственности в нескольких штатах США. Идея состояла в том, чтобы создать в этих штатах систему управления изображениями, позволяющую сканировать требуемые законом бумажные документы и таким образом автоматизировать ввод данных и сбор оплаты услуг.

Я была в коллективе, где в то время присутствовали также два канадских эксперта в той же области из Британской Колумбии. ''Мы рассматривали использование системы управления изображениями, –– сказали эксперты, –– но пришли к выводу, что будет лучше совсем отказаться от бумаги и перейти к использованию системы ввода данных через веб. Поскольку законодательство предписывает использование бумажных документов, мы разработали проект закона, который позволил бы избавиться от бумаги, и лоббировали законодателей с тем, чтобы новый закон был принят. На это ушло больше года, однако в конце концов закон приняли и после этого перейти к системе на основе веб было сравнительно быстро и легко. В настоящее время во всех провинциях Канады принято аналогичное законодательство.

К моему отчаянию никто в США не подумал внедрить аналогичную безбумажную технологию. Законы, требующие использования бумажных документов, настолько вошли в плоть и кровь государственного аппарата, что, кажется, никому даже в голову не пришло, что бумажные документы в данном случае — это привычка, которую легко можно устранить, поступив так, как в Канаде. Созданная в результате система, ориентированная на работу с изображениями бумажных документов, получилась очень дорогостоящей в реализации и по-прежнему требующей ручного ввода данных с изображений документов. (Мэри Поппендик)

Критическая цепь

Критической цепью (Critical Chain) Голдратт назвал применение теории ограничений к проектам. Голдратт полагает, что основное ограничение для проекта (разработку продукта он рассматривает как проект) создается, когда приблизительные оценочные сроки воспринимаются как обязательные. Людей, работающих над проектом, просят оценить объем работы, но они в основном вынуждены строить догадки, поскольку каждый проект уникален, и потому проекты по определению содержат массу неопределенностей. Так как заведомо известно, что оценка будет рассматриваться как нечто обязательное, оценивающие стараются приспособиться к обстоятельствам, включая в оценку значительный запас "на всякий случай". Однако, даже если все будет идти хорошо, указанное при оценке время все равно будет израсходовано, поскольку те, кто оценивал, не захотят чтобы кто-то подумал, что оценка была преднамеренно завышена.[16]

Голдратт пишет, что приблизительные оценки должны и восприниматься как таковые, а запас времени "на всякий случай", "присоединенный" к оценкам трудоемкости тех или иных задач, должен быть ликвидирован и это время следует аккумулировать в резервном фонде времени проекта. При подобном подходе отдельные задачи будут выполнены значительно быстрее, а резервный фонд времени в случае необходимости может быть использован для любой задачи (задач) проекта чтобы компенсировать вариацию (variation). В своей книге *Agile Estimating and Planning*[17] ("Гибкая оценка и планирование") Майк Кон (Mike Cohn) подробно объясняет, как применять эту рекомендацию Голдратта к программному обеспечению.

Следует иметь в виду, что резервный фонд времени проекта не ускорит выполнение проекта сам по себе, если не отказаться от старой практики, проявляющейся в предположениях и механизмах оплаты. Критическая цепь не заработает, пока все (от старшего

[16] Там же.
[17] Cohn M. *Agile Estimating and Planning*, Prentice Hall, *2006*.

менеджмента до тех, кто делает оценку) не откажутся от предположения, что решение задач должно осуществляться в сроки, диктуемые оценками. В действительности, если для решения половины задач проекта потребуется времени больше, чем это предусмотрено оценкой, желаемое улучшение достигнуто не будет.

Дополнительная проблема с использованием критической цепи в разработке программного обеспечения связана с предположением, что все задачи, решение которых необходимо для завершения разработки, должны быть известны заранее, так же, как и зависимости между ними. Как было ранее показано, подобные предположения некорректны для большинства проектов создания ПО. Прежде всего, необходимо отказаться от идеи, что все требования должны быть известны в начале процесса разработки, затем следует сосредоточить творческие усилия на устранении зависимостей.

На уровне проекта резервный фонд времени критической цепи, возможно, имеет смысл, однако на уровне задач (являющихся составляющими проекта) существование подобных резервных фондов ничто иное, как привычка приспосабливаться. Если удастся устранить зависимости между задачами и создать сплоченный коллектив, способный осуществлять разработку небольшими "порциями" (или инкрементами), которые по завершении их разработки можно было бы передавать заказчику, больше не будет проблемы, для решения которой была изобретена критическая цепь. Эксперт в теории ограничений сочтет это естественным результатом применения упомянутой теории. После этого остается внимательно присмотреться к старой привычке к ограничениям, которая оказалась встроена в новый подход (с использованием критической цепи) и создать новые правила, ликвидирующие ставшие ненужными привычки.

Привычка приспосабливаться

В нашей повседневной практике, процессах и правилах скрыты множество привычек к реальным или воображаемым ограничениям (или дефициту). Эта приспособленность так вошла в нашу плоть и кровь, что часто мы даже не осознаем, что она есть. Нередко эта приспособленность предназначена для обхода конкретных проблем, которые больше не существуют или которые не существовали для нас никогда. Например, раннее определение цены и планирование в основном является приспособлением к существующему механизму финансирования, который обуславливает выделение всех денег в начале проекта. Когда проекты финансируются поэтапно (или инкрементно), как это имеет место с разработкой большинства новых продуктов, нет причин определять эти параметры в начале цикла разработки.

Среда разработки 30 лет назад, 20 лет назад и даже 10 лет назад была абсолютно не такой, как сейчас, и наша приспособленность к ограниченности компьютерной памяти, быстродействию процессора, вычислительным мощностям и функциональным возможностям программного обеспечения больше неуместна. Объектно-ориентированное программирование, автоматизированное блочное и приемочное тестирование, ориентированные на сервис архитектуры (service-oriented architectures), ориентированная на темы документация (topic-oriented documentation) и огромное количество всевозможных средств для организации процесса разработки небольшими партиями, появились в последние годы и постоянно совершенствуются. Веб является мощным средством распространения инноваций, варьирующихся от программирования для Open Source до мощных поисковых возможностей. В столь быстро изменяющейся среде целесообразно постоянно выявлять и устранять свои привычки и приспособленность к ограничениям и нехваткам, которые больше не существуют.

Гипотеза

Когда уже решено, куда вы решили отправиться, с учетом того, что вы уже знаете о бережливых инициативах, следующим шагом должно стать создание гипотезы о том, как начать внедрение бережливого подхода в ваших условиях. Мы (авторы) полагаем, что это эффективнее всего начать следующим образом.

- Обучать лидеров трудовых коллективов и руководителей младшего звена (а не ответственных за внедрение новых методов (change agents)).
- Поддерживать стремление улучшать процессы (а не просто придерживаться документации).
- Использовать ограниченное число критериев (все критерии на уровне системы), которые обеспечивали бы соответствующее функционирование на уровне подсистем.

Обучение

В начале XX века, в то время как Сакичи Тоеда занимался производством ткацких станков в Японии, когда Генри Форд только что основал свою автомобильную компанию, а Фредерик Тейлор опубликовал свои идеи о науке управлять, ученый по имени Чарльз Аллен (Charles Allen) в Нью-Бедфорде (New Bedford), Массачусетс, разработал программу производственного обучения. Для этого он избрал новый подход — он полагал, что наиболее эффективно учеба осуществляется на практике, поэтому он учил людей, как следует учиться непосредственно во время практической работы.

К 1917 году в Массачусетсе была создана модель Американского профессионального обучения (American vocational education), которая впоследствии распространилась по всей стране. Директором этой программы стал Чарльз Аллен. В то время продолжалась первая мировая война и Совет США по морским перевозкам (US Shipping Board) решил построить тысячу новых морских судов. Возникла необходимость в десятикратном увеличении числа квалифицированных кораблестроителей и Совет по морским перевозкам поручил Чарльзу Аллену взять на себя ответственность за их обучение. В течение следующих двух лет 88000 рабочих прошли через программу обучения, разработанную Алленом, что рассматривалось как большой успех. Свой опыт и философию Аллен суммировал в книге *The Instructor, the Man and the Job*[18] ("Инструктор, человек, работа").

Чарльз Аллен исходил из того, что в основном обучение происходит на работе, поэтому он сосредоточился на обучении руководителей низшего звена (supervisors) тому, как следует учить рабочих. Его первое предположение состояло в том, что такие руководители прекрасно знают работу, которой они призваны руководить. Программы обучения Аллена учили руководителей, как лучше передать это знание новым рабочим в четыре этапа: подготовка, изложение, использование и проверка. В то время как в автомобильной отрасли производственным инженерам предлагали найти "наилучший способ" выполнения работы и рассказать рабочим, что нужно делать, Аллен учил руководителей

[18] Allen C. *The Instructor, the Man and the Job: A Handbook for Instructors of Industrial and Vocational Subjects*, J. B. Lippincott Co., *1919*.

низшего звена из судостроительной отрасли сотрудничать с рабочими в решении проблем. Его философией было: "Если учащийся не учится, значит, учитель не учит".

Перенесемся на 20 лет вперед — в 1940 год. Совет США по военному производству (US War Production Board) столкнулся с аналогичной проблемой. Резкое увеличение производства самолетов и боеприпасов создало огромную потребность в квалифицированных рабочих, которая еще более усиливалась призывом большинства молодых людей на военную службу. Учитывая успех Аллена, Совет по военному производству разработал Программу обучения на работе (Training Within Industry — TWI), которая была ориентирована на обучение руководителей низшего звена тому, как обучать рабочих. Ниже перечислены основные составляющие этой программы.[19]

- **Профессиональный инструктаж (Job Instruction — JI).** Данная часть программы обучения основывалась на предположении, что опытные рабочие, возможно, очень хороши в своем деле, но у них мало опыта в передаче своих знаний другим. Используя четыре этапа Аллена, руководителей низшего звена учили, как:
  - **а)** подготовить рабочего;
  - **б)** изложить суть операции;
  - **в)** предложить рабочему использовать приобретенный навык — попробовать выполнять соответствующую операцию;
  - **г)** довести дело до конца, часто проверяя результаты, задавая наводящие вопросы, постепенно сводя наставничество к минимуму, по мере приобретения рабочим опыта.
- **Методы работы (Job Methods — JM):** Данная часть программы обучения основывалась на предположении, что рабочие имеют много хороших идей о том, как усовершенствовать процесс, но, возможно, их не поощряли реализовывать эти идеи. Руководителей учили помочь рабочим:
  - **а)** разделить работу на операции;
  - **б)** ставить под сомнение каждую мелочь;
  - **в)** создать новый метод в сотрудничестве с коллегами;
  - **г)** широко пропагандировать новый метод, добиться его одобрения и применять его.
- **Атмосфера на работе (Job Relations — JR).** Данная часть программы обучения учила руководителей низшего звена применять к каждому человеку индивидуальный подход и призвана была помочь им разрешать "человеческие" проблемы эффективно и справедливо. Руководителей учили:
  - **а)** разбираться в вопросе;
  - **б)** взвешивать обстоятельства и не торопиться с выводами;
  - **в)** вовремя принимать меры;
  - **г)** проверять результаты.

Вслед за Алленом обучающая программа TWI исходила из того, что руководители низшего звена знают, как выполнять работу своих подчиненных и поэтому могут играть

[19] Huntzinger J. *The Roots of Lean*, `www.lean.org/Community/Resources/ThinkersCorner.cfm`.

роль преподавателей и наставников. Интересно заметить, что шестой и седьмой пункты Деминга посвящены тем же вопросам, что и программа обучения TWI.

1. Поощряйте учебу. Менеджеры должны знать, как делать работу, которой они управляют, и быть способны обучать рабочих. Менеджеры также должны учиться, чтобы понимать производство.
2. Будьте лидером. Работа менеджера состоит в том, чтобы помочь людям лучше выполнять свою работу и устранять помехи в системе, которые мешают им выполнять работу с гордостью. Наибольшая проблема Америки — недостаточное использование способностей людей.

Вероятно, два миллиона руководителей низшего звена в США прошли обучение по программе TWI, что привело к значительному повышению производительности и резкому сокращению конфликтов и недовольства. Когда война подошла к концу в 1945 году, закончилась и программа TWI — по крайней мере в США; TWI-обучение было предложено в странах, восстанавливающихся после войны, и было особенно хорошо принято в Японии. Как и с идеями Деминга, компания Toyota внедрила идеи TWI у себя, добавив собственный опыт и превратив все это вместе взятое в программу Toyota. По сей день руководители в компании Toyota проходят обучение, основывающееся на идеях Чарльза Аллена, которые были реализованы в программе TWI.[20]

Стремление улучшать процессы

Один из руководителей в компании Toyota, Теруюки Миноура (Teruyuki Minoura) утверждает, что буква "T" в аббревиатуре TPS соответствует слову "thinking" (интеллектуальный) и что величайшая сила "Thinking Production System" (Системы интеллектуального производства) компании Toyota в том, как она воспитывает людей. "При системе с "выталкиванием" ('push' system) существует малый спрос на интеллект рабочих, поскольку они просто производят, руководствуясь данными им инструкциями, — утверждает он. — В противоположность этому при системе с "вытягиванием" ('pull' system) от рабочего требуется использовать свой интеллект, чтобы создать процесс производства, в котором он один должен решать, что должно быть сделано и как быстро это должно быть сделано."

Большинство программ совершенствования процесса слишком сильный акцент делает на документацию и совершенно выпускают из виду поощрение каждого работника думать каждую минуту о том, как может быть усовершенствовано выполнение им работы. Конечно, многие компании полагают, что они поощряют рабочих думать об улучшениях. Многие годы мы имеем систему внесения предложений, поощряющую людей подавать свои предложения в письменном виде, и многие годы результативность этой системы приближается к нулю.

Проблема в том, что большинство идей из программ внесения предложений и мозговых штурмов передаются кому-то другому для оценки и иногда для реализации. Это ошибка. Реализовывать хорошие идеи должны те, кто их предлагает. Не следует предлагать людям "опускать" свои идеи в систему внесения предложений, чтобы их реализовал кто-то другой. Они не должны просто добавлять свои идеи в длинный список хороших идей. Когда хоро-

[20] Smalley A. *TPS vs. Lean and the Law of Unintended Consequences*, Там же.

шая идея улучшить что-то становится чьей-то заботой, ее авторы уже не могут влиять на ее реализацию и неявное (tacit) знание проблемы при этом также теряется.

Все работники должны постоянно присматриваться к используемому в настоящий момент процессу и должны поощряться использовать эффективные методы решения проблем для испытания новых идей и внедрения тех из них, которые работают. Документация должна существовать как основа этого процесса решения проблем. Она должна разрабатываться, использоваться, поддерживаться и без труда изменяться трудовыми коллективами.

Если мы думающие люди, ориентированные на создание потребительской ценности, необходимо тщательно присмотреться к программам оценки, основывающимся на документации. Как правило, подобные программы оценивают зрелость организации по тому, насколько точно она (организация) придерживается документированных процедур. Хотя причина этого кажется на первый взгляд достаточно безобидной, на практике программа оценки обычно оказывает давление на организацию, чтобы та заморозила свою документацию и, следовательно, не меняла свои процессы. Результатом подобного подхода является беспокоящая тенденция лишить рабочих, занятых непосредственным созданием ценности, "права думать". Очень часто рабочие поощряются делать именно то, что предписывает документация, в то время как акцент следовало бы сделать на поощрении совершенствовать документацию. Основанные на документации программы оценки проявляют тенденцию ценить неизменность документации. В действительности постоянные изменения в документации должны рассматриваться как признак, что в данной организации научились думать.

Критерии

Поиск эффективных критериев для оценки работы коллективов разработчиков всегда был непростой задачей, поскольку результаты часто не очевидны в течение некоторого времени после завершения разработки. Это ведет к широкому использованию критериев, основывающихся на предположении, что если каждый этап процесса оптимизирован, результаты процесса также будут оптимальны. Это некорректный взгляд на то, как работает система. Хотя верно, что оптимизация каждого этапа процесса приведет к улучшению результатов (особенно, если речь идет о неуправляемом процессе), однако верно и то, что локальная оптимизация в конечном итоге саботирует оптимизацию системы в целом.

Попытки улучшить ложные критерии создают неправильные стимулы, часто ведущие к непредвиденным результатам. Например, программное обеспечение поставляется вовремя, без перерасхода бюджета, с запланированным набором функциональных возможностей, однако при этом абсолютно не удовлетворяет заказчиков.

Ложные критерии

Моя коллега, которую я назову Мишель (не настоящее имя) пару лет назад возглавила организацию, занимающуюся созданием программного обеспечения, которая была в плачевном состоянии. Производительность разработчиков измерялась строками программного кода в час. А производительность тестировщиков — числом обнаруженных дефектов (чем больше дефектов выявлено, тем выше производительность).

Трудно себе представить два критерия, которые были бы более неудачны. Оценка строками программного кода в час поощряла разработчиков выдавать количество, не заботясь о создаваемой потребительской ценности. Оценка на основе количества выявленных дефектов мешала тестировщикам сотрудничать с разработчиками с целью создания свободного от дефектов кода.

Первым делом Мишель изменила критерии. Прекратился подсчет строк программного кода, и коллективы (как разработчиков, так и тестировщиков) больше не вознаграждались за выявленные дефекты. За последние два года, основываясь на здравом смысле и применяя проверенные подходы, Мишель смогла трансформировать то, что она возглавила, в эффективную организацию. (Том Поппендик)

Итак, какие критерии следует использовать для создания нужных стимулов? Мы предлагаем вместо увеличения числа критериев сократить их число и найти критерии уровня системы, которые обеспечивали бы соответствующее функционирование на уровне подсистем. В бережливых организациях такие критерии хорошо известны; это время цикла (cycle time), финансовые результаты и удовлетворение заказчика. Давайте посмотрим, как они могут быть использованы в разработке программного обеспечения.

Время цикла

Наиболее фундаментальным критерием бережливости является время цикла: сколько времени (в среднем) потребуется, чтобы пройти путь от создания концепции до получения прибыли или от получения "заказа" до создания программного обеспечения, готового к вводу в эксплуатацию у заказчика? Этот единственный критерий является мерилом мощности и возможностей вашего процесса на системном уровне. Кроме того, он позволяет выявить каждый изъян (или источник потерь) в системе: каждый случай недостатка квалификации, нехватки мощности или неудачной реализации увеличивает время цикла. Кроме того, время цикла увеличивает попытка сделать слишком много сразу, так же, как и сложность, излишние зависимости и отсутствие толерантности к изменениям.

При определении времени цикла не следует измерять минимально возможное время функционирования системы. Это плохая идея измерять, как хорошо вы умеете проталкивать работу, поскольку в бережливой среде такое проталкивание не является ни необходимым, ни приемлемым. Вопрос не в том, как быстро вы сумеете осуществить одиночную поставку, а в том, как быстро вы можете осуществлять поставки или реагировать на запросы заказчика постоянно.

Целью организации-разработчика является прежде всего установить повторяющееся, надежное время цикла для каждой разновидности работы, а затем сокращать этот временной промежуток с помощью непрекращающегося процесса усовершенствования. Этот единственный критерий стимулирует разного рода полезные действия в каждом подразделении организации, поскольку он заставляет ориентироваться на конечный результат и побуждает к сотрудничеству.

Для сокращения времени цикла необходимо полноценное участие всех людей, занятых в создании потребительской ценности. Этот показатель стимулирует взаимодействие, повышение качества и совершенствование стандартов. Почти все другие показатели, которые поддаются оценке (способность укладываться в сроки, предоставляемое качество, быстрота реагирования на запросы рынка и даже степень загрузки ресурсов), повысятся при сокращении времени цикла. В то же время, если вы попытаетесь оптимизировать подсистемы, время цикла системы, возможно, будет увеличиваться.

Хотелось бы увидеть несколько примеров того, что именно нужно измерять, чтобы вычислить время цикла.

Когда дефект попадает в ваш список дефектов, зафиксируйте соответствующую дату. Когда он будет устранен, подсчитайте, как долго он находился в списке. Постоянно фиксируйте эти числа: среднее время цикла, среднее отклонение от времени цикла, средний "возраст" дефектов в списке и т.п.

Когда некоторая работа попадает в журнал ожидающей своей очереди работы, зафиксируйте соответствующую дату. Если после этого данная работа будет разделена на более мелкие "порции", каждая такая "порция" будет иметь исходную дату. Если два работы из журнала будут объединены, новая объединенная работа получит более раннюю дату. Когда работа будет выполнена и соответствующий продукт готов к поставке, вычтите исходную дату из даты поставки. Полученная разница будет равна времени цикла. Вычислите среднее время цикла и среднее отклонение от времени цикла в каждом релизе. Также вычислите среднее время ожидания для работ, все еще находящихся в журнале.

Но в журнале содержатся очень трудоемкие работы, к выполнению которых никто еще даже не приступал и заказчики уже успели забыть о них.

В этом случае, возможно, было бы полезно разделить содержимое журнала предстоящей работы на три категории.

А. Работы, которые заказчики признают в качестве своих заказов (т.е. ожидают их выполнения), а также работы, трудоемкость которых поддается оценке.

Б. Работы, которые требуется разделить на более мелкие "порции", чтобы можно было определить архитектуру, выполнить общее планирование и т.п.

В. Очень трудоемкие работы.

Если работа может принадлежать нескольким категориям, ее следует отнести к наивысшей категории. Например, если заказчики ожидают выполнения своего заказа, значит, соответствующую работу однозначно следует поместить в категорию А.

Помечайте текущей датой каждую перемену в журнале предстоящей работы. Добавьте новую дату, когда работа переместится в категорию Б из В. Добавьте еще одну дату, когда работа переместится в категорию А.

Когда работа из журнала будет выполнена и соответствующий продукт готов к вводу в эксплуатацию у заказчика (deployment), вычислите три категории времени цикла.

1. Время, прошедшее после перемещения данной работы в категорию А.
2. Время, прошедшее после перемещения данной работы в категорию Б.
3. Время, прошедшее после перемещения данной работы в категорию В.

При каждом релизе вычислите среднее время цикла и среднее отклонение от времени цикла для каждой из трех категорий. О чем говорят эти цифры? Может быть, нужно добавить еще какие-нибудь категории?

Подсчитайте также средний "возраст" работ категории А, оставшихся в журнале предстоящей работы. Если данная величина значительно превосходит время двух релиз-циклов, значит, вы еще не ограничили количество работы в очереди (или в журнале) возможностями вашей организации ее выполнить.

Финансовая отдача

Большинство проектов, связанных с разработкой программного обеспечения, финансируется с расчетом получить прибыль. Даже государственные и некоммерческие организации делают инвестиции, руководствуясь тем же расчетом. Мы полагаем, что основным критерием успеха разработки должен быть коммерческий успех. Это означает, есте-

ственно, что можно начинать разработку, только если расчеты получить прибыль реалистичны и после начала разработки *следует постоянно оценивать реальные результаты*. Конечно, оценка вероятности получить прибыль требует времени и кроме того, здесь могут присутствовать иные факторы. Однако основной принцип состоит в том, что, если вы определитесь, чего хотите, вы с большей вероятностью это получите.

Если конечной целью разработки является получение приносящего прибыль продукта, тогда коллектив разработчиков должен реализовать модель прибылей и убытков (profit and loss — P&L) для этого продукта, чтобы нужным образом направлять свою работу для создания наиболее прибыльного продукта. Для внутренней разработки, разработки по субподряду или некоммерческой разработки коллективу целесообразно применить такой показатель, как отдача от инвестиций (return on investment — ROI) или иной экономический показатель (например, производительность (throughput)), который использовался для обоснования инвестиции.[21]

Мы обнаружили, что большинство компаний старается оградить свои коллективы разработчиков от финансовых вопросов, связанных с их работой. Однако в каждом коллективе, где нам довелось побывать, были бы рады понять экономические цели своей работы и принимать решения с учетом этих целей, а затем испытать чувство успеха, приходящее при достижении целей.

Иногда цель не связана с получением прибыли: это может быть увеличение аудитории общественной радиостанции или числа детей, участвующих в компании вакцинации. В любом случае первое, что необходимо сделать, это объяснить коллективу разработчиков, в чем в данном случае будет состоять успех (а в качестве последнего шага необходимо дать оценку этому успеху (или признать отсутствие такового) и сделать так, чтобы успех был виден всем).

Удовлетворение заказчика

Выдающиеся решения не оставляют заказчиков равнодушными. Удовлетворяться должны истинные, фундаментальные запросы заказчиков, и созданный по их заказу продукт не должен уступать конкурирующим продуктам. Целью бережливого подхода к разработке программного обеспечения должен быть поиск способов порадовать заказчика, поняв его ситуацию и предложив коренное решение его проблемы. В своей книге *The Ultimate Question*[22] ("Главный вопрос") Фред Рейхельд (Fred Reichheld) утверждает, что компании, которые радуют заказчиков, приобретают устойчивое конкурентное преимущество, в то время как компании, вызывающие недовольство заказчиков, лишатся их, как только у последних появится более привлекательная альтернатива.

Рейхельд предлагает единственный простой критерий, позволяющий судить, насколько довольны заказчики: чистая оценка готовности заказчика рекомендовать организацию, создавшую данный продукт (net promoter score). Задайте заказчикам один простой вопрос: насколько вы готовы рекомендовать наш продукт или услугу? Оценивать следует по 11-балльной системе (0–10), где 0 означает "не буду рекомендовать ни в коем случае", а 10 — "буду рекомендовать непременно". Ответы следует группировать сле-

---

[21] Poppendieck M., Poppendieck T. *Lean Software Development: An Agile Toolkit* Addison-Wesley, p. 83–91. Примеры моделей P&L и ROI для коллективов разработчиков можно найти в указанной книге.

[22] Reichheld F. *The Ultimate Question: Driving Good Profits and True Growth*, Harvard Business School Press, *2006*.

дующим образом: оценки 9 и 10 определяют данного заказчика как “промоутера”. Оценки 0–6 означают, что данный заказчик сыграет для вас роль “очернителя”. Наконец, оценки 7 и 8 свидетельствуют, что заказчик испытывает равнодушие по отношению вашей организации. (Система оценок, аналогичная данной, используется в школах (американских), где сумма баллов, равная 90–100, соответствует оценкам A или B, в то время как любая сумма, меньшая 70, рассматривается как провал.)

Для того чтобы вычислить оценку готовности рекомендовать, вычтите число процентов “очернителей” из числа процентов “промоутеров”. В результате вы получите число в диапазоне от –100% до 100%. Для средней компании эта оценка может быть равна 10%. Для по-настоящему хороших компаний данная оценка не должна быть ниже 50%. А отрицательные значения — причина для обеспокоенности. В своей книге Рейхельд представляет аргументы, что оценка готовности рекомендовать (о которой речь шла выше) тесно связана как с долей рынка, занимаемой оцениваемой организацией, так и с ее прибыльностью, что превращает данную оценку в удобный показатель, выражающий одним показателем удовлетворенность заказчика.

Дорожная карта

> Дорога в тысячу миль начинается с первого шага.[23]

Бережливая разработка — это дорога, которая начинается с места, где вы в данный момент находитесь, и ведет далеко в будущее. В качестве напутственного слова мы предлагаем краткую дорожную карту, которая должна помочь вам в пути.

1. **Начните оттуда, где вы в настоящее время находитесь.** Как вы создаете потребительскую ценность и получаете прибыль?
2. **Найдите самое крупное ограничение (или дефицит).** Что именно является крупнейшей проблемой, ограничивающей вашу способность создавать ценность и получать прибыль?
3. **Постарайтесь предвидеть главную угрозу.** Что является главной угрозой вашей способности создавать ценность и получать прибыль в долговременной перспективе?
4. **Оцени*т*е вашу культуру.** Создайте и постоянно укрепляйте культуру глубокого уважения к рабочим, непосредственно создающим потребительскую ценность, а также к партнерам. Устраните все, что мешает появлению профессиональной гордости.
5. **Организуйте обучение.** Обучайте лидеров трудовых коллективов, а также менеджеров, как возглавлять, как учить и как помогать рабочим использовать упорядоченный подход к улучшению производственных процессов.
6. **Постарайтесь решить крупнейшую проблему.** Устраните крупнейшее ограничение, мешающее работать трудовым коллективам. Если нужно, проведите серию быстрых экспериментов, которые бы в конце концов указали путь к решению.
7. **Устраняйте привычки к ограничениям.** Выявите правила, позволяющие мириться с ограничениями. Определите, какими должны быть новые правила.

---

[23] Лао-цзы (604 до н.э .– 531 до н.э.), *Путь Лао-цзы*; данную цитату можно перевести иначе: “Дорога в тысячу миль начинается с места, где ты стоишь”.

8. **Выполняйте оценку.** Определите, улучшились ли такие показатели, как время цикла (из конца в конец системы), истинная прибыльность и подлинное удовлетворение заказчика.
9. **Совершенствуйте.** Внедряйте улучшения, которые проверены и поддерживаются результатами.
10. **Повторяйте цикл.** После того как самая крупная проблема будет устранена, самой крупной станет другая проблема. Найдите ее и повторите цикл.

Попробуйте это

Мы предлагаем 21-этапную программу реализации бережливого подхода к разработке программного обеспечения. Каждый шаг содержит только краткий набросок того, что должно быть сделано. Детали можно найти в соответствующих главах этой книги. Рассматривайте это как рекомендации — попробуйте их и посмотрите, дают ли они нужные результаты для вашей организации.

Оптимизировать целое

1. **Осуществлять бережливые инициативы для всего потока создания ценности и продукта в целом.** Объектом бережливых инициатив должен быть весь поток создания ценности, начинающийся и заканчивающийся заказчиком. Следует сосредоточиться на продукте в целом, а не только на программном обеспечении. Постройте карту потока создания ценности и выявляйте все, что мешает потоку, а также случаи переделки уже сделанной работы. Приводите в порядок вышедшие из строя процессы и обеспечивайте недостающие функциональные возможности системы.
2. **Использовать соответствующие показатели.** Велика вероятность, что в потоке создания ценности используются локальные показатели, применение которых ведет к оптимизации составляющих. Перестаньте использовать локальные показатели и используйте взамен показатели, оценивающие поток создания ценности в целом!
3. **Сокращать стоимость пересечения организационных барьеров.** Если в потоке создания ценности имеются большие задержки, скорее всего, это обусловлено существованием барьеров между организациями или между подразделениями одной организации. Присмотритесь к этим барьерам и оцените, во что обходится их преодоление. Любая мера, ускоряющая поток создания ценности через эти барьеры, почти наверняка приведет к сокращению стоимости пересечения этих барьеров.

Уважать людей

4. **Организовать обучение естественных лидеров трудовых коллективов и руководителей низшего звена.** Вместо обучения руководителей производства основное внимание следует уделить обучению естественных лидеров трудовых коллективов. Кроме того, таким лидерам следует предоставлять поддержку и время для реализации принципов бережливости на своих участках работы.

5. **Ответственность и право принимать решения следует переместить на низший уровень.** Бережливые инициативы должны реализовываться создающими потребительскую ценность коллективами, руководимыми своими естественными лидерами и руководителями низшего звена. Следует предоставить самим коллективам создавать свои производственные процессы, оказывая им в этом всяческую поддержку.
6. **Воспитывать профессиональную гордость.** Множество вещей мешают профессиональной гордости: предпочтение индивидуальным поощрениям перед коллективными, плохие условия труда, неприемлемые сроки, отсутствие времени для должных тестирования и рефакторинга, навязанные процессы, рутинная (больше подходящая для роботов) работа и т.п. Старайтесь искоренить все перечисленное: не создавайте конфликтных стимулов между членами коллектива, автоматизируйте рутинные задачи, предоставьте работникам самим определить оптимальные методы выполнения работы, никогда не требуйте соблюдения невозможных сроков. Поощряйте приверженность стоящим задачам и стремление получить высококачественные результаты. Это окажет куда больший положительный эффект, чем индивидуальные стимулы и премии.

Доставлять быстро

7. **Выполнять работу небольшими партиями.** Следует стремиться сокращать объемы проектов. Также следует сокращать релиз-цикл. И конечно же список ожидающей своей очереди работы должен быть максимально короткий.
8. **Ограничить объем предстоящей работы возможностями ее выполнить.** Следует добиться, чтобы коллектив работал с постоянным ритмом; это позволит определить его производительность. Работа из журнала ожидающей своей очереди работы должна выбираться в темпе, соответствующем имеющейся производительности; прежде чем оттуда будет "взята" следующая работа, текущая работа должна быть выполнена. Следует стремиться ограничивать размеры подобных очередей, и новая работа не должна помещаться в очередь, если там нет для нее свободного места.
9. **Основное внимание уделять времени цикла, а не загрузке (или использованию) ресурсов.** Не следует беспокоиться по поводу загрузки ресурсов. Взамен нужно обратить внимание на время, требующееся для поставки продукта на рынок или заказчику.

Не торопиться брать обязательства

10. **Отказаться от представления, что хорошей практикой является начинать разработку, имея подробную спецификацию.** Параллельная (concurrent) разработка предполагает составление спецификации в процессе создания продукта. Подобная разработка экономит деньги и время, обеспечивает лучшие результаты и позволяет принимать решения на основе последних данных. Иными словами, ничего плохого в параллельной разработке нет, и если это так, зачем же тогда придерживаться идеи, что подробная спецификация должна быть создана до начала процесса разработки? Если предварительное создание спецификации необходимо, чтобы обеспечить авансовое финансирование, тогда лучше перейти к инкрементному (или постепенному)

финансированию. А если необходимость предварительного создания спецификации диктуется контрактом с фиксированными расценками, в этом случае лучше отдать предпочтение какой-либо иной разновидности контракта.

11. **Устранять зависимости.** Вместо того чтобы беспокоиться по поводу зависимостей, делайте все возможное, чтобы устранить их. Делимая архитектура системы имеет фундаментальное значение. Старайтесь устранить все зависимости и терпите их только когда нет другого выхода.
12. **Старайтесь иметь несколько альтернатив.** Для каждого необратимого решения старайтесь поддерживать несколько вариантов и не отказывайтесь ни от одного из этих вариантов до последнего момента. Старайтесь исследовать и понять каждый вариант решения, чтобы получить необходимую информацию и уверенность для лучшего выбора. Во всех прочих случаях старайтесь создавать толерантный к изменениям программный код, содержите его чистым и простым и в случае необходимости без колебаний вносите в него изменения.

Создавать знание

13. **Над отдельными элементами архитектуры работают многофункциональные коллективы.** Следует сделать так, чтобы архитектура будущей системы позволяла разделить проект на логические модули, работать над которыми смогли бы **многофункциональные** коллективы, представляющие все этапы в потоке создания ценности. У каждого коллектива должны быть руководство и стимулы, которые бы поддерживали приверженность (общим целям), прозрачность деятельности и интенсивную обратную связь. Такие коллективы должны часто обмениваться информацией и постоянно учиться.
14. **Поддерживать атмосферу постоянного совершенствования.** Создавать условия, когда каждый коллектив постоянно стремится совершенствовать свои процессы.
15. **Использовать проверенные методы решения проблем.** Коллективы разработчиков должны использовать цикл PDCA или научный метод, или какое-либо сочетание этих методов решения проблем. Сначала следует создать гипотезу, затем провести серию быстрых экспериментов, создать лаконичную документацию и реализовать лучшую альтернативу.

Встраивать качество

16. **Осуществлять синхронизацию.** Целенаправленно сокращайте объемы незавершенной работы. Создавайте тесты в первую очередь. Тестируйте программный код так скоро, как это возможно. Не создавайте список дефектов; останавливайте работу и устраняйте дефекты в момент их обнаружения. Максимально часто осуществляйте интегрирацию созданного программного кода. Используйте вложенную синхронизацию для осуществления интеграции с более крупными и сложными системами.
17. **Автоматизировать.** Следует понимать, что люди склонны ошибаться. Поэтому создаваемое программное обеспечение следует защищать, широко применяя автоматизацию. Автоматизируйте каждый поддающийся этому процесс так быстро,

как только можно. Автоматизируйте процессы тестирования, сборки, инсталляции и любые другие. Однако осуществлять это следует так, чтобы люди не исключались из процессов с тем, чтобы они могли эти процессы совершенствовать.

18. **Использовать рефакторинг.** Содержите библиотеку кода чистой и простой, без какого-либо дублирования. Подвергайте рефакторингу программный код, тесты и документацию, чтобы свести сложность к минимуму.

Ликвидировать потери

19. **Обеспечить маркетинговое и техническое руководство.** Обеспечить персональную ответственность за выработку глубокого понимания того, что ценят пользователи, и тесно с этим связанное глубокое понимание, что технология может дать. Следует также проследить, чтобы разрабатывалось как раз то, что нужно.
20. **Не создавать ничего кроме потребительской ценности.** Обеспечить, чтобы все действия в процессе были направлены на создание ценности или улучшение возможностей создавать ценность. Следует регулярно замерять эффективность цикла процесса (process cycle efficiency — PCE) и постоянно стремиться улучшить этот показатель.
21. **Создавать меньше программного кода.** Следует агрессивно ограничивать функциональные возможности системы только теми, которые абсолютно необходимы для создания ценности. Создать в организации атмосферу нетерпимости к излишней сложности.

Список литературы

Google Philosophy—The Things Google Has Found to Be True `www.google.com/corporate/tenthings.html`.

Inditex: The Future of Fast Fashion The Economist, June 18, 2005.

PC Magazine Technical Excellence Awards — *Web Applications: Google* PC Magazine *December 14, 1999*, Volume 18, Number 22, p. 104.

Allen C. *The Instructor, the Man and the Job: A Handbook for Instructors of Industrial and Vocational Subjects* J. B. Lippincott Co., Philadelphia, 1919.

Материал, основанный на работе Алена, по-прежнему используется в компании Toyota.

Appleton B. *Articles on Agile Software Configuration Management*, `www.cmwiki.com/AgileSCMArticles`.

Astels D. *Test-Driven Development*, Prentice Hall, 2004.

Заслуживающее премии издание. Для тех, кто реализует у себя разработку через тестирование (test-driven development), это как раз подходящая книга.

Augustine S. *Managing Agile Projects*, Addison-Wesley, 2005.

Прекрасное пособие по практике гибкой (agile) разработки.

Austin R. D. *Measuring and Managing Performance in Organizations*, Dorset House, 1996.

Вы получите результаты в соответствии с показателями, которые у вас используются. Невозможно оценивать абсолютно все; что же в этой ситуации делать? Это одна из лучших книг, посвященных оценке производительности.

Balle F., Balle M. *Lean or Six Sigma*, `www.lean.org/library/leanorsigma.pdf`.

Beck K., Andres C. *Extreme Programming Explained, Second Edition.* Addison-Wesley, 2005.

Книга об экстремальном программировании, с которой все началось. Первое издание было тепло встречено читателями. Второе издание — это совершенно новая книга, и это прекрасно.

Boehm B., Turner R. *Balancing Agility and Discipline: A Guide for the Perplexed*, Addison-Wesley, 2004.

Чем руководствоваться: текущим (только что приобретенным) знанием или заранее составленным планом?

Brin S., Page L. *The Anatomy of a Large-Scale Hypertextual Web Search Engine*, Computer Networks and ISDN Systems, 30(1–7):107–117, April 1998.

Brooks F. *The Mythical Man-Month, Anniversary Edition*, Addison-Wesley, 1995.

Классическое издание (которое впервые было опубликовано в 1975 г.), выдержавшее проверку временем. Книга демонстрирует, как мало изменилось за тридцать лет.

Carrison D. *Deadline! How Premier Organizations Win the Race Against Time*, AMACOM (American Management Association), 2003.

Истории о том, как несколько грандиозных проектов были выполнены в невероятно сжатые сроки. Особого внимания заслуживает глава, посвященная истории создания лайнера Boeing 777.

Christensen C. *The Innovator's Dilemma*, Harvard Business School Press, 1997; Harper Business, 2000.

Замечательная книга о том, как появление революционных технологий почти всегда приводит к освобождению от должности руководителей-маркетологов.

Christensen C., Cook S., Hall T. *Marketing Malpractice: The Cause and the Cure*, Harvard Business Review, December 2005.

Clark K. B., Fujimoto T. *Product Development Performance: Strategy, Organization, and Management in the World Auto Industry*, Harvard Business School Press, 1991.

Книга демонстрирует, что ключом к целостности продукта (product integrity) является информационный поток (с рынка в коллектив разработчиков, а затем к каждому члену коллектива).

Clark M. *Pragmatic Project Automation: How to Build, Deploy and Monitor Java Applications*. Pragmatic Press, 2004.

Защита от ошибок в бережливом производстве осуществляется через автоматизацию. Как это делается на практике, продемонстрирует данная книга.

Cohn M. *Agile Estimating and Planning*, Addison-Wesley, 2005.

Данная книга для тех, кто интересуется, как осуществляется планирование в гибком производстве.

Cohn M. *User Stories Applied*, Addison-Wesley, 2004.

Подробности о том, как использовать "истории" для осуществления процесса разработки.

Collins J. C. *Good to Great*, Harper Business, *2001*.

Пять основных условий, позволяющих создать успешную организацию. Очень полезная книга!

Constantine L., Lockwood L. *Software for Use*, Addison-Wesley, 1999.

Авторитетная книга об организации процесса разработки, ориентированного на нужды конечного пользователя.

Cusumano M. A., Selby R. W. *Microsoft Secrets, paperback edition*, Simon & Schuster, 1998.

Книга (впервые опубликованная в 1995 г.) о том, как компания Microsoft разрабатывала программное обеспечение в 90-х годах: небольшие коллективы, делимая архитектура, ежедневная интеграция, допускающий повторное использование программный код.

de Geus A. *The Living Company: Habits for Survival in a Turbulent Business Environment*, Harvard Business School Press, 1997, 2002.

Deming W. E. *Out of the Crisis*, MIT Press, 2000.

Denne M., Cleland-Huang J. *Software by Numbers: Low-Risk, High Return Development*, Prentice Hall, 2004.

Данная книга продемонстрирует читателю (на основе экономического анализа) преимущества поэтапного ввода в эксплуатацию (у заказчика) созданного программного

обеспечения. Содержит убедительные обоснования для использования гибкой (agile) методологии разработки ПО.

Drucker P. *Management Challenges for the 21st Century*, Harper Business, 1999.
(Книга Питера Друкера *Задачи менеджмента в 21 веке* была переведена на русский язык и выпущена издательством "Диалектика" в 2007 году.)

Evans E. *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Addison-Wesley, 2004.
Чрезвычайно полезная книга, демонстрирующая, что понимание среды заказчика является решающим условием создания успешного программного обеспечения.

Evans P., Wolf B. *Collaboration Rules*, Harvard Business Review, July–August 2005.

Feathers M. *Working Effectively with Legacy Code*, Prentice Hall, 2005.
Если вам приходится иметь дело с унаследованным программным кодом, эта книга для вас!

Ferdows K., Lewis M. A., Machuca J. A. D. *Rapid-Fire Fulfillment*, Harvard Business Review, November 2004.

Fielding R. T. *Shared Leadership in the Apache Project* Communications of the ACM, April 1999.

Fowler M. *Strangler Application*, `www.martinfowler.com/bliki/StranglerApplication.html`.

Fowler M., et al *Refactoring*, Addison-Wesley, 1999.
Классическая книга о рефакторинге.

Freedman D. H. *Corps Business: The 30 Management Principles of the US Marines*, Harper Business, 2000.
Замечательная книга о руководстве, менеджменте и о том, как выявлять и использовать способности "рядовых" рабочих.

George M., Wilson S. *Conquering Complexity in Your Business: How Wal-Mart, Toyota, and Other Top Companies Are Breaking Through the Ceiling on Profits and Growth*, McGraw-Hill, 2004.

Gilbreth F. B. Jr., Carey E. G. *Belles on Their Toes*, T.Y. Crowell Co., 1950.

Gilbreth F. B. Jr., Carey E. G. *Cheaper by the Dozen*, T.Y. Crowell Co., 1948.

Goldratt E. *Beyond the Goal: Eliyahu Goldratt Speaks on the Theory of Constraints (Your Coach in a Box)*, Coach Series, Gildan Audio, 2005.

Goldratt E., Cox J. *The Goal, 2nd Revised Edition*, North River Press, 1992.
Если вы не прочли этот "бизнес-роман" (впервые опубликованный в 1984 году) о теории ограничений (применительно к производству), значит, вы не заметили классическое произведение.

Hamel G. *Management Innovation*, Harvard Business Review, February 2006.

Hammonds K. H. *How Google Grows...and Grows...and Grows*, Fast Company, Issue 69, April 2003.

Huntzinger J. *The Roots of Lean—Training within Industry: The Origin of Japanese Management and Kaizen*, `http://artoflean.com/documents/pdfs/Handout_2c_Roots_of_Lean _and_TWI.pdf`.

Иные документы (помимо указанного), использованные для создания TWI, можно также найти на том же сайте.

Kano N., Seraku, N., Takahashi F., Tsuji S. *Attractive Quality and Must-Be Quality, Hinshitsu*, Quality (April 1984.), p. 39–48.

Quality — журнал японского общества управления качеством (Japanese Society for Quality Control)

Katzenbach J. R., Smith D. K. *The Wisdom of Teams*, Harvard Business School Press, 1992.

Kennedy M. *Product Development for the Lean Enterprise: Why Toyota's System Is Four Times More Productive, and How You Can Implement It*, Oaklea Press, 2003.

Книга для тех, кто интересуется, как компания, где изобрели бережливый подход (речь идет о компании Toyota), осуществляет разработку продуктов.

Kerievsky J. *Refactoring to Patterns*, Addison-Wesley, 2005.

Лучшее практическое руководство по рефакторингу. (Книга Джошуа А. Кериевски *Рефакторинг с использованием шаблонов* была переведена на русский язык и выпущена ИД "Вильямс" в 2008 году.)

Liker J., Meier D. *The Toyota Way Fieldbook*, McGraw Hill, 2006.

Liker J., Morgan J. *The Toyota Product Development System: Integrating People, Process, and Technology*, Productivity Press, 2006.

Liker J. *The Toyota Way: 14 Management Principles from the World's Greatest Manufacturer*, McGraw Hill, 2004.

Только две главы, посвященные созданию моделей автомобилей Lexus и Prius, оправдывают цену всей книги.

Lohmeyer D., Pogreb S., Robinson S. *Who's Accountable for IT?* McKinsey Quarterly, December 7, 2004.

MacCormack A. *Product-Development Practices That Work: How Internet Companies Build Software*, MIT Sloan Management Review, Vol. 40, № 2, 2001.

MacGibbon S. P., Schumacher J. R., Tinaikar R. S. *When IT's Customers Are External*, McKinsey Quarterly, Q1 2006.

Marick B. *Agile Testing Directions*, `www.testing.com/cgi-bin/blog/2003/08/21-agile-testing-project-1`.

Marick B. *Approaches to Legacy Code*, `www.testing.com/cgi%2Dbin/blog/2005/05/11`.

Martin R. C. *Agile Software Development: Principles, Patterns, and Practices*, Prentice Hall, 2002.

Содержит глубокие знания о применении объектно-ориентированных концепций.

McAfee A. *Do You Have Too Much IT?* MIT-Sloan Management Review, Spring 2004.

McAfee A. *Enterprise 2.0: The Dawn of Emergent Collaboration*, MIT Sloan Management Review, Spring 2006.

Meszaros G., Bohnet R. *Test-Driven Porting*, Agile 2005 Experience Report, `http://agile2005.org/XR23.pdf`.

Miller L. *Case Study of Customer Input for a Successful Product*, Agile 2005 Experience Report, http://agile2005.org/XR19.pdf.

Mira M., Westblom U., Forssander S., Andersson G., Medin M., Ebarasi S., Fahlgren T., Johansson S. E., Tornquist S., Holmgren M. *Taxonomy of Problem Management Activities*, proceedings of thr 5th European Conference on Software Maintenance and Reengineering, March 200, pp. 1–10.

Moore G. A. *Crossing the Chasm, revised edition*, Harper Business, 2002;
Данная книга впервые была опубликована в 1991 г.
(Книга Джеффри А. Мура *Преодоление пропасти: маркетинг и продажа хайтек-товаров массовому потребителю* была переведена на русский язык и выпущена ИД "Вильямс" в 2006 году.)

Mugridge R., Cunningham W. *Fit for Developing Software: Framework for Integrated Tests*, Prentice Hall, 2005.
Каждый, кому приходится осуществлять автоматизированное приемочное тестирование, включая рефакторинг унаследованного программного кода с помощью приемочных тестов, должен прочитать эту книгу.

Neward T. Mike C. *Pragmatic Project Automation for .NET*, Pragmatic Press, 2007.

Nonaka I., Takeuchi H. *The Knowledge Creating Company: How Japanese Companies Create the Dynamics of Innovation*, Oxford University Press, 1995.

Norwegian Computer Society. *PS2000 Standard Contract—English version*, February, 2006, http://dataforeningen.no/?module=Articles;action=ArticleFolder.publicOpenFolder;ID=1044.

O'Reilly C. A. III, Pfeffer J. *Hidden Value: How Great Companies Achieve Extraordinary Results with Ordinary People*, Harvard Business School Press, 2000.

Ohno T. *Toyota Production System: Beyond Large Scale Production*, Productivity Press, 1988.
Книга, которую обязательно следует прочесть, написанная "отцом" производственной системы Toyota. Легкая в чтении, увлекательная и глубокая.

Poppendieck M., Poppendieck T. *Lean Software Development: An Agile Toolkit*, Addison-Wesley, 2003.
Если в вашей компании ревализуется бережливая инициатива и если вы занимаетесь разработкой программного обеспечения, вам обязательно следует прочесть данную книгу.

Poppendieck M. *Unjust Deserts*, Better Software Magazine, August 2004.
Размышления о вознаграждении за сделанную работу в бережливой и гибкой (lean/agile) организации, занятой разработкой программного обеспечения.

The "Thinking" Production System: TPS as a Winning Strategy for Developing People on the Global Manufacturing Environment, Public Affairs Division, Toyota Motor Corporation, www.toyota.co.jp/en/special/tps/tps.html.

Rapp W. V. Co-Principal Investigator. *Automobiles: Toyota Motor Corporation: Gaining and Sustaining Long-Term Advantage Through Information Technology*. Case Prepared for Columbia Project: Use of Software to Achieve Competitive Advantage by The College of International Relations, Ritsumeikan University Kyoto, Japan, April 2000

Raymond E. S. *The Cathedral and the Bazaar*, www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/, 2000.

Reichheld F. *The Ultimate Question: Driving Good Profits and True Growth*, Harvard Business School Press, 2006.

Reinertsen D. G. *Managing the Design Factory*, The Free Press, New York, 1997.
Замечательная книга о бережливом подходе к созданию продуктов.

Rogers P., Blenko M. *Who Has the D? How Clear Decision Roles Enhance Organizational Performance*, Harvard Business Review, January 2006.

Sabbagh K. *21st Century Jet: The Building of the 777*, produced by Skyscraper Productions for KCTS/Seattle and Channel 4 London. 1995.

Sapolsky H. *The Polaris System Development: Bureaucratic and Programmatic Success in Government*, Harvard University Press, 1972.

Schwaber K., Beedle M. *Agile Software Development with SCRUM*, Prentice Hall, 2001.
Книга-введение в методологию гибкой разработки Scrum, которую обязательно следует прочитать.

Schwaber K. *Agile Project Management with Scrum*, Microsoft Press, 2004.
Книга содержит множество примеров и советов о том, как применять методологию гибкой разработки Scrum.

Shingo S. *Study of Toyota Production System from an Industrial Engineering Viewpoint*, Productivity Press, 1981.

Shore J. *Quality with a Name*, www.jamesshore.com/Articles/Quality-With-a-Name.html

Smalley A. *TPS vs. Lean Additional Perspectives*, http://artoflean.com/documents/pdfs/TPS_versus_Lean_additional_perspectives_v3.pdf.

Smalley A. *TPS vs. Lean and the Law of Unintended Consequences*, www.superfactory.com/articles/smalley_tps_vs_lean.htm.

Smith P. G., Reinertsen D. G. *Developing Products in Half the Time, Second Edition*, John Wiley and Sons, 1998.
Данная книга (которая впервые была опубликована в 1991 г.) все еще не потеряла своего значения как классическое издание по ускоренной разработке продуктов.

Sobek D. K. II, *Principles that Shape Product Development Systems: A Toyota-Chrysler Comparison*, University of Michigan, 1997.
Данная диссертация была представлена на соискание ученой степени доктора философии (Industrial and Operations Engineering) в университете Мичигана

Spear S., Bowen H. K. *Decoding the DNA of the Toyota Production System*, Harvard Business Review, September–October 1999.

Stalk G. *Time—The Next Source of Competitive Advantage*, Harvard Business Review, July 1988.

Sutherland J. *Future of Scrum: Parallel Pipelining of Sprints in Complex Projects*, Agile 2005 Proceedings, `http://jeffsutherland.com/scrum/SutherlandFutureOfScrum-Agile2005.pdf`.

Taylor S., Schroeder K. *Inside Intuit*, Harvard Business School Press, 2003.

Toyoda E. *Toyota: Fifty Years in Motion*, Kodansha, 1987.
(Даная книга впервые была опубликована на японском языке в 1985 году.)

Public Affairs Report, Toyota Motor Corporation, October 8, 2003, `www.toyota.co.jp/en/special/tps/tps.html`.

Ulrich D., Kerr S., Ashkenas R. *The GE Workout*, McGraw-Hill, 2002.
Данное здание можно использовать в качестве пособия для организации кайдзен-мероприятий

Van Schooenderwoert N. *Embedded Agile Project by the Numbers with Newbies*, Proceedings, Agile 2006 Conference, Minneapolis, *July 2006*.

Van Schooenderwoert N., Morsicato R. *Taming the Embedded Tiger—Agile Test Techniques for Embedded Software*, Proceedings, Agile Development Conference, Salt Lake City, June 2004.

Vasilash G. S. *Engaging the ES 300*, Automotive Design and Production, September 2001.

Vise D. A. *Google's Missing Piece*, Washington Post, February 10, 2005.

Weick K., Sutcliffe K. *Managing the Unexpected: Assuring High Performance in the Age of Complexity*, Jossey-Bass, 2001.
Захватывающая книга, о том, что делает безопасными такие по определению опасные места, как авианосцы, химические предприятия и места аварий.

Womack J., Jones D. T. *Lean Solutions: How Companies and Customers Can Create Value and Wealth Together*, Free Press, 2005.

Womack J., Jones D., Roos D. *The Machine That Changed the World*, Rawson Associates, 1990.

Womack J. P., Jones D. T. *Lean Thinking*, Simon & Schuster, *1996*, *Second Edition*, Free Press, 2003.
Почти в течение десятилетия это лучшая книга, посвященная бережливому подходу. Новое, пересмотренное издание остается классикой.

`www.lean.org`
На сайте содержится огромное количество информации о принципах бережливого мышления.

Предметный указатель

3

3M, 75

B

BMI, 58
Boeing 777, 131

E

eBay, 73

F

Fujitsu, 58

G

Google, 63; 119

H

Honda, 224

I

Intuit, 70

O

Open Source, 73; 199; 211

P

PatientKeeper, 15; 113
PDA, 87; 113
Product Owner, 15; 187

Q

Quicken, 70

S

Scrum, 15
Scrum Product Owner, 144
ScrumMaster, 144
Smart Design, 224

T

Toyoda, 25
Toyota, 26; 224

Z

Zara, 85

А

Алан Мюлелли, 132
Андон, 149
Аутсорсинг, 215

Б

Бережливая разработка продуктов, 34
Бережливая цепь поставок, 34
Бережливое производство, 32
Бережливый подход
    семь принципов, 44
Блочные тесты, 201

В

ВАА, 218
Вариация, 118
Ввод в эксплуатацию, 89
Вложенная синхронизация, 205
Время цикла, 116; 237

Г

Генри Форд, 24
Главный инженер, 72

Д

Диагностические тесты, 203

Ж

Журнал предстоящей работы, 15; 187

З

Закон Литтла, 118

И

Интеллектуальное производство, 224
итеРативный процесс, 186
Итерация, 186

К

Кайдзен-мероприятие, 177
Канбан, 148
Карта потока создания ценности, 98
Кийтиро Тойода, 26
Критическая цепь, 231

М

Менеджер продукта, 187
Множественное проектирование, 165
Модель Кано, 68

Н

Непрерывная интеграция, 204

О

Общее время цикла, 110
Останов производственной линии, 30

П

Партия работы, 123
Приемочные тесты, 202
Программа Polaris, 181
Продукт, 78
Проект, 78
Производственная система Toyota, 25; 26; 159
Производственный цикл, 113

Р

Разработка через тестирование, 49; 201
Релиз-цикл, 123
Реляционные контракты, 220
Рефакторинг, 165; 170

С

Сакичи Тойода, 25
Сигаэо Синго, 27; 90
Система с "вытягиванием", 31; 127
Среднее время цикла, 119

Т

"точно вовремя", 26; 28; 29; 165
Тайити Оно, 26
Теория ограничений, 229
Тестирование качества, 203
Тесты простоты использования, 203
Технологичность, 77

У

У. Эдвардс Деминг, 134
14 заповедей, 135
Унаследованное ПО, 171

Ч

Чемпион, 72; 187

Ш

Шесть сигм, 228

Э

Эйдзи Тойода, 27
Эффективность цикла процесса, 109; 110

ЭФФЕКТИВНАЯ РАБОТА С УНАСЛЕДОВАННЫМ КОДОМ

Майкл К. Физерс

www.williamspublishing.com

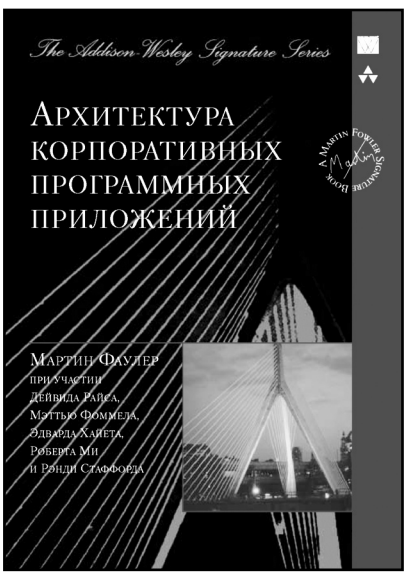
Эта книга посвящена практическим вопросам эффективной работы с унаследованным кодом. В ней освещаются механизмы внесения изменений в унаследованный код, способы переноса его фрагментов в среду тестирования, особенности написания тестов для безопасного изменения и реорганизации кода, приемы точного определения мест для подобных изменений, а также подходы к обращению с унаследованным процедурным кодом. Кроме того, в книге представлены способы разрыва зависимостей для работы с обособленными фрагментами кода и безопасного внесения в них изменений. Книга адресована тем, кто имеет опыт разработки прикладного программного обеспечения и его сопровождения.

ISBN 978-5-8459-1530-6

в продаже

АРХИТЕКТУРА КОРПОРАТИВНЫХ ПРОГРАММНЫХ ПРИЛОЖЕНИЙ

Мартин Фаулер



www.williamspublishing.com

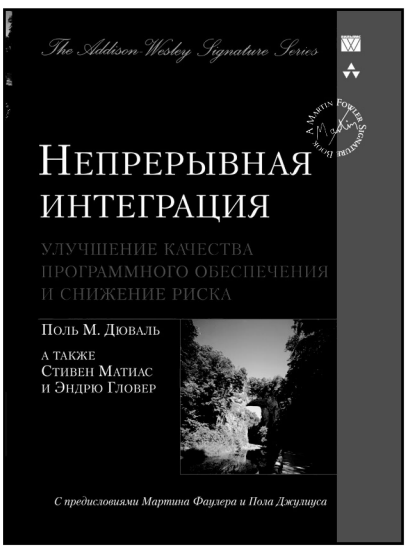
Книга дает ответы на трудные вопросы, с которыми приходится сталкиваться всем разработчикам корпоративных систем. Автор, известный специалист в области объектно-ориентированного программирования, заметил, что с развитием технологий базовые принципы проектирования и решения общих проблем остаются неизменными, и выделил более 40 наиболее употребительных подходов, оформив их в виде типовых решений. Результат перед вами — незаменимое руководство по архитектуре программных систем для любой корпоративной платформы. Это своеобразное учебное пособие поможет вам не только усвоить информацию, но и передать полученные знания окружающим значительно быстрее и эффективнее, чем это удавалось автору. Книга предназначена для программистов, проектировщиков и архитекторов, которые занимаются созданием корпоративных приложений и стремятся повысить качество принимаемых стратегических решений.

ISBN 978-5-8459-0579-6

в продаже

НЕПРЕРЫВНАЯ ИНТЕГРАЦИЯ
УЛУЧШЕНИЕ КАЧЕСТВА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ И СНИЖЕНИЕ РИСКА

Поль М. Дюваль,
Стивен Матиас
и Эндрю Гловер



www.williamspublishing.com

В этой книге рассматриваются некоторые из наиболее типичных процессов разработки программного обеспечения: компиляция кода, определение данных и манипулирование ими в базе данных; осуществление проверки, просмотр кода и в конечном итоге развертывание программного обеспечения. Но главное, в ней описано, как непрерывная интеграция способна снизить риски, которые подстерегают при создании приложений. В системе непрерывной интеграции большинство этих процессов автоматизировано, и они запускаются после каждого изменения разрабатываемого программного обеспечения. В этой книге обсуждаются аспекты автоматизации непрерывной интеграции, большинство предоставляемых ей преимуществ в области повторяемых и склонных к ошибкам процессов.

ISBN 978-5-8459-1408-8 **в продаже**