Чистий КОДЕР

РОБЕРТ С. МАРТІН

Чистий КОДЕР

КОДЕКС ПОВЕДІНКИ ДЛЯ ПРОФЕСІЙНИХ РОЗРОБНИКІВ

ВИДАВНИЧИЙ ДІМ
ФАБУЛА
#PRO

УДК 004.41
М29

Авторизований переклад з англомовного видання під назвою
The Clean Coder: A Code of Conduct for Professional Programmers,
1-ше видання, автора Роберта С. Мартіна, опубліковано Pearson Education, Inc,
що виступає як Addison Wesley Professional,
Copyright © 2011 Pearson Education, Inc.

Authorized translation from the English language edition, entitled The Clean Coder:
A Code of Conduct for Professional Programmers, 1st Edition by Robert C. Martin,
published by Pearson Education, Inc, publishing as Addison Wesley Professional,
Copyright © 2011 Pearson Education, Inc.

*Усі права збережено. Жодної частини цієї книжки не може бути відтворено
або передано в будь-якій формі або будь-якими засобами, електронними
чи механічними, включно з фотокопією, записом чи будь-якою системою
зберігання та пошуку інформації, без письмового дозволу Pearson Education, Inc.*

*All rights reserved. No part of this book may be reproduced or transmitted in any form
or by any means, electronic or mechanical, including photocopying, recording or by any
information storage retrieval system, without permission from Pearson Education, Inc.*

Видання українською мовою опубліковано ТОВ Видавничий дім «Фабула» © 2023
Ukrainian language edition published by Publishing House Fabula LLC, Copyright © 2023

Мартін Роберт С.

М29 Чистий кодер: Кодекс поведінки для професійних розробників / пер. з англ. Г. Якубовська. — Харків : ВД Фабула, 2023. — 256 с.

ISBN 978-617-522-082-5

Роберт Мартін, також відомий як Дядечко Боб,— знакова постать у світі розробки ПЗ, блискучий професіонал, міжнародний консультант, один із тих, хто створював 2001 року всесвітньо відомий Agile-маніфест. Кожна його наступна книжка — джерело безцінного досвіду.

У «Чистому кодері» автор викладає свої очікування від професійного розробника у всіх можливих аспектах: із погляду управлінських взаємодій, тайм-менеджменту, зовнішнього тиску, співпраці в команді та вибору відповідних інструментів. Не оминає він і питань трудової етики. Ви також дізнаєтеся, що навіть гуру програмування далеко не завжди є професіоналами. Натомість Роберт Мартін пропонує читачеві шлях до справжнього розробницького професіоналізму — і робить це надзвичайно цікаво й дотепно.

УДК 004.41

Copyright © 2011 Pearson Education, Inc.
© Г. Якубовська, пер. з англ., 2023
© ВД «Фабула», макет, 2023

ISBN 978-617-522-082-5

ЩО КАЖУТЬ ПРО КНИЖКУ «ЧИСТИЙ КОДЕР»

Дядечко Боб Мартін однозначно піднімає планку своєю останньою книжкою. У ній він викладає свої очікування від професійного програміста з погляду управлінських взаємодій, тайм-менеджменту, тиску, співпраці та вибору відповідних інструментів. Мартін також пояснює, що, окрім навичок TDD і ATDD, кожен розробник, який вважає себе професіоналом, не лише повинен знати, а й дотримуватися цих вимог — аби молода професія розробника програмного забезпечення стабільно зростала й розвивалася.

Маркус Ґертнер,
старший розробник програмного забезпечення в it-agile GmbH
www.it-agile.de
www.shino.de

Деякі технічні книжки надихають і навчають; деякі захоплюють і розважають. І лише в унікальних випадках книжка охоплює всі ці чотири напрямки одночасно. Як на мене, праці Роберта Мартіна завжди поєднують ці напрямки, і «Чистий кодер» — не виняток. Читайте, вивчайте та живіть з уроками цієї книжки, і тоді ви точно зможете вважати себе фахівцем із програмного забезпечення.

Джордж Баллок,
старший розробник програмного забезпечення в Microsoft Corp

Якби ступінь з інформатики вимагала обов'язкового читання після закінчення навчання, то це була би найперша книга із пропонованого переліку. У реальному світі ваш поганий код не зникає, коли закінчується термін, ви не отримуєте високу оцінку за марафонське кодування в ніч перед здачею завдання, і, що найгірше, вам доводиться мати справу з людьми. Отже, гуру кодування не обов'язково є професіоналами. Натомість «Чистий кодер» описує шлях до професіоналізму — і робить це надзвичайно цікаво й дотепно.

Джефф Овербі,
Університет Іллінойсу в Урбана-Шампейн

«Чистий кодер» — це набагато більше, ніж набір правил чи рекомендацій. У цій книжці містяться важко набута мудрість і знання, які зазвичай отримують шляхом багаторічних проб і помилок або працюючи підмайстром у майстра. Якщо ви називаєте себе професіоналом із програмного забезпечення, вам знадобиться ця книжка.

Р. Л. Богетті,
провідний системний дизайнер у Baxter Healthcare
www.RLBogetti.com

Між 1986 і 2000 роками я тісно співпрацював із Джимом Ньюкірком, колегою з *Teradyne, Inc*. Ми з ним поділяли пристрасть до програмування та чистого коду. Ми проводили ночі, вечори та вихідні дні разом, експериментуючи з різними стилями програмування та методами дизайну. Ми постійно обговорювали численні бізнес-ідеї. Зрештою ми разом створили *Object Mentor, Inc*. Я багато чому навчився у Джима, коли ми разом реалізовували наші задуми. Але одним із найважливіших було його ставлення до трудової етики; це було найголовніше, що я намагався перейняти. Джим — справжній професіонал. Я пишаюся тим, що працював із ним, і з повним правом називаю його своїм другом.

ЗМІСТ

Вступне слово 13
Передмова 19
Подяки 23
Про автора 29
Вступ 31

Розділ 1. Професіоналізм **37**
Будьте обережні у своїх проханнях 38
Брати відповідальність 38
По-перше, не зашкодьте 41
Трудова етика 47
Бібліографія 53

Розділ 2. Казати «ні» 55
Протилежні ролі 58
Високі ставки 62
Бути командним гравцем 63
Вартість вимовленого «так» 69
Неможливий код 76

Розділ 3. Казати «так» **79**
Мова зобов'язань 81
Як навчитися говорити «так» 86
Висновки 90

Розділ 4. Написання коду 91
Підготовка 93
Зона потоку 96
Блокування 99
Відлагодження 101
Спостерігайте за собою 105
Запізнення 106
Допомога 109
Бібліографія 112

Розділ 5. Розробка, орієнтована на тестування 113
Вердикт винесено 115
Три закони TDD 116
Чим TDD не є 120
Бібліографія 121

Розділ 6. Практика 123
Додзьо кодування 127
Розширюємо свій досвід 131
Висновок 132
Бібліографія 132

Розділ 7. Приймальні тести 133
Вимоги до комунікації 134
Приймальні тести 139
Висновки 151

Розділ 8. Стратегії тестування 153
Контроль якості не повинен нічого виявити 154
Піраміда автоматизації тестування 155
Висновки 160
Бібліографія 160

Розділ 9. Тайм-менеджмент 161
Зустрічі 163
Мана концентрації 168
Розподіл часу та помідори 170
Уникання 172
Безвихідь 172
Бруд, болото та інший безлад 173
Висновки 174

Розділ 10. Оцінки 175
Що таке оцінка? 178
PERT 182
Оцінювання завдань 185
Закон великих чисел 188
Висновок 188
Бібліографія 189

Розділ 11. Тиск 191
Уникання тиску 194
Як скористатися тиском 196
Висновки 197

Розділ 12. Співпраця 199
Програмісти проти людей 201
Мозочок 207
Висновки 208

Розділ 13. Команди і проєкти 209
Чи можна змішувати? 210
Висновки 214
Бібліографія 214

Розділ 14. Наставництво, навчання і майстерність 215
Диплом — і нічого більше 216
Наставництво 217
Учнівство 223
Майстерність 227
Висновки 228

Додаток А. Налаштування 229
Інструменти 231
Управління вихідним кодом 232
IDE/Редактор 237
Відстеження завдань 239
Безперервне складання 241
Інструменти модульного тестування 241
Інструменти тестування компонентів 243
Інструменти інтеграційного тестування 244
UML/MDA 245
Висновки 248

Додаток В. Покажчик 249

ВСТУПНЕ СЛОВО

Якщо ви все ж таки взяли в руки цю книжку, то я вважаю, що ви працюєте в галузі програмного забезпечення. Чудово, і я також! Оскільки ж я спромігся привернути вашу увагу, то дозвольте мені розповісти, чому я сам узяв би цю книжку до рук.

Усе почалося нещодавно. Завіса підіймається, світло, камера, Чарлі…

Кілька років тому я працював у середній за розмірами корпорації з продажу суворо регульованої продукції. Ви напевне знаєте цей тип компаній: вона цілком займала триповерхову будівлю, у директорів і вище були особисті кабінети, і близько тижня витрачалося на те, щоби зібрати потрібних людей для зустрічі в одному офісі.

Ми працювали на дуже конкурентному ринку, коли уряд представив новий продукт. Несподівано з'явилося нове коло потенційних клієнтів, і все, що нам потрібно було зробити — це змусити їх купити цей продукт. Це означало, що ми мали до певного терміну подати заяву до федерального уряду, пройти аудит і буквально наступного дня вийти на ринок.

Наш відділ маркетингу знову і знову наголошував на важливості цих дат. Одна помилка — й уряд не дозволить нам вийти на ринок протягом року, а якщо клієнти не зможуть зареєструватися в перший же день, то всі вони підпишуть контракти з кимось іншим, і ми залишимося на узбіччі.

Це було того типу середовище, у якому одні люди скаржаться і ремствують, а інші вважають, що «великий тиск народжує діаманти».

Я був технічним менеджером проєкту — мене підвищили із відділу розробки, і відповідав за запуск вебсайту в належний день, аби

потенційні клієнти могли завантажувати інформацію та, найважливіше, реєстраційні форми. Моїм напарником був бізнес-менеджер проєкту — далі я називатиму його Джо. Роль Джо полягала у співпраці з іншою стороною: він мав займатися продажами, маркетингом та нетехнічними вимогами. Він також належав до тих хлопців, які полюбляють коментарі типу «великий тиск народжує діаманти».

Якщо ви досить довго працювали в корпоративній Америці, то, ймовірно, бачили, як тицяють пальцем, звинувачують та відчувають відразу до роботи, що цілком природно. Наша компанія знайшла цікаве розв'язання цієї проблеми з Джо і мною.

Наче у Бетмена з Робіном, наша робота полягала в тому, щоби довести справу до кінця. Я щодня зустрічався з технічною командою, ми щодня складали розклад, з'ясовували оптимальний шлях, а потім усували всі можливі перешкоди на цьому шляху. Якщо комусь було потрібне програмне забезпечення, ми йшли за ним. Якщо вони «намагалися» налаштувати брандмауер, але «Боже, а як же моя обідня перерва?» — і ми приносили їм обід. Якщо хтось висловлював бажання попрацювати над нашою задачею щодо налаштувань конфігурації, але мав інші пріоритети, ми з Джо вирушали на переговори з їхнім безпосереднім керівником.

Потім із менеджером.

Потім із директором.

Інакше кажучи, ми робили все можливе.

Сказати, що ми ламали стільці і несамовито галасували, було би перебільшенням, але ми використовували всі методи, наявні в нашому арсеналі, щоби досягти мети, попутно винайшли кілька нових підходів і зробили це з дотриманням етичних норм, чим я пишаюся досі.

Я думав про себе як про члена команди, який не гребує написанням SQL-виразу або роботою з кодування. У той час і про Джо я думав так само, як і про інших членів команди, а не як про когось, хто стоїть над нею.

Згодом я зрозумів, що Джо не поділяє цієї думки. То був дуже сумний день для мене.

Була п'ятниця, 13.00; вебсайт мав запрацювати наступного понеділка вранці.

Ми завершили. *DONE*. Усі системи були готові, ми також перебували у повній готовності. Я зібрав усю технічну команду на фінальну нараду, залишалося тільки натиснути кнопку. Разом із нами були не лише технічні спеціалісти, а й фахівці з маркетингу та власники продуктів.

Ми пишалися собою. Це був чудовий момент.

Потім увійшов Джо.

Він сказав щось на кшталт: «У мене погані новини. У юридичного відділу немає готових реєстраційних форм, тому поки що ми не можемо розпочати роботу».

Це не було надскладною проблемою; у нас траплялися затримки з тією чи іншою задачкою протягом усього проєкту. Я був готовий до чогось подібного, і моя відповідь була: «Добре, партнере, давай зробимо це зараз. Юридичний відділ міститься на третьому поверсі, хіба не так?»

Але подальші події здавалися дуже дивними.

Замість того щоби погодитися, Джо запитав: «Про що ти говориш, Метт?»

Я сказав: «Ти добре знаєш. Це звичайна річ. Ідеться про чотири PDF-файли, чи не так? Вони існують, юристи просто мають їх схвалити. То давай просто зависнемо в їхніх кабінетах, деякий час подивимося на них зловісним поглядом і нарешті доведемо справу до кінця».

Джо натомість заявив: «Ні, ми просто випустимо продукт наприкінці наступного тижня. Нічого жахливого».

Ви, мабуть, можете передбачити решту розмови. Звучало це приблизно так:

Метт: Але чому? Вони можуть зробити це години за дві.

Джо: Це може зажадати більше часу.

Метт: Але в них попереду весь уік-енд. Купа часу. Давай зробимо це!

Джо: Метт, ці люди — професіонали. Ми не можемо тупо дивитися на них і наполягати, щоби вони пожертвували своїм особистим відпочинком заради нашого маленького проєкту.

Метт *(пауза)*: Джо! Як ти думаєш, що ми робили разом із командою інженерів протягом останніх чотирьох місяців?

Джо: Так, але ж вони професіонали…

Пауза.

Зітхання.

То що зробив Джо? Як ви гадаєте?

Тоді я вважав, що саме технічний персонал є професіоналами в найкращому сенсі цього слова.

Але, згадуючи про це зараз, я вже не є настільки впевненим.

Давайте подивимося на цю взаємодію Бетмена й Робіна ще раз, з іншої точки зору.

Я вважав, що скеровую команду до досягнення найкращих результатів, але Джо, схоже, грав у гру з неявним припущенням, що технічний персонал є його суперником. Подумайте про це: навіщо тоді було бігати, ламати стільці та закликати людей до самовіддачі? Хіба ми не повинні були запитати їх, коли вони будуть готові, отримати від них певну відповідь, повірити в отриману відповідь і надихатися цією вірою?

Безумовно, як професіонали — могли… і водночас не могли. Джо не довіряв нашим відповідям і почувався цілком комфортно, керуючи командою технічних спеціалістів на мікрорівні, і водночас із якоїсь причини найбільше довіряв команді юристів і не мав бажання керувати ними на мікрорівні.

Що це все означає?

Так чи інакше, команда юристів продемонструвала справжній професіоналізм на відміну від технічної команди.

Так чи інакше, ця група переконала Джо, що їм не потрібна нянька, що вони не граються в ігри і що до них треба ставитися як до однолітків, яких поважають.

Ні, я не думаю, що це стосується будь-яких дипломів та атестатів, котрі вивішують на стінах кабінетів, або кількох додаткових років навчання в коледжі, хоча ті роки в коледжі могли включати помітну частину прихованого засвоєння правил соціальної поведінки.

Після того дня, що залишився в далекому минулому, я поставив собі питання: як мусить змінитись технічна професія, щоби мене вважали професіоналом.

Так, у мене було кілька ідей. Я часом вів блог, багато читав, мені вдалося покращити власну ситуацію на роботі і допомогти ще кільком людям. Проте я не мав жодної книжки, у якій був би викладений чіткий та зрозумілий план змін.

Але одного разу, цілком несподівано, мені запропонували написати рецензію на чернетку книжки — тієї самої, яку ви зараз тримаєте в руках.

Ця книжка крок за кроком розповіла мені, як саме представити себе і взаємодіяти з іншими людьми як професіонал. І це були не банальні кліше, не порожні міркування на аркушах паперу, а конкретні кроки до того, що варто зробити і як це зробити.

У деяких випадках приклади були наведені з дослівною точністю, і деякі із цих прикладів містили відповіді, контраргументи, пояснення і поради, що треба робити, коли інша людина намагається «просто ігнорувати вас».

Гей, погляньте — це знову той самий Джо, але цього разу сцена матиме геть інакший вигляд.

Замість того щоб ухилятися від зобов'язань, технічний персонал ретельно їх виконує. Замість того щоб ухилятися від оцінок або дозволити комусь іншому займатися плануванням (а потім нарікати на це), технічна команда самоорганізується і бере на себе реальні зобов'язання.

А тепер уявіть, що співробітники дійсно працюють спільно. Коли розробники заходять у глухий кут, вони беруть слухавку, і системний адміністратор береться до роботи. Коли Джо приходить перевірити, чи виконується задача, йому не потрібно «піднімати пил» — він бачить, що адміністратор бази даних ретельно працює, а не просто переглядає вебсторінки в Google. Також й оцінки часу, необхідного для виконання задач, що він отримує від колег, здаються абсолютно реальними, і у Джо не виникає відчуття, що проєкт виконується десь у проміжку між обідом і перевіркою електронної пошти. Усі хитрощі та спроби маніпулювати розкладом не супроводжуються словами: «Ми постараємось», натомість він чує: «Це наш обов'язок; якщо хочеш поставити якісь власні цілі, не соромся».

Я підозрюю, що через деякий час Джо почне думати про технічну команду як про професіоналів. І він матиме рацію.

Кроки, необхідні для перетворення вашого статусу із фахівця на професіонала, ви знайдете в розділах книжки Дядечка Боба.

Ласкаво просимо на наступний щабель у вашій кар'єрі! Сподіваюся, що вам це сподобається.

Метью Хойзер,
натураліст програмних процесів

ПЕРЕДМОВА

28 січня 1986 року об 11:39 ранку за східним часом на 73 секунді після запуску і на висоті 48 тисяч футів космічний шатл «Челленджер» був розірваний на шматки через несправність правого твердопаливного ракетного прискорювача (SRB). Семеро астронавтів, серед яких була вчителька середньої школи Кріста МакОліфф, загинули. Вираз обличчя матері Крісти, коли вона спостерігала за смертю своєї дочки за дев'ять миль над її головою, переслідує мене донині.

«Челленджер» розпався через те, що гарячі вихлопні гази з несправного SRB просочилися з-поміж секцій його корпусу, охопивши корпус зовнішнього паливного бака. Це призвело до руйнування хвостового кріплення правого твердопаливного прискорювача і навантажених структур зовнішнього паливного бака, на якому власне і трималась уся конструкція. Елементи комплексу стали зміщувати-

ся відносно один одного, що призвело до пошкодження паливного бака, детонації компонентів палива (рідких водню та кисню) та руйнування конструкції в результаті величезних аеродинамічних перенавантажень. Аберантні вектори сили змусили весь корабель, швидкість якого перевищувала 1,5 Маха, обертатися, й аеродинамічні напруження швидко розірвали його корпус на шматки.

Між круглими секціями SRB були розташовані два ущільнювальні кільця із синтетичного каучуку. Коли секції скріплялися болтами, ущільнювальні кільця стискалися, затуляючи зазор, через який могли би проникнути вихлопні гази.

Але ввечері перед запуском шатла температура на стартовому майданчику впала до 17° за Фаренгейтом (–8,3 °C), що було на 23 градуси нижче за мінімальну задану температуру для ущільнювальних кілець і на 33 градуси нижче за будь-який попередній запуск. Унаслідок цього каучукові ущільнювальні кільця стали занадто жорсткими, щоб належним чином блокувати гарячі гази. Після запуску SRB виник стрибок тиску, оскільки гарячі гази швидко накопичувалися. Секції прискорювача роздулися і зсунулися назовні, послабивши тиск на ущільнювальні кільця. Надмірна жорсткість ущільнювальних кілець більше не дозволяла їм утримувати тиск, тому частина розжарених газів просочилася назовні й випаровувала ущільнювальні кільця у місті кріплення секції прискорювача до зовнішнього паливного бака.

Інженери компанії *Morton Thiokol*, які розробили SRB, знали, що існують проблеми з ущільнювальними кільцями, і повідомили про це менеджерам *Morton Thiokol* і NASA ще сім років тому. Дійсно, ущільнювальні кільця під час попередніх запусків отримували подібні пошкодження, але не настільки, щоби призвести до катастрофи. Запуск при найбільш низькій температурі спричинив кільцям найбільшої шкоди. Інженери розробили певний підхід до розв'язання цієї проблеми, але її усунення було відкладено.

Інженери вже тоді підозрювали, що ущільнювальні кільця на холоді втрачають еластичність. Вони також знали, що температура для запуску «Челленджера» того разу була нижчою за будь-який попередній запуск, і значно нижчою за «червону лінію». Інакше кажучи, інженери знали, що ризик занадто великий, і діяли, виходячи із цього знання. Вони писали меморандуми, більш схожі на сигнали за допомогою величезних червоних прапорів. Вони наполегливо закликали *Thiokol* і менеджерів NASA не здійснювати запуск. На одинадцятій годині наради, що відбулася за кілька годин до запуску, ті самі інженери представили свої уточнені дані. Вони благали, вмовляли і протестували. Але в кінцевому рахунку менеджери їх проігнорували.

Коли настав час запуску, деякі з інженерів відмовилися дивитися трансляцію, бо боялися вибуху на майданчику. Але коли «Челленджер» велично піднявся в небо, вони почали розслаблятися. За секунди до катастрофи, коли вони спостерігали, як корабель перевищує швидкість в 1 Мах, один із них сказав, що, здається, вони таки «ухилилися від кулі».

Незважаючи на всі протести, службові записки та заклики інженерів, менеджери вирішили, що все знають краще. Їм здавалося, що інженери перестаралися. Вони не довіряли даним технічних фахівців та їхнім висновкам. До того ж, запуск здійснювався під величезним фінансовим і політичним тиском. Тому менеджери просто *сподівалися*, що все обійдеться.

Ці менеджери були не дурнями — вони були злочинцями. Того холодного ранку життя семи чудових чоловіків і жінок, а також надії всього покоління, яке прагнуло здійснювати космічні подорожі, були знищені, бо ці керівники поставили свої страхи, надії та інтуїтивні сподівання вище за висновки експертів. Вони ухвалили рішення, на яке не мали права, знехтувавши авторитетом людей, які були насправді *обізнаними*,— інженерів.

Але як бути з інженерами? Звичайно, інженери зробили те, що й повинні були зробити. Вони повідомили своїх менеджерів і наполегливо відстоювали свою позицію. Вони скористалися відповідними каналами і використали всі необхідні протоколи. Вони робили все, що могли, *всередині* системи — і все одно менеджери їх здолали. Тож, здавалося б, інженери могли не відчувати провини за те, що трапилося.

Але інколи я ставлю собі питання: чи не переслідував когось із них уві сні образ матері Крісті МакОліфф і чи не виникало в них бажання зателефонувати Дену Разеру[1]?

[1] Ден Разер — відомий американський тележурналіст, ведучий вечірніх новин на CBS. Багато разів у своїх коментарях саркастично дорікав NASA щодо «невиправданих затримок» із черговими стартами космічних човників за програмою «Space Shuttle».— *Прим. пер.*

Про цю книжку

Ця книжка — про професіоналізм у сфері програмного забезпечення. Вона містить багато дієвих порад і відповідей на наступні запитання:

- Ким є професіонал у сфері програмного забезпечення?
- Як поводиться професіонал?
- Як професіонал справляється з конфліктами, надщільними графіками роботи та нерозумними керівниками?
- Коли та як професіонал має сказати «ні»?
- Як професіонал справляється з тиском?

Але поряд із прагматичними порадами у цій книжці ви знайдете позицію, що з часом стає все більш нагальною. Це позиція чесності, честі, самоповаги та гордості. Це готовність взяти на себе величезну відповідальність — бути майстром та інженером своєї справи. Ця відповідальність включає в себе гарне та чисте виконання роботи, зважене спілкування та чіткі оцінки. А також керування своїм часом та ухвалення непростих рішень щодо ризиків та винагороди.

Але ця відповідальність включає ще одну річ — найважчу. Як інженер ви маєте глибокі знання щодо своїх систем та проєктів, яких не може мати жоден менеджер. Із цим знанням щільно пов'язана необхідність *відповідально діяти*.

Бібліографія

McConnell, Malcolm. *Challenger 'A Major Malfunction'*. NY: Simon & Schuster, 1987.

"Space Shuttle Challenger disaster".— http://en.wikipedia.org/wiki/Space_Shuttle_Challenger_disaster

ПОДЯКИ

Моя кар'єра є серією співпраць і комбінацій. Хоча в мене було багато особистих мрій і прагнень, мені здавалося, що я завжди знайду, із ким ними поділитися. У цьому сенсі я відчуваю себе трохи ситхом[1]: «Завжди є двоє».

Перше співробітництво, котре я міг би вважати професійним, було із Джоном Марчезе у віці 13 років. Ми разом планували створювати комп'ютери. Я був мізком, а він — м'язами. Я показував йому, куди припаяти дріт, і він його припаював. Я казав йому, де встановити реле, і він його встановлював. Нам було дуже весело, і ми витратили на цю забавку сотні годин. Таким чином ми побудували чимало об'єктів з реле, кнопками, підсвічуванням, навіть із телетайпами! Звісно, ці штуки не робили нічого корисного, але виглядали дуже вражаюче, і ми дуже наполегливо над ними працювали. До Джона: мої подяки!

Під час навчання у середній школі я познайомився із Тімом Конрадом — це було на уроці німецької мови. Тім був дуже розумний. Коли ми з ним об'єдналися, щоби створити знову ж таки комп'ютер, то вже Тім був мізками, а я — м'язами. Він навчив мене азам електроніки і познайомив із PDP-8. Ми з ним збудували працюючий електронний 18-розрядний двійковий калькулятор, що міг додавати, віднімати, множити і ділити. На це ми витратили вихідні дні протягом року і всі весняні, літні та різдвяні канікули. Ми шалено працювали, і зрештою все почало діяти дуже добре. Тіме, дякую!

Ми з Тімом також навчилися програмувати комп'ютери. Це було нелегко зробити 1968 року, але нам вдалося. Отримавши книжки про асемблер PDP-8, Fortran, Cobol та PL/1, ми їх буквально ковтали. Ми

[1] Ситхи (англ. *Sith*) — вигадані персонажі із всесвіту «Зоряних війн».— *Прим. перекл.*

писали програми, що не мали жодних шансів на виконання, бо ми не мали доступу до комп'ютера. Але ми все одно написали їх із щирої любові до цієї справи.

У старшій школі навчальна програма з інформатики викладалася на другому курсі. Там був телетайп ASR-33, підключений до модема з комутованим доступом на 110 бод, а в Іллінойському технологічному інституті створили обліковий запис у системі розподілу часу комп'ютера Univac 1108. Ми з Тімом одразу стали фактичними операторами цього пристрою. Ніхто інший не міг навіть наблизитися до нього.

Модем підключали, піднявши слухавку й набравши номер. Коли у слухавці чувся вереск модема-відповідача, треба було натиснути кнопку «Orig» на телетайпі, завдяки чому вихідний модем видавав власний вереск. Далі ви клали слухавку, і з'єднання для передачі даних починало працювати.

Той телефон мав замок на циферблаті. Ключ від нього тримали в себе викладачі. Але це не мало значення, бо ми дізналися, що можна набрати телефонний номер (будь-який), швидко натискаючи гачок перемикача телефону відповідна до номера, що набирається, кількість разів. Я був ударником у рок-гурті, тому мав доволі непогані рефлекси і міг зателефонувати до заблокованого модема менш, ніж за 10 секунд.

У нашій комп'ютерній лабораторії було два телетайпи. Один працював онлайн, а інший — автономно. Обидва застосовувалися студентами для написання програм. Студенти друкували свої програми на телетайпах із підключеним перфоратором. Кожне натискання клавіші залишало отвір на паперовій стрічці. Програми створювалися на IITran, надзвичайно потужній інтерпретованій мові.

Студенти часто залишали ті паперові стрічки в кошику біля телетайпів. Після школи ми з Тімом підключали комп'ютер (звісно ж, натискаючи гачок перемикача), завантажували стрічки в пакетну систему IITran, а потім кидали слухавку. При 10 символах за секунду це була не дуже швидка процедура. Приблизно за годину ми передзвонювали і отримували роздруківки, знову ж таки зі швидкістю 10 символів за секунду. Телетайп не розділяв роздруківки студентів, видаляючи сторінки. Він просто друкував їх підряд, тому ми розрізали їх ножицями, скріплювали скріпками і клали у вихідний кошик.

Ми з Тімом були господарями і богами цього процесу. Навіть викладачі залишали нас самих, коли ми перебували в тій кімнаті. Ми робили свою справу, і вони це знали. Нас ніколи не просили це робити, ніколи не казали нам, що ми можемо і чого не можемо робити. Однак нам так і не дали ключа від телефонного апарата. Щойно ми потрапляли туди, викладачі залишали приміщення — тобто нам дали дуже довгий повідець. Моїм викладачам математики — містеру Макдерміту, містеру Фогелю і містеру Робієну — дякую!

Потім, після виконання домашніх завдань, ми починали гратися. Ми писали програму за програмою, реалізуючи безліч божевільних і дивних речей. Зокрема, ми написали програми, що зображували кола та параболи в ASCII на телетайпі. Ми написали програми випадкового блукання та генератори випадкових слів. Ми обчислили 50 факторіалів до останньої цифри. Ми годинами винаходили програми для написання, а потім випробовували їх на практиці.

Через два роки Тім, наш приятель Річард Ллойд і я були найняті розробниками в *ASC Tabulating* у Лейк-Блафф (штат Іллінойс). Нам із Тімом тоді ледве виповнилося по 18 років. Ми вирішили, що навчання в коледжі — марна трата часу, і що ми мусимо негайно розпочати наші кар'єри. Там ми зустріли Білла Хорі, Френка Райдера, Великого Джима Карліна і Джона Міллера. Саме вони надали деяким молодим людям можливість дізнатися, що таке насправді професійне програмування. Це був не лише позитивний і не лише негативний досвід. Але він, безперечно, був пізнавальним. Усім їм, а також Річарду, який був справжнім каталізатором і керував більшою частиною цього процесу, дякую!

Після того як у віці 20 років я залишив роботу в *ASC Tabulating* і занепав духом, я працював майстром із ремонту газонокосарок у крихітній майстерні мого зведеного брата. Однак це виходило в мене вкрай погано, тому і йому довелося мене звільнити. Дякую, Вес! За рік по тому я почав працювати в *Outboard Marine Corporation*. На той час я вже був одружений, і моя дружина народила дитину. Звідти мене теж звільнили. Дякую, Джон, Ральф і Том!

Потім я пішов працювати в *Teradyne*, де зустрів Расса Ешдауна, Кена Фіндера, Боба Копіторна, Чака Студі і CK Srithran (зараз — Кріс Аєр). Кен був моїм босом, Чак і CK були моїми приятелями. Я багато чого навчився в них. Дякую, хлопці!

Потім був Майк Кер'ю. У *Teradyne* ми з ним склали вельми динамічний дует і разом написали кілька систем. Якщо ви хотіли щось зробити, і зробити це швидко, ви мали звертатися до Боба і Майка. Нам було дуже весело разом. Дякую, Майку!

Джеррі Фіцпатрик також працював у *Teradyne*. Ми познайомилися, коли разом грали в *Dungeons & Dragons*, і швидко подружилися. Ми написали програмне забезпечення для «Commodore 64» з метою підтримки користувачів *D&D*. Ми також розпочали новий проєкт у *Teradyne* під назвою «Електронний реєстратор». Ми працювали разом кілька років, і Джеррі став для мене й досі залишається вірним другом. Дякую, Джеррі!

Я провів рік в Англії, працюючи на *Teradyne*. Там я працював в одній команді з Майком Кергозу. Ми з ним разом будували численні плани, хоча більшість із них стосувалися велосипедів та пабів. Але він був відданим розробником, дуже зосередженим на якості та дисципліні (хоча, можливо, він і не погодився би у цьому зі мною). Дякую, Майку!

Повернувшись з Англії 1987 року, я почав складати подальші плани разом із Джимом Ньюкірком. Ми обидва залишили *Teradyne* (із різницею в декілька місяців) і приєдналися до стартапу, що мав назву *Clear Communications*. Ми провели там кілька років разом, намагаючись заробити мільйони, які так і не з'явилися. Але нас це не розчарувало. Зрештою, ми разом заснували *Object Mentor*. Дякую, Джиме!

Джим — найбільш безпосередня, найбільш дисциплінована і зосереджена людина, із якою мені доводилося працювати. Він навчив мене стількох речей, що мені не під силу тут усе це перелічити. Натомість я присвятив йому цю книжку.

Існує також багато інших людей, із якими я будував плани на майбутнє і співпрацював, і таких, які глибоко вплинули на моє професійне життя. Це Лоуелл Ліндстром, Дейв Томас, Майкл Физерс, Боб Косс, Бретт Шухарт, Дін Вамплер, Паскаль Рой, Джефф Лангр, Джеймс Греннінг, Браян Баттон, Алан Френсіс, Майк Хілл, Ерік Мід, Рон Джеффріс, Кент Бек, Мартін Фаулер, Грейді Буч та нескінченний список інших. Дякую, дякую всім!

Звісно, найбільшою моєю прихильницею була моя прекрасна дружина Енн Марі. Я одружився з нею, коли мені було 20, а їй за три дні до того виповнилося 18. Протягом 38 років вона була моїм постійним супутником, моїм кермом і вітрилом, моєю любов'ю і моїм життям. Я з радістю проведу ще мінімум чотири десятиліття поруч із нею.

А тепер моїми помічниками та однодумцями є мої діти. Я постійно співпрацюю зі своєю старшою дочкою Анжелою. Вона тримає мене в тонусі і ніколи не дозволяє забути про призначену зустріч чи зобов'язання. Я складаю бізнес-плани разом зі своїм сином Мікою, засновником *8thlight.com*. Його голова працює в бізнесі набагато краще за мою. Наше останнє починання — *cleancoders.com* — це щось дуже захопливе!

Мій молодший син Джастін щойно почав працювати з Мікою у *8thlight.com*. Моя молодша дочка Джина — інженер-хімік, вона працює в *Honeywell*. Із цими двома серйозні справи тільки розпочинаються!

Ніхто у житті не навчить вас більше і краще за ваших дітей. Дякую, діти!

ПРО АВТОРА

Роберт С. Мартін (Дядечко Боб) працює програмістом із 1970 року. Він є засновником і президентом *Object Mentor, Inc.*, міжнародної фірми, що складається з досвідчених розробників і менеджерів розробки програмного забезпечення, які спеціалізуються на допомозі компаніям у виконанні їхніх проєктів. *Object Mentor* пропонує найбільшим корпораціям по всьому світу консультації з покращення процесів, з об'єктно-орієнтованого ПЗ, навчання та підвищення кваліфікації.

Роберт Мартін опублікував десятки статей у різних галузевих журналах і є постійним доповідачем на міжнародних конференціях і виставках. Він також є автором та редактором багатьох книжок, серед яких найбільш визначними є:

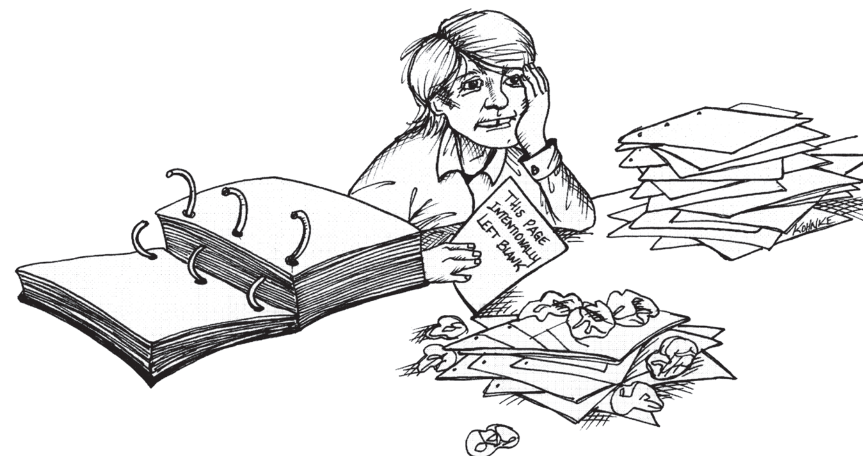
- Designing Object Oriented C++ Applications Using the Booch Method
- Patterns Languages of Program Design 3
- More C++ Gems
- Extreme Programming in Practice
- Agile Software Development: Principles, Patterns, and Practices
- UML for Java Programmers

- Clean Code (у перекладі українською: Мартін, Роберт С. Чистий код: створення, аналіз і рефакторинг. Харків, «Фабула», 2020.)

Залишаючись лідером у сфері розробки програмного забезпечення, Мартін три роки поспіль працював головним редактором «C++ Report», а також першим головою *Agile Alliance*. Він також є засновником *Uncle Bob Consulting, LLC* і співзасновником (разом зі своїм сином Мікою Мартіном) фірми *The Clean Coders LLC*.

ВСТУП

(Не пропускайте цього, воно вам знадобиться)



Я припускаю, що ви щойно взяли цю книжку до рук саме тому, що ви розробник і вас цікавить поняття професіоналізму. Так і повинно бути! Бо професіоналізм — це те, чого так не вистачає нашій професії.

Я також розробник. Я займаюся програмуванням більше 42 років[1] і за цей час — скажу відверто — бачив геть усе. Мене звільняли. Я здобув усесвітнє визнання. Я був керівником групи, начальником, рядовим працівником і навіть виконавчим директором. Я працював із видатними розробниками і недолугими «слимаками»[2]. Я займався розробкою як передових вбудованих програмно-апаратних комплексів, так і корпоративних систем нарахування зарплати. Я програмував на COBOL, FORTRAN, BAL, PDP-8, PDP-11, C, C++, Java, Ruby,

[1] Не треба панікувати *(тут і далі всі примітки, за винятком особливо оговорених, належать авторові).*

[2] Термін невизначеного походження.

Smalltalk та на багатьох інших мовах та системах. Я працював із ненадійними людьми, і я працював із висококваліфікованими професіоналами. Саме останнім і присвячено цю книжку.

На її сторінках я спробую визначити, що означає бути професійним програмістом. Я опишу ті атрибути, ознаки та дії, що, на мою думку, притаманні справжнім професіоналам.

Звідки я знаю, що це за атрибути, ознаки та дії? Тому що дізнався про все це із власного досвіду. І коли я отримав свою першу посаду розробника, нікому навіть на думку не спадало називати мене професіоналом.

Це трапилося 1969 року, мені тоді було лише 17 років. Мій батько переконав місцеву фірму *ASC* найняти мене на неповний робочий день. (Так, мій батько це вміє. Одного разу він на моїх очах став на шляху машини, що розганялася, підняв руку і наказав: «Стояти!» Машина зупинилася. Моєму татові взагалі важко відмовити.) Отже, мене взяли на роботу. Моє робоче місце розташовувалося в кімнаті, де зберігалися всі інструкції до комп'ютерів IBM, і мене змусили вносити до них описи оновлень за кілька останніх років. Саме тоді я вперше побачив фразу: «Сторінка навмисно залишена порожньою» («This page intentionally left blank»).

Через кілька днів удосконалення інструкцій начальник запропонував мені написати просту програму на Easycoder[1]. Я був у захваті, адже до цього я ще не написав жодної програми для справжнього комп'ютера. Втім, я швидко переглянув кілька книжок по Autocoder і приблизно уявляв, із чого треба починати.

Моя програма мала прочитати записи з магнітної стрічки і змінити ідентифікатори цих записів. Значення нових ідентифікаторів починалися з 1 і збільшувалися на 1 для кожного наступного запису. Записи з оновленими ідентифікаторами мали перезаписуватися на нову стрічку.

Начальник показав мені полицю, на якій лежали стоси коробок із червоними і синіми перфокартами. У них чергувалися карти червоного та синього кольорів, кожна пачка налічувала приблизно

[1] Easycoder був асемблером для комп'ютера Honeywell H200, схожим на Autocoder для комп'ютера IBM 1401.

200 карток. Кольорові смуги на картах містили вихідний код бібліотеки підпрограм, яку зазвичай використовували розробники. Вони просто знімали верхню пачку зі стосу (переконавшись, що взяли тільки червоні або тільки сині карти) і клали її в кінець своєї програмної «колоди».

Моя програма була написана на так званих програмних формулярах. Програмні формуляри являли собою великі прямокутні аркуші паперу, розділені на 25 рядків та 80 стовпців. Кожен рядок відповідав одній картці. Програма записувалася на формулярі великими літерами за допомогою олівця № 2. В останніх шістьох стовпцях кожного рядка записувався її номер тим самим олівцем. Порядкові номери зазвичай збільшувалися на 10, аби пізніше в стос можна було вставити нові карти.

Формуляри передавалися операторам підготовки даних. У компанії працювало кілька десятків жінок, які брали формуляри з великої скриньки та «набивали» їх на клавішних перфораторах. Ці машини були дуже схожі на друкарські машинки, але не друкували знаки на папері, а кодували їх, пробиваючи отвори в перфокартах.

Наступного дня оператори повернули мою програму внутрішньою поштою. Маленький стос перфокарт був загорнутий у формуляр і перетягнутий резинкою. Я пошукав на перфокартах помилки набору. Начебто все нормально. Я поклав бібліотечну колоду наприкінці свого стосу програм і відніс операторам.

Комп'ютери були встановлені в машинному залі з регульованим мікрокліматом та фальш-підлогою (для прокладання кабелів). Я постукав у двері, суворий оператор забрав у мене колоду й поклав її в ящик. Моя програма буде запущена, коли до неї дійде черга.

Наступного дня я одержав свою колоду назад. Вона була загорнута в опис результатів перевірки і перетягнута резинкою. (Так, у ті дні ми використовували *дуже багато* резинок!)

Я відкрив лістинг і побачив, що програма не пройшла компіляцію. Повідомлення про помилки у коді виявилися надто складними для мого розуміння, тому я відніс їх до свого начальника. Він переглянув перелік повідомлень, щось пробурмотів під ніс, зробив кілька позначок, узяв колоду й наказав іти за ним.

Він пройшов до операторської і сів за вільний перфоратор. Одну за одною виправив усі карти з помилками і додав ще кілька карток. А тоді коротко пояснив суть того, що робив, але я далеко не все усвідомив, бо все промайнуло в одну мить.

Він відніс нову колоду до машинного залу й постукав у двері. Кинув кілька чарівних слів оператору, а потім і сам пройшов до залу, поманивши мене за собою. Оператор встановив магнітні стрічки на накопичувачі та завантажив колоду, поки ми спостерігали за ним. Закрутилися стрічки, заторохтів принтер — і на цьому все скінчилося. Програма запрацювала.

Наступного дня начальник подякував мені за допомогу, і моя робота на цьому завершилася. Вочевидь, фірма *ASC* вирішила не гаяти час на навчання сімнадцятирічного новачка.

Утім, мій зв'язок з *ASC* на цьому не завершився. Через кілька місяців я отримав там постійну роботу з обслуговування принтерів у вечірню зміну. Ці принтери друкували будь-яку нісенітницю з образів, що зберігалися на стрічці. Я мав своєчасно заправляти принтери папером, ставити стрічки з образами, видаляти «зажований» папір і взагалі стежити за тим, аби пристрої працювали.

Усе це відбувалося 1970 року. Я не міг собі дозволити навчання в коледжі, та й воно мене не надто приваблювало. Війна у В'єтнамі ще не скінчилася, і в студентських містечках було неспокійно. Я продовжував вивчати книжки з COBOL, Fortran, PL/1, PDP-8 та асемблера для IBM 360. Я плекав надію обійтися без навчання і якнайшвидше отримати роботу розробника.

Через дванадцять місяців я досягнув цієї мети. Мене підвищили до штатного розробника в *ASC*. Я і двоє моїх друзів, Річард і Тім, яким теж було по 19 років, працювали разом із трьома іншими розробниками над бухгалтерською системою, що мала працювати в режимі реального часу, для фірми, котра займалася вантажними перевезеннями. Ми працювали на Varian 620i. То був простий мінікомп'ютер з архітектурою, подібною до PDP-8, за винятком того, що він мав 16-розрядні слова і два регістри. Програмування велося на асемблері.

Ми написали кожен рядок коду у цій системі. І я дійсно маю на увазі *кожен* рядок. Ми написали операційну систему, обробники переривань, драйвери вводу/виводу, *файлову систему* для дисків, систему підвантаження зображення і навіть компонувальник із динамічною переадресацією. Не кажучи вже про код програми. Ми написали все це за 8 місяців, працюючи по 70–80 годин на тиждень, аби вкластися в божевільний термін. Тоді я отримував 7200 доларів на рік.

Система запрацювала вчасно. А потім ми пішли.

Ми звільнилися раптово і з озлобленням. Річ у тім, що після пекельної роботи та успішної здачі системи, компанія підвищила нам заробітну платню на якихось два відсотки. Ми відчували себе ображеними й ображеними. Деякі з нас знайшли роботу в інших місцях і просто подали заяви про звільнення.

На жаль, я обрав інший, далеко не найкращий шлях. Ми з приятелем вдерлися до кабінету директора і звільнилися з неабияким скандалом. Це принесло нам емоційне задоволення, що тривало приблизно добу.

Наступного дня я зрозумів, що в мене немає роботи. Мені ледве виповнилося 19 років, і я був безробітним без будь-якого диплома. Я пройшов співбесіди на кілька вакансій у галузі програмування, але вони завершилися невдало. Наступні чотири місяці я пропрацював у майстерні з ремонту газонокосарок, що належала моєму зведеному братові. На жаль, ремонтника з мене не вийшло, і врешті-решт мені довелося піти. Я впав у депресію.

Я засиджувався до третьої години ночі, поїдаючи піцу і дивлячись старі фільми жахів на старому чорно-білому телевізорі моїх батьків з антеною, схожою на заячі вуха. Поступово кошмари на екрані почали просочуватися в моє життя. Я валявся в ліжку до полудня, бо не хотів бачити черговий сумний день. Я поступив на курси математичного аналізу в регіональному коледжі, однак провалив іспит. Моє життя летіло під укіс.

Мати нарешті поговорила зі мною, пояснивши, що так жити неможливо і що я був ідіотом, коли звільнився, не знайшовши нової роботи, а ще й пішов зі скандалом разом із приятелем. Вона сказала мені, що звільнятися, не маючи нової роботи, взагалі не можна, і робити це слід спокійно, тверезо та одноосібно. А ще вона сказала, що мені варто зателефонувати колишньому начальнику

і попроситися на те саме місце. І додала: «Тобі потрібно скуштувати пиріг скромності».

Дев'ятнадцятирічні хлопці не схильні визнавати власні помилки, і я, звісно, не був винятком. Проте обставини взяли гору над моєю гординею. Зрештою я зателефонував начальнику та відкусив великий шматок від того пирога скромності. І це спрацювало! Він охоче запропонував мені зарплатню у 6800 доларів на рік, а я, зі свого боку, охоче прийняв його пропозицію.

Наступні вісімнадцять місяців я працював на попередньому місці, звертаючи увагу на кожну дрібницю і намагаючись стати по-справжньому цінним співробітником. Винагородою мені стали посадові підвищення та збільшення заробітної платні. Все йшло якнайкраще. Коли я пішов із цієї компанії, ми залишилися в добрих відносинах, а мені на той час уже запропонували набагато кращу роботу.

Мабуть, вам здається, що я засвоїв отриманий урок і через те став професіоналом? Нічого подібного. Це був лише перший із численних уроків, які мені ще треба було засвоїти. Пізніше мене звільнили з однієї компанії за зірваний через безтурботність графік і мало не звільнили з іншої за випадкове розголошення конфіденційної інформації. Я брався за ризиковані проєкти і блискуче завалював їх, не звертаючись по допомогу, яка, як я вважав, була мені не потрібна. Я завзято захищав свої технічні рішення, навіть тоді, коли вони суперечили потребам замовників. Я прийняв на роботу геть некваліфіковану людину, яка стала важким тягарем для мого наймача. І що найгірше — через мої організаційні помилки звільнили двох інших людей.

Тому краще ставтеся до цієї книжки як до каталогу моїх помилок, висповідання моїх гріхів, а також збірки порад, що допоможуть вам уникнути аналогічних помилок.

1 ПРОФЕСІОНАЛІЗМ

О, смійся, Кертіне, старий! Це чудовий жарт, який зіграв із нами Господь, чи доля, чи природа — як тобі завгодно. Але хай би хто його зіграв, безперечно, він мав почуття гумору! Ха!

Говард, персонаж роману Б. Травена «Скарб Сьєрра-Мадре»

Отже, ви хочете стати професійним розробником, чи не так? Ходити з гордо піднятою головою та оголошувати всьому світу: «Я професіонал!» І щоби люди дивилися на вас з повагою, а матері вказували на вас і говорили своїм дітям, що ті мають бути схожими на вас. Ви ж хочете цього, так?

Будьте обережні у своїх проханнях

Термін «професіоналізм» має багато смислових відтінків. Звичайно, професіоналізм — це своєрідний почесний знак і привід для гордості, але він також є ознакою відповідальності і підзвітності. Зрозуміло, що ці сторони професіоналізму нерозривно пов'язані між собою. Ви не можете пишатися тим, за що не несете жодної відповідальності.

Бути непрофесіоналом набагато простіше. Непрофесіонали не несуть відповідальності за свою роботу — вони залишають відповідальність своїм роботодавцям. Якщо непрофесіонал робить помилку, то сміття за ним прибирає роботодавець. Але якщо професіонал робить помилку, то він сам усуває її наслідки.

А якщо у ваш модуль закрадеться помилка, що обійдеться вашій компанії у 10 тисяч доларів? Непрофесіонал зниже плечима, скаже: «Бува, що поробиш», і продовжить писати наступний модуль. Професіонал має виписати своїй компанії чек на 10 тисяч[1]!

Дивна річ: коли йдеться про ваші особисті гроші, все виглядає трохи інакше. Згодні? Але це відчуття постійно присутнє у професіоналів, бо воно є суттю професіоналізму. Тому що професіоналізм — це відповідальне ставлення до справи.

Брати відповідальність

Ви же прочитали «Попередній вступ», чи не так? Якщо не прочитали — поверніться і прочитайте, бо він створює широкий контекст для решти матеріалу цієї книжки.

[1] Сподіваюся, що подібні речі трапляються з ним не часто.

Щоби по-справжньому зрозуміти, чому так важливо брати на себе відповідальність, мені довелося пережити на власному досвіді наслідки відмови від неї.

У 1979 році я працював на компанію *Teradyne*. Я був «провідним інженером» і відповідав за розробку програми, що управляла мікрокомп'ютерною системою для вимірювання якості роботи телефонних ліній. Центральний мінікомп'ютер підключався по виділених або комутованих лініях на швидкості 300 бод до десятків периферійних мікрокомп'ютерів, що керували вимірювальним обладнанням. Код був написаний на асемблері.

Нашими користувачами були сервіс-менеджери великих телефонних компаній. Кожен із них відповідав за 100 тисяч телефонних ліній. Моя система допомагала менеджерам знаходити та виправляти проблеми в телефонних лініях ще до того, як вони будуть помічені клієнтами. У такий спосіб скорочувалася кількість скарг клієнтів, що їх вимірювала та використовувала комісія з комунального обслуговування для регулювання тарифів, які призначали телефонні компанії. Інакше кажучи, ця система була надзвичайно важливою.

Щоночі проводилася «нічна перевірка»: центральний мінікомп'ютер наказував кожному з периферійних мікрокомп'ютерів протестувати всі телефонні лінії, що були під його контролем. Уранці центральний комп'ютер отримував список пошкоджених ліній разом із характеристиками наявних дефектів. За даними звіту сервіс-менеджери будували графіки роботи ремонтників, аби пошкодження виправлялися ще до надходження скарг від клієнтів.

Іноді я розсилав кільком десяткам замовників нову версію свого ПЗ. «Розсилав» — цілком відповідне слово. Я записував програму на стрічки та надсилав ці стрічки своїм клієнтам. Клієнти завантажували стрічки, а потім перезапускали свої системи.

Чергова версія містила виправлення низки незначних дефектів і нову функцію, що її вимагали наші клієнти. Зазвичай ми обіцяли реалізувати цю нову функцію до певної дати. Я ледве встигав підготувати стрічки в ніч перед зазначеною датою, щоби клієнти отримали їх вчасно.

За два дні до дедлайну мені зателефонував Том, менеджер нашого експлуатаційного відділу. За його словами, кілька клієнтів поскаржилися на те, що «нічна перевірка» не відбулася і вони не отримали звітів. У мене завмерло серце, бо для того, щоби вчасно підготувати чергову версію програми, я не зробив тестування нового коду. Я протестував основну функціональність системи, але на тестування програми перевірки телефонних ліній у цілому було потрібно багато годин, а мені належало видати програму негайно. Жодна з виправлених помилок не містилася в коді перевірки телефонних ліній, тому я відчував себе у безпеці.

Втрата звіту «нічної перевірки» була серйозною справою. Це означало, що ремонтники отримають менше роботи, яку пізніше їм доведеться надолужувати. Також деякі клієнти могли помітити дефект і поскаржитися. Втрати даних за цілу ніч було достатньо, щоби сервіс-менеджер зателефонував Тому і влаштував йому прочуханку.

Я ввімкнув тестову систему, завантажив оновлену програму й запустив перевірку. Програма пропрацювала кілька годин, а згодом аварійно завершилася. Код не працював. Якби я протестував його до постачання, то дані не були би втрачені, а сервіс-менеджери не смикали би Тома.

Я зателефонував Тому й повідомив, що мені вдалося виявити проблему. Під час розмови з'ясувалося, що багато інших клієнтів уже зверталися до нього з подібною проблемою. Потім він поцікавився, коли я зможу виправити помилку. Я відповів, що поки не знаю, але працюю над цим. Далі я сказав йому, що клієнтам варто було би повернутися до попередньої версії програми. Том розсердився і заявив, що таке повернення буде, так би мовити, подвійним ударом для клієнтів: вони втрачають дані за цілу ніч і не можуть використовувати нову функцію.

Помилки виявилися глибоко прихованими, а тестування потребувало кількох годин. Перші виправлення не спрацювали. Другі — також. Мені знадобилося кілька спроб, а отже, і кілька днів, аби розібратися в проблемі. Весь цей час Том телефонував мені кожні кілька годин і питав, коли буде виправлена помилка. Він також переповідав мені все, що йому казали клієнти, і скаржився на те, як незручно йому було пропонувати їм скористатися попередніми програмами.

Зрештою я знайшов дефект, відправив оновлення, і все повернулося до норми. Том, який не був моїм начальником, поступово охолонув, і весь цей епізод начебто пішов у минуле. Але коли інцидент було вичерпано, мій безпосередній керівник прийшов до мене і сказав: «Це не повинно повторитися». І я з ним погодився.

Поміркувавши, я зрозумів, що розсилка програми без тестування коду була безвідповідальним вчинком. Я знехтував тестуванням, аби доповісти, що оновлення було відправлено вчасно. При цьому я думав лише про власну репутацію. Мене не хвилював ані клієнт, ані мій роботодавець. Насправді ж треба було ще від самого початку вчинити відповідально — тобто повідомити Тому, що тестування не завершено і що я не готовий надіслати програму у визначений термін. Це було би неприємно, Том би засмутився. Але клієнти не втратили би своїх даних, і все обійшлося би без скарг та розгніваних дзвінків.

Не зашкодьте

Отже, як саме ми беремо на себе відповідальність? Існують певні принципи. Клятва Гіппократа може здатися трохи пишномовною, але що може бути кращим? І, зрештою, це цілком логічно, адже найперше зобов'язання і головна ціль професіонала-початківця — це застосовувати свої здібності лише на благо.

Яку шкоду може заподіяти розробник? Із суто програмного погляду, він може нашкодити функціональності і структурі продукту. Давайте поглянемо, як цього уникнути.

Не зашкодьте функціональності

Звісно, ми хочемо, щоби наші програми працювали. Багато хто з нас став розробником після того, як йому вдалося змусити працювати свою першу програму, і це почуття хочеться відчувати знову і знову. Але у працездатності наших програм зацікавлені не лише ми. Наші клієнти та роботодавці хочуть того ж. Не забувайте — вони платять нам за створення програм, що роблять саме те, що їм потрібно.

Функціональність програм страждає від помилок. Отже, однією з ознак професіоналізму повинно бути написання програм без помилок.

«Але зачекайте! — чую я від вас.— Це ж неможливо, бо програмний код надто складний, аби написати його без помилок».

Безсумнівно, ви маєте слушність. Програмний код надто складний, щоби створити його без помилок. На жаль, це не позбавляє нас відповідальності. Людський організм також надто складний, щоби досконало вивчити його, але лікарі все одно присягаються не завдавати шкоди. І якщо навіть вони не намагаються уникнути відповідальності, то з якого дива цього прагнемо ми?

Я чую ваші заперечення: «То що, ми мусимо писати абсолютно досконалий код?»

Ні, я хочу сказати, що ви маєте відповідати за власну недосконалість. Те, що у вашому коді виникають помилки, не означає, що ви за них не відповідаєте. Написати ідеальну програму практично неможливо, але відповідальність за всі недоліки несете саме ви, і ніхто інший.

Це вірна ознака справжнього професіонала — уміння відповідати за власні помилки, поява яких практично неминуча. Отже, мій професіонал-початківець, насамперед навчися вибачатися. Вибачення потрібні, але їх недостатньо. Не можна робити ті самі помилки знову і знову. У міру професійного становлення частота помилок у вашому коді має асимптотично наближатися до нуля. Вона ніколи не сягне цієї позначки, але ви відповідальні за те, щоби вона була якомога ближчою до нуля.

Контроль якості не повинен нічого виявити

Коли ви передаєте остаточну версію продукту до служби контролю якості, ви повинні розраховувати на те, що контроль не виявить жодних проблем. Дуже непрофесійно передавати на контроль якості свідомо дефектний код. А який код є свідомо дефектним? Будь-який, у якому ви *не впевнені*!

Деякі фахівці використовують службу контролю якості для виявлення помилок. Вони розраховують на те, що контроль виявить помилки та поверне їх список. Деякі компанії навіть виплачують премії службі контролю якості за виявлені помилки. Що більше помилок — то більша премія.

Річ навіть не в тім, що це дуже витратна практика, яка завдає шкоди компанії і продукту. І не в тім, що така поведінка руйнує графіки і підриває довіру до організації в команди розробників. І навіть не в тім, що це банальний прояв лінощів і безвідповідальності. Передавати на контроль якості код, працездатність якого ви не можете гарантувати,— непрофесійно. Така поведінка порушує правило «не нашкодьте».

Чи знайде служба контролю якості помилки? Можливо, тож приготуйтеся вибачатися, а потім подумайте, чому ці помилки уникли вашої уваги, і зробіть щось для того, щоби це не повторилося знову.

Коли служба контролю якості, або ще гірше — *користувач*, виявляє помилку, це має вас здивувати, засмутити і налаштувати на запобігання повторенню подібних речей у майбутньому.

Ви мусите мати впевненість, що ваш код працює

Як *дізнатися*, чи працює ваш код? Це легко. Протестуйте його. Потім протестуйте ще раз. Протестуйте згори до низу. Протестуйте знизу догори. Протестуйте всіма відомими способами!

Можливо, вас турбує, що таке ретельне тестування коду вимагає надто багато часу. Зрештою у вас є графіки і терміни, яких потрібно дотримуватися. І якщо витрачати час на тестування, то коли писати код? Ви маєте рацію! Тому тестування слід автоматизувати. Напишіть модульні тести, що можна виконувати будь-якої миті, і запускайте їх якнайчастіше.

Яка частина коду має тестуватися цими автоматизованими модульними тестами?

Мені насправді потрібно відповідати на це питання? Весь код! Весь, без винятку.

Гадаєте, я пропоную стовідсоткове тестове покриття коду? Ні, я *не пропоную*. Я *вимагаю* цього. Кожен написаний вами рядок коду мусить бути протестований. І на цьому крапка.

Скажете, це нереалістично? Але чому? Ви пишете код, тож очікуєте, що він буде виконуватися. Якщо ж ви очікуєте, що ваш код буде виконуватися, то повинні знати, чи він працює. А дізнатися цього можна лише одним засобом — тестуванням.

Я є головним автором та виконавцем проєкту з відкритим кодом FitNesse. На момент написання цієї книжки, розмір FitNesse сягнув 60 KSLOC, 26 із яких містяться у понад 2 тисячах модульних тестів. За даними EMMA, покриття цих 2 тисяч тестів складає близько 90 % коду.

Чому не більше? Тому що EMMA бачить не всі рядки, що виконуються! Я вважаю, що ступінь покриття набагато вищий. Чи становить він 100 %? Ні, бо 100 % — це асимптотична межа.

Але ж деякі частини коду важко тестувати! Так, але тільки тому, що цей код був недоладно *спроєктований*. Отже, код слід проєктувати з огляду на подальшу *простоту тестування*. І найкраще для цього — написати ваші тести перш ніж писати код, що має їх проходити.

Цей принцип широко використовується в методології розробки через тестування (TDD, Test Driven Development), що буде детально описана в одному з наступних розділів.

Автоматизований контроль якості

Уся процедура контролю якості за допомогою FitNesse полягає у виконанні модульних та прийомних тестів. Якщо тести проходять успішно, видається продукт. При цьому процедура контролю займає близько трьох хвилин, і її можна виконати будь-якої хвилини.

Звичайно, через помилку у FitNesse ніхто не постраждає і не втратить мільйони доларів. З іншого боку, FitNesse має багато тисяч користувачів і *дуже* невеликий список дефектів.

Безумовно, деякі системи настільки критичні, що короткого автоматизованого тесту недостатньо для визначення їхньої готовності до

розгортання. Водночас вам як розробнику потрібен відносно швидкий і надійний механізм перевірки, що написаний код працює і не заважає роботі інших частин системи. Отже, автоматизовані тести здатні щонайменше повідомити, що *система з великою ймовірністю* пройде контроль якості.

Не зашкодьте структурі

Справжньому професіоналові відомо, що додання функціональності на шкоду структурі — остання справа. Структура коду забезпечує його гнучкість. Порушуючи структуру, ви руйнуєте майбутнє коду.

Усі програмні проєкти базуються на фундаментальному припущенні щодо простоти змін. Якщо ви порушуєте цей принцип, створюючи негнучкі структури, то тим самим підриваєте економічну модель, закладену в основу всієї галузі.

Коротше кажучи: *внесення змін не має призводити до надмірних витрат*.

На жаль, занадто багато проєктів грузнуть у болоті поганої структури. Завдання, що колись виконувалися за лічені дні, починають забирати тижні, а згодом і місяці. Керівництво у відчайдушних спробах надолужити втрачені темпи наймає додаткових розробників для пришвидшення роботи. Але нові розробники лише погіршують ситуацію, поглиблюючи наявні структурні недоліки і створюючи нові перешкоди.

Про принципи і патерни проєктування, що сприяють створенню гнучких, зручних у супроводі структур, написано дуже багато[1]. Професійні розробники постійно тримають ці правила у пам'яті і намагаються будувати програмні архітектури, виходячи з них. Однак є один нюанс, про який часто забувають: *якщо ви хочете, щоби ваш код був гнучким, його необхідно перевіряти на гнучкість!*

Єдиний спосіб переконатися, що ваше програмне забезпечення легко змінювати,— це спробувати внести до нього зміни. І якщо зробити

[1] Неявне посилання на книжку Р. Мартіна «Agile Software Development: Principles, Patterns, and Practices».— *Прим. перекл.*

це виявляється складніше, ніж передбачалося, варто переробити структуру коду, щоби наступні зміни вносилися простіше.

Коли потрібно робити такі зміни? *Завжди!* Щоразу під час роботи з модулем треба потроху вдосконалювати його структуру. Кожне читання коду має призводити до вдосконалення структури.

Цю ідеологію іноді називають *безжальним рефакторингом*. Я називаю цей принцип «правилом бойскаута»: завжди залишайте модуль чистішим, ніж до перевірки. Робіть добрі справи в коді кожного разу, як його бачите.

Такий підхід цілком суперечить ставленню деяких фахівців до програмного коду. Вони вважають, що часті зміни в робочому коді є небезпечними. Ні! Небезпечно залишати код у статичному, недоторканому стані. Якщо не перевіряти код на гнучкість, то в тих випадках, коли треба внести зміни, він може виявитися занадто «жорстким».

Чому багато розробників бояться вносити часті зміни до свого коду? Вони бояться його порушити! А чому вони цього бояться? Тому що не мають тестів.

І знову ми повертаємось до тестів. Якщо у вас є автоматизований тестовий пакет, що покриває майже 100 % коду, і якщо цей пакет можна швидко застосувати у будь-який момент часу, то *ви взагалі не боятиметеся змінювати код*. А як довести, що ви не боїтеся змінювати код? Змінювати його якомога частіше.

Професійні розробники настільки впевнені у своєму коді та тестах, що залюбки вносять випадкові, спонтанні зміни. Вони можуть ні з сього ні з того перейменувати клас. Помітивши задовгий метод під час читання модуля, звісно ж, перероблять його на коротший. Вони можуть перетворити команду `switch` на поліморфну конструкцію або згорнути ієрархію наслідування у ланцюговий виклик (chain-of-command). Коротше кажучи, професіонали відносяться до коду так само, як скульптор відноситься до глини,— постійно розминають його та надають нових форм.

Трудова етика

За свою кар'єру ви відповідаєте самі. Ваш роботодавець аж ніяк не повинен піклуватися про вашу популярність на ринку праці. Він не повинен навчати вас, відправляти на конференції або купувати вам найновіші фахові видання. Всім цим ви маєте займатися *самі*. І горе тому розробникові, який довірить власну кар'єру своєму роботодавцю.

Деякі роботодавці погоджуються купувати вам книжки, відправляти вас на семінари та конференції. Чудово, вони роблять вам велику послугу. Але не думайте, що вони мусять це робити! Якщо ваш роботодавець не робить цього за вас, поміркуйте, як зробити це самотужки.

Ваш роботодавець також не повинен надавати вам час для навчання. Деякі роботодавці можуть це робити, а деякі навіть вимагають, аби ви займалися підвищенням кваліфікації. Але і в цьому разі вони роблять вам послугу, і ви маєте віддячити їм. Подібних послуг ніколи не варто очікувати.

Ви зобов'язані своєму роботодавцю деяку кількість часу та інтелектуальних зусиль. Візьмемо, наприклад, стандартний для США сорокагодинний робочий тиждень. Ці сорок годин ви мусите провести за розв'язанням *проблем вашого роботодавця*, а не *ваших особистих проблем*.

Краще заплануйте собі шістдесят робочих годин на тиждень. Перші сорок ви працюєте на компанію, а решту двадцять — на себе. Протягом цих 20 годин ви читаєте, практикуєтеся, навчаєтеся та іншим доступним чином розвиваєте свої навички.

Ви можете заперечити: «А як же моя сім'я? Моє особисте життя? Я маю жертвувати всім заради свого роботодавця?»

Але я не маю на увазі *весь* ваш особистий час — лише двадцять додаткових годин на тиждень. Якщо ви будете використовувати обідню перерву на читання або прослуховування подкастів і ще дев'яносто хвилин на день на вивчення нової мови — це вирішить усі проблеми.

Давайте трохи порахуємо. У тижні 168 годин. Сорок дістається вашому роботодавцю, ще двадцять — вашій кар'єрі. Залишається 108. П'ятдесят шість витрачається на сон, тож залишаються 52 години.

Можливо, ви не хочете брати подібні зобов'язання. І це цілком нормально, але тоді не вважайте себе професіоналом. Професіонали не шкодують часу на вдосконалення у професії.

Можливо, ви вважаєте, що робота мусить залишатися на робочому місці і її не слід тягнути із собою додому. Згоден! Ці двадцять годин ви маєте працювати не на свого роботодавця, а на власну кар'єру.

Інколи ці два напрями збігаються, і робота, яку ви виконуєте для роботодавця, виявляється виключно корисною для вашої кар'єри. У такому разі витратити на неї частину зі згаданих двадцяти годин буде цілком розумно. Але не забувайте, що ці години призначені виключно для *вас*. Вони повинні використовуватися для того, щоби підвищити вашу професійну цінність.

Може здатися, що цей шлях веде до професійного вигорання. Навпаки — він допомагає *уникнути* вигорання. Можна припустити, що ви стали розробником завдяки своєму ентузіазму і любові до програмування, а ваше бажання стати професіоналом зумовлене саме цим ентузіазмом. Тоді впродовж цих двадцяти годин ви займатиметеся тим, що *зміцнить* ваш ентузіазм. Отже, ці години мають виявитися цікавими!

Знати свою сферу

Вам відомо, що таке діаграма Нассі—Шнейдермана? І якщо невідомо, то чому? А чим відрізняються стейт-машини Mealy та Moore? Це ви повинні знати. Чи зможете ви написати процедуру швидкого сортування (quicksort), не звертаючись до опису алгоритму? Вам відомо, що таке «трансформаційний аналіз» (Transform Analysis)? Чи могли би ви виконати функціональну декомпозицію діаграми інформаційного потоку? А що означає термін «бродячі дані» (Tramp Data) або «примирення» (Conascence)? Навіщо потрібна таблиця Парнаса?

За останні п'ятдесят років у нашій галузі з'явилося багато нових ідей, дисциплін, методів, інструментів і термінів. Які з них вам відомі? Адже кожен, хто хоче стати професіоналом, мусить засвоїти вагому частину цих знань і постійно збільшувати розмір цієї частини.

Навіщо все це знати? Хіба ж наша сфера не прогресує так стрімко, що старі ідеї швидко втрачають актуальність? Перша частина цього питання цілком очевидна: безумовно, відбувається стрімкий прогрес.

Але зазначте: цей прогрес у багатьох відношеннях має другорядне значення. Дійсно, нам уже не доводиться по 24 години чекати на завершення компіляції. Сьогодні ми пишемо системи, розмір яких вимірюється гігабайтами, працюємо у глобальній мережі, що надає миттєвий доступ до інформації. Але, з іншого боку, ми пишемо ті самі команди `if` і `while`, що і 50 років тому. Багато чого змінилося, але багато чого залишилося незмінним.

Друга частина питання так само хибна. Лише частина ідей останніх 50 років утратили актуальність. Дещо відійшло на другий план, це правда. Концепція каскадної розробки (waterfall development), наприклад, явно перестала бути популярною. Однак це не означає, що ми не повинні знати, що це за концепція, у чому полягають її сильні та слабкі аспекти.

Водночас переважна більшість ідей останніх 50 років анітрохи не втратила своєї цінності. А деякі стали навіть ще ціннішими.

Згадайте формулювання Дж. Сантаяни: «Ті, хто не пам'ятає минулого, приречені на його повторення».

Нижче наведений *мінімальний* перелік тем, у яких має розбиратися кожен розробник:

- **Патерни проєктування.** Ви маєте бути здатні описати всі 24 патерни із книжки «Банди Чотирьох» і мати практичне уявлення про багато інших патернів із книжок «Pattern-Oriented Software Architecture».
- **Принципи проєктування.** Ви повинні знати принципи SOLID і добре розумітися на принципах компонентного проєктування.
- **Методи.** Ви мусите розуміти суть методологій XP, Scrum, Lean, Kanban, каскадної розробки, структурного аналізу та структурного проєктування.
- **Дисципліни.** Вправляйтеся у практичному застосуванні розробки через тестування (TDD), об'єктно-орієнтованого проєктування, структурного програмування, безперервної інтеграції та парного програмування.
- **Артефакти.** Ви повинні навчитися працювати з UML, DFD, структурними діаграмами, мережами Петрі, діаграмами переходів, блок-схемами і таблицями рішень.

Безперервне навчання

Несамовитий темп змін у нашій галузі потребує від розробників постійного вивчення великих обсягів матеріалу — і лише для того, щоби залишатися в курсі справи. Горе тим архітекторам, які перестають програмувати,— вони швидко опиняються без роботи. Горе розробникам, які перестають вивчати нові мови,— їм залишається дивитися, як галузь обходиться без них. Горе тим розробникам, які не вивчають нові дисципліни та методології,— на них чекає занепад на тлі подальшого процвітання колег.

Чи підете ви до лікаря, який не знає, що зараз відбувається в медицині і не читає медичних журналів? Чи звернетесь до консультанта з податків, який не стежить за податковим законодавством та прецедентами? Тож навіщо роботодавцю наймати розробника, який не прагне бути в курсі новітніх віянь?

Читайте книжки, статті, блоги, твіти. Відвідуйте конференції. Відвідуйте спеціалізовані групи. Беріть участь у дослідницьких групах. Вивчайте те, що лежить за межами звичної сфери вашої роботи. Якщо ви розробник .NET — вивчайте Java. Якщо ви програмуєте на Java — вивчайте Ruby. Якщо ви програмуєте на C — вивчайте Lisp. А якщо вам закортить серйозно потренувати мозок, вивчайте Prolog та Forth!

Тренування

Професіонали тренуються. Справжні професіонали ретельно працюють над тим, щоби їхні навички були «відточені» і завжди готові до застосування. Недостатньо виконувати повсякденну роботу, вважаючи її тренуванням. Повсякденна робота — це виконання обов'язків, а не тренування. Тренування починається тоді, коли ви цілеспрямовано застосовуєте свої навички *за межами* робочих обов'язків з єдиною метою — вдосконалення цих навичок.

Що означатиме тренування для розробника? На перший погляд, сама концепція виглядає абсурдно. Але давайте трохи затримаємося та подумаємо. Як музиканти вдосконалюють свою майстерність? Звісно, не на концертах, а під час занять. Як вони це роблять? Серед іншого, за допомогою спеціальних вправ, гам та етюдів. Музиканти

повторюють їх знову і знову, щоби тренувати пальці та розум і підтримувати свою майстерність на належному рівні.

Але як можуть тренуватися розробники програмного забезпечення? У цій книжці цілий розділ присвячений різним методам тренування, тому зараз я не заглиблюватимуся в подробиці. Особисто я, наприклад, часто застосовую метод повторення простих вправ на кшталт «гри у боулінг» або «розкладання на прості множники». Я називаю ці вправи *ката*[1]. Існує багато різних ката, із яких можна вибрати те, що краще вам підійде.

Ката зазвичай мають вигляд простого завдання із програмування — наприклад, написання функції, що розкладає ціле число на прості множники. Метою виконання ката є не пошук рішення, адже ви вже знаєте, як вирішується подібне завдання. Ката тренують ваші пальці і ваш мозок.

Я щодня виконую одну-дві ката, часто на початку занурення в роботу. Я пишу їх на Java або на Ruby чи Clojure, чи якоюсь іншою мовою, якої я мушу дотримуватися в робочому стані. Я використовую ката для тренування конкретних навичок — скажімо, привчаючи пальці використовувати клавіші прискореного доступу, або прийомів рефакторингу.

Ставтеся до ката як до десятихвилинної ранкової розминки і десятихвилинної релаксації ввечері.

Спільна робота

Другий найкращий спосіб чогось навчитися — спільна робота з іншими людьми. Професійні розробники навмисно намагаються програмувати, тренуватися, проєктувати чи планувати разом. Вони багато чого дізнаються один від одного й виконують свою роботу швидше та з меншою кількістю помилок.

Це не означає, що ви повинні витрачати 100 % свого робочого часу на спільну роботу. Одноосібна робота також дуже важлива. І хай би як я любив програмувати в парі, мені конче потрібно час від часу попрацювати насамоті.

[1] Ката — систематизована послідовність прийомів у японських бойових мистецтвах, пов'язаних між собою принципами ведення бою з уявним противником.

Менторство

Найкращий спосіб чогось навчитися — навчати інших. Факти запам'ятовуються найшвидше тоді, коли вам потрібно повідомити їх людям, за яких ви відповідаєте. Отже, найбільшу користь від викладання передусім отримує викладач. До того ж, найкращий спосіб ввести нових людей в організацію — посидіти з ними і пояснити, що та як працює. Фахівці беруть на себе персональну відповідальність за навчання новачків. Вони не кидають новачків у річку, надаючи їм можливість або самотужки навчитися плавати, або піти на дно.

Знання предметної сфери

Кожен професіонал повинен розуміти предметну область програмованих ним рішень. Якщо ви пишете бухгалтерську систему, ви повинні розумітися на бухгалтерії. Якщо ви пишете додаток для туристичної фірми, ви повинні розумітися в туристичному бізнесі. Бути експертом не обов'язково, але до вивчення теми необхідно ставитись відповідально.

Починаючи проєкт у новій для себе області, прочитайте одну-дві книжки на відповідну тему. Проведіть співбесіди з клієнтом та користувачами стосовно основ конкретної предметної області. Порадьтеся з експертами, постарайтеся усвідомити їхні принципи та цінності.

Найгірший різновид непрофесіоналізму — програмувати виключно за специфікацією, не розуміючи, чи є ця специфікація придатною для виконання певного завдання. Ви повинні мати достатні знання у предметній області заради того, щоби розпізнати та виправити можливі специфічні помилки.

Розуміння інтересів роботодавця/замовника

Проблеми роботодавця — це *ваші* проблеми. Ви повинні розуміти їх і намагатися знайти найкращі рішення. У ході розробки системи уявіть себе на місці свого роботодавця і переконайтеся в тому, що можливості, які ви розробляєте, дійсно відповідають його потребам.

Розробникам легко ототожнювати себе один з одним. Але коли йдеться про роботодавців, багато хто опиняється у протиріччі з ними. Фахівці всіма силами прагнуть уникнути цього.

Скромність

Програмування — творча діяльність. Під час написання коду ми робимо щось буквально з нічого. Ми рішуче наводимо лад серед хаосу. Ми впевнено і в найменших деталях керуємо поведінкою машини, яка в іншому разі могла би завдати непоправної шкоди. Отже, програмування є актом виключно амбітним.

Професіонали знають, що вони зарозумілі, а не смиренні. Професіонал знає свою роботу й пишається нею. Професіонал упевнений у своїх здібностях і свідомо йде на ризик, спираючись на цю впевненість. Професіонал не може бути легкодухим або боязким.

Однак професіонал також знає, що в окремих випадках він зазнає невдачі, його оцінки ризиків виявляться неправильними, а його здібності — недостатніми; тоді він подивиться у дзеркало і побачить, що звідти йому усміхається самовпевнений бовдур.

Отже, перетворившись на мішень для кепкувань, професіонал сміється гучніше за всіх. Сам він ніколи не висміює колег, але приймає заслужені глузування і легко відмахується від незаслужених. Він не знущається з колег, які припустилися помилок, бо знає, що наступним може опинитися він сам. Професіонал розуміє не тільки свої переваги, але й те, що доля рано чи пізно знайде в нього слабкі місця. Коли ж її постріл влучить у ціль, залишиться тільки дотримуватися поради Говарда: сміятися.

Бібліографія

Martin, Robert C. *Principles, Patterns, and Practices of Agile Software Development*. NJ: Prentice Hall, 2002.

2 КАЗАТИ «НІ»

Зроби або не роби зовсім. Не треба намагатися.

Магістр Йода

На початку 1970-х років я та двоє моїх дев'ятнадцятирічних друзів працювали в компанії *ASC* над системою обліку в реальному часі для профспілки «Teamsters Union» в Чикаго. Якщо вам на думку спадають такі імена, як Джиммі Хоффа[1],— то ви маєте рацію. У 1971 році могутня загальноамериканська профспілка вантажоперевізників мала величезний вплив.

Наша система мала запрацювати до певної дати. Заради цього в розробку були вкладені великі гроші, а наша команда працювала по 60–80 годин на тиждень, аби встигнути вчасно.

Десь за тиждень до дати запуску систему нарешті зібрали докупи. У ній залишалося безліч помилок та нерозв'язаних проблем, і ми почали гарячково позбавлятися їх, керуючись довжелезним списком. Нам бракувало часу, щоб їсти та спати, не кажучи вже про те, щоби думати.

Нашим начальником в *ASC* був такий собі Френк, відставний полковник ВПС. Це був один із тих гучних і наполегливих керівників, які зазвичай пропонують обирати: або ти негайно стрибаєш з парашутом із висоти 10 тисяч футів, або робиш те саме, але вже без парашута. Ми, дев'ятнадцятирічні молокососи, боялися навіть в очі йому дивитися.

Френк сказав, що все має бути зроблено до наперед зазначеної дати. Нема тут про що міркувати. Коли настане термін, система має бути готовою. Крапка. І жодних «але».

Мій безпосередній бос, Білл, був набагато приємнішим хлопцем. Він пропрацював із Френком чимало років і добре розумів, що можливо, а що ні. Але й він сказав нам, що система має запрацювати до встановленої дати хай би що.

Система запрацювала вчасно. І все це увінчалося оглушливим провалом.

[1] Джим Хоффа — президент профспілки «Teamsters Union» у 1957–1971 рр., широко відомий своєю корумпованістю, зв'язками з мафією і нецільовим використанням пенсійних накопичень членів профспілки. У 1975 році безслідно зник при загадкових обставинах.— *Прим. перекл.*

Штаб-квартира профспілки «Teamsters Union» зв'язувалася з нашим комп'ютером, розташованим за 30 миль, через дюжину напівдуплексних терміналів, що працювали зі швидкістю 300 бод. Кожен термінал зависав що пів години. Ми стикалися із цією проблемою і раніше, але не мали часу на моделювання трафіку, який оператори введення даних несподівано створили для нашої системи.

Ситуація погіршувалася тим, що відривні аркуші, що друкувалися на телетайпах ASR35, підключених до нашої системи по 110-бодних телефонних лініях, зависали на середині друку. Проблема вирішувалася виключно перезавантаженням. Тобто всі оператори, чиї термінали ще працювали, мали завершити поточну роботу. Коли вся робота припинялася, вони телефонували нам, і ми перезавантажували комп'ютер. Операторам терміналів, що зависли, доводилося все робити наново. І це відбувалося не рідше одного разу на годину.

Через пів дня офіс-менеджер «Teamsters Union» наказав нам відключити систему і не вмикати, доки вона не запрацює нормально. Зрештою вони втратили половину робочого дня, а всі дані довелося заново вводити в старій системі.

Несамовите ревіння Френка розносилося по всій будівлі. Це тривало довго, дуже довго. Потім Білл і системний аналітик Джаліль зазирнули до нас і спитали, скільки часу знадобиться на забезпечення стабільної роботи системи. Я відповів: «Чотири тижні».

На їхніх обличчях відбився жах. «Ні,— сказали вони,— система має запрацювати до п'ятниці».

Я сказав: «Послухайте, система ледве запрацювала минулого тижня. Нам потрібно розібратися з безліччю проблем. На це знадобиться мінімум чотири тижні».

Але Білл і Джаліль залишалися непохитними: «Ні, все має запрацювати у п'ятницю. Ви можете хоча би спробувати?»

Тоді керівник нашої команди сказав: «Гаразд, ми спробуємо».

П'ятницю було обрано вдало — навантаження наприкінці тижня набагато нижче. Нам удалося знайти багато дефектів і виправити їх до понеділка. Але навіть після цього вся система ледве трималася, нагадуючи картковий будиночок. Проблеми із зависаннями, як і раніше,

відбувалися кілька разів на день. Також виникали інші проблеми. Тільки за кілька тижнів система була доведена до стану, коли масові скарги припинилися, і начебто стало можливим відносно нормальне життя.

І тоді, як я вже розповідав у Вступі, ми всі звільнилися. А фірма зіткнулася зі справжньою кризою: їй довелося наймати нових розробників, аби покінчити зі скаргами клієнтів, що не припинялися.

Хто винен у цьому фіаско? Звісно, проблема частково була створена стилем керівництва Френка. Тактика загального залякування заважала йому усвідомлювати реальний стан речей. Зрозуміло й те, що Білл і Джаліль мали протистояти тиску Френка набагато активніше. І наш керівник команди не повинен був погоджуватися на вимогу видати продукт до п'ятниці. Та й мені варто було наполягати на своєму «ні» замість того, щоби погоджуватися з лідером команди.

Професіонали говорять правду тим, хто наділений владою. Їм не бракує сміливості, щоби відповісти «ні» начальству.

Але як сказати «ні» керівникові? Це ж ваш *начальник*! Хіба ви не повинні робити те, що наказує начальник?

Ні! Кажіть «ні», якщо ви професіонал.

Це рабам забороняється говорити «ні». Наймані працівники теж неохоче кажуть «ні». Але професіоналу *належить* говорити «ні» у певних випадках. Ба більше: успішним менеджерам украй потрібні люди, у яких стає сміливості сказати «ні».

Тільки так можна справді чогось досягти.

Протилежні ролі

Одному з рецензентів книжки цей розділ активно не сподобався. Він сказав, що через нього ледь не відклав книжку. Йому доводилося створювати команди, у яких не було антагоністичних відносин, усі члени групи працювали разом у злагоді та без будь-якої конфронтації.

Я радий за цього рецензента, але не впевнений у тому, що його команди дійсно були позбавлені конфронтації настільки, наскільки це йому здавалося. А якщо так — не маю впевненості, що вони були цілком ефективними. Із власного досвіду мені відомо, що важкі рішення краще приймати виходячи з конфронтації протилежних ролей.

Керівники — люди, які мають свої обов'язки, і більшість керівників знають, як виконувати свою роботу належним чином. Частина цієї роботи полягає в тому, щоби якомога жорсткіше переслідувати та обстоювати *свої* цілі.

Так само й розробники — люди, які мають свої обов'язки, і більшість із них знає, як виконувати свою роботу належним чином. І якщо вони належать до справжніх професіоналів, то якомога жорсткіше переслідуватимуть і захищатимуть *власні* цілі.

Коли керівник говорить вам, що сторінка входу в систему має бути готова до завтрашнього дня, він переслідує одну зі своїх цілей. Тобто виконує свою роботу. Якщо ви добре знаєте, що зробити сторінку до завтрашнього дня неможливо, то, відповідаючи: «Добре, я спробую», ви не виконуєте свою роботу. Виконати її у цей момент можна лише одним-єдиним способом — заявити: «Ні, це неможливо».

Але хіба ви не маєте виконувати розпорядження начальства? Ні, насправді ваш начальник розраховує на те, що ви захищатимете свої цілі так само жорстко, як він захищає свої. І таким чином ви вдвох прийдете до *оптимального рішення*.

Оптимальним результатом є мета, загальна для вас і вашого керівника. Фокус у тому, щоби визначити цю мету, а для цього зазвичай потрібні перемовини.

Перемовини можуть виявитися цілком приємними:

> Майк: Пауло, сторінка входу в систему потрібна мені до завтрашнього дня.
>
> Паула: Ого! Вже завтра? Добре, я спробую.
>
> Майк: Чудово, дякую!

Ввічлива розмова, жодної конфронтації. Обидві сторони розійшлися з усмішками на обличчях. Дуже мило.

Але при цьому обидві сторони поводилися непрофесійно. Паула чудово знає, що робота над сторінкою займе більше одного дня, тому вона просто бреше. Можливо, вона не вважає свої слова брехнею. Можливо, вона *справді хоче спробувати* і сподівається, що якимсь дивом їй вдасться виконати обіцянку. Але зрештою вона все одно бреше.

З іншого боку, Майк вважає її «я спробую» за «так». І це просто безглуздо: він повинен знати, що Паула намагається уникнути конфронтації, тому мусить виявити наполегливість і запитати: «Мені здається, чи в тебе дійсно є сумніви? Ти впевнена, що зможеш зробити сторінку до завтрашнього дня?»

Або ще одна приємна розмова:

> Майк: Пауло, сторінка входу в систему потрібна мені до завтрашнього дня.
>
> Паула: Пробач, Майку, але мені знадобиться більше часу.
>
> Майк: Як гадаєш, коли вона буде готова?
>
> Паула: Як щодо двох тижнів — нормально?
>
> Майк (записує щось у щоденнику): Добре, дякую.

Приємна, але дуже неефективна і цілком непрофесійна розмова. Обидві сторони зазнали невдачі в пошуках оптимального результату. Замість того, щоби питати, чи влаштують Майка два тижні, Паула мала висловитися ствердно: «Мені знадобиться на це два тижні, Майку».

З іншого боку, Майк сприйняв запропонований термін без питань, ніби його особисті цілі не мають жодного значення. Цікаво, чи не збирається він просто повідомити свого начальника, що демонстрацію демоверсії програми замовникові доведеться відкласти через Паулу? Така пасивно-агресивна поведінка морально погана.

В обох випадках жодна зі сторін не має спільної мети. Жодна зі сторін не намагалася досягти оптимального результату.

Давайте спробуємо наступний варіант:

> Майк: Пауло, сторінка входу до системи потрібна мені до завтрашнього дня.

> Паула: Ні, Майку, тут роботи на два тижні.
>
> Майк: Два тижні? За оцінками проєктувальників, вона мала забрати три дні, а минуло вже п'ять!
>
> Паула: Проєктувальники помилилися, Майку. Вони видали свою оцінку до того, як служба маркетингу сформулювала остаточні вимоги. У мене залишилось роботи ще на десять днів. Ти бачив мої оновлені оцінки на вікі?
>
> Майк (із суворим виглядом, незадоволеним голосом): Це неприпустимо, Пауло! Завтра я представлятиму клієнтам демоверсію, тож маю продемонструвати, що сторінка входу працює.
>
> Паула: Яка частина сторінки входу має працювати до завтрашнього дня?
>
> Майк: Мені потрібна сторінка входу! Я мушу мати можливість увійти в систему.
>
> Паула: Майку, я можу зробити макет сторінки входу, що дозволить входити до системи. Нині її найпростіший варіант уже працює. Макет не перевіряє ім'я користувача та пароль і не надсилає забутий пароль електронною поштою. У верхній частині немає банера із фірмовим логотипом, не працює кнопка звернення до довідкової системи і підказки. Сторінка не зберігає cookie для наступного входу й не встановлює обмежень для доступу. Але ввійти до системи ви зможете. Це тебе влаштує?
>
> Майк: Отже, вхід працюватиме?
>
> Паула: Так, працюватиме.
>
> Майк: Чудово, Пауло, ти мене врятувала! (Відходить із задоволеним виглядом.)

Сторони досягли оптимального результату. Для цього вони спершу сказали «ні», а потім виробили взаємоприйнятне рішення. Вони діяли як професіонали. У розмові був елемент конфронтації і кілька незручних моментів, але це неминуче, коли дві людини наполегливо переслідують різні цілі.

А як щодо «чому»?

Можливо, ви думаєте, що Паулі слід детальніше пояснити, чому робота над сторінкою потребує набагато більше часу. Із власного

досвіду можу сказати, що причини менш важливі, ніж сам факт. А факт полягає в тому, що на сторінку знадобиться два тижні. Чому саме два тижні — це вже має другорядне значення.

Водночас пояснення може допомогти Майку зрозуміти (а отже, й ухвалити) цей факт. І якщо Майк має технічну кваліфікацію і темперамент для розуміння, такі пояснення можуть виявитися корисними. З іншого боку, Майк може не погодитися з висновками Паули. Можливо, він вирішить, що вона все робить неправильно. Він може сказати, що їй не потрібен такий обсяг тестування або рецензування, або що, припустимо, крок 12 можна виключити з опису. Занадто багато подробиць мають тенденцію до перетворення на мікроменеджмент.

Високі ставки

Казати «ні» найважливіше тоді, коли ставки високі. Що вища ставка, то більша цінність вимовленого «ні».

Здавалося б, очевидне твердження. Якщо ризик настільки великий, що від успіху залежить виживання компанії, ви повинні без жодних вагань надати керівництву найточнішу інформацію. А це часто означає відмову.

> Дон, директор із розробки: Отже, за нашими поточними прогнозами проєкт «Золотий Гусак» буде завершено за 12 тижнів від сьогоднішнього дня з похибкою плюс-мінус 5 тижнів.
>
> Чарлз, виконавчий директор *(чверть хвилини сидить мовчки, поступово багровіє)*: Тобто ти хочеш сказати, що ми запізнюємося на 17 тижнів?
>
> Дон: Так, імовірно.
>
> Чарлз *(встає, Дон підводиться на мить пізніше)*: Хай йому грець, Доне! Усе мало бути готове ще три тижні тому! Замовник із *Galitron* телефонує щодня і питає, де його чортова система. І я маю сказати їм, що доведеться чекати ще чотири місяці? Запропонуй щось краще!
>
> Дон: Чаку, я тобі казав три місяці тому, одразу після реорганізації, що нам знадобиться ще чотири місяці. Я маю на увазі те, що ти скоротив мій штат на двадцять відсотків! Чи повідомив ти тоді *Galitron*, що ми затримаємо здачу продукту?
>
> Чарлз: Ти чудово знаєш, що не повідомив. Ми не можемо дозволити собі втратити це замовлення, Доне. *(Чарлз робить паузу, блідне.)* Без *Galitron* нам кришка. Ти ж це знаєш, еге ж? А тепер, після цієї затримки, я боюсь… Що мені казати раді директорів? *(Знову сідає в крісло, намагаючись зберегти самовладання.)* Доне, ти маєш щось вигадати!
>
> Дон: Я нічого не можу зробити, Чаку. Ми це вже обговорювали. *Galitron* не збирається скорочувати вимоги і не погоджується на проміжні версії. Вони хочуть, аби інсталяція проводилася лише один раз, і на цьому все закінчувалося. Я просто не зможу це зробити швидше. Нічого не вийде.
>
> Чарлз: Чорт забирай! Навіть якщо я скажу, що від цього залежить твоя робота?
>
> Дон: Якщо мене звільнять, оцінка не зміниться.
>
> Чарлз: Усе, розмову завершено. Повертайся до своєї команди і ретельно стеж за тим, аби проєкт рухався. А мені потрібно зробити кілька неприємних дзвінків…

Звісно, Чарлз мав повідомити все це керівництву *Galitron* ще три місяці тому, коли він уперше почув новий прогноз. Принаймні зараз він чинить правильно, повідомляючи інформацію замовникові (і раді директорів). Але якби Дон не настояв на своєму, ці дзвінки могли бути відкладені на ще пізніший термін.

Бути командним гравцем

Усі ми чули, наскільки важливо «бути командним гравцем». Це означає, що ви виконуєте свої функції настільки добре, наскільки це можливо, і допомагаєте своїм колегам, якщо вони вскочуть у халепу. Людина, яка вміє працювати в команді, постійно спілкується з іншими, звертає увагу на своїх колег та старається якомога краще виконувати свої обов’язки.

Уміння працювати в команді зовсім не означає, що ви повинні погоджуватися з усіма.

Розгляньмо наступну ситуацію:

Паула: Майку, у мене свіжі прогнози. Група згодна з тим, що демоверсія буде готова за вісім тижнів плюс-мінус один тиждень.

Майк: Пауло, ми вже запланували здачу демоверсії, це має відбутися за шість тижнів.

Паула: Навіть не спитавши нашої думки? Майку, ти не можеш звалити все це на нас!

Майк: Це вже зроблено.

Паула *(зітхає)*: Гаразд, я повернуся до команди і подивлюся, що ми зможемо видати за шість тижнів, але це буде не вся система. Деякі функції будуть відсутні, а завантаження даних — неповним.

Майк: Пауло, замовник хоче побачити повну демоверсію.

Паула: Цього не буде, Майку.

Майк: Дідько! Гаразд, зроби перелік того, що ви зможете зробити, і передай мені завтра вранці.

Паула: Добре, зроблю.

Майк: Чи не можна якось прискорити роботу? Може, варто працювати ефективніше, більш креативно?

Паула: Куди вже ефективніше, Майку! Ми добре знаємо завдання, і на його реалізацію нам знадобиться вісім чи дев'ять тижнів, але не шість.

Майк: Можна попрацювати понаднормово.

Паула: Від цього все тільки сповільниться. Пам'ятаєш, що сталося останнього разу, коли ми змусили людей працювати понаднормово?

Майк: Так, але цього разу не станеться.

Паула: Усе буде так само, Майку, повір мені. Нам потрібно вісім чи дев'ять тижнів, але не шість.

Майк: Добре, склади план, але постійно міркуй, як упоратися за шість тижнів. Напевно, щось можна вигадати.

Паула: Ні, Майку, не можна. Я можу скласти план на шість тижнів, але там не буде багатьох важливих функцій і даних. Лише так — і ніяк інакше.

Майк: Домовилися, але я впевнений, що ви зможете створити диво, якщо спробуєте.

(Паула йде, хитаючи головою.)

Пізніше, на зустрічі в директора:

Дон: Отже, Майку, замовник розраховує отримати демоверсію за шість тижнів. І сподівається, що все працюватиме.

Майк: Так, усе буде готове. Мої люди проводять за комп'ютерами дні та ночі, і ми впораємося. Можливо, доведеться попрацювати понаднормово й виявити творчий підхід, але ми це зробимо!

Дон: Як добре, що ви розумієте загальні інтереси!

Хто ж *насправді* «розуміє загальні інтереси» у цій ситуації?

Паула діє у спільних інтересах, бо вона, у міру своїх можливостей, повідомляє, що можна, а що не можна зробити. Вона жорстко захищає свою позицію, незважаючи на вмовляння та скиглення Майка. Що ж до Майка, то він діє на користь команди з однієї людини — самого Майка. Очевидно, що він не думає про інтереси Паули, бо щойно пообіцяв замість неї зробити те, чого вона зробити не зможе. Майк також не бере до уваги інтереси Дона (хоча сам він із цим не погодився би), оскільки бреше директорові.

Чому ж Майк так чинить? Він хоче, щоби Дон бачив у ньому людину, яка розуміє інтереси компанії, а також вірить у свою здатність маніпулювати Паулою заради досягнення шеститижневого терміну. Однак не треба вважати Майка злим та зіпсованим; він просто занадто впевнений, що йому вдасться змусити інших людей робити те, що йому треба.

Намагання

Найгірше, що може зробити Паула у відповідь на маніпуляції Майка,— сказати: «Добре, я спробую». Не хочу приплітати сюди магістра Йоду, але в даному разі він має рацію: «*Не треба намагатися*».

Вам не подобається ця думка? Може, ви вважаєте, що намагатися щось зробити корисно? Адже Колумб не відкрив би Америку, якби не спробував?

Слово «*спробувати*» багатозначне. У цьому разі я маю на увазі значення «докласти додаткових зусиль». Які додаткові зусилля може докласти Паула, щоби демоверсія була готова вчасно? Якщо це можливо, то складається враження, що її команда раніше працювала не на повну силу. Мабуть, вони тримали додатковий резерв до цієї миті[1].

Обіцяючи «спробувати», ви визнаєте, що раніше стримувалися, що у вас у наявності додатковий резерв, яким можна скористатися. Ви зізнаєтеся також у тому, що мета може бути досягнута за допомогою додаткових зусиль. Ба більше: ви фактично берете на себе зобов'язання застосувати ці додаткові зусилля для досягнення мети. Отже, обіцяючи *спробувати*, ви обіцяєте *досягти успіху*. І таким чином звалюєте на себе важкий тягар. Якщо спроба не призведе до бажаного результату, це розглядатиметься як провал.

Чи маєте ви додаткове незадіяне джерело енергії? І навіть якщо маєте, чи зможете досягти мети? А може, просто створюєте умови для майбутнього провалу?

Обіцяючи спробувати, ви обіцяєте змінити свої плани. Попередніх планів виявилося замало. Обіцяючи спробувати, ви кажете, що у вас є новий план. Що це за план? Як ви мусите змінити свою поведінку? Що збираєтеся робити інакше тепер, коли «намагаєтеся»?

Якщо ви не маєте іншого плану, якщо не зміните свою поведінку, якщо все йтиме так само, як і до вашої обіцянки, то що тоді означає це «спробуємо»?

Якщо ви не тримаєте частини можливостей у резерві, якщо у вас немає нового плану, якщо ви не маєте наміру змінити свою поведінку й цілком упевнені у вихідній оцінці, то ваша обіцянка «спробувати» є докорінно непорядною. Ви *брешете*. І, мабуть, ви робите це для того, щоби «зберегти обличчя» й уникнути конфронтації.

Підхід Паули був кращим. Вона знов і знов нагадувала Майкові, що вихідна оцінка була невизначеною. Вона весь час наполягала на «восьми чи дев'яти» тижнях, підкреслювала існуючу невизначеність і жодного разу не відступила. Вона жодного разу не казала, що якісь додаткові зусилля, новий план чи зміна поведінки здатні зменшити цю невизначеність.

[1] Як казав Фогхорн Легхорн: «Я завжди тримаю свої пір'я пронумерованими саме для такої надзвичайної ситуації». (Фогхорн Легхорн — півень, персонаж, що з'являється у 29 мультфільмах, створених студією «Warner Bros. Animation» у 1946–1964 рр.)

Три тижні по тому:

Майк: Пауло, за три тижні здаємо демоверсію. Замовники хочуть подивитися, як працює пересилання файлів.

Паула: Майк, це не входило до узгодженого списку функцій.

Майк: Я знаю, але вони вимагають.

Паула: Добре, тоді з демоверсії доведеться викинути єдиний вхід або резервне копіювання.

Майк: У жодному разі! Вони хочуть бачити і ці функції!

Паула: Тобто вони хочуть повністю реалізовану функціональність? Ти це маєш на увазі? Я ж казала, що це неможливо.

Майк: Пробач, Пауло, але замовник не поступається. Вони хочуть побачити все й одразу.

Паула: Цього не буде, Майку. Просто не буде.

Майк: Та гаразд, Пауло, можна ж хоча би *спробувати*?

Паула: Майку, я можу *спробувати* літати повітрям. Можу *спробувати* перетворити свинець на золото. Можу *спробувати* перепливти Атлантичний океан. Як гадаєш, у мене вийде?

Майк: Послухай, я ж не вимагаю неможливого.

Паула: Ні, Майку, саме вимагаєш.

(Майк усміхається, киває та відвертається, збираючись піти.)

Майк: Я вірю в тебе, Пауло, і знаю, що ти мене не підведеш.

Паула *(у спину Майкові)*: Майку, ти мене чуєш? Це погано скінчиться.

(Майк просто махає рукою, не обертаючись.)

Пасивна агресія

Паула мусить ухвалити цікаве рішення. Вона підозрює, що Майк не розповідає Донові про її оцінки. Виходячи із цього, вона може просто дозволити Майку рухатися обраним шляхом і зрештою впасти у прірву. Їй доведеться подбати лише про те, щоби в їхньому листуванні були зареєстровані всі копії відповідних службових записок.

Коли вибухне катастрофа, Паула зможе продемонструвати всім і кожному, *що* й *коли* вона казала Майку. Це пасивно-агресивна тактика — дозволити Майку самому накинути на себе петлю.

Також Паула може спробувати запобігти катастрофі, звернувшись до Дона безпосередньо. Безумовно, є ризик, але у цьому полягає суть розуміння інтересів команди. Коли прямо на вас мчить вантажний потяг і ніхто, крім вас, його не бачить, у вас є вибір: або мовчки відійти вбік і подивитися, як він задавить решту людей, або закричати: «Потяг! Негайно забирайтеся геть!»

Два дні по тому:

Паула: Майку, ти повідомив Донові про мої оцінки? Він довів до відома замовника, що в демоверсії не працюватиме функція пересилання файлів?

Майк: Пауло, ти ж казала, що зробиш це для мене.

Паула: Ні, Майку, я цього не казала. Навпаки — я наполягала на тому, що це неможливо. Ось копія службової записки, що я надіслала тобі після нашої розмови.

Майк: Так, але ти сказала, що спробуєш, хіба ні?

Паула: Ми це вже обговорювали, Майку. Пам'ятаєш: свинець і золото?

Майк *(зітхає)*: Послухай, Пауло, це дуже потрібно. Просто нагально. Будь ласка, роби що завгодно, але ти просто зобов'язана додати цю функціональність вчасно.

Паула: Майку, ти помиляєшся. Я зобов'язана зробити щось інше — повідомити Донові про справжній стан справ, якщо цього не зробиш ти.

Майк: Це порушення субординації, так не можна.

Паула: А я і не хочу нічого порушувати, Майку, але ти мене змушуєш.

Майк: Ох, Пауло…

Паула: Послухай, Майку, ми не встигнемо реалізувати всю функціональність демоверсії вчасно. Зрозумій це нарешті. І припини вмовляти мене працювати більше та напруженіше. Припини обманювати себе, сподіваючись, що я якимсь дивом витягну кролика з капелюха. Зрозумій: ти повинен повідомити про це Донові, і зробити це прямо *сьогодні*.

Майк *(із виряченими очима)*: Сьогодні?

Паула: Так, Майку, сьогодні. А тоді ти, я і Дон улаштуємо зустріч, на якій обговоримо функціональність, що ввійде до демоверсії. Якщо ця зустріч не відбудеться, я буду змушена особисто звернутися до Дона. Ось копія службової записки, де все це пояснюється.

Майк: Ти просто прикриваєш себе!

Паула: Майку, насправді я намагаюся прикрити *нас обох*. Ти взагалі уявляєш, що станеться, коли замовник з'явиться тут, очікуючи побачити повну демоверсію, а ми її не зможемо пред'явити?

Чим закінчиться історія Паули і Майка? Розгляньте можливі варіанти самі. Суть у тому, що Паула поводилася цілком професійно. Вона казала «ні» у правильно вибрані моменти, і робила це належним чином. Вона відповіла «ні», коли на неї тиснули, щоби вона змінила свою оцінку. Вона відповіла «ні» на всі спроби маніпуляції, лестощів і благань. І, що найважливіше,— відповіла «ні» самообману та бездіяльності Майка. Паула діяла на користь команді. Майкові була потрібна допомога, і вона використала все, що було в її можливостях, аби йому допомогти.

Вартість вимовленого «так»

Найчастіше, ми вважаємо за краще казати «так». І справді — у згуртованій команді люди намагаються знайти шлях до згоди. Керівники та розробники в добре керованих командах ведуть переговори доти, доки не вироблять взаємоузгоджений план дій.

Але, як ми вже бачили, іноді, аби отримати «так», варто не боятися вимовити «ні».

Візьмемо до прикладу наступну історію, опубліковану Джоном Бланком у його блозі[1]. Вона відтворена тут із його дозволу. Під час ознайомлення спитайте себе, коли і як йому належало сказати «ні».

[1] Див.: http://raptureinvenice.com/?p=63

ХОРОШИЙ КОД СТАВ МІФОМ?

У юнацькому віці ви вирішуєте стати розробником. У старших класах ви навчаєтеся писати код за принципами об'єктно-орієнтованого програмування. У коледжі ви застосовуєте засвоєні принципи в таких галузях, як штучний інтелект і 3D-графіка.

А після вступу в коло професіоналів для вас починається нескінченна робота з написання якісного на комерційному рівні, простого в супроводі «ідеального» коду, що має витримати випробування часом.

Якість комерційного рівня? Уф! Це доволі кумедно.

Я вважаю, що мені щастить. Я люблю патерни проєктування. Мені подобається вивчати теорію ідеального програмування. Я запросто можу вести годинну дискусію із приводу невдалого вибору ієрархії наслідування моїм партнером по XP, наполягаючи, що відносини типу HAS-A у багатьох ситуаціях кращі за IS-A. Але останнім часом мені не дає спокою одне питання.

Чому в сучасному програмуванні став неможливим по-справжньому хороший код?

Типова проєктна пропозиція

Працюючи за контрактом, я проводжу свої дні (і ночі) за розробкою мобільних програм для численних клієнтів. І за ті багато років, що я цим займаюся, я зрозумів, що саме вимоги роботи «на замовника» не дозволяють мені писати по-справжньому якісні програми — такі, які мені хотілося би створювати.

Перш ніж я почну, дозвольте запевнити вас, що мені не можна дорікнути в нестачі наполегливості. Мені подобається тема чистого коду. Я не знаю нікого, хто би прагнув ідеальної архітектури програмного продукту так само, як я. Проблема криється у виконанні, і не з тієї причини, про яку ви зараз подумали.

Дозвольте розповісти вам історію.

Наприкінці минулого року одна добре відома компанія провела конкурс на розробку застосунку. Ця компанія — дуже великий оператор роздрібної торгівлі; для збереження конфіденційності назвемо її *Gorilla Mart*. Представники замовника зазначили, що їм потрібно активізувати свою присутність в області застосунків для iPhone, а замовлений застосунок повинен бути готовий до «чорної п'ятниці». У чому полягає проблема? Сьогодні вже перше листопада. На створення програми залишаються чотири тижні. А ще й *Apple* зазвичай вимагає до двох тижнів на затвердження програм. Виходить, програма має бути написана… ЗА ДВА ТИЖНІ?!

Так. У нас два тижні на створення програми. І, на жаль, наша заявка перемогла. (У бізнесі фігура замовника відіграє важливу роль, ніде подітися.)

> «Нічого страшного,— каже Керівник № 1 із *Gorilla Mart*,— застосунок нескладний. Усе, що нам потрібно,— показати користувачеві кілька продуктів із нашого каталогу й дати йому можливість побачити адреси найближчих магазинів. На нашому сайті це вже працює. Ми дамо вам готову графіку. Ймовірно, ви зможете використати… Як це називається? А, так — жорстке кодування (hardcode)!»
>
> У розмову вступає Керівник № 2: «І ще нам знадобляться купони на знижки, які користувач зможе пред'являти на касі. Відверто кажучи, ця програма пишеться "на один раз". Тож давайте зробимо її, а потім для наступної фази буде "з нуля" написана інша, більша та краща».

Так воно й відбувається. Незважаючи на роки наполегливих нагадувань про те, що кожна необхідна замовнику функціональність завжди виявляється складнішою, ніж здається з його пояснень, ви погоджуєтесь. Ви дійсно вірите, що цього разу все буде зроблено за два тижні. Так! Ми впораємося! Цього разу все буде інакше! Декілька зображень та мережевих операцій для отримання адрес магазинів. XML! Це ж так просто! Поїхали!

Але одного дня виявляється достатньо, щоби я знову повернувся до реальності.

> Я: Отже, дайте мені інформацію, потрібну для комунікації з вебслужбою, щоби я міг отримати адреси магазинів.
>
> Замовник: А що таке вебслужба?
>
> Я: …………….

Саме так воно й відбувалося. Дані про адреси магазинів, що виводилися у правому верхньому куті їхнього вебсайту, надавалися зовсім не вебслужбою — вони генерувалися кодом Java. Але не з API. Ще й хостинг забезпечувався стратегічним партнером *Gorilla Mart*.

А ось і мерзенна «третя сторона»…

У тій ситуації мені вдалося отримати від *Gorilla Mart* лише поточний список магазинів у вигляді файлу Excel. Код пошуку довелося писати «з нуля».

Пізніше в той самий день я отримав наступний удар: замовник забажав, аби дані про продукти і купони могли щотижня змінюватися. Про жорстке кодування даних можна забути! Виходить, за два тижні доведеться написати не лише додаток для iPhone, а й сервер на PHP, потім інтегрувати його з… Що? Контролем якості теж мені доведеться займатися?

Аби компенсувати збільшений обсяг роботи, нам доведеться програмувати трошки швидше. Забудьте про патерн «Абстрактна фабрика», використовуйте великий та потворний цикл `for` замість композиції, бо ж немає часу!

Отак хороший код поступово ставав неможливим.

Два тижні до завершення

Запевняю вас, попередні два тижні виявилися досить паршивими. Перші два дні були втрачені через багатогодинні зустрічі щодо наступного проєкту. Зрештою на роботу в мене залишилося 8 днів. У перший тиждень я пропрацював 74 години, а наступного… Боже, я навіть не пам'ятаю, це стерлося з моїх синапсів. І напевно, це на краще.

Я провів ці вісім днів за шаленим програмуванням. Я використав усі можливі засоби, щоби впоратися зі своєю роботою: копіювання/вставку (*aka* повторне використання коду), «чарівні числа» (щоб уникнути дублювання визначень констант з їх наступним — о жах! — повторним введенням) — і ЖОДНИХ модульних тестів! (Кому потрібні червоні маркери в такий час, вони тільки відбивають бажання працювати!)

Код вийшов доволі поганим, а я не мав часу на рефакторинг. Утім, для таких термінів він був не дуже поганий — адже все одно писався «на викид», так? Щось із цього здається вам знайомим? Зачекайте, далі буде ще цікавіше.

Накладаючи остаточні штрихи (перш ніж перейти до написання серверного коду), я почав поглядати на кодову базу з думкою, що, може, все не так уже й погано. Адже програма, зрештою, працює, а сам я вижив! Якби не той телефонний дзвінок.

> «Боб у нас працює зовсім недавно, він був дуже зайнятий і не міг зателефонувати раніше. А тепер він каже, що користувачі повинні вводити адреси своєї електронної пошти, щоб отримати купони. Він ще не бачив програму, але вважає, що це чудова ідея! Крім того, нам знадобиться система побудови звітів для отримання введених адрес із сервера. Добре придумано і не надто дорого. (Стривайте, останнє, здається, цілком позичено в «Монті Пайтон»!) І якщо вже мова зайшла про купони, вони повинні мати обмежений термін дії, а термін дії ми маємо призначати самі. Так, а ще…»

А тепер повернемось на крок назад. Що ми знаємо про хороший код? Хороший код має бути легко розширюваним. Простим у супроводі. Він має легко модифікуватися. Він має читатися, як проза. Так от, мій код не був хорошим.

І ще одне. Якщо ви хочете підвищити свою кваліфікацію як розробника, завжди пам'ятайте, що замовник регулярно збільшує обсяг роботи. Він завжди намагається додати у застосунок нові можливості. Він завжди хоче вносити зміни, і завжди — НА ПІЗНІЙ СТАДІЇ. Ось проста формула для підрахунку того, чого слід очікувати:

(кількість керівників)^2

+ 2 * кількість нових керівників

+ кількість дітей у Боба

= ДНІ, ДОДАНІ В ОСТАННЮ ХВИЛИНУ

Сучасні керівники — такі самі люди, як ми. Вони мусять забезпечувати сім'ї (якщо сатана дозволив їм завести сім'ю). Вони бажають, аби застосунок виявився успішним (це шанс для підвищення!). Але головна проблема полягає в тому, що вони намагаються претендувати на свою частку успіху в проєкті. Після того як усе буде сказано і зроблено, вони хочуть вказати на деяку функцію або архітектурне рішення, котре вони могли би вважати особистою заслугою.

Але повернемось до нашого проєкту. Ми додали ще кілька днів і реалізували введення адреси електронної пошти. А потім я зомлів від утоми.

Клієнти ніколи не переймаються так, як ви

Клієнти, незважаючи на всі їхні заяви про очевидну терміновість, ніколи не переймаються порушенням графіка більше, ніж ви. У день завершення роботи над програмою я розіслав повідомлення із фінальним варіантом застосунку всім ключовим учасникам. Керівникам, менеджерам і так далі. «ГОТОВО! Ось вам версія 1.0! СЛАВА ВСЕВИШНЬОМУ!»

Я натиснув кнопку «Відправити», відкинувся у кріслі і з самовдоволеною посмішкою почав уявляти, як замовники несуть мене на руках, а на 42-й вулиці проходить парад, де мене вінчають лаврами як найвидатнішого розробника всіх часів. Принаймні моя особа мала би бути на їхній рекламі, чи не так?

Хоч як дивно, але фірма-замовник не поспішала мене хвалити. Я взагалі не знав, що вони думають із цього приводу. Я не отримав від них жодної реакції. Жодного повідомлення. Скидалося на те, що керівництво *Gorilla Mart* вирішило, що цей етап уже позаду, і звернулося до наступного проєкту.

Гадаєте, я брешу? Тоді переконайтеся самі. Я подав заявку до *Apple* з незаповненим описом застосунку. Я запитав опис у *Gorilla Mart*, але ніхто мені не відповів, а чекати було ніколи (див. попередній абзац). Я знову написав. І знову отримав той самий результат. Тоді я підключив до цього наше керівництво. Двічі мені телефонували, і двічі я чув: «Нагадайте, що вам було потрібно?» МЕНІ БУВ ПОТРІБЕН ОПИС ЗАСТОСУНКУ!

Тиждень по тому почалося тестування програми інженерами *Apple*. Зазвичай це час радощів, але для мене то був час смертельного жаху. Як і очікувалося, через день програма була відхилена з найбільш жалюгідної та непереконливої причини, яку я тільки можу собі уявити: «У застосунку відсутній опис». Із функціональністю все гаразд, проте бракує опису. Саме тому застосунок для *Gorilla Mart* не був готовий до «чорної п'ятниці».

Мене це невимовно роздратувало. Я пожертвував спілкуванням із сім'єю заради двотижневого робочого марафону, а в *Gorilla Mart* за цілий тиждень ніхто не потурбувався створити навіть опис програми! Однак ми таки отримали той клятий опис — але через годину після відмови *Apple*.

І якщо до цього моменту я відчував звичайне роздратування, то за півтора тижні остаточно розлютився. Виявилось, що замовник не надав нам реальних даних. Продукти та купони на сервері були фіктивними. Або умовними, якщо хочете. Код купона дорівнював 1234567890 — тобто був просто взятий зі стелі. Фальшована нісенітниця.

А потім настав доленосний ранок, коли я зайшов на портал — І ТОЙ ЗАСТОСУНОК ВИЯВИВСЯ ДОСТУПНИМ! Разом із фіктивними даними і всім іншим! Я із жахом телефонував усім, кого згадав, і відчайдушно волав: «МЕНІ ПОТРІБНІ ДАНІ!» Жіночий голос запитав, із ким мене з'єднати — з рятувальниками чи з поліцією, і я кинув слухавку. Але потім я таки додзвонився до *Gorilla Mart* з отим своїм «МЕНІ ПОТРІБНІ ДАНІ!» І я ніколи не забуду їхньої відповіді:

> «Вітаю! Це Джон. У нас змінився віцепрезидент, і ми вирішили відмовитись від випуску застосунку. Видаліть його з App Store, гаразд?»

Зрештою, виявилось, що приблизно 11 людей зареєстрували свої адреси у базі даних. Це означало, що, теоретично, всі вони можуть заявитися до *Gorilla Mart* із фальшивим купоном. Це могло перетворитися на суцільний жах.

Загалом у всіх твердженнях замовника правдою було лише одне: той код справді писався «на викид». Єдина проблема полягала в тому, що він взагалі ніколи не був використаний.

Результат? Поспіх у розробці, повільний вихід на ринок

Мораль цієї історії: ключові учасники проєкту (зовнішній замовник або внутрішній керівник) придумали схему, яка мала би змусити розробників швидко писати код. Чи це ефективно? Ні. Швидко? Так. Ось як працює ця схема:

- **Повідомити розробникам, що застосунок дуже простий.** Це створює в команди спотворене уявлення про масштаби роботи. Розробники швидко беруться за роботу, а тим часом…
- **Функціональність проєкту розширюється, причому робоча команда вважається винною в тому, що не передбачила ту чи іншу потребу заздалегідь.** У нашому випадку жорстке кодування даних мало призвести до ускладнення оновлень. Як я міг цього не зрозуміти одразу? Згодом я зрозумів, але до того отримав брехливі обіцянки від замовника. Або інший варіант: замовник наймає «нову людину», яка знаходить якийсь очевидний недогляд. А що як завтра замовник повідомить, що вони найняли на роботу Стіва Джобса і потрібно додати алхімічні трансформації? А далі…
- **Далі учасників проєкту постійно підганяють, щоби робота була завершена вчасно.** Розробники працюють на максимальній швидкості (і з максимальним ризиком помилок, але хто звертатиме на це увагу?). До дедлайну залишається кілька днів? Але навіщо казати, що термін можна перенести, якщо робота

йде дуже продуктивно? Краще використати це у своїх інтересах! Потім, коли дедлайн настає, додається ще кілька днів, потім тиждень — і це після того, як ви щойно відпрацювали двадцятигодинну зміну, щоби все було завершено вчасно. Все як на знаменитій карикатурі з віслюком і морквиною — за винятком того, що з віслюком обходяться набагато краще, ніж з вами.

Схема добре продумана. Але чи варто звинувачувати її творців, упевнених у тому, що вона працює? Просто вони не бачать того кошмарного коду, який створюється в таких умовах. І так відбувається знов і знов, незважаючи на результати. В умовах глобалізованої економіки, коли корпорації тримаються тільки за долар, а підвищення котирування акцій пов'язується зі скороченням штатів, понаднормовою роботою та офшорною розробкою, описана стратегія економії на розробниках робить майже неможливим створення якісного коду. Якщо ми, розробники, не чинитимемо належного опору, то за допомогою прохань, наказів або погроз нас змусять писати вдвічі більший обсяг коду за вдвічі менший час.

Неможливий код

Коли у цій історії Джон запитує: «Хіба хороший код став неможливим?», насправді він ставить інше питання: «Хіба професіоналізм став неможливим?» Зрештою у цій сумній історії постраждав не лише код. Постраждали родина Джона, його роботодавець, замовники та користувачі. У програші виявилися *всі*[1].

І цей програш пояснюється непрофесіоналізмом. Хто тут діяв непрофесійно? Джон недвозначно натякає, що то були керівники *Gorilla Mart*. Схема, описана ним наостанок, ясно вказує на їхню непорядну поведінку. Але чи була ця поведінка поганою? Я так не вважаю.

Вони хотіли отримати застосунок для iPhone до «чорної п'ятниці». Вони були готові заплатити за нього. Вони знайшли когось, хто погодився взятися за цю роботу. То у чому їх звинувачувати?

Так, у спілкуванні явно виникали проблеми. І цілком очевидно, що керівники фірми-замовника не знали, що таке вебслужба. Це звичайна справа — один підрозділ великої корпорації знати не знає,

[1] За винятком, можливо, безпосереднього роботодавця Джона, хоча, тримаю парі, він також програв.

чим займається інший. Але все це слід передбачити. Джон навіть визнає це, коли пише: «Незважаючи на роки постійних нагадувань про те, що кожна необхідна замовнику функціональність завжди виявляється складнішою, ніж здається з його пояснень…»

Отже, якщо винуватцем була не *Gorilla Mart*, то хто?

Можливо, безпосередній роботодавець. Джон натякає на це в поблажливій фразі: «У бізнесі фігура замовника відіграє важливу роль». Може, роботодавець Джона давав нерозумні обіцянки керівникам *Gorilla Mart*? Або чинив тиск на Джона (прямий чи опосередкований), аби ці обіцянки виправдалися? Джон не говорить про це, тому нам залишається про це здогадуватися.

Але якщо так, то за що у цій історії відповідає сам Джон? Особисто я покладаю всю відповідальність виключно на нього. Саме Джон погодився на двотижневий термін, добре знаючи, що проєкти зазвичай виявляються складнішими, ніж здаються спочатку. Це він погодився написати серверну частину на PHP і на вимогу щодо додання реєстрації електронною поштою та введення обмежень термінів дії купона. Це Джон працював по двадцять годин на добу та по дев'яносто годин на тиждень і відмовився від своєї сім'ї та нормального життя, щоби не зірвати терміни здачі проєкта.

Чому Джон так вчинив? Він говорить про це цілком виразно: «Я натиснув кнопку "Відправити", відкинувся у кріслі і з самовдоволеною посмішкою почав уявляти, як замовники несуть мене на руках, а на 42-й вулиці проходить парад, де мене вінчають лаврами як найвидатнішого розробника всіх часів». Коротше кажучи, Джон забажав бути героєм. Він побачив шанс здобути славу і вхопився за нього.

Фахівці часто роблять героїчні справи, але не тому, що прагнуть бути героями. Професіонали стають героями, коли добре виконують свою роботу без порушень термінів і перевитрат бюджету. Прагнучи стати «героєм дня», Джон діяв непрофесійно. Він мав сказати «ні» у відповідь на запропонований йому двотижневий термін. А якщо не сказав — то мав зробити це, виявивши, що жодної вебслужби для отримання даних не існує. Він повинен був сказати «ні» у відповідь на вимогу додати реєстрацію електронної пошти та обмеження терміну дії купонів. Він повинен був сказати «ні» на все, що призводило до непотрібних жертв і перевитрат часу.

Але найголовніше — Джон мусив сказати «ні» своєму внутрішньому рішенню щодо того, що виконати роботу у зазначений термін можна лише одним способом — створивши плутанину в коді. Зверніть увагу на те, що Джон говорить про хороший код і модульні тести:

«Аби компенсувати збільшений обсяг роботи, нам доведеться програмувати трошки швидше. Забудьте про патерн «Абстрактна фабрика», використовуйте великий та потворний цикл `for` замість композиції, бо ж немає часу!»

І ще раз:

«Я провів ці вісім днів за шаленим програмуванням. Я використав усі можливі засоби, щоби впоратися зі своєю роботою: копіювання/вставку (aka повторне використання коду), «чарівні числа» (щоб уникнути дублювання визначень констант із їх наступним — о жах! — повторним введенням) — і ЖОДНИХ модульних тестів! (Кому потрібні червоні маркери в такий час, вони тільки відбивають бажання працювати!)»

Усі ці рішення і стали справжньою причиною катастрофи. Джон прийняв той факт, що досягти успіху можна лише через непрофесійну поведінку, й отримав те, що заслужив.

Можливо, це звучить надмірно суворо. Я цього не хотів. У попередніх розділах я описував, як неодноразово робив ту саму помилку у власній кар'єрі. Спокуса «виглядати героєм» або «тим, хто вирішує проблеми» — надто велика. Однак усі ми повинні розуміти, що відмова від професійних принципів не вирішує проблем, а лише створює їх.

З огляду на сказане, я нарешті можу відповісти на запитання Джона: «Хіба хороший код став неможливим? Хіба професіоналізм став неможливим?»

Ось моя відповідь: «Ні!»

3 КАЗАТИ «ТАК»

А ви знаєте, що це я винайшов голосову пошту? Присягаюся! Загалом нас, власників патенту на голосову пошту, було троє: Кен Файндер, Джеррі Фітцпатрік і я. Це було на початку 1980-х, коли ми працювали на компанію *Teradyne*. Виконавчий директор доручив нам створити продукт нового типу, і ми винайшли «електронного секретаря» (скорочено ЕС).

Усім зараз добре відомо, що таке «електронний секретар». Це одна з тих кошмарних машин, що відповідають на дзвінки в компаніях і ставлять всілякі ідіотські питання, на які потрібно відповідати натисканням кнопок («Аби перейти на англійську, натисніть 1»).

Наш ЕС відповідав на дзвінок і просив ввести ім'я потрібної людини, після чого викликав її через внутрішній зв'язок. ЕС повідомляв про виклик і запитував, чи з'єднати її з тим, хто телефонував. Якщо відповідь була позитивною, він установлював зв'язок і відключався.

Ви могли повідомити ЕС про своє поточне місцезнаходження. Також йому можна було вказати кілька телефонних номерів. Якщо ви перебували в іншому офісі, ЕС знаходив вас. Якщо ви були вдома, ЕС знову ж таки знаходив вас. Якщо ви перебували десь в іншому місті, ЕС також знаходив вас. А якщо в нього це все ж таки не виходило, він записував повідомлення для вас від того, хто вам телефонував. Саме в *Teradyne* було вперше застосовано голосову пошту.

Як не дивно, компанія *Teradyne* не спромоглася придумати, яким чином їй продати «електронного секретаря». Проєкт вийшов за рамки бюджету і був перетворений на систему CDS (Craft Dispatch System) для передачі фахівцям з ремонту телефонів інформації про наступне завдання. Також *Teradyne* відмовилася від патенту, не повідомивши нас про це (!). Наступний власник патенту подав заявку на три місяці пізніше за нас (!!)[1].

Після того як ЕС перетворився на CDS (але ще задовго до відмови від патенту), я чекав на виконавчого директора компанії, сидячи… на дереві. Перед фасадом офісу ріс великий дуб, я забрався на нього і чекав, поки під'їде його «Ягуар». Урешті-решт я перехопив його біля дверей і попросив приділити мені кілька хвилин — і він погодився.

[1] Не те, щоби цей патент обіцяв мені якісь гроші. Я продав його *Teradyne* за один долар — відповідно до мого трудового договору (і не отримав цей долар).

Я сказав, що проєкт ЕС потрібно запускати заново і що ми напевно зможемо заробити на ньому. Але співрозмовник здивував мене, заявивши: «Добре, Бобе, підготуй план. І покажи, як ми зможемо на цьому заробити. Якщо ти це зробиш, а я у твій план повірю, ми знову запустимо ЕС».

От цього я не очікував. Я передбачав, що він скаже: «Ти маєш рацію, Бобе. Давай запустимо проєкт, а я поміркую, як на цьому заробити». Але ні, він переклав цей тягар на мене. Не можу сказати, що це мене втішило — я ж все-таки розробник, а не фінансист. Мені хотілося працювати над проєктом ЕС, а не відповідати за прибутки та збитки. Але я не хотів виказувати розчарування, тому подякував йому і вийшов з офісу зі словами: «Дякую, Рассе, я неодмінно це зроблю… коли знайду вільний час».

А тепер дозвольте мені передати слово Рою Ошерову, який зараз розповість, якою жалюгідною виявилася ця заява.

Мова зобов'язань

Автор — Рой Ошеров

Сказати. Мати на увазі. Зробити.

Обіцянка складається із трьох частин.

1. Ви *кажете*, що ви це зробите.
2. Ви *маєте на увазі* те, про що казали.
3. Ви *виконуєте* обіцяне.

Але чи часто нам зустрічаються люди (звісно, це не ми з вами!), які виконують усі три частини?

- **Ви питаєте хлопця з IT-відділу**, чому мережа працює так повільно, він каже: «Ага, треба було би придбати нові маршрутизатори». І ви вже *знаєте*, що нічого зроблено не буде.
- **Ви просите учасника групи** провести ручне тестування перед складанням вихідного коду, а він відповідає: «Звісно. Постараюся зробити до кінця дня». І чомусь вам *здається*, що завтра необхідно поцікавитися, чи провів він тестування.

- **Начальник** входить в офіс, бурмочучи: «Нам треба працювати швидше». І ви *розумієте*, що насправді це особисто вам потрібно працювати швидше, бо сам начальник нічого робити не збирається.

Лише невелика частина людей, обіцяючи щось, відповідально ставляться до своїх слів і роблять те, що обіцяли. Дехто навіть щиро збирається виконати обіцянку, але нічого не робить. Набагато більше людей обіцяють, не збираючись хоч щось робити. Ви, мабуть, чули, як хтось із ваших знайомих каже: «Друже, мені дійсно слід схуднути»,— і при цьому ви переконані, що нічого робити він не буде. Такі речі відбуваються постійно.

Звідки ж виникає це дивне відчуття, що люди здебільшого надто легковажно ставляться до власних обіцянок?

Що гірше, нас часом підводить інтуїція. Інколи нам *хочеться* повірити, що співрозмовник відповідально ставиться до своїх слів, хоча насправді це не так. Ми *хочемо* вірити в те, що обіцяє загнаний у глухий кут розробник,— наприклад, що двотижневий проєкт буде завершений за тиждень… хоча насправді це теж не так.

Замість того, щоби покладатися на інтуїцію, можна чітко визначити за деякими мовними ознаками, наскільки відповідально люди ставляться до своїх слів. Правильно обираючи слова, можна позбавитися проблем із пунктами 1 і 2 наведеного переліку. Коли ми обіцяємо щось зробити, то мусимо відповідально ставитися до своїх слів.

Ознаки порожніх обіцянок

Ретельно вибирайте формулювання, які використовуєте у своїх *обіцянках*, тому що за *словами* часто можна судити про подальший перебіг подій. Якщо вам не вдається підібрати потрібні слова, найімовірніше, ви недостатньо відповідально ставитеся до сказаного або не вірите у його здійсненність.

Декілька прикладів слів і виразів, що є характерними ознаками порожніх обіцянок:

- **«Потрібно/повинен»:** «Нам треба це зробити якнайшвидше», «Мені слід схуднути», «Хтось повинен про це подбати».

- **«Сподіваюся/добре»:** «Сподіваюся, це буде зроблено до завтрашнього дня», «Сподіваюся, ми ще зустрінемося й обговоримо цю тему», «Добре би викроїти час для цього», «Було би добре, якби комп'ютер працював швидше».
- **«Давайте» (без «я»):** «Давайте зустрінемося пізніше», «Давайте доробимо цю штуку».

Якщо ви почнете звертати увагу на ці ключові слова, то помітите, як часто вони звучать довкола вас — і навіть тоді, коли ви самі щось кажете іншим.

Ви помітите, як наполегливо ми намагаємося уникнути відповідальності за сказане.

І це *неприпустимо* — покладатися на подібні обіцянки у своїй роботі. Утім, перший крок уже зроблено: ви почали дізнаватися про ознаки безвідповідального ставлення до обіцянок.

Отже, ми знаємо, як виглядають порожні обіцянки. А як розпізнати серйозну обіцянку?

Ознаки серйозних обіцянок

Усі висловлювання, наведені в попередньому підрозділі, мають щось спільне: вони припускають, що особисто ви жодним чином не впливаєте на те, що відбувається, і не несете за це персональну відповідальність. У більшості випадків люди поводяться так, наче вони є *жертвами ситуації*, а не контролюють її.

Але насправді *особисто ви* ЗАВЖДИ можете хоч якось вплинути на ситуацію, тому ви завжди можете відповідально пообіцяти *щось* зробити.

Певними ознаками серйозних обіцянок є вирази типу: «Я зроблю те й те… до такого часу» («Я завершу роботу над цим модулем до вівторка»).

Чим важлива ця фраза? *Ви стверджуєте, що щось зробите, називаючи конкретний час завершення.* Ви говорите не про когось іншого, а тільки про себе. Ви кажете виключно про те, що *зробите* особисто ви. Ви не *сподіваєтеся*, що це буде зроблено, і не уточнюєте «*…якщо буде час*», а демонструєте чіткий намір *виконати* обіцяне.

Давши подібне усне зобов'язання, ви не зможете відмовитись від нього без порушення обіцянки. Ви сказали, що щось зробите, і тепер залишається тільки один із двох варіантів: ви або зробите, або не зробите. Якщо не зробите, інші люди можуть цілком справедливо запитати, чого насправді варті ваші обіцянки. До того ж вам буде соромно повідомляти їм про невиконання обіцяного (якщо вони чули вашу обіцянку).

Неприємна перспектива, чи не так?

Даючи серйозну обіцянку, ви берете на себе повну відповідальність щодо обіцяного перед аудиторією, що складається мінімум з однієї людини. Ви стоїте не перед дзеркалом і не перед монітором комп'ютера. Ви розмовляєте з іншою людиною та обіцяєте щось зробити. Із цього починається відповідальність. Поставте себе в ситуацію, що змусить вас щось зробити.

Перехід на мову зобов'язань допоможе вам здолати шлях від відповідального ставлення до слів — до виконання обіцяного.

Утім, існує кілька причин, через які промовець може *не поставитися відповідально* до своїх обіцянок, бо ці причини перешкоджають їх виконанню.

Виконання обіцянки залежить від іншої людини

Обіцяйте лише те, що є під *повним* вашим *контролем*. Наприклад, якщо вашою метою є завершення модуля, у роботі над яким бере участь інша команда, ви не можете обіцяти завершити модуль із повною інтеграцією. Однак ви *можете* обіцяти виконати конкретні дії, що наблизять досягнення кінцевої мети. Ви можете:

- поспілкуватися годину-другу з Гері з інфраструктурної групи, щоби зрозуміти залежності;
- створити інтерфейс, що абстрагує залежності вашого модуля від інфраструктури іншої групи;
- зустрічатися не менше трьох разів на тиждень із відповідальним за створення застосунку для забезпечення працездатності ваших змін у системі складання, що використовується в компанії;
- створити власну процедуру складання, під час якої виконуються інтеграційні тести.

Відчуваєте різницю?

Якщо кінцева мета залежить від когось іншого, обіцяйте лише конкретні дії, що сприятимуть досягненню кінцевої мети.

Це не спрацює, бо я не впевнений, що це можна зробити

Навіть якщо досягнення кінцевої мети неможливе, ви можете взяти на себе зобов'язання стосовно виконання дій, що наближають її досягнення. Ба більше, перевірка досяжності мети може бути однією з таких дій!

Замість того щоби обіцяти виправити всі 25 помилок, що залишилися до виходу фінальної версії (що може виявитися неможливим), можна пообіцяти виконати конкретні дії, що наближають вас до цієї мети.

- Перебрати всі 25 помилок та спробувати відтворити їх.
- Поспілкуватися з автором кожного повідомлення про помилку та побачити її відтворення.
- Витратити весь час на виправлення помилок.

Це не спрацює, бо іноді я просто не встигаю

Що ж, буває. Могло статися щось непередбачене — життя залишається життям. Але ви все одно прагнете виправдати очікування. У такій ситуації краще *якнайшвидше* змінити очікування.

Якщо ви не можете виконати свою обіцянку, дуже важливо якнайшвидше повідомити про це тому, кому ви обіцяли.

Що швидше ви повідомите про невдачу всі зацікавлені сторони, то більша ймовірність того, що команда матиме час зупинитися, наново оцінити поточну ситуацію і вирішити, чи можна щось зробити або змінити (наприклад, пріоритети). Можливо, це дозволить виконати або змінити вихідні зобов'язання.

Кілька прикладів:

- Ви призначили зустріч у кафе з колегою, але застрягли у транспортній пробці. Ви сумніваєтеся, що вам вдасться виконати обіцянку вчасно опинитися на місці. Щойно ви зрозумієте, що запізнюєтеся, зателефонуйте колезі та повідомте йому про це. Можливо, ви знайдете інше місце чи відкладете зустріч.

- Ви пообіцяли виправити помилку, котра, на перший погляд, здавалася цілком звичайною. Якоїсь миті ви розумієте, що помилка виявилася набагато підступнішою за ваші уявлення про неї, і піднімаєте білий прапор. Далі команда може розробити перелік дій для виконання цього зобов'язання (робота в парах, опрацювання потенційних рішень, мозковий штурм) або змінити пріоритети і переключити вас на розв'язання простішої помилки.

Майте на увазі: якщо ви нікого не повідомите про потенційну проблему якнайшвидше, то ніхто не зможе допомогти вам у виконанні ваших зобов'язань.

Резюме

Сама ідея особливої «мови обіцянок» виглядає трохи дивно, але вона допоможе вирішити багато проблем спілкування, із якими розробники стикаються у своїй повсякденній роботі: прогнози, терміни, непорозуміння під час особистого спілкування. Вас вважатимуть серйозним розробником, який дотримується свого слова,— а це, можливо, одне з найкращих визначень, що існують у нашій галузі.

Як навчитися говорити «так»

Я попросив у Роя дозволу на публікацію цього допису, бо він зачепив мене за живе. Вище я довго пояснював, як навчитися казати «ні». Однак не менш важливо навчитися казати «так».

Зворотний бік «спроби»

Уявіть, що Пітерові доручено внести зміни до системи оцінювання. За його власною оцінкою ця робота вимагає п'ять-шість днів. Він також вважає, що підготовка документації щодо змін потребує кількох годин. У понеділок вранці Мардж, його начальниця, цікавиться поточним станом справ:

> Мардж: Пітере, чи будуть готові зміни в системі оцінювання до п'ятниці?

> Пітер: Я думаю, це можливо.
>
> Мардж: Разом із документацією?
>
> Пітер: Спробую її зробити.

Можливо, Мардж не відчуває нерішучості у заявах Пітера, але насправді він нічого не обіцяє. Мардж ставить такі запитання, що потребують чіткої відповіді «так» або «ні», а Пітер дає невизначені відповіді.

Зверніть увагу на слово «спробую». У попередньому розділі ми визначили його як «докладення додаткових зусиль», але тут Пітер використовує значення «може, вийде, а може, і ні».

Йому варто було би відповісти інакше:

> Мардж: Пітере, чи будуть готові зміни в системі оцінювання до п'ятниці?
>
> Пітер: Не виключено, але це може бути і понеділок.
>
> Мардж: Разом із документацією?
>
> Пітер: На документацію знадобиться ще кілька годин. Теоретично можу встигнути до понеділка, але може статися так, що документація буде готова лише у вівторок.

У цьому випадку Пітер говорить чесніше — він описує існуючу невизначеність. Можливо, Мардж вдасться щось зробити із цією невизначеністю. А можливо, й ні.

Дисципліноване прийняття зобов'язань

> Мардж: Пітере, мені потрібна чітка відповідь: «так» або «ні». Чи буде готова до п'ятниці система оцінювання разом із документацією?

Мардж ставить абсолютно правильне питання. Її мета — забезпечити дотримання графіка, тому їй потрібна однозначна відповідь щодо п'ятниці. Як повинен відповісти Пітер?

> Пітер: У такому разі я маю сказати «ні». Найбільш рання дата, коли зміни і документація будуть точно готові,— це вівторок.

Мардж: Ти обіцяєш зробити все до вівторка?

Пітер: Так, до вівторка все буде готове.

А якщо для Мардж аж надто важливо, щоби зміни і документація були готові саме до п'ятниці?

Мардж: Пітере, із вівторком у нас будуть великі проблеми. Віллі, наш штатний технічний письменник, звільниться в понеділок. Він матиме лише п'ять днів на підготовку інструкції для користувача. Якщо документація щодо системи оцінювання не буде готова в понеділок вранці, то він не зробить це вчасно. Ти не міг би спочатку скласти документацію?

Пітер: Ні, спочатку потрібно внести зміни, бо документація будується за вихідними даними тестових запусків.

Мардж: Може, зміни разом із документацією якось можна завершити до ранку понеділка?

Тепер Пітер має ухвалити рішення. Цілком можливо, що модифікація системи оцінювання буде завершена у п'ятницю; можливо, навіть документація буде готова ще до того, як він піде на вихідні. А якщо робота вимагатиме більше часу, ніж він вважає, можна виділити кілька годин у суботу. Що він має сказати Мардж?

Пітер: Знаєш, Мардж, якщо я попрацюю кілька годин у суботу, то, ймовірно, все буде готове вранці у понеділок.

Чи вирішує це проблему Мардж? Ні. Висловлювання Пітера просто змінює ступінь ймовірності успіху. Пітер мав би говорити інакше:

Мардж: То я можу розраховувати на ранок понеділка?

Пітер: Можливо, але гарантувати не можу.

Не виключено, що таке формулювання не влаштує Мардж.

Мардж: Послухай, Пітере, мені потрібна певність. Ти можеш *обіцяти*, що роботу буде закінчено до ранку понеділка?

У цей момент у Пітера виникає спокуса схалтурити. Можливо, робота буде завершена швидше, якщо обійтися без написання тестів. Або без рефакторингу. Можна також відмовитися від повного регресійного тестування. Саме в такі моменти і виявляється професіоналізм.

По-перше, всі ці припущення хибні. Робота не буде завершена швидше, якщо Пітер не напише тести. Вона не буде завершена швидше, якщо він не здійснить рефакторинг або пропустить повне регресійне тестування. Багаторічний практичний досвід учить нас тому, що порушення правил лише уповільнює роботу.

По-друге, професіонал має підтримувати певні стандарти. Його код має пройти тестування, тому без тестів не обійтися. Його код має бути чистим та елегантним. І він повинен переконатися, що зміни не порушують роботу інших компонентів системи. Пітер як професіонал уже дав зобов'язання дотримуватись цих стандартів. Усі інші зобов'язання відходять на другий план, тому про будь-які негідні витівки слід забути.

Пітер: Ні, Мардж, найраніший термін, який я дійсно можу гарантувати,— це вівторок. Вибач, якщо це порушує твої плани, але нам доводиться рахуватися з реальністю.

Мардж: Чорт, я дуже розраховувала, що ти впораєшся швидше. Ти цілком упевнений?

Пітер: Так, я впевнений, що робота може тривати до вівторка.

Мардж: Добре, доведеться поговорити з Віллі. Може, йому вдасться змінити свій графік.

У цьому випадку Мардж погодилася з відповіддю Пітера і звернулася до пошуку інших можливостей. А якщо всі можливості вичерпано? Якщо Пітер був останньою надією?

Мардж: Послухай, Пітере, я розумію, що тобі буде нелегко, але мені дуже потрібно, щоби все було готове до ранку понеділка. Це дуже важливо. Чи можна хоч щось зробити заради цього?

Тепер Пітеру доводиться думати про понаднормову роботу, на яку, ймовірно, буде витрачена більша частина вихідних. Він має абсолютно об'єктивно оцінити свою працездатність та резерви. Легко сказати — «попрацюю на вихідних»; набагато важче знайти в собі достатньо сил для якісного виконання роботи.

Професіонали знають межі своїх можливостей. Вони знають, який обсяг роботи можуть виконати понаднормово, і знають, чим за це

доведеться розплачуватися. У цьому випадку Пітер цілком упевнений, що кількох додаткових годин буде достатньо.

> Пітер: Добре, Мардж, ось що я скажу. Зараз я зателефоную додому та обговорю можливість роботи у вихідні зі своєю родиною. Якщо вони не проти, то все буде зроблено до ранку понеділка. Я навіть сам прийду в понеділок вранці і простежу, щоб у Віллі не виникло питань. Але потім я вирушу додому й відпочиватиму до середи. Домовилися?

Пітер говорить абсолютно чесно. Він упевнений у тому, що при понаднормовій роботі він упорається зі змінами та написанням документації. Він також знає, що кілька днів після цього від нього не буде жодної користі на роботі.

Висновки

Професіонал не повинен відповідати «так» на все, що від нього вимагають. Проте він має докласти максимум зусиль для того, щоби це «так» стало можливим. Коли професіонали говорять «так», вони використовують такі формулювання, що не викликають у співрозмовника жодних сумнівів у надійності їхніх обіцянок.

4 НАПИСАННЯ КОДУ

У попередній книжці[1] я описав структуру і природу *чистого коду*. У цьому розділі буде розглянутий сам *акт* написання коду, а також контекст, що його оточує.

[1] Мартін, Роберт С. Чистий код: створення, аналіз і рефакторинг. Харків, «Фабула», 2020.

Коли мені було 18 років, я досить швидко набирав текст, але мені доводилося дивитися на клавіші. Я не вмів друкувати наосліп. Одного вечора я провів кілька годин за перфоратором IBM 029, намагаючись не дивитися на клавіші під час набору програми, записаної на кількох формулярах. Після набору я перевірив усі перфокарти та викинув ті, що містили помилки.

Спочатку я помилявся досить часто. Але до кінця вечора набір ішов майже ідеально. За цей довгий вечір я зрозумів, що якість сліпого друку переважно залежить від *упевненості*. Мої пальці вже знали, де є потрібні клавіші; мені залишалося тільки набути впевненості, що я не помиляюся. Серед іншого мені допомагало те, що я відразу *відчував*, коли роблю помилку. До кінця вечора я майже миттєво розпізнавав помилки введення й одразу викидав зіпсовану перфокарту, навіть не дивлячись на неї.

Уміння інтуїтивно відчувати власні помилки дуже важливе. Не лише при наборі тексту, а й у всьому іншому. Наявність почуття помилки означає, що ви набагато швидше замикаєте цикл зворотного зв'язку і навчаєтеся на власних помилках. Я вивчав й успішно опановував чимало інших дисциплін з того дня, проведеного за IBM 029. І в усіх випадках ключем до майстерності виявлялися впевненість у собі та відчуття помилки.

У цьому розділі описаний мій особистий набір правил і принципів написання коду. Ці правила і принципи стосуються не самого коду, а моєї поведінки, настрою та відношень із колегами під час написання коду. Вони характеризують мій розумовий, моральний та емоційний контекст. У них коріняться мої впевненість і почуття помилки.

Можливо, ви не погодитеся із чимось зі сказаного. Зрештою, все це суто особистий досвід. Ба більше, якісь принципи можуть викликати у вас протест або відторгнення. І це нормально — я аж ніяк не пропоную абсолютних істин. Це надбання однієї людини, яка прагла стати професіональним розробником.

Не виключаю, однак, що ознайомившись із моїми особистими поглядами на написання коду, ви знайдете в них щось корисне і для себе.

Підготовка

Написання коду — інтелектуально складне і стомливе заняття. Існує небагато дисциплін, що потребують такого рівня концентрації та уваги. Причина полягає в тому, що при написанні коду вам доводиться «жонглювати» відразу кількома суперечливими факторами.

1. Насамперед ваш код має працювати. Вам належить розібратися в суті проблеми і зрозуміти, як її розв'язувати. Ви повинні переконатися, що написаний вами код адекватно репрезентує обране рішення. Ви мусите простежити за кожною дрібницею, не порушуючи правил мови, платформи, поточної архітектури та інших специфічних особливостей поточної системи.
2. Ваш код повинен вирішувати завдання, сформульоване замовником. Часто вимоги замовника не відображають усього, що необхідно для виконання завдання. Ви маєте зрозуміти це й обговорити із замовником, аби написаний код відповідав його справжнім потребам.
3. Ваш код має добре вписуватися в існуючу систему. Він не повинен призводити до підвищення її жорсткості, крихкості чи непрозорості. Вам потрібно добре організувати керування залежностями. Коротше кажучи, ваш код має відповідати принципам якісного проєктування[1].
4. Ваш код має нормально читатися і сприйматися іншими розробниками. Проблема не зводиться до написання виразних коментарів. Насамперед ви повинні будувати код у такий спосіб, щоби в його структурі виявлялися ваші наміри. Зробити це непросто. Цілком можливо, що це найскладніша навичка, яку мусить опанувати розробник.

«Жонглювати» всіма цими факторами важко. Підтримувати необхідну концентрацію та увагу протягом тривалого часу ще важче. Додайте до цього проблеми і подразники, пов'язані з роботою в команді та організації, а також піклування про повсякденне життя. Отже, відволікаючих факторів більш ніж достатньо.

[1] Martin, Robert C. *Agile Software Development: Principles, Patterns and Practices*. Upper Saddle River, NJ: Prentice Hall, 2003.

Якщо ж ви не в змозі цілком зосередитися на своїй роботі, то отримаєте поганий код. У ньому буде купа помилок. Він матиме хибну структуру, буде непрозорим і заплутаним, не відповідатиме справжнім потребам замовника. Коротше кажучи, код доведеться переробляти чи переписувати. Робота без зосередженості завжди марнотратна.

Якщо ви втомилися або не можете зосередитися, *не пишіть код*. Краще подумайте над тим, як усунути відволікаючі фактори і заспокоїти свій розум.

Нічне програмування

Найгірший мій код був написаний о третій годині ночі. Це було 1988 року, коли я працював у телекомунікаційному стартапі *Clear Communications*. Ми проводили багато часу за створенням «колективної праці». Звичайно, мріючи з часом розбагатіти.

Якось дуже пізно ввечері (точніше, дуже рано-вранці) — для розв'язання проблеми синхронізації я організував відправку моїм кодом повідомлення самому собі через систему диспетчеризації подій (ми називали це «надіслати пошту»). Рішення було *хибне*, але о 3 годині ночі воно здавалося страшенно привабливим. Справді, після 18 години безперервного програмування (не кажучи вже про 70-годинний робочий тиждень) мені нічого іншого на думку *вже* не спадало.

Пам'ятаю, я дуже пишався своєю багатогодинною роботою. Пам'ятаю, я відчував свою *цілеспрямованість*. Пам'ятаю, мені здавалося, що працювати о 3 години ночі — доля справжнього професіонала. Як же я помилявся!

Цей код, мов бумеранг, повертався до нас знову і знову. Він породив помилкову архітектуру, яку використовували всі та недоліки якої постійно доводилося маскувати обхідними рішеннями. Код приводив до появи дивних помилок синхронізації та незрозумілих циклів зворотного зв'язку. Одне повідомлення призводило до відправки іншого, потім ще одного — і так — до безкінечності. У нас завжди не було часу на те, щоб переписати цю халтуру (нам так здавалося), але в нас завжди вистачало часу для накладання чергової «латки», щоб обійти її. Сміття накопичувалося, а мій нічний код обростав

все більшими наслідками та побічними ефектами. Через багато років ця історія перетворилася на популярний жарт нашої групи. Коли я втомлювався чи впадав у відчай, колеги казали: «Дивіться! Боб збирається надіслати собі пошту!»

Мораль цієї історії: не пишіть код, коли ви втомилися. Відданість справі та професіоналізм виявляються у дисципліні, а не у тривалості роботи. Обов'язково слідкуйте за сном, здоров'ям та способом життя, щоб ви могли щодня присвятити роботі вісім *хороших* годин.

Програмування у засмучених почуттях

Ви колись намагалися писати код після серйозної сварки з дружиною чи приятелем? Чи помічали, що у цей час у вас у мозку запускається фоновий процес, який намагається розв'язати конфлікт чи, принаймні, пригадати його? Іноді тиск цього фонового процесу віддається у грудях або десь у животі. Ви відчуваєте занепокоєння, ніби випили занадто багато кави або коли. І це відволікає від роботи.

Коли я турбуюся через суперечку з дружиною, розбіжності із замовником чи хворобу дитини, я не можу сконцентруватися на роботі. Моя концентрація стає хиткою. Я ловлю себе на тому, що дивлюсь на екран, тримаючи руки на клавіатурі,— і нічого не роблю. Кататонія. Параліч. Я перебуваю за мільйон миль від робочого місця, цілком захоплений своєю фоновою проблемою, замість того, щоб займатися своєю справою.

Іноді мені вдається змусити себе *думати* про код. Я примушую себе написати один-два рядки. Зусиллям волі домагаюся проходження одного-двох тестів. Але я не можу робити так і далі, бо знову поринаю у стан заціпенілої бездушності і нічого не бачу широко відкритими очима, переживаючи власні «фонові неприємності».

Я дізнався, що писати код у такі моменти марно. Код, що я напишу, можна негайно відправляти на сміття. Замість роботи над кодом, я мушу спочатку розібратися зі своїми проблемами.

Звичайно, більша частина проблем не розв'язується за годину чи дві. До того ж, наші роботодавці навряд чи змиряться з нашою нездатністю працювати в той час, поки ми переживаємо особисті нега-

разди. Тому ви повинні навчитися завершувати фоновий процес або, принаймні, знижувати його пріоритет, аби він не відволікав вас постійно.

Я розв'язую цю проблему через розподіл часу. Замість того, щоби змушувати себе програмувати, поки особисті проблеми турбують мене, я виділяю спеціальний час, наприклад, годину, на усунення причини занепокоєння. Якщо моя дитина захворіла, я телефоную додому та перевіряю її стан. Якщо я посварився із дружиною, я телефоную їй і обговорюю наші розбіжності. Якщо маю проблеми із грошима, то обмірковую можливе рішення. Я знаю, що проблема навряд чи буде остаточно розв'язана за цю годину, але мені з великою долею імовірності вдасться зменшити рівень занепокоєння та притлумити фоновий процес.

В ідеалі на боротьбу з переживаннями мав би витрачатися особистий час. Соромно гаяти робочий час подібним чином. Професійні розробники витрачають свій особистий час, аби робочі години використовувалися якомога продуктивніше. Це означає, що ви маєте спеціально викроїти час вдома для розв'язання проблем і не приносити їх на роботу.

З іншого боку, якщо ви вже перебуваєте на роботі, а тривожні думки підривають вашу продуктивність, краще витратити годину на їх вирішення, ніж змушувати себе писати код, який пізніше все одно буде викинутий (або ще гірше — із яким вам доведеться жити).

Зона потоку

Про надпродуктивний стан, що називається «потоком» (flow), писалося багато. Деякі програмісти називають його «зоною» ("the Zone"). Вам, імовірно, знайоме це відчуття граничної концентрації свідомості, якої можуть досягати розробники при написанні коду. У цьому стані вони почуваються всемогутніми і непогрішними. Тому і прагнуть увійти у цей стан, а їхня самооцінка часто визначається тим, яку частину часу вони в ньому проводять.

А тепер невелика порада від того, хто неодноразово бував у «зоні» та повертався з неї: *уникайте «зони»*. Насправді цей стан далеко не

такий продуктивний і, тим більше, непогрішний. Це лише помірно медитативний стан, у якому розумові здібності знижуються, поступаючись відчуттю швидкості.

Дозвольте мені пояснити це. У «зоні» *можна* написати більше коду. Якщо ви практикуєте розробку через тестування (TDD), то ви швидше подолаєте цикл «червоний/зелений/рефакторинг». І у вас з'явиться легка ейфорія через *відчуття* досягнення мети. Проблема полягає в тому, що під час перебування в «зоні» втрачається частина «загальної картини», тому ви швидше ухвалюєте рішення, які пізніше вам доведеться виправляти. Код швидше пишеться у «зоні», але згодом вам доведеться частіше повертатись до нього.

Тепер, коли я відчуваю, що моя свідомість поступово входить до «зони», я на кілька хвилин відходжу від цієї межі. Я намагаюся прояснити свої думки: відповідаю на електронну пошту, переглядаю твіти. Якщо час наближається до полудня — роблю перерву на ланч. Якщо я працюю в команді, то знаходжу когось для парного програмування.

Парне програмування має одну важливу перевагу: удвох практично неможливо ввійти у «зону». Стан «зони» неконструктивний, тоді як парна робота потребує безперервного інтенсивного спілкування. Мені доводиться часто чути, що одна із претензій до парного програмування полягає саме в тому, що воно блокує входження до «зони». Чудово! Зона — це *не* те місце, де вам належить перебувати.

Загалом, це *не зовсім* так. У деяких ситуаціях «зона» — саме те місце, де варто опинитися. Наприклад, коли ви тренуєтеся!

Але ми поговоримо про це в іншому розділі.

Музика

Коли я працював у *Teradyne* наприкінці 1970-х, то мав окреме приміщення. Як системний адміністратор нашої PDP 11/60 я був одним із небагатьох розробників, які мали власний термінал. Це був термінал VT100 зі швидкістю передачі даних у 9600 бод, підключений до PDP11 через кабель RS232, що я особисто проклав з офісу до машинного залу.

У моїй кімнаті стояла стереосистема. То був старий програвач, підсилювач до нього й виносні динаміки. Я мав досить велику колекцію вінілу, включно з «Led Zeppelin», «Pink Floyd»… ну, загалом ви зрозуміли.

Зазвичай я вмикав стереосистему перед тим, як почати писати код. Мені здавалося, що музика сприяє концентрації. Проте я помилявся.

Якось я передивлявся модуль, що редагував під час прослуховування початкової частини «The Wall». Коментарі в коді містили слова з пісні та згадки про звукові вставки: ревіння пікіруючих бомбардувальників, дитячий плач і т. п.

І тоді я збагнув, що читач такого коду набагато більше дізнається про музичну колекцію автора (тобто, мене), ніж про проблему, заради розв'язання якої було написано цей код.

Я також зрозумів, що просто не можу добре писати код під час прослуховування музики. Музика заважає мені зосередитись. До того ж, на прослуховування витрачаються ресурси, необхідні моєму мозку для створення чистого, добре спланованого коду.

Можливо, у вас усе інакше. Можливо, музика *допомагає* вам писати код. Я знаю багатьох людей, які програмують у навушниках. Я припускаю, що музика допомагає їм, але також підозрюю, що насправді вона лише допомагає їм увійти до «зони».

Перешкоди

Уявіть, що ви програмуєте на своєму робочому місці. Як ви відреагуєте, коли хтось звернеться до вас по допомогу? Зірветеся? Зміряєте нахабу холодним поглядом? Чи мова вашого тіла натякне, що ви конче зайняті? Коротше: ви відреагуєте неввічливо або зупините поточну роботу і люб'язно допоможете, виявивши таке саме ставлення, котре хотіли би бачити від інших, якби самі звернулися до них?

Нечемні відповіді часто коріняться в «зоні». Можливо, вас дратує те, що хтось витягує вас звідти або заважає вашим спробам увійти до «зони». Хай там як, але грубість часто пояснюється вашим зв'язком із «зоною».

Але в деяких випадках винуватцем виявляється не «зона», а ви самі. Припустімо, що ви намагаєтеся розібратися у складному питанні, що потребує цілковитої концентрації. Це завдання може бути вирішене кількома засобами.

Парна робота чудово допомагає впоратися з перешкодами. Ваш партнер занурюється в контекст завдання, а ви можете працювати з телефонними дзвінками і питаннями колег. Коли ви повертаєтеся до напарника, він допомагає вам швидко відновити інтелектуальний контекст на момент виникнення перешкоди.

Також відчутну допомогу надає TDD. Якщо якийсь тест не проходить, він визначає поточний контекст вашої роботи. Розібравшись із перешкодою, ви повертаєтеся до нього і продовжуєте працювати над усуненням проблеми.

Зрозуміло, вам не вдасться *повністю уникнути перешкод*, що неминуче відволікатимуть вас і призводитимуть до втрат часу. Але згадайте, що наступного разу допомога може знадобитися і вам. Отже, професіоналізм проявляється у ввічливій готовності прийти на допомогу.

Блокування

Іноді код просто «не приходить». Це бувало зі мною, і я бачив, як подібне траплялося з іншими людьми. Ви сидите за комп'ютером — і нічого не відбувається.

Тоді ви знаходите інші справи. Читаєте електронну пошту або Twitter. Переглядаєте книжки, графіки і документи. Влаштовуєте зустрічі, починаєте дискусії. Коротше кажучи, робите *що завгодно*, щоби знову не опинитися наодинці з комп'ютером.

Чому виникають такі «творчі кризи»? Ми вже торкнулися багатьох причин цього явища. Особисто для мене головним фактором є нестача сну. Якщо я мало сплю, то просто не здатен програмувати. Також важливу роль відіграють тривога, страх і депресія.

Дивно, але ця проблема має дуже просте рішення. Воно спрацьовує майже завжди, легко реалізується і забезпечує необхідний імпульс для написання великого обсягу коду.

Ось воно: знайдіть партнера для парного програмування.

Просто неймовірно, наскільки ефективно це працює. Тільки-но ви сідаєте з кимось поруч, усі проблеми, що заважали роботі, негайно зникають. Робота в парі призводить до *фізіологічних* змін. Я не знаю, що конкретно відбувається, але виразно це відчуваю. Чи то в мозку, чи то в тілі відбувається певна біохімічна зміна, що дозволяє подолати застій і знову взятися за роботу.

Але візьміть до уваги, що це рішення не ідеальне. Іноді активність триває годину-дві, а далі виникає настільки серйозне виснаження, що доводиться кидати партнера і шукати місце для відпочинку. Іноді, навіть коли я працюю в парі, у мене вистачає сил лише погоджуватись із тим, що робить напарник. І все ж таки для мене типовою реакцією на роботу в парі є відчутне відновлення творчого потенціалу.

Творчий внесок

Існують й інші заходи запобігання застою. Я досить давно дізнався, що результати творчої роботи залежать від творчого внеску.

Я читаю багато книжок на різні теми. Це матеріали із програмування, політики, біології, астрономії, фізики, хімії, математики та багатьох інших сфер. Однак у кінцевому рахунку я виявив, що наукова фантастика найкраще стимулює мою творчу активність.

Для вас це може бути щось геть інше — гарний детектив, поетична збірка чи навіть любовний роман. Напевно, річ у тім, що творчість породжує творчість. Також варто враховувати елемент ескапізму. Години, проведені вдалині від повсякденних турбот, ще й під активним впливом цікавих творчих ідей, викликають майже непереборне бажання щось створити самому.

Не всі форми творчої діяльності працюють однаково ефективно. Перегляд телебачення, як правило, не впливає на мій творчий стан. Похід до кінотеатру працює краще, але ненабагато. Музика не допомагає створювати код, але сприяє створенню презентацій та підготовці відеоматеріалів. І все ж таки з усіх форм творчих джерел ніщо не впливає на мене краще, ніж стара добра «космічна опера».

Відлагодження

Один із найгірших сеансів відлагодження (debugging session) за всю мою кар'єру стався 1972 року. Термінали, що були підключені до бухгалтерської системи вантажоперевізників, зависали один-два рази на день. Свідомо відтворити помилку було неможливо. Помилка не віддавала переваг якомусь конкретному терміналу або застосунку. Неважливо, чим займався користувач перед зависанням: тільки-но термінал працював бездоганно, а наступної хвилини безнадійно зависав.

На діагностику були потрібні тижні. Тим часом вантажоперевізники дратувалися все більше й більше. Щоразу при зависанні користувачеві доводилося припиняти роботу і чекати, поки решта користувачів теж завершать свої поточні операції. Після цього вони телефонували нам, і ми перезавантажували центральний комп'ютер. Це був справжній кошмар.

Першу пару тижнів ми витратили на звичайне опитування користувачів, які працювали за терміналами, що зависали. Ми запитували, чим вони займалися в той момент і що робилося перед тим. Інших користувачів ми питали, чи не помітили вони чогось незвичайного на *своїх* терміналах у момент зависання. Опитування проводилося телефоном, тому що термінали були встановлені в передмісті Чикаго, а ми працювали в офісі, розташованому на 30 миль північніше.

У нас не було журналів, лічильників чи відлагоджувачів (debuggers). Взаємодія із внутрішнім станом системи здійснювалася через індикатори і тумблери на передній панелі. Ми могли зупинити комп'ютер і переглянути вміст пам'яті по словах. Однак займатися цим понад п'ять хвилин було неможливо, тому що вантажоперевізникам була потрібна їхня система в робочому стані.

Ми витратили кілька днів на написання нескладного інспектора, що працював у режимі реального часу. Ним можна було керувати з телетайпа ASR-33, який служив нам консоллю. Інспектор дозволяв читати та змінювати вміст пам'яті під час роботи системи. Ми додали журнальні повідомлення, що виводились на телетайп у критичних ситуаціях. Ми створили в пам'яті лічильники подій, що запам'ятовували інформацію про стан перегляду пам'яті інспектором.

І, звичайно, весь цей код створювався «з нуля» на асемблері і тестувався вечорами, коли система не використовувалася.

Робота терміналів керувалася перериваннями. Символи, що передаються терміналам, зберігалися у циклічних буферах. Щоразу під час передачі символу послідовним портом спрацьовувало переривання і до відправки готувався наступний символ у циклічному буфері.

Згодом з'ясувалося, що термінал зависав через розсинхронізацію трьох змінних, що керували циклічним буфером. Ми і гадки не мали, чому це відбувалося, але то було хоч щось. Десь у 5К рядків коду супервізора ховалася помилка, що некоректно працювала з одним із цих покажчиків.

Нова інформація також дозволила нам знімати блокування з терміналів вручну! Ми могли за допомогою інспектора передати цим трьом змінним значення за замовчуванням, і термінали, як за помахом чарівної палички, знову починали працювати. Незабаром ми написали невеличкий фрагмент коду, що перевіряв синхронізацію лічильників та відновлював їх у разі потреби. Спочатку код запускався спеціальним тумблером переривання користувача на передній панелі, коли замовники телефоном повідомляли про зависання. Пізніше ми просто виконували код відновлення щосекунди.

Через місяць проблема із зависанням зникла — принаймні з погляду вантажоперевізників. Іноді один із їхніх терміналів зупинявся на пів секунди, але за наявності базової швидкості передачі у 30 символів за секунду ніхто цього не помічав.

Але чому відбувалася десинхронізація лічильників? Мені було 19, і я був сповнений рішучості розібратися.

Автором коду супервізора був Річард, який на той час поїхав на навчання до коледжу. Ніхто з нас не розумівся на цьому коді, бо Річард ставився до свого творіння дуже ревно. Код належав *йому*, і нам не було дозволено розбиратися в ньому. Але тепер Річарда не було, тому я відшукав його лістинг завтовшки кілька дюймів і почав переглядати сторінку за сторінкою.

Циклічні буфери у цій системі були звичайними структурами даних FIFO, тобто чергами. Прикладні програми заносили символи з одного кінця черги, поки вона не заповнювалася. Обробники переривань витягували символи з іншого кінця черги, коли принтер був готовий їх прийняти. Якщо у черзі не було символів, принтер зупинявся. Помилка змушувала програми вважати, що черга заповнена, а обробники переривань — що вона порожня.

Обробники переривань виконувались в іншому програмному потоці, окремо від решти коду. Отже, лічильники та змінні, доступні для обох обробників та іншого коду, мали бути захищені від паралельного оновлення. У нашому випадку це означало, що перед виконанням будь-якого коду, що працював із цими трьома змінними, необхідно було заборонити переривання. До того моменту, коли я взявся за лістинг, мені вже було зрозуміло: треба шукати ділянку коду, що працює зі змінними без попередньої заборони переривань.

Звичайно, зараз ми би використали багато потужних інструментів для пошуку всіх місць, де код торкався цих змінних. За лічені секунди ми би знайшли всі рядки коду, що працюють зі змінними. За хвилини можна було би визначити, де саме автор забув заборонити переривання. Однак ця історія відбувалася 1972 року, і в мене таких інструментів не було. Були лише мої власні очі.

Я переглянув кожну сторінку коду в пошуках змінних. На жаль, змінні використовувалися *скрізь*. Майже на кожній сторінці програма зверталася до них у такий спосіб. Для багатьох звернень переривання не заборонялися — вони обмежувалися читанням, отже, були нешкідливими. До того ж, у цьому конкретному асемблері було неможливо перевірити, чи доступна змінна лише для читання, без аналізу логіки коду. Щоразу, коли змінна була прочитана, пізніше вона могла бути оновлена і збережена. І якщо при цьому переривання не було заборонено, вміст змінних легко міг бути зіпсований.

Мені знадобилося кілька днів інтенсивного вивчення коду, але зрештою я виявив помилку. У середині коду знайшлося одне місце, у якому одна із трьох змінних оновлювалася без попередньої заборони переривань.

Я занурився в обчислення. Вразливість існувала протягом двох мікросекунд. У системі дюжина терміналів передавали дані на швидкості 30 символів за секунду, тож переривання відбувалися кожні 3 мікросекунди або близько до того. З урахуванням розмірів

супервізора й тактової частоти процесора, зависання від цієї вразливості мали відбуватися із частотою приблизно 1–2 рази на день. Ось воно!

Звичайно, я виправив помилку, але в мене не вистачило сміливості відключити автоматичний запуск перевірки та виправлення лічильників. Тому я й досі не впевнений у тому, що в системі не було іншої діри.

Час відлагодження

Із якоїсь невідомої причини розробники не вважають відлагодження природною частиною процесу розробки. Їм здається, що воно схоже на фізіологічну потребу: її *задовольняють* просто тому, що це неминуче. Однак час відлагодження стає компанії рівно в таку суму, що і час написання коду, тому будь-які заходи щодо його скорочення є корисними.

Сьогодні я проводжу за відлагодженням набагато менше часу, ніж десять років тому. Я не проводив точних вимірювань, але, за моїми оцінками, тривалість відлагодження скоротилася аж у десять разів. Я досяг цього радикального скорочення за рахунок переходу на методологію розробки через тестування (TDD), що буде розглянута в наступному розділі.

Незалежно від того, використовується TDD або інша методологія аналогічної ефективності[1], ви як професіонал зобов'язані прагнути звести час відлагодження до нуля. Звісно, нуль — мета асимптотична, але від цього вона не перестає бути метою.

Лікарі не люблять наново робити операції пацієнтам, аби виправити власні попередні помилки. Адвокати не люблять повторно братися за провалені справи. Лікар чи адвокат, який часто припускається помилок, не вважатиметься професіоналом. Так само й розробник, що регулярно помиляється, чинить непрофесійно.

[1] Я не знаю жодної дисципліни, що була би настільки ефективною, як TDD, але, можливо, вона вам відома.

Спостерігайте за собою

Програмування — марафон, а не спринт. Неможливо виграти забіг, набравши максимальну швидкість на старті. Перемагає той, хто економить сили та обирає розумний темп. Марафонець мусить дбати про свій організм як до, так і *під час* змагань. І професійні розробники також бережуть сили і творчий потенціал.

Вмійте зупинятися

Не можете піти додому, доки не виконали своє завдання? Піти не тільки можна, а й потрібно! Творче мислення та інтелектуальна діяльність — недовговічні стани нашого розуму. Коли ми втомлюємося, вони зникають. Якщо ви будете силою примушувати втомлений мозок працювати в пізню годину, найімовірніше, це призведе лише до додаткової втоми та зниження ймовірності того, що задачу вдасться розв'язати в душі або в машині.

Якщо ви зайшли у глухий кут, якщо ви втомилися — відключіться і зробіть перерву. Дайте своїй підсвідомості відпочити. Уважно ставлячись до своїх ресурсів, ви зробите більше за менший час і з меншими зусиллями. Самі визначайте темп роботи для себе та своєї команди. Вивчіть закономірності у проявах ваших творчих здібностей та осяянь і використовуйте їх замість того, щоби діяти проти них.

Дорогою додому

Багато задач були успішно розв'язані в машині, коли я повертався з роботи додому. Водіння потребує великих витрат нетворчих інтелектуальних ресурсів. Ваші очі, руки і частина розуму зайняті керуванням машиною, і ви відволікаєтесь від того, чим займалися під час роботи. І в цьому відволіканні є щось невизначене, що допомагає вашому розуму знаходити інші, більш творчі шляхи в пошуках рішень.

Душ

Несподівано багато складних задач було розв'язано мною в душі. Можливо ранкові водні процедури допомагають моєму мозкові остаточно прокинутися і пригадати всі рішення, що виникли в ньому, поки я спав.

Працюючи над задачею, ви іноді опиняєтеся так близько від неї, що вже не бачите всіх можливих варіантів. Ви просто не помічаєте елегантних рішень, тому що творча частина вашого розуму пригнічена зайвою зосередженістю. Іноді найкраще для пошуку рішення — повернутися додому, повечеряти, подивитися телевізор і лягти спати, а наступного ранку прокинутися і прийняти душ.

Запізнення

Ви *відставатимете* від графіка. Це відбувається навіть із найкращими з нас. Це відбувається з найдбайливішими. Іноді оцінки часу виявляються помилковими, а отримання результату запізнюється.

Головний трюк у боротьбі із запізненням — це раннє його виявлення та загальна прозорість. Найгірше, коли ви до останнього моменту запевняєте оточуючих, що роботу буде завершено вчасно, а потім підводите всіх. *Не робіть цього*. Натомість *регулярно* перевіряйте темпи просування проєкту стосовно кінцевої мети, маючи на увазі три обґрунтовані кінцеві дати: найкращу, номінальну та гіршу[1]. Будьте за можливості чесними у своїх оцінках. *Не включайте надію* в свої оцінки! Повідомте всі три дати своїй команді і ключовим учасникам проєкту і щодня оновлюйте їх.

Надія

Що робити, коли із цих дат випливає, що ви *можете* не встигнути до дедлайну? Наприклад, через десять днів починається торгово-промислова виставка, де має бути презентований ваш продукт. З іншого боку, потрійна оцінка часу готовності підсистеми, над якою ви працюєте, дорівнює 8/12/20.

[1] Набагато докладніше про це — у Розділі 10.

Не сподівайтеся, що вам вдасться зробити все за 10 днів! Сподівання вбивають проєкти. Вони зривають графіки і руйнують репутації. Надія наразить вас на великі неприємності. Якщо виставка розпочнеться за 10 днів, а номінальна оцінка складає 12 днів, ви *не встигаєте* до визначеного терміну. Переконайтеся, що група та ключові учасники проєкту розуміють ситуацію, і не заспокоюйтесь доти, доки не буде вироблений альтернативний план. І не дозволяйте решті учасників проєкту сподіватися.

Поспіх

А якщо начальник запрошує вас для бесіди віч-на-віч і прохає встигнути до зазначеного терміну? Якщо він наполягає і підкреслює, що ви маєте зробити все можливе для цього? *Не відступайте від оцінок!* Вихідні оцінки завжди точніші за будь-які зміни, що робляться під тиском. Повідомте начальникові, що ви вже розглянули всі варіанти (бо ви їх дійсно розглянули) і що прискорити роботу можна лише одним способом — відкиданням частини функціональності. *Не піддавайтеся спокусі прискорити темп!*

Горе нещасному розробнику, який поступиться під тиском. Він почне шукати обхідні шляхи і працюватиме понаднормово в марній надії створити диво. Це неминучий шлях до катастрофи, тому що він надає вам, вашій команді і ключовим учасникам проєкту невиправдані сподівання. Все це випливає з небажання дивитися в обличчя реальності та відкладення неприємних, але необхідних рішень.

Поспіх — безглуздий. Ви не змусите себе програмувати швидше. Ви не змусите себе швидше розв'язувати задачі. А якщо спробуєте — тільки вповільните роботу і влаштуєте хаос, що також уповільнить роботу інших учасників.

Отже, ви повинні відповісти начальникові, команді і ключовим учасникам проєкту так, аби в них не залишилося необґрунтованих надій.

Понаднормові

Начальник каже: «А якщо збільшити робочий день на кілька годин? Скажімо, працювати по суботах? Напевно, якось можна викроїти зайві години, щоби встигнути до визначеної дати».

Понаднормова робота можлива, а іноді просто необхідна. Інколи можна вчасно завершити роботу, працюючи по 10 годин на добу з одним-двома вихідними на місяць. Однак це дуже ризиковано. Навряд чи двадцятивідсоткове збільшення тривалості робочого дня дозволить вам виконати на двадцять відсотків більше роботи. А ще важливіше те, що понаднормова робота протягом більш ніж двох-трьох тижнів *напевно* призведе до провалу.

Отже, на понаднормову роботу можна погоджуватися лише за наявності деяких умов: 1 — особисто ви можете її собі дозволити; 2 — аврал триватиме недовго, не більше двох тижнів; 3 — у вашого керівництва є резервний план на випадок, якщо ваші зусилля завершаться невдачею.

Останній критерій має вирішальне значення. Якщо ваш начальник не може пояснити, що він збирається робити у разі провалу, не погоджуйтесь на понаднормову роботу.

Хибна готовність

Імовірно, найгірший із усіх виявів непрофесіоналізму з боку розробника — це спроба видати недороблений продукт за готовий. Іноді це просто відверта брехня, що само по собі погано. Але набагато небезпечнішою є інша ситуація — спроба підвести раціональну основу під нове визначення «готовності». Ми переконуємо себе, що зроблено *достатньо*, і переходимо до наступного завдання. І при цьому говоримо, що роботу можна виконати пізніше, коли в нас знайдеться більше часу.

Ця хвороба є дуже заразною. Якщо одному розробнику така поведінка зійде з рук, інші наслідуватимуть його приклад. Я бачив, як подібні речі доходили до жахливих крайнощів. Один із моїх клієнтів під «готовністю» розумів реєстрацію змін у базі даних. І при цьому код міг навіть не компілюватися! Дуже легко об'явити задачу «готовою», якщо нічого не має працювати.

Коли команда потрапляє до цієї пастки, начальство чує від її членів, що все йде нормально. Всі звіти про хід виконання роботи показують, що її буде завершено до відповідного терміну. Ситуація нагадує пікнік сліпих на залізничних рейках: ніхто не бачить потяга, завантаженого нерозв'язаними проблемами, що наближається, поки не стає надто пізно.

Дайте визначення готовності

Проблема хибної готовності розв'язується створенням незалежного визначення готовності. Для цього слід доручити бізнес-аналітикам і фахівцям із тестування створити автоматизовані приймальні тести[1], без проходження яких продукт не може вважатися готовим. Тести пишуться тестовими мовами — як-от FitNesse, Selenium, RobotFX, Cucumber тощо. Вони мають бути зрозумілими для бізнесменів і ключових учасників проєкту, не пов'язаних із його технічною стороною, і виконуватися наскільки можливо часто.

Допомога

Програмування — по-справжньому *важка* робота. Що ви молодші, то менше в це віриться. Зрештою програмний код — лише послідовність команд `if` і `while`. Але в міру накопичення досвіду ви дізнаєтеся, що найважливішу роль грає спосіб об'єднання цих команд. Не можна просто звалити їх докупи і сподіватися на краще. Натомість систему необхідно ретельно розбити на невеликі, зрозумілі блоки, що повинні бути якнайменше пов'язані один з іншим,— а це насправді складно.

Програмування настільки складне, що жодній людині не впоратися із цією роботою самотужки. Навіть самому кваліфікованому фахівцю необхідні думки та ідеї інших розробників.

[1] Див. Розділ 7.

Як допомагати іншим

Відповідальні розробники мають бути готовими допомагати один одному. Розробник, який ізолюється у своєму офісі чи кабінці й відмовляється відповідати на запитання колег, порушує професійну етику. Ваша робота не настільки важлива, щоби ви не могли пожертвувати дещицею часу для допомоги іншим. Честь професіонала зобов'язує пропонувати колегам допомогу тоді, коли це потрібно.

Це не означає, що ви повинні відмовитись від виконання власних завдань. Але, наприклад, ви можете повідомити, що між 10:00 та ланчем вас не можна турбувати, а з 13:00 до 15:00 ваші двері відчинені для всіх.

Також будьте уважні до стану роботи ваших колег. Якщо хтось відчуває труднощі, запропонуйте допомогу. Дивовижно, наскільки ефективною часом виявляється така допомога. Річ не в тім, що ви набагато розумніші за свого колегу; просто свіжа думка часом стає потужним каталізатором для розв'язання нагальних проблем.

Коли ви комусь допомагаєте, сядьте й напишіть код разом. Заплануйте на допомогу щонайменше годину на день, або навіть більше. Можливо, вам знадобиться менше часу, але не варто поспішати. Зрештою від такої співпраці ви отримаєте більше, ніж віддасте.

Як приймати допомогу

Коли хтось пропонує вам допомогу, будьте вдячні. Прийміть її з подякою і поставтеся до неї з усією увагою. *Не відмовляйтеся від допомоги*, посилаючись на те, що вам бракує часу. Виділіть на розмову близько тридцяти хвилин. Якщо особливої користі від запропонованої допомоги не видно, чемно вибачтесь і завершіть розмову з подякою. Пам'ятайте: професіоналізм зобов'язує вас не лише пропонувати, а й приймати запропоновану допомогу.

Навчіться *просити* про допомогу. Коли ви зайшли в глухий кут або не можете розібратися в задачі, попросіть когось допомогти вам. Якщо ви перебуваєте у спільній кімнаті, просто скажіть уголос: «Мені потрібна допомога». В інших випадках можна скористатися Twitter, електронною поштою або телефоном. Звернення по допомогу також є частиною професійної етики. Непрофесійно залишатися у безвиході, коли допомога цілком доступна.

Думаєте, зараз я намалюю райдужну картину: хор виконавців спірічуелз співає пронизливий гімн, пухнасті зайчики скачуть верхи на єдинорогах, і всі разом вирушають до країни надій і змін? Нічого подібного. Розробники зазвичай виглядають самовпевненими, егоцентричними інтровертами. Ми обираємо цю сферу не тому, що дуже любимо ближніх. Більшість із нас звертається до програмування тому, що нам подобається концентруватися на дрібницях, жонглювати безліччю абстрактних концепцій і будь-яким іншим засобом доводити собі, що ми є носіями видатного інтелекту, а зовсім не для того, щоби розбиратися із проблемами інших людей.

Так, це стереотип. Так, це узагальнення із багатьма винятками. Однак реальність виглядає саме так: розробники не схильні до співпраці[1]. Проте співпраця відіграє виключно важливу роль в ефективному програмуванні. Оскільки ж для багатьох із нас співпраця не є інстинктом, нам потрібні *методології*, що спонукають нас до співпраці.

Менторинг

Далі цій темі буде присвячений цілий розділ. А поки дозвольте мені просто сказати, що відповідальність за навчання менш кваліфікованих розробників покладається на їх досвідчених колег. Навчальних курсів недостатньо. Книжок недостатньо. Ніщо не дозволить молодому розробникові досягти високої продуктивності швидше, ніж власне бажання в поєднанні з ефективним навчанням у старших товаришів. Отже, ще одна сторона професійної етики полягає в тому, що досвідчені фахівці беруть під опіку молодших і навчають їх. Водночас професійним обов'язком недосвідченого розробника є пошук наставника і запозичення його досвіду.

[1] Це набагато більше стосується чоловіків, ніж жінок. У мене відбулася чудова розмова з @desi (Desi McAdam, засновницею DevChix) про те, що насправді мотивує жінок-розробниць. Я сказав їй, що написати працюючу програму для мене — це те саме, що вполювати дикого звіра. Вона відповіла, що для неї та інших жінок, із якими вона розмовляла, процес написання коду був актом створення чогось і подальшого плекання творіння.

Бібліографія

Martin, Robert C. *Clean Code, Upper Saddle River*, NJ: Prentice Hall, 2009; Мартін, Роберт С. Чистий код: створення, аналіз і рефакторинг. Харків, «Фабула», 2020.

Martin, Robert C. *Agile Software Development: Principles, Patterns and Practices*. Upper Saddle River, NJ: Prentice Hall, 2003.

5 РОЗРОБКА, ОРІЄНТОВАНА НА ТЕСТУВАННЯ

Минуло понад десять років відтоді, як методологія розробки через тестування (TDD, Test Driven Development) з'явилася в нашій галузі. Спочатку вона застосовувалася на хвилі екстремального програмування (XP, eXtreme Programming), але з того часу була використана Scrum і майже всіма іншими гнучкими (Agile) методологіями. Навіть команди, які не користуються гнучкими методологіями, застосовують TDD.

Коли 1998 року я вперше почув про «випереджувальне тестування» («Test First Programming»), то поставився до нього скептично. Та й хто би тоді вчинив інакше? Як це — починати роботу з написання модульних тестів? Навіщо робити такі дурниці? Але на той час я вже мав тридцятирічний досвід професійного програмування; я бачив, як у галузі з'являються та зникають численні нові ідеї, і чудово розумів, що нічого не варто заперечувати, особливо якщо це рекомендує така людина, як Кент Бек.

Отже, 1999 року я вирушив до Медфорда (штат Орегон), аби зустрітися з Кентом і навчитися в нього нової методології. Результат виявився просто вражаючим!

Ми з Кентом засіли в його офісі і почали програмувати просте завдання на Java. Я хотів написати якийсь примітивний код, але Кент заперечив і провів мене по всьому процесу крок за кроком. Спочатку він написав крихітну частку модульного тесту, яку і кодом не можна було назвати. Потім написав код, достатній для того, щоби компілювався тест. А потім він написав ще один тест і ще трохи коду.

Подібний робочий цикл цілком суперечив усьому моєму досвіду. Я звик писати код не менше години, перш ніж намагатися відкомпілювати або запустити його. Але Кент виконував свій код буквально кожні 30 секунд або близько того. Це виглядало неймовірно!

Але найцікавішим було те, що цей робочий цикл був мені відомий! Я стикався з ним багато років тому, коли ще дитиною[1] програмував

[1] Як я вважав тоді, дитиною є будь-яка людина молодше 35 років. У свої двадцять я витратив значну кількість часу, пишучи безглузді маленькі ігри інтерпретованими мовами. То були ігри про війни в космосі, пригодницькі ігри, ігри про скачки, ігри зі зміями, навіть азартні ігри.

ігри інтерпретованими мовами на зразок Basic або Logo. У цих мовах не було складання як такого: ви просто додавали рядок коду й запускали програму. Робочий цикл відбувався дуже швидко. І тому програмування цими мовами було *дуже продуктивним*.

Але в *реальному програмуванні* такий робочий цикл здавався абсурдним. Тут ви витрачали багато часу на написання коду, а потім ще більше часу на те, щоби змусити його компілюватися. І ще більше часу потребувало відлагодження. Адже я був розробником на C++, чорт забирай, а в C++ процеси складання і компонування могли тривати хвилинами, а то й годинами. Тридцятисекундні робочі цикли здавались немислимими.

Однак переді мною сидів Кент, який писав свою програму на Java із тридцятисекундними циклами — і без найменшого натяку на те, що робота вповільниться. І тоді до мене дійшло, що ця проста методологія дозволяє програмувати справжніми мовами з тривалістю робочого циклу, типовою для Logo!

І я капітально «підсів» на неї!

Вердикт винесено

Відтоді я дізнався, що TDD є чимось більшим, ніж простий трюк для скорочення робочого циклу. Ця методологія має безліч переваг, що будуть описані нижче.

Але спочатку я маю повідомити таке:

- Вердикт винесений!
- Дебати завершені.
- Команда GOTO шкідлива.
- А TDD — працює.

Так, за минулі роки про TDD написано багато суперечливих статей і блогів. Спочатку в них можна було знайти багато серйозної критики і обґрунтованих сумнівів, але в наші дні ці дискусії припинилися. Хай би хто що казав, а TDD працює.

Я знаю, що це твердження здається занадто радикальним, але, зрештою, хірургам уже не треба доводити корисність миття рук. І я не думаю, що розробникам слід захищати TDD.

Як можна називати себе професіоналом, якщо ви не знаєте, чи працює ваш код? А як можна дізнатися, що ваш код працює, якщо ви не тестуєте його при кожному внесенні змін? І як тестувати код при кожному внесенні змін, не маючи автоматизованих модульних тестів з дуже високим покриттям? Але чи можливо створити автоматизовані модульні тести з дуже високим покриттям без застосування TDD?

Утім, останню пропозицію варто розглянути докладніше. Що це, власне, таке TDD?

Три закони TDD

1. Новий робочий код пишеться тільки після написання невдалого модульного тесту.
2. Ви пишете такий обсяг коду модульного тесту, що є необхідним для того, щоби цей тест не проходив (якщо код тесту не компілюється, вважається, що він не проходить).
3. Ви пишете такий обсяг робочого коду, що є достатнім для проходження модульного тесту, який на цю мить не проходить.

Ці три закони змусять вас використовувати робочий цикл тривалістю близько 30 секунд. Спочатку ви пишете невелику частину модульного тесту. За ці лічені секунди ви згадуєте в коді ім'я класу або функції, які ще не написані, і це призводить до збою компіляції модульного тесту. Отже, далі ви маєте написати робочий код, із яким тест відкомпілюється. Але писати більше коду не можна, тому переходьте до написання додаткового коду модульного тесту.

Цикл повторюється знову і знову. Додаємо невеликий фрагмент у тестовий код. Додаємо невеликий фрагмент у робочий код. Два кодові потоки зростають одночасно, перетворюючись на взаємодоповнюючі компоненти. Відповідність між тестами і робочим кодом нагадує відповідність між антитілом та антигеном.

Молитва про вигоди

Упевненість

Якщо ви приймете TDD як професійну методологію, то писатимете десятки тестів щодня, сотні тестів щотижня, тисячі тестів щороку. І всі ці тести постійно будуть у вас під рукою і запускатимуться при кожному внесенні в код будь-яких змін.

Я є основним автором і відповідальним за супровід FitNesse[1] — системи приймального тестування на базі Java. На момент написання цієї книжки код FitNesse складався з 64 000 рядків, із яких 28 000 містилися у 2200 окремих модульних тестах. Ці тести забезпечують покриття щонайменше 90 % робочого коду[2], а їхнє виконання триває близько 90 секунд.

Щоразу, коли я змінюю якусь частину FitNesse, то запускаю модульні тести. Якщо вони проходять успішно, я отримую практично повну впевненість, що зміни нічого не порушили. Наскільки повну? У будь-якому разі достатню, щоб опублікувати оновлену версію!

Весь процес контролю якості FitNesse зводиться до команди `ant release`. Ця команда збирає FitNesse «з нуля», а потім запускає всі модульні та приймальні тести. Якщо всі тести проходять успішно, я публікую результат.

Зниження щільності дефектів

Зараз FitNesse не є критично важливою програмою. Якщо у FitNesse закрадеться помилка, ніхто не помре і ніхто не втратить мільйони доларів. Виходячи із цього, я можу собі дозволити опублікувати нову версію на підставі лише проходження тестів. З іншого боку, FitNesse має тисячі користувачів, і при тому, що за останній рік кодова база розширилася на 20 000 рядків, мій список дефектів складається лише з 17 позицій (більшість із яких мають суто косметичну природу). Таким чином, я знаю, що щільність дефектів у FitNesse є надзвичайно низькою.

---

[1] http://fitnesse.org.

[2] Дев'яносто відсотків — це мінімум. Насправді цифра вища. Точну кількість важко обчислити, оскільки інструменти покриття не можуть бачити код, що виконується у зовнішніх процесах або у блоках перехоплення.

І цей ефект не є унікальним. Істотне зниження кількості дефектів при використанні TDD описане в цілій низці звітів[1] та досліджень[2]. Від IBM до Microsoft, від Sabre до Symantec — компанія за компанією і команда за командою повідомляють про зниження кількості дефектів у 2, 5 і навіть 10 разів. Справжній професіонал не може ігнорувати такі показники.

Сміливість

Чому ми не виправляємо невдалий код одразу ж, як бачимо його? Зазвичай наша перша реакція на безладну функцію: «Ну й мішанина, слід усе виправити». Друга реакція: «Нехай це зробить хтось інший!» Чому? Тому що ви знаєте, що, ледве торкнувшись коду, ви ризикуєте його «зламати»; а якщо код буде «зламаний», то відповідальність за нього автоматично переходить до вас.

А якщо ви будете *впевнені*, що впорядкування коду нічого не порушить? Що ви матимете таку впевненість, про яку я щойно згадав? Якщо ви просто натискаєте кнопку і за 90 секунд *дізнаєтеся*, що зміни нічого не порушили, а *принесли користь*?

Це одна з найбільших переваг TDD. Якщо у вас є пакет тестів, якому можна довіряти, ви втрачаєте страх перед внесенням змін. Побачивши невдалий код, ви просто чистите його — і все. Код стає глиною, із якої виліплюються прості та елегантні структури.

Коли розробник позбавляється страху перед чисткою коду, він його чистить! Чистий код легше зрозуміти, простіше змінювати та легше розширювати. Зі спрощенням коду ймовірність дефектів стає ще нижчою. Відбувається *стабільне поліпшення* кодової бази — замість «загнивання коду», звичного для нашої галузі.

Хіба професійний розробник може дозволити, щоби «загнивання» тривало?

Документація

Чи доводилося вам колись застосовувати сторонній фреймворк? Фірма-розробник зазвичай надсилає красиво оформлену інструкцію, створену технічними письменниками. Типова версія містить

[1] http://www.objectmentor.com/omSolutions/agile_customers.html.

[2] Див. бібліографію до цього розділу.

27 глянцевих ілюстрацій із кружечками та стрілочками; на звороті кожної ілюстрації присутній абзац тексту з описом налаштування, розгортання та інших операцій із цією структурою. А наприкінці, десь у додатку, ховається маленький кривенький розділ із прикладами коду.

Куди ви насамперед подивитесь у такому посібнику? Якщо ви розробник, то звернетеся до прикладів коду. Ви зробите це, бо знаєте: код розповість усю правду. 27 глянцевих ілюстрацій із кружечками та стрілочками виглядають дуже симпатично, але якщо ви хочете дізнатися, як використовувати код,— треба читати його.

Кожен модульний тест, написаний із дотриманням трьох наведених тут законів, є прикладом використання системи, сформульованим у вигляді програмного коду. Якщо ви дотримувалися трьох законів, то у вашому коді матиме місце модульний тест, який описує створення кожного об'єкта в системі для будь-якого способу створення таких об'єктів. У ньому буде модульний тест, що описує виклик кожної функції в системі для будь-якого осмисленого способу виклику. Для кожної операції, із приводу якої у вас можуть виникнути питання, буде наявний модульний тест, що докладно описує її виконання.

Модульні тести є документами. Вони описують нижній архітектурний рівень системи. Вони однозначні, точні, написані мовою, зрозумілою для найширшої аудиторії, і достатньо точні та формальні для виконання. Це найкраща низькорівнева документація, яка взагалі можлива. Який професіонал не забезпечив би таку документацію?

Архітектура

Якщо ви дотримуєтеся трьох законів і пишете тести перш, ніж робочий код, то стикаєтеся з дилемою. Часто ви знаєте, який код потрібно написати, але три закони вимагають спочатку написати модульний тест, який точно буде невдалим, бо код іще не існує! Тобто вам пропонується тестувати ще не написаний код.

Проблема з тестуванням коду полягає в необхідності ізоляції цього коду. Часто буває важко тестувати функцію, що викликає інші функції. Для того щоби написати такий тест, необхідно якимсь чином відокремити функцію від решти. Інакше кажучи, необхідність тестування змушує вас продумати *архітектуру програми*.

Якщо ви не починаєте з написання тестів, то ніщо не завадить вам звалити всі функції купою, що не піддається тестуванню. Якщо тести пишуться пізніше, можливо, вам вдасться протестувати вхідну та вихідну поведінку цієї купи, але, ймовірно, із тестуванням окремих функцій виникнуть великі проблеми.

Таким чином, дотримання трьох законів та випереджаюче створення тестів сприяють більш якісній архітектурі з меншою кількістю прив'язок. А хіба професіонал відмовиться від інструментів, чиє застосування призводить до вдосконалення архітектури?

«Але ж я можу написати тести пізніше!» — заперечите ви. Ні, не можете. Звісно, *деякі* тести дійсно можна написати пізніше. Можна навіть наблизитися до високого покриття, якщо ви ретельно його виміряєте. Проте тести, написані пізніше, лише *захищають* від помилок. А тести, написані з випередженням, активно *атакують* їх. Тести, створені пізніше, пишуться розробником, який сформував код і знає, як вирішувалося завдання. Такі тести жодним чином не можна порівнювати за повнотою та актуальністю з тестами, написаними заздалегідь.

Вибір професіоналів

Зі всього сказаного вище випливає, що TDD — вибір професіоналів. Ця методологія дарує впевненість, додає сміливості розробникам, знижує кількість дефектів, вдало формує документацію і покращує архітектуру. За такої кількості доказів на користь TDD відмова від її використання цілком може вважатися проявом *непрофесіоналізму*.

Чим TDD не є

З усіма своїми перевагами TDD — не релігія і не панацея. Дотримання умов трьох законів не гарантує жодної з перерахованих переваг. Поганий код можна написати навіть при попередньому написанні тестів. Та й самі тести теж можуть виявитися написаними невдало.

У певних ситуаціях усі три закони виявляються непрактичними і навіть невідповідними. Такі ситуації трапляються рідко, але вони

можливі. Тому жоден професійний розробник не стане застосовувати методологію, котра в конкретній ситуації приносить більше шкоди, ніж користі.

Бібліографія

Maximilien E. M., Williams L. Assessing Test-Driven Development at IBM.— http://collaboration.csc.ncsu.edu/laurie/Papers/MAXIMILIEN_WILLIAMS.PDF

George B., Williams L. An Initial Investigation of Test-Driven Development in Industry.— http://collaboration.csc.ncsu.edu/laurie/Papers/TDDpaperv8.pdf

Janzen D., Saiedian H. Test-driven development concepts, taxonomy, and future direction.— *IEEE Computer*. V. 38, Is. 9, pp. 43–50.

Nagappan N., Maximilien E. M., Bhat Th., Williams L. Realizing quality improvement through test driven development: results and experiences of four industrial teams. Springer Science + Business Media, LLC 2008.— http://research.microsoft.com/en-us/projects/esm/nagappan_tdd.pdf

6 ПРАКТИКА

Усі фахівці відточують свою майстерність за допомогою спеціальних вправ. Музиканти грають гами. Футболісти стрибають через шини. Лікарі тренуються у накладанні швів та виконанні інших хірургічних прийомів. Адвокати репетирують промови. Солдати беруть участь у навчаннях. Професіонали тренуються скрізь, де ефективність виконання роботи має особливо важливе значення. Цей розділ повністю присвячений розвитку навичок програмування.

Деякі основи практики

Концепція тренування у програмуванні з'явилася досить давно, але власне тренуванням вона була визнана лише на початку нового тисячоліття. Імовірно, перший формальний приклад тренувальної програми було надруковано на сторінці 6 відомого підручника[1]:

```
main()
{
  printf("hello, world\n");
}
```

Хто з нас не писав цю програму у тій чи іншій формі? Ми використовуємо її для того, щоби випробувати нове середовище програмування або нову мову. Коли ми пишемо і виконуємо цю програму, це є достатнім доказом того, що ми так само можемо написати і відкомпілювати *будь-яку іншу* програму.

Коли я був молодший, освоєння нового комп'ютера зазвичай починалося для мене з написання програми SQINT, що обчислює квадрати цілих чисел. Я писав її на асемблері, BASIC, FORTRAN, COBOL та безлічі інших мов. Усі численні версії цієї програми також доводили, що я можу змусити комп'ютер зробити те, що потрібно.

Перші персональні комп'ютери з'явилися на початку 1980-х років. Щоразу, отримавши можливість попрацювати за одним із них (то були VIC-20, Commodore-64 або TRS-80), я писав невелику програму для виведення нескінченного потоку символів — '\' і '/'. Шаблони, що будувалися такими програмами, тішили око і виглядали набагато складнішими за маленьку програму, котра їх будувала.

І хоча ці програми були суто навчальними, програмісти спеціально *не тренувалися*. Відверто кажучи, це нам просто не спадало на думку. Ми були надто зайняті написанням коду, щоби піклуватися про вдосконалення майстерності. Та й навіщо? У ті роки програмування не вимагало швидкої реакції або спритних пальців. Перші екранні редактори з'явилися лише наприкінці 1970-х років. Переважна більшість нашого робочого часу витрачалася на очікування компіляції або відлагодження довгих потворних потоків коду. Короткі цикли

[1] Kernighan B.W., Ritchie D. M. *The C Programming Language*. Upper Saddle River, NJ: Prentice Hall, 1975.

TDD ще не були винайдені, тому ті нетривіальні можливості, що відкриваються завдяки тренуванню, просто були не потрібні.

Двадцять два нулі

Але від початку епохи програмування спливло досить багато часу. Деякі обставини *відчутно* змінилися, інші взагалі не змінилися.

Однією із перших машин, для яких я програмував, була PDP-8/I. Цей комп'ютер мав тактову частоту 1,5 мс. Основна пам'ять містила 4096 12-бітних слів. Він був розміром із холодильник і споживав купу електроенергії. Його дисковий накопичувач дозволяв зберігати 32K 12-бітних слів, а взаємодія з оператором здійснювалася через телетайп зі швидкістю передачі 10 символів за секунду. Нам здавалося, що це дуже потужна машина, на якій можна творити дива.

Нещодавно я придбав новий ноутбук MacBook Pro. Він оснащений двоядерним процесором із тактовою частотою 2,8 ГГц, має 8 Гб оперативної пам'яті, SSD-накопичувач ємністю у 512 Гб і 17-дюймовий РК-екран із роздільною здатністю 1920 × 1200. Я ношу його в наплічнику. Він легко вміщується в мене на колінах і споживає менше 85 Вт.

Порівняно із PDP-8/I мій ноутбук працює у вісім тисяч разів швидше, має у два мільйони разів більше пам'яті, у 16 мільйонів разів більшу ємність запам'ятовуючого пристрою, споживає лише 1 % потужності, займає 1 % місця і коштує у 25 разів дешевше.

Підрахуємо:

$$8000 \times 2\,000\,000 \times 16\,000\,000 \times 100 \times 100 \times 25 = 6{,}4 \times 1022.$$

Це *величезне* число. Різниця становить 22 *порядки*! Це відстань в ангстремах від Землі до Альфи Центавра або кількість електронів у срібному доларі — інакше кажучи, велике, дуже велике число. І тепер такий комп'ютер стоїть на моїх колінах — і, мабуть, на ваших теж!

І що ж я роблю із цією міццю, що зросла на 10 у 22 ступені? Приблизно те саме, що робив і на PDP-8/I. Я пишу команди `if`, цикли `while` та команди *присвоєння*.

О, інструменти для написання цих команд помітно покращилися, і мови стали потужнішими. Але сама природа команд за цей час

не змінилася. Код 2010 року буде зрозумілий програмісту 1960-х років. Глина, із якої ми ліпимо свої програми, не змінилася за чотири десятиліття.

Тривалість робочого циклу

Але *процес* роботи серйозно змінився. У 1960-х роках я міг чекати день або два, щоби отримати результати компіляції. Наприкінці 1970-х років програма із 50 000 рядків компілювалася приблизно за 45 хвилин. Навіть у 1990-ті роки довге складання вважалося нормою.

У наші дні програмісти не чекають на результати компіляції[1]. У їхньому розпорядженні з'явилася така немислима обчислювальна міць, що цикл «червоний-зелений-рефакторинг» прокручується буквально за секунди.

Наприклад, я веду проєкт FitNesse, написаний на Java, що складається із 64 000 рядків коду. Повне складання з усіма модульними та інтеграційними тестами забирає менше чотирьох хвилин. Якщо тести проходять, я публікую нову версію продукту. Отже, *весь процес контролю якості — від вихідного коду до розгортання — забирає менше чотирьох хвилин*. Час компіляції дуже малий. Тести виконуються за лічені секунди. Отже, цикл компіляція/тестування може здійснюватися *по 10 разів за хвилину*!

Звісно, не завжди варто працювати з такою швидкістю. Часто краще пригальмувати і подумати[2]. Але в деяких ситуаціях прокручування циклу з максимальною швидкістю виявляється *надзвичайно* продуктивним.

Для швидкого виконання будь-яких операцій необхідні тренування. Швидке прокручування циклу «код/тест» вимагає майже моментального прийняття *рішень*. А для швидкого прийняття рішень необхідно безпомилково розпізнавати величезну кількість ситуацій та проблем, а також *знати*, що треба зробити для їх розв'язання.

[1] Той факт, що деякі програмісти чекають на складання, є драматичним і свідчить про необережність. У сучасному світі час складання має вимірюватися секундами, а не хвилинами і, звісно, не годинами.

[2] Це техніка, яку Річ Хіккей назвав HDD (Hammock-Driven Development) — «техніка розробки в гамаку».

Уявіть двох майстрів бойових мистецтв під час поєдинку. Кожен намагається зрозуміти, що зараз спробує зробити інший, і правильно зреагувати за лічені мілісекунди. Під час бою вам не вдасться зупинити час, проаналізувати ситуацію і підібрати кращу відповідь. У бою потрібно *реагувати*. Ваше *тіло* реагує, а тим часом розум працює над стратегією вищого рівня.

Поки ви прокручуєте цикл «код/тест» кілька разів за хвилину, ваше *тіло* знає, які клавіші слід натискати. Певна частина мозку розпізнає ситуацію і реагує, видаючи протягом кількох мілісекунд адекватне рішення, а тим часом мозок концентрується на проблемі вищого рівня.

І в бойових мистецтвах, і в програмуванні швидкість залежить від *тренованості*. І в обох випадках тренування проходить приблизно однаково: ми обираємо кілька пар «проблема/рішення» і повторюємо їх знову і знову, поки не досягнемо автоматизму.

Уявіть гітариста, скажімо, Карлоса Сантану. Музика з його голови безпосередньо переходить у пальці. Він не замислюється над положенням пальців чи прийомами гри. Його розум звільнений для високорівневих мелодій та гармонійних поєднань, тоді як тіло перетворює їх на низькорівневі рухи пальців.

Але для досягнення такої свободи потрібна *практика*. Музикант тренується, виконуючи етюди, гами і рифи знову і знову, поки не знатиме їх напам'ять.

Додзьо кодування

Із 2001 року я проводжу демонстрацію TDD, яку називаю «Грою в кеглі» («The Bowling Game»)[1]. Ця нескладна вправа забирає близько 30 хвилин. Вона виявляє конфлікт в архітектурі, розвивається до кульмінаційної точки і наостанок видає сюрприз. Я написав цілий розділ про цей приклад в «Agile Software Development: Principles, Patterns and Practices».

За минулі роки я проводив цю демонстрацію сотні, а може й тисячі разів. Я досяг такої майстерності, що міг би повторити її уві сні.

[1] Вона стала дуже популярним ката, і пошук у Google видасть чимало прикладів. Оригінал шукайте тут: http://butunclebob.com/ArticleS.UncleBob.TheBowlingGameKata.

Я звів до мінімуму кількість натискань клавіш, підібрав найзручніші імена змінних й оптимізував структуру алгоритму так, що вона стала мало не ідеальною. Хоча тоді я й не знав цього, це було моє перше ката.

У 2005 році я відвідав конференцію XP2005 у Шеффілді (Велика Британія). Там я брав участь у презентації під назвою «Додзьо програмування» («Coding Dojo»), яку проводили Лорен Боссавіт та Еммануель Галло. Усі присутні відкрили свої ноутбуки і програмували разом із доповідачами, які застосовували методологію TDD для реалізації гри Конвея «Життя». Доповідачі назвали цю вправу «ката» і повідомили, що початкова ідея[1] належала «Прагматику» Дейву Томасу[2].

Відтоді багато програмістів стали використовувати метафору бойових мистецтв для своїх тренувальних сеансів. Назва «Додзьо програмування»[3] теж прижилася. Іноді кілька програмістів зустрічаються і тренуються разом, як майстри бойових мистецтв. Іноді тренування проходять поодинці — теж за аналогією з бойовими мистецтвами.

Близько року тому я навчав групу розробників в Омасі. За ланчем вони запросили мене на свій сеанс «Додзьо програмування». Я спостерігав за тим, як двадцять розробників відкрили свої ноутбуки і, клавіша за клавішею, почали повторювати дії свого наставника, який виконував ката «Гра в кеглі».

У додзьо застосовуються різні види вправ. Деякі з них наведено нижче.

Ката

У бойових мистецтвах терміном *ката* називається певний набір ретельно відпрацьованих рухів, що імітують дії однієї зі сторін у поєдинку. Їхньою метою (звісно, гіпотетичною) є досягнення повної досконалості. Майстер прагне навчити своє тіло ідеально виконувати кожен рух і об'єднувати окремі прийоми у плавну серію. Добре виконувані ката виглядають неймовірно гарно.

[1] http://codekata.pragprog.com.
[2] Ми використовуємо прізвисько «Прагматичний», аби відокремити його від «Великого» Дейва Томаса з OTI.
[3] http://codingdojo.org/

Але попри красу, ката призначені не для сценічного виконання. Майстри виконують їх, аби навчити розум і тіло реагувати на конкретні бойові ситуації. Відпрацьовані рухи мають виконуватися автоматично та інстинктивно, щоби спрацьовувати саме тоді, коли вони необхідні.

Програмні ката є конкретним набором відпрацьованих натискань клавіш і пересувань миші, що імітують розв'язання певної програмної задачі. Ви не розв'язуєте завдання, бо вже знаєте рішення. Натомість ви тренуєтеся у виконанні дій та прийнятті рішень, необхідних для розв'язання задачі.

І у цьому випадку метою є асимптотичне наближення до досконалості. Вправа повторюється знову і знову, щоби навчити ваш мозок реагувати, а пальці слухатися. Під час тренування можуть виявитися деякі покращення і підвищення ефективності ваших дій або самого рішення.

Відпрацювання кількох ката допомагає запам'ятати «гарячі клавіші» та ідіоми навігації. Це також добре працює при вивченні таких дисциплін, як TDD і CI (Continuous Integration, безперервна інтеграція). Але найважливіше полягає в тому, що ката допомагають закріпити в підсвідомості пари «завдання/розв'язання», і зіткнувшись із цими завданнями в реальному програмуванні, ви будете заздалегідь знати, як вони вирішуються.

Розробник, так само, як і майстер бойових мистецтв, повинен знати різні ката і регулярно практикувати їх, щоби вони не зникли з пам'яті. Описи багатьох ката є за адресами http://katas.softwarecraftsmanship.org та http://codekata.pragprog.com.

Ось деякі з моїх улюблених ката:

- Гра в кеглі: http://butunclebob.com/ArticleS.UncleBob.TheBowlingGameKata
- Прості числа: http://butunclebob.com/ArticleS.UncleBob.ThePrimeFactors-Kata
- Перенесення слів: http://thecleancoder.blogspot.com/2010/10/craftsman-62-dark-path.html

Якщо вам до смаку складніші випробування, спробуйте вивчити ката настільки добре, щоби виконувати їх під музику. Зробити це вкрай важко.

Вадза

Коли я займався джіу-джитсу, переважна частина часу відводилася відпрацюванню *вадза*. Вадза нагадують ката, але в них беруть участь двоє людей. Усі прийоми точно запам'ятовуються і відтворюються напарниками. Один грає роль нападника, інший обороняється. Рухи повторюються знову і знову, а напарники обмінюються ролями.

Розробники можуть тренуватись аналогічним чином у грі, яка має назву *пінг-понг*[1]. Двоє напарників обирають ката чи просте завдання. Один пише модульний тест, а інший має змусити цей тест проходити. Потім вони обмінюються ролями.

Якщо напарники обирають стандартну ката, то результат наперед відомий, а розробники відпрацьовують роботу з клавіатурою і мишею, а також чіткість запам'ятовування ката. З іншого боку, якщо напарники обирають нове завдання, гра стає цікавішою. Розробник, який пише тест, переважно контролює процес розв'язання завдання. Також він має право встановлювати обмеження. Наприклад, якщо розробники вирішують реалізувати алгоритм сортування, автор тесту може встановити обмеження щодо швидкості та витрат пам'яті, ускладнивши таким чином завдання партнера. У такому вигляді гра стає змагальною — і більш цікавою.

Рандорі

Рандорі — вільний спаринг. Під час тренувань із джіу-джитсу ми визначали бойові сценарії, а потім їх відпрацьовували. Інколи одна людина мала захищатися, тоді як інші послідовно атакували її. Іноді двоє і більше нападників виступали проти одного, який захищався (зазвичай це був наш сенсей, і майже завжди перемога залишалася за ним). Іноді одна пара вступала в єдиноборство з іншою — і т. п.

Імітація двобою не має якогось аналога у програмуванні; проте у багатьох програмних додзьо грають у гру, що теж має назву рандорі. Вона дуже схожа на вадза для двох партнерів, які розв'язують спільну задачу. Однак у рандорі грають багато учасників, а правила

[1] http://c2.com/cgi/wiki?PairProgrammingPingPongPattern.

трохи інші. На екрані, що проєктується на стіну, один із учасників пише тест. Інший учасник забезпечує проходження тесту, а потім пише наступний тест. Хід передається по колу або учасники просто шикуються у чергу. І кожного разу такі вправи виявляються *дуже* кумедними.

Залишається дивуватися, як багато можна дізнатися з таких вправ. Ви отримуєте глибоке уявлення про те, як інші люди підходять до розв'язання завдань. Отримана інформація допомагає розширити ваш світогляд і підвищити кваліфікацію.

Розширюємо свій досвід

Професійні розробники часто страждають від одноманітності розв'язуваних задач. Роботодавці часто обмежуються однією мовою, платформою і предметною областю, у якій змушений працювати розробник. Якщо ви не подбаєте про розширення власного світогляду, це може призвести до небажаного обмеження резюме й менталітету. Такі розробники нерідко виявляються непідготовленими до змін, що періодично виникають у нашій галузі.

Проєкти з відкритим кодом

Один зі способів «триматися на передньому краї» запозичений із практики адвокатів та лікарів: виконуйте суспільно-корисну роботу, беручи участь у проєктах із відкритим кодом. Таких проєктів дуже багато; однак немає кращого способу поповнити ваш творчий арсенал, ніж попрацювати над завданням, що принесе користь іншим людям.

Якщо ви програмуєте на Java — візьміть участь у проєкті Rails. Якщо ви пишете великий обсяг на C++ для свого роботодавця, знайдіть проєкт на Python і долучіться до нього.

Етика тренування

Професійні розробники тренуються у вільний час. Ваш роботодавець не зобов'язаний дбати про підтримку вашої кваліфікації або розширення вашого резюме. Пацієнти не платять лікарям за те, що вони тренуються в накладанні швів. Футбольні вболівальники (зазвичай) не платять за те, щоби подивитися, як гравці бігають на тренуваннях. Глядачі не платять за те, щоби послухати, як музиканти грають гами. І роботодавці розробників також не зобов'язані оплачувати їм час тренувань.

Оскільки тренування проводяться у ваш особистий час, ви, зі свого боку, не зобов'язані використовувати ті мови і платформи, що використовуєте на своїй основній роботі. Виберіть будь-яку мову і продемонструйте ваші здібності поліглота. Якщо ви працюєте на платформі .NET, потренуйтеся у використанні Java або Ruby — вдома або під час обідньої перерви.

Висновок

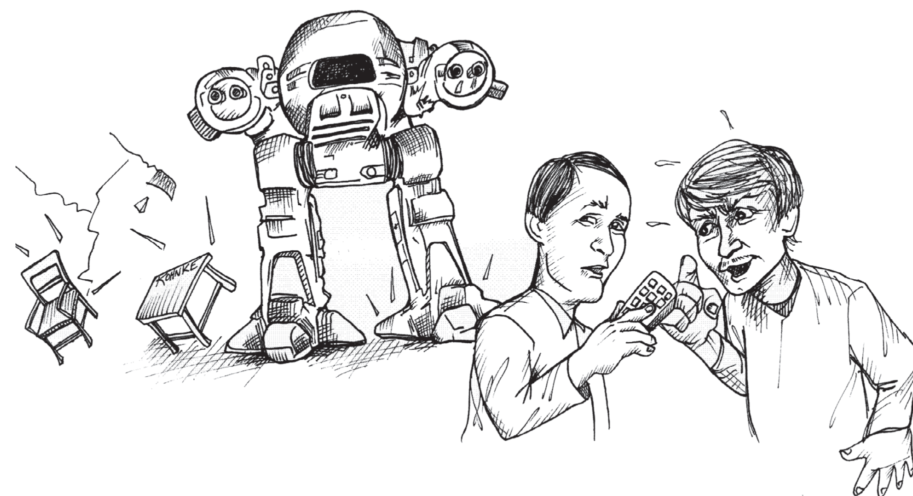
Усі професіонали у такий чи інший спосіб тренуються. Вони роблять це, бо хочуть якомога краще впоратися з дорученою їм справою. Ба більше, вони роблять це у свій вільний час, оскільки знають, що відповідальність за підвищення їхньої кваліфікації лежить на них самих, а не на роботодавцях. За тренування вам ніхто не платитиме. Ви практикуєтеся, щоби стабільно отримувати платню за основну роботу, і до того ж, непогану.

Бібліографія

Kernighan B.W., Ritchie D. M. *The C Programming Language*. Upper Saddle River, NJ: Prentice Hall, 1975.

Martin R. C. *Agile Software Development: Principles, Patterns and Practices*. Upper Saddle River, NJ: Prentice Hall, 2003.

7 ПРИЙМАЛЬНІ ТЕСТИ



Роль професійного розробника передбачає як програмування, так і спілкування. Пам'ятайте, що принцип «сміття на вході/сміття на виході» застосовується і до розробників, тому професійні розробники стежать за тим, аби їхнє спілкування з іншими членами команди і бізнесменами було точним та плідним.

Вимоги до комунікації

Одним із найпоширеніших аспектів спілкування між розробниками і бізнесом є розробка вимог. Бізнесмени описують те, що, на їхню думку, їм потрібно, а розробники створюють те, що, на їхню думку, їм описали. Принаймні так має бути. На практиці інформаційний обмін відбувається надзвичайно складно, а ризик помилок залишається дуже високим.

У 1979 році під час роботи в *Teradyne* мене відвідав Том, керівник відділу монтажу та обслуговування. Він попросив мене навчити його працювати в текстовому редакторі ED-402, щоби він міг створити просту систему обробки заявок на усунення несправностей.

ED-402 був комерційним редактором, написаним для комп'ютера M365 — клона PDP-8. Це був дуже потужний текстовий редактор із вбудованою мовою сценаріїв, котру ми використовували для різноманітних операцій із текстом.

Том не був розробником, але застосунок, що він собі уявляв, виглядав досить простим. Він припускав, що я швидко навчу його, а він самотужки напише програму. Я наївно думав те ж саме. Адже мова сценаріїв, зрештою, являла собою звичайну макромову команд редагування з найпростішими умовними та циклічними конструкціями.

Отже, ми всілися за комп'ютер, і я запитав, що має робити його застосунок. Том почав з екрану введення даних. Я показав, як створити текстовий файл зі сценарними командами і як ввести у сценарій символьне представлення команд редагування. Але коли я глянув Томові в очі, то отримав у відповідь абсолютно порожній погляд. Він не зрозумів жодного слова з мого пояснення.

Тоді я вперше зустрівся із цим явищем. Для мене символьне представлення команд редагування було простим і природним: наприклад, для представлення команди *Control+B* (команда переміщення курсору на початок поточного рядка) у файл сценарію просто додавали символи ^B. Але Тому це здавалося безглуздим: він міг перейти від «редагування файлу» до «редагування файлу, що редагував файл». Том не був дурним. Напевно, він просто зрозумів, що завдання набагато складніше, ніж йому здавалося спочатку, і не захотів витрачати свій час та розумові сили для вивчення такої хитромудрої концепції, як використання редактора для управління редактором.

Мало-помалу я розпочав самостійно реалізовувати програму Тома, поки він сидів і спостерігав. За 20 хвилин мені стало ясно, що він уже думає не про те, як зробити все самому, а про те, як пояснити мені, що саме йому потрібно.

На це довелося витратити цілий день. Том описував якусь можливість, а я реалізовував її, поки він спостерігав. Робочий цикл становив трохи більше 5 хвилин, тому не було потреби відходити і займатися чимось іншим. Він просив мене зробити «щось», і за 5 хвилин реалізація цього «щось» уже працювала.

В інших випадках Том малював те, що йому було треба, на аркуші паперу. Деякі з його вимог важко було реалізувати засобами ED-402; тоді я пропонував щось інше. Зрештою, ми сходилися на чомусь, що могло сяк-так працювати, і я реалізовував узгоджений варіант.

Але потім ми випробовували той варіант, і Том змінював своє рішення, кажучи щось на кшталт: «Так, але це не зовсім те, що мені потрібне. Давай спробуємо інакше».

Годину за годиною ми робили спроби та експериментували, надаючи застосунку потрібної форми. Ми випробували одне, потім інше, потім ще щось. Незабаром я зрозумів, що Том — скульптор, а я — інструмент у його руках.

Зрештою Том отримав бажаний застосунок, не маючи при тому найменшого уявлення про те, як побудувати наступний застосунок самостійно. З іншого боку, я отримав важливий урок стосовно того, як замовник визначає свої потреби. Виявилося, що його уявлення про функціональність програми часто не витримують зіткнення з комп'ютером.

Передчасна точність

І бізнесмени, і розробники раз у раз потрапляють до пастки *передчасної точності*. Бізнесмени хочуть точно знати розмір прибутку, перш ніж погодитися на участь у проєкті. Розробники хочуть точно знати, що їм належить зробити, перш ніж переходити до оцінювання проєкту. Обидві сторони бажають мати точну інформацію, яку неможливо отримати наперед, до того ж, часто вони готові витрачати цілі статки на спроби її отримання.

Принцип невизначеності

На жаль, на папері все виглядає зовсім не так, як у робочій системі. Коли бізнесмени бачать, як виглядає реалізація їхніх вимог у системі, то розуміють, що хотіли зовсім не цього. А іноді вже після знайомства з реалізацією вони починають краще розуміти, що їм насправді потрібно,— і найчастіше це те, що вони побачили.

У гру вступає «ефект спостерігача», або принцип невизначеності. Коли ви демонструєте нову можливість бізнес-учасникам, у них з'являється більше інформації, ніж було раніше, а ця нова інформація впливає на їхнє сприйняття системи загалом.

Отже, що точніше ви формулюєте свої вимоги, то швидше вони втрачають актуальність під час реалізації системи.

Тривожність із приводу оцінки

Розробники також потрапляють у пастку точності. Вони знають, що мають видати оцінку системи, і часто вважають, що їхня оцінка має бути точною. Нічого подібного!

По-перше, навіть за наявністю ідеальної інформації у ваших оцінках буде присутній величезний розкид. По-друге, принцип невизначеності знищує ранню точність ущент. Вимоги із часом змінюватимуться, а це призводить до того, що точність втрачає сенс.

Професійні розробники розуміють, що оцінки можуть і мають базуватися на вимогах із низькою точністю, і визнають, що ці оцінки *є лише оцінками*. Вони завжди включають у свої оцінки діапазони похибки, щоби бізнес-учасники розуміли наявність невизначеності (див. Розділ 10).

Пізня неоднозначність

Розв'язання проблеми передчасної точності полягає в тому, щоби відкласти точні визначення на якомога пізніший термін. Професійні розробники не надають вимог до того моменту, поки не почнеться розробка. Однак такий підхід може призвести до іншої біди: пізньої неоднозначності.

Серед ключових учасників проєкту нерідко виникають розбіжності. У таких випадках часто простіше «заговорити» проблему, ніж розв'язати її. Тоді виникають такі формулювання тієї чи іншої вимоги, із якими начебто згодні всі, хоча реального розв'язання проблеми немає. Колись Том Демарко висловився із цього приводу так: «Неоднозначність у документі з вимогами виникає через розбіжності між учасниками»[1].

Звісно, неоднозначність не завжди виникає через суперечки або розбіжності. Іноді ключові учасники проєкту просто вважають, що ті, хто виконує вимоги, вже знають, що мається на увазі.

Можливо, ці формулювання цілком зрозумілі їм самим у певному контексті, але означають щось зовсім інше для розробника, який читає документ. Подібна контекстна неоднозначність також може виникати при особистому спілкуванні замовників з розробниками.

> Сем (зацікавлена особа): Так, ще потрібно організувати резервне копіювання журнальних файлів.
>
> Паула: Як часто?
>
> Сем: Щодня.
>
> Паула: Зрозуміло. І де зберігатимуться резервні копії?
>
> Сем: Що ти маєш на увазі?
>
> Паула: Напевно, вони мають зберігатися в певній папці?
>
> Сем: Так, мабуть.
>
> Паула: І як вона називатиметься?
>
> Сем: Може, `backup`?
>
> Паула: Так, нормально. Значить, журнал щодня зберігатиметься в каталозі `backup`. У який час?

---

[1] XP Immersion 3, May, 2000.— http://c2.com/cgi/wiki?TomsTalkAtXpImmersionThree.

Сем: Щодня.

Паула: Ні, у який саме час доби?

Сем: У будь-який.

Паула: О дванадцятій?

Сем: Ні, тільки не під час торгів на біржі. Краще опівночі.

Паула: Добре, нехай буде опівночі.

Сем: Чудово, дякую!

Паула: Завжди рада допомогти.

Пізніше Паула розповідає про завдання своєму колезі Пітеру:

Паула: Отже, журнал повинен копіюватися в папку з ім'ям `backup` щодня опівночі.

Пітер: Зрозуміло. А як буде називатися файл?

Паула: Я думаю, `log.backup` підійде.

Пітер: Добре.

В іншому офісі Сем спілкується телефоном із замовником.

Сем: Так, так, журнальні файли зберігатимуться.

Карл: Дуже важливо, щоби жодного журналу не було втрачено. Не можна виключати, що нам доведеться колись повернутися до будь-якого із цих файлів навіть через місяці або роки — у випадку збою живлення, якоїсь катастрофи, судової суперечки тощо.

Сем: Не турбуйтеся, я щойно говорив із Паулою. Журнали зберігатимуться в папці з ім'ям `backup` щодня опівночі.

Карл: Добре, це те, що треба.

Ви помітили неоднозначність? Замовник очікує, що зберігатимуться всі журнали, а Паула впевнена, що достатньо зберегти журнал за останній день. Якщо замовник захоче повернутися до резервної копії місячної давності, він знайде лише дані останнього дня.

У цьому разі і Паула, і Сем виявилися не на висоті. Професійні розробники (і зацікавлені особи) повинні стежити за тим, щоб із вимог була виключена будь-яка неоднозначність.

Це *складне* завдання, і мені відомий лише один спосіб його розв'язання.

Приймальні тести

Термін «приймальні тести» (acceptance tests) перевантажений безліччю значень і занадто поширений. Одні вважають, що йдеться про тести, що виконуються користувачами перед прийманням чергової версії продукту. Інші розуміють під цим терміном контроль якості. У цьому розділі під терміном «приймальні тести» розуміються тести, написані за участі зацікавлених сторін і розробників *для перевірки виконання вимог*.

Що таке «виконано»?

Одна з найпоширеніших неоднозначностей, із якими стикаються професійні розробники, пов'язана із самим поняттям «виконання». Коли розробник каже, що виконав своє завдання, то що саме мається на увазі? Що він готовий передати свій код в експлуатацію із цілковитою впевненістю? Що його код готовий до контролю якості? А може, що він дописав свій код і навіть зміг один раз виконати його, але ще не тестував?

Я працював із командами, які мали різні уявлення про терміни «виконано» (done) та «готово» (complete). В одній групі використовувалися терміни «виконано» та «цілком виконано» (done-done).

Професійні розробники мають лише одне визначення: «виконано» значить *виконано*. Це означає, що весь код написаний, усі тести пройдено, служба контролю якості та зацікавлені особи прийняли результат. Виконано!

Але як досягти такого рівня виконання, не знижуючи темпу переходу між ітераціями? Слід створити низку автоматизованих тестів, проходження яких означатиме виконання всіх перелічених умов! Якщо приймальні тести вашої підсистеми проходять успішно, то роботу *виконано*.

Професійні розробники розширюють визначення вимог до автоматизованих приймальних тестів. Вони спілкуються із зацікавленими особами та спеціалістами з контролю якості, прагнучи, щоби автоматизовані тести цілком відтворювали визначення «виконано».

Сем: Так, ще потрібно організувати резервне копіювання журнальних файлів.

Паула: Як часто?

Сем: Щодня.

Паула: Зрозуміло. І де зберігатимуться резервні копії?

Сем: Що ти маєш на увазі?

Паула: Напевно, вони мають зберігатися у певній папці?

Сем: Так, мабуть.

Паула: І як вона називатиметься?

Сем: Може, `backup`?

Том (тестувальник): Зачекайте, `backup` — надто загальна назва. Що саме зберігатиметься у цій папці?

Сем: Резервні копії.

Том: Резервні копії чого?

Сем: Журнальних файлів.

Паула: Але ж журнальний файл лише один?

Сем: Ні, їх багато. По одному щодня.

Том: Тобто у нас буде один активний журнал і багато резервних копій?

Сем: Звісно.

Паула: О! А я думала, тобі потрібні тимчасові резервні копії.

Сем: Ні, замовник хоче, щоби вони зберігалися необмежено довго.

Паула: Для мене це новина. Добре, що вчасно порозумілися.

Том: Ім'я підкаталогу має повідомляти, що саме в ньому зберігається.

Сем: Там зберігатимуться старі неактивні журнали.

Том: Тоді ми назвемо його `old_inactive_logs`.

Сем: Чудово.

Том: І коли саме буде створюватися ця папка?

Сем: Що ти маєш на увазі?

Паула: Папка створюватиметься під час запуску системи, але тільки в тому разі, якщо вона не існує.

Том: Зрозуміло, це наш перший тест. Я запускаю систему й перевіряю, чи створено папку `old_inactive_logs`. Потім я додаю файл до цієї папки, завершую роботу, знову запускаю систему і перевіряю, що папка і файл містяться в належному місці.

Паула: На виконання цього тесту потрібно чимало часу. Ініціалізація системи вже займає близько 20 секунд, і час лише збільшується. До того ж, я не хочу збирати систему наново під час кожного запуску приймальних тестів.

Том: Що ти пропонуєш?

Паула: Створимо клас `SystemStarter`. Основна програма завантажуватиме його з групою об'єктів `StartupCommand`, побудованих на базі патерна `Command`. Потім, під час запуску системи, `SystemStarter` наказує всім об'єктам `StartupCommand` виконати свої команди. Один із цих об'єктів `StartupCommand` створює папку `old_inactive_logs`, але тільки в тому разі, якщо вона не існує.

Том: Гаразд, тоді мені залишається лише протестувати цього нащадка `StartupCommand`. Я напишу для цього простий тест FitNesse.

Том прямує до дошки.

Том: Перша частина матиме приблизно такий вигляд:

```
given the command LogFileDirectoryStartupCommand
given that the old_inactive_logs directory does not exist
when the command is executed
then the old_inactive_logs directory should exist
and it should be empty
```

Том *(додає)*: Друга частина матиме приблизно такий вигляд:

```
given the command LogFileDirectoryStartupCommand
given that the old_inactive_logs directory exists
and that it contains a file named x
When the command is executed
Then the old_inactive_logs directory should still exist
and it should still contain a file named x
```

Паула: Так, цього має вистачити.

Сем: А це все дійсно необхідно?

Паула: Семе, яке із цих двох тверджень недостатньо важливе для перевірки?

Сем: Просто хочу сказати, що для написання всіх цих тестів знадобиться чимало зайвої роботи.

Том: Так, але не більше, ніж для написання плану ручного тестування. І *геть менше* ніж для багаторазового виконання тестів вручну.

Взаємодія сторін

Основні цілі приймальних тестів — взаємодія сторін, ясність і точність вимог. Наслідком погодження приймальних тестів є досягнення розуміння планованої поведінки системи з боку розробників, зацікавлених сторін і тестувальників. Досягнення такої ясності є обов'язком для всіх сторін. Що ж до професійних розробників, то вони вважають своїм обов'язком працювати із зацікавленими сторонами і тестувальниками, а також вживати заходи для того, щоби всі сторони знали, що саме вони мають намір створити.

Автоматизація

Приймальні тестування *завжди* мають бути автоматизованими. У певних моментах життєвого циклу програмних продуктів є місце для ручного тестування, але *такі* тести ніколи не повинні виконуватися вручну. Причина проста: витрати.

Погляньте на рис. 7.1. Руки, які ви на ньому бачите, належать менеджеру з контролю якості великої інтернет-компанії. У документі, що він тримає, міститься його план ручного тестування. Він має цілу армію тестувальників з інших країн, які виконують цей план кожні шість тижнів. Кожне тестування коштує приблизно мільйон доларів. Менеджер щойно повернувся із зустрічі, на якій його керівник повідомив, що бюджет тестування буде урізаний приблизно на 50 %, а тепер питає: «Яку саме половину цих тестів не слід виконувати?»

Назвати те, що відбулося, катастрофою, означає не сказати нічого. Витрати на ручне тестування настільки великі, що фірма вирішила відмовитися від нього і просто жити далі, *не знаючи, чи працює половина її продуктів!*

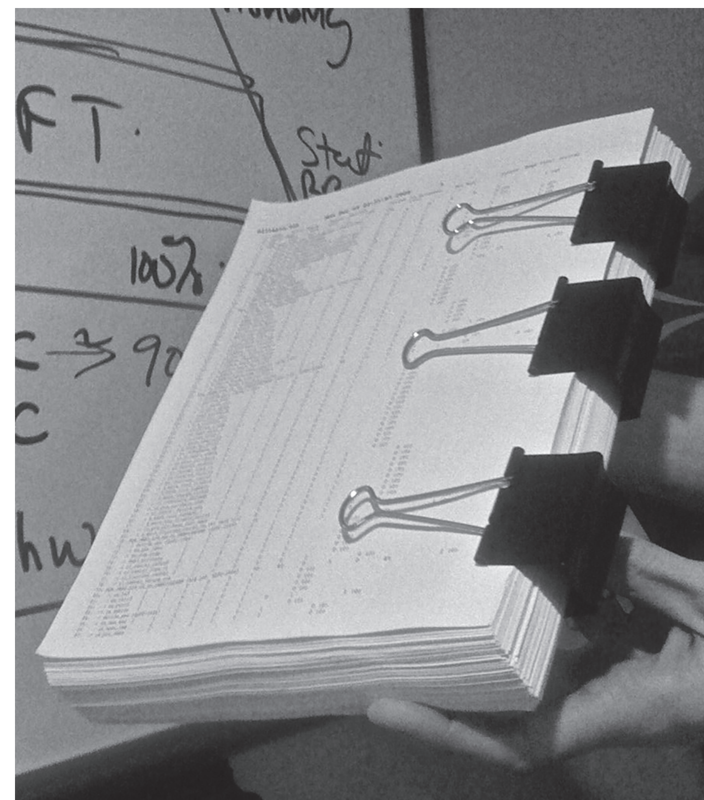


Рис. 7.1. План ручного тестування

Професійні розробники не допускають виникнення подібних ситуацій. Витрати на проведення автоматизованих тестувань настільки незначні в порівнянні з витратами на ручне тестування, що написання сценаріїв, котрі запускаються вручну, не має жодного економічного сенсу. Професійні розробники вважають за обов'язок простежити за тим, аби приймальні тести проводилися в автоматизованому режимі.

Існує безліч програмних інструментів — як комерційних, так і з відкритим кодом,— що автоматизують приймальні тестування. FitNesse, Cucumber, cuke4duke, robot framework та Selenium — і це далеко не повний список. У всіх цих інструментах автоматизовані тестування визначаються в такій формі, що не тільки розробники можуть читати їх, розуміти і навіть створювати.

Додаткова робота

Зауваження Сема щодо зайвої роботи зрозуміле. На перший погляд здається, що написання подібних приймальних тестів вимагатиме значних зусиль. Але, подивившись на рис. 7.1, ми побачимо, що це зовсім не зайва робота. Написання цих тестів є просто роботою з визначення системи. Лише на такому рівні деталізації ми, розробники, можемо зрозуміти, що роботу «виконано». Лише на такому рівні деталізації зацікавлені особи проєкту здатні переконатися, що система, за яку вони платять, робить те, що потрібно. І лише на такому рівні деталізації можлива успішна автоматизація тестування. Тож не варто дивитися на ці тести як на зайву роботу — краще розглядайте їх як суттєву економію часу та грошей. Тести запобігають помилкам у реалізації системи та допомагають *дізнатися*, коли ваша робота завершена.

Хто й коли пише приймальні тести?

В ідеальному світі зацікавлені сторони проєкту і служба контролю якості співпрацюють у написанні цих тестів, а розробники перевіряють їх на логічну несуперечність. У реальному світі зацікавлені сторони рідко знаходять час або бажання занурюватися на потрібний рівень деталізації, тому доручають цю справу бізнес-аналітикам, спеціалістам з контролю якості або навіть розробникам. Якщо виявиться, що тести писатимуть розробники, принаймні простежте за тим, аби це були не ті розробники, котрі займаються реалізацією функціональності, що тестується.

Бізнес-аналітики зазвичай пишуть «оптимістичні» версії тестів, бо ці тести переважно описують аспекти, що мають комерційну цінність. Служба контролю якості зазвичай пише «песимістичні» тести з перевіркою різноманітних граничних умов, винятків та аномальних випадків. І це зрозуміло, бо завдання контролю якості — дбати про все, що може піти не так.

Відповідно до принципу «пізньої точності» приймальні тести слід писати якнайпізніше, зазвичай за кілька днів до реалізації. В Agile-проєктах тести пишуться *після* вибору функцій наступної ітерації чи спринту.

Перші приймальні тести мають бути готові до першого дня ітерації. Нові тести повинні з'являтися щодня до середини ітерації, коли мають бути готові всі тести. Якщо до середини ітерації деякі приймальні тести ще не готові, слід залучити кількох розробників до їх термінового доопрацювання. Якщо це відбувається часто, включіть до команди додаткових бізнес-аналітиків та/або спеціалістів із контролю якості.

Роль розробника

Робота з реалізації певного аспекту системи починається тоді, коли для цього аспекту готові всі приймальні тести. Розробники виконують приймальні тестування та переконуються в тому, що ті не проходять. Потім вони працюють над тим, щоби повністю інтегрувати приймальні тести із системою і починають роботу з реалізації потрібного аспекту, забезпечуючи проходження тестів.

Паула: Пітере, допоможеш мені?

Пітер: Звісно, а що трапилося?

Паула: Ось наш приймальний тест. Як бачиш, він не проходить:

```
given the command LogFileDirectoryStartupCommand
given that the old_inactive_logs directory does not exist
when the command is executed
then the old_inactive_logs directory should exist
and it should be empty
```

Пітер: Так, усі результати червоні. Жоден сценарій ще не написаний. Давай я напишу перший:

```
|scenario|given the command _|cmd|
|create command|@cmd|
```

Паула: А ми вже маємо операцію createCommand?

Пітер: Так, у пакеті `CommandUtilitiesFixture`, що я написав минулого тижня.

Паула: Добре, давай запустимо тест.

Пітер *(запускає тест)*: Перший рядок став зеленим, переходимо до наступного.

Не звертайте уваги на всі ці сценарії (scenarios) і пристосування (fixtures). Це лише службовий код, який необхідно написати для зв'язування тестів із системою, що тестується. Достатньо сказати, що всі перераховані інструменти надають ті чи інші засоби пошуку за шаблоном для розпізнавання та розбору інструкцій тесту та подальшого виклику функцій, що передають дані системі, яка тестується. Усе це робиться досить просто, а отримані сценарії та пристосування можуть використовуватись у різних тестах.

Суть у тому, що розробник повинен пов'язати приймальні тести із системою, а потім забезпечити їхнє проходження.

Обговорення тестів та пасивно-агресивна позиція

Автори тестів — люди, і вони припускаються помилок. Іноді при переході до реалізації стає очевидним, що тест виглядає безглуздо. Тести бувають надто заплутаними чи громіздкими. Вони можуть базуватися на безглуздих передумовах. А іноді вони просто невірні. Усе це може бути дуже неприємним, якщо ви розробник, котрий повинен забезпечити проходження тесту.

Ваше завдання як професійного розробника — обговорити ситуацію з автором тесту для покращення. Ніколи не обирайте пасивно-агресивну позицію, коли кажете собі: «Як написано в тесті, так я і зроблю».

Пам'ятайте: професіонал зобов'язаний допомогти своїй команді створити програму настільки якісну, наскільки це можливо. А це означає, що всі мають приділяти увагу можливим помилкам та промахам колег і спільно працювати над їхнім виправленням.

Паула: Томе, із цим тестом щось не те:

```
ensure that the post operation finishes in 2 seconds.
```

Том: Як на мене, все гаразд. Наша вимога полягає в тому, що користувачі не повинні чекати більше двох секунд. У чому проблема?

Паула: Проблема в тому, що ми можемо гарантувати виконання вимоги лише у статистичному сенсі.

Том: На мою думку, це лише слова. У вимогах ясно сказано: «Дві секунди».

Паула: Правильно, і ми можемо забезпечити цей результат у 99,5 % випадків.

Том: Пауло, у вимогах цього немає.

Паула: Ми живемо в реальному світі. Я не можу надати інших гарантій.

Том: Сем буде розлючений!

Паула: Взагалі-то я з ним уже кілька разів обговорювала цю тему. Він погоджується із тим, що типова тривалість операції буде не більше двох секунд.

Том: Ну і як мені писати цей тест? Я не можу сказати: «операція зазвичай закінчується за дві секунди».

Паула: Сформулюй на статистичному рівні.

Том: Пропонуєш виконати операцію тисячу разів і перевірити, що вона забрала понад дві секунди в п'яти або менше випадках? Абсурд.

Паула: Ні, це забере дуже багато часу. А як щодо цього:

```
execute 15 post transactions and accumulate times.
ensure that the Z score for 2 seconds is at least 2.57
```

Том: А що це за z-показник?

Паула: Та просто статистика. А як тобі наступне формулювання:

```
execute 15 post transactions and accumulate times.
ensure odds are 99.5 % that time will be less than 2 seconds
```

Том: Так, це вже зрозуміліше, але чи можна довіряти математиці?

Паула: Я обов'язково включу всі проміжні обчислення до звіту з тестування, щоби ти міг перевірити їх, якщо в тебе залишаться сумніви.

Том: Добре, мене це влаштовує.

Приймальні та модульні тести

Не плутайте приймальні (acceptance) тести із *модульними (unit)*. Модульні тести пишуться розробниками для розробників. Вони є формальними архітектурними документами з описом нижнього рівня структури та поведінки коду. Їхніми читачами будуть не бізнесмени, а такі ж розробники.

Приймальні тести створюються бізнесменами *для* бізнесменів (навіть коли, зрештою, їх пишете ви, розробник). Вони є формальними описами вимог, що визначають поведінку системи з погляду бізнесу. Їх читачами є бізнесмени *та* розробники.

Можливо, у когось виникне спокуса позбутися зайвої роботи, припустивши, що тести двох видів є чимось надмірним. Але, хоча модульні та приймальні тести часто тестують те саме, жодної надмірності в них немає.

По-перше, навіть якщо вони тестують те саме, при цьому використовуються різні шляхи та механізми. Модульні тести заглиблюються у внутрішню реалізацію системи і викликають методи конкретних класів. Приймальні тести звертаються до системи на значно вищому рівні — рівні API або навіть рівні інтерфейсу користувача. Отже, шляхи виконання цих тестів дуже різняться.

Але справжня причина, через яку ці тести не можна назвати надмірними, полягає в тому, що *тестування не є їхньою головною функцією*. Той факт, що вони щось перевіряють, є вторинним. Модульні та приймальні тести насамперед є документами і лише потім — тестами. Їхня головна мета — формальне документування архітектури, структури і поведінки системи. Автоматична перевірка архітектури, структури і поведінки надзвичайно корисна, але справжньою метою є саме документування.

Графічні інтерфейси та інші складності

Графічний інтерфейс важко визначити заздалегідь. Теоретично це можливо, але рідко вдається зробити вдало. Річ у тім, що естетика — предмет суб'єктивний, а отже, мінливий. Розробники люблять *повозитися* зі своїми графічними інтерфейсами, довести їх до пуття і відшліфувати. Вони намагаються використовувати різні шрифти, кольори, макети та схеми виконання операцій. Графічні інтерфейси перебувають у постійному розвитку.

Ця обставина ускладнює написання приймальних тестів для графічних інтерфейсів. Завдання вирішується за рахунок такого проєктування системи, коли графічний інтерфейс можна розглядати як API, а не як набір кнопок, повзунків, таблиць і меню. На перший погляд це здається дивним, але насправді йдеться просто про якісне проєктування.

У сфері проєктування програмних систем є принцип, що зветься принципом єдиного обов'язку (SRP, Single Responsibility Principle). Він проголошує, що при проєктуванні слід розділяти аспекти системи, які можуть змінюватися з різних причин, і групувати разом ті аспекти, що змінюються за тими самими причинами. Графічні інтерфейси не є винятком.

Макет, формат і схема виконання операцій у графічному інтерфейсі можуть змінюватися з естетичних причин або з міркувань ефективності, але базова функціональність графічного інтерфейсу залишається незмінною. Тому при написанні приймальних тестів для графічного інтерфейсу варто користуватись базовими абстракціями, котрі не дуже часто змінюються.

Наприклад, на сторінці може бути кілька кнопок. Замість створення тестів, що імітують натискання кнопок за їх положенням на сторінці, слід передбачити можливість імітації натискань з ідентифікацією за іменами. Ще краще, якщо кожна кнопка матиме унікальний ідентифікатор. При написанні тесту набагато зручніше обирати кнопку з ідентифікатором `ok_button`, ніж кнопку у стовпці 3 рядка 4 таблиці елементів.

Вибір вдалого інтерфейсу для тестування

І все ж таки краще писати тести, що активізують функції тестованої системи через API, а не через графічний інтерфейс. При цьому має застосовуватися той самий API, що використовується графічним інтерфейсом. Тут немає нічого нового: фахівці в галузі проєктування десятиліттями казали нам, що графічний інтерфейс слід відокремлювати від бізнес-логіки.

Тестування через графічний інтерфейс завжди створює проблеми, якщо ви не обмежуєтеся тестуванням *лише* графічного інтерфейсу. Річ у тім, що графічний інтерфейс з великою ймовірністю зміниться, а це робить тести дуже крихкими. Коли кожна зміна інтерфейсу порушує роботу тисячі тестів, ви починаєте відмовлятися від тестів або припиняєте змінювати графічний інтерфейс. Жоден із цих варіантів вдалим не назвеш. Отже, пишіть тести бізнес-логіки так, аби вони проходили через API, що міститься за графічним інтерфейсом.

Деякі приймальні тести визначають поведінку самого графічного інтерфейсу. Звісно, такі тести повинні проходити через графічний інтерфейс. Однак вони не тестують бізнес-логіку, а отже, не вимагають її зв'язування з графічним інтерфейсом. Тому на час тестування власне графічного інтерфейсу краще ізолювати його від бізнес-логіки і замінити останню заглушками.

Обмежтеся мінімальним обсягом тестів графічного інтерфейсу. Вони занадто крихкі через мінливість графічного інтерфейсу. Що більше у вас тестів графічного інтерфейсу, то менша ймовірність того, що вони залишаться незмінними.

Безперервна інтеграція

Простежте за тим, аби всі модульні та приймальні тести запускалися кілька разів на день у механізмі *безперервної інтеграції*. Цей механізм повинен ініціюватися системою керування вихідним кодом. Щоразу, коли хтось вносить у базу новий модуль, механізм безперервної інтеграції має ініціювати складання з наступним запуском усіх тестів у системі. Результати запуску мають надсилатися всім членам команди.

Стоп-сигнал

Дуже важливо, щоби тести безперервної інтеграції виконувалися постійно. Вони мають завжди проходити без помилок. У разі помилки вся команда припиняє займатися поточними справами і спрямовує зусилля на те, щоби забезпечити успішне проходження всіх тестів. Складання, що завершилося помилкою в системі з безперервною інтеграцією, має розглядатися як екстрена подія, своєрідний «стоп-сигнал».

Я спілкувався з командами, які недостатньо серйозно ставилися до помилок у тестах. Такі команди зазвичай були «надто зайняті», щоби розв'язати проблему негайно, тому тести відкладалися до кращих часів. В одному випадку непрацюючі тести були просто відключені, бо розробників дратували повідомлення про помилки. Пізніше, вже після передачі продукту замовникові, вони раптом згадали, що забули повернути тести до збірки. Це з'ясувалося вже після того, як розлючений замовник закидав їх повідомленнями про помилки.

Висновки

Передавати інформацію щодо подробиць складно. Це стосується розробників і зацікавлених сторін проєкту, які обговорюють подробиці програми. Занадто вже легко кожній зі сторін махнути на все рукою і *вважати*, що інша сторона її розуміє. Надто вже часто сторони погоджуються на тому, що зрозуміли одна одну, і розходяться з абсолютно різними поглядами і переконаннями.

Мені відомий лише один спосіб ефективного виключення комунікаційних помилок у спілкуванні розробників із зацікавленими сторонами проєктів — написання автоматизованих приймальних тестів. Ці тести формалізовані, повністю однозначні і залишаються синхронізованими із застосунком. Вони є ідеальним документом, що визначає вимоги до проєкту.

8 СТРАТЕГІЇ ТЕСТУВАННЯ

Професійні розробники тестують свій код. Проте тестування не зводиться до написання кількох модульних чи приймальних тестів. Написання цих тестів — справа корисна, але цього не достатньо. Будь-якій команді професійних розробників потрібна хороша *стратегія тестування*.

1989 року я працював у *Rational* над першою версією *Rose*. Щомісяця, або близько того, начальник служби контролю якості оголошував день «полювання на помилки». Усі учасники проєкту — від розробників до керівників, від секретарів до адміністраторів баз даних — сідали за Rose і намагалися викликати збій у програмі. За різні типи помилок були призначені призи. Помилка, що вела до аварійного завершення програми, винагороджувалася вечерею в ресторані для двох персон. Той, хто виявляв найбільше помилок, міг виграти поїздку на вихідні до Монтеррея.

Контроль якості не повинен нічого виявити

Я вже казав це колись і скажу знову. Незважаючи на те що у вашій компанії може існувати окремий підрозділ контролю якості, котрий займається тестуванням програмних продуктів, команда розробників повинна прагнути, щоби контроль не виявив жодних дефектів.

Звісно, ця мета навряд чи постійно досяжна. Якщо група розумних людей рішуче і непохитно прагне виявити всі дефекти і недоліки продукту, вона, напевно, щось знайде. І все ж таки, щоразу, коли служба контролю якості щось виявляє, це має стати тривожним сигналом для розробників. Вони повинні запитати себе, як це сталося, і вжити заходів для запобігання подібним негараздам у майбутньому.

Служба контролю якості — це частина команди

Зі сказаного вище може скластися враження, що служба контролю якості і група розробки протистоять одна одній, а їхні стосунки мають антагоністичний характер. Цього не повинно бути.

Служба контролю якості і група розробки спільно працюють заради підвищення якості системи. Оптимальна роль для фахівців з контролю якості полягає у створенні специфікацій та опису параметрів системи.

Служба контролю якості як специфікатор

Служба контролю якості повинна працювати разом із бізнес-стороною для створення автоматизованих приймальних тестів, що є справжньою специфікацією і задокументованими вимогами до системи. Послідовно, від ітерації до ітерації, вони отримують інформацію про вимоги із боку бізнесу та перетворюють їх на тести, які описують бажану для розробників поведінку системи (див. Розділ 7). Зазвичай бізнес-сторона створює «оптимістичні», а служба контролю якості — «песимістичні» тести з перевіркою різноманітних граничних умов, винятків та аномальних випадків.

Створення опису параметрів системи

Ще одна роль служби контролю якості — застосування дисципліни дослідницького тестування[1] для опису характеристик істинної поведінки працюючої системи та передачі інформації про неї групі розробки та бізнес-стороні. У цій ролі служба контролю якості *не* займається інтерпретацією вимог — вона ідентифікує фактичну поведінку системи.

Піраміда автоматизації тестування

Професійні розробники зазвичай застосовують для створення модульних тестів методологію розробки через тестування (TDD). Команди професійних розробників використовують приймальні тести для складання специфікації системи і механізм безперервної інтеграції (див. Розділ 7) для запобігання регресії. Однак ці тести є лише частиною загальної картини. Хай би якими корисними були модульні і приймальні тести, нам також знадобляться тести вищого рівня, що стежитимуть за тим, аби контроль якості не виявляв жодних дефектів. На рис. 8.1 зображено піраміду автоматизації тестування[2] —

[1] http://www.satisfice.com/articles/what_is_et.shtml.
[2] Cohn M. Succeeding with Agile, Boston, MA: Addison-Wesley, 2009. P. 311–312.

графічне представлення різноманітних тестів, необхідних у професійно організованій розробці.

Рис. 8.1. Піраміда автоматизації тестування

Модульні тести

У фундаменті піраміди розташовуються модульні тести. Вони пишуться розробниками для розробників мовою програмування системи. Метою цих тестів є визначення специфікації системи на нижньому рівні. Розробники пишуть ці тести перед написанням робочого коду з метою розкриття своїх планів. Виконання тестів у контексті безперервної інтеграції допомагає простежити, щоби вихідні наміри розробників були успішно реалізовані.

Модульні тести повинні забезпечувати настільки близьке до 100 % покриття коду, наскільки це є можливим. Загалом покриття має перевищувати 90 %, причім воно має бути *реальним* — на відміну від псевдо-тестів, що виконують код без перевірки тверджень стосовно його поведінки.

Компонентні тести

До цієї категорії входить частина приймальних тестів, що згадувалися в попередньому розділі. Зазвичай ці тести пишуться для окремих компонентів системи. У компоненти системи інкапсулюються бізнес-правила, тому тести компонентів стають приймальними тестами для бізнес-правил.

Як показано на рис. 8.2 компонентний тест ізолює окремий компонент системи. Він передає компоненту вхідні дані, отримує від нього результати та перевіряє їх на відповідність вхідним даним. Для ізоляції інших компонентів системи використовуються макети та тест-дублери.

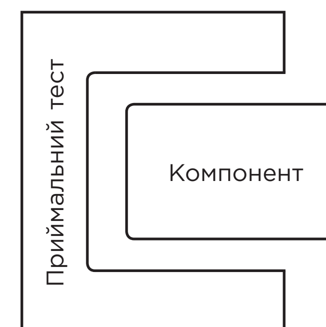


Рис. 8.2. Компонентний приймальний тест

Компонентні тести пишуться фахівцями з контролю якості та бізнес-стороною за допомогою розробників. Вони формуються в середовищах компонентного тестування на кшталт FitNesse, JBehave або Cucumber. (Компоненти графічного інтерфейсу тестуються у спеціалізованих середовищах на кшталт Selenium або Watir.) Тести мають бути написані так, аби бізнес-сторона могла читати та інтерпретувати їх — або навіть створювати самотужки.

Компонентні тести забезпечують покриття приблизно половини системи. Вони переважно орієнтовані на «оптимістичні» ситуації та найбільш очевидні граничні умови, винятки та аномальні ситуації. Переважна більшість «песимістичних» ситуацій покривається модульними тестами, і на рівні компонентних тестів вони не мають сенсу.

Інтеграційні тести

Тести цієї категорії мають сенс лише у великих системах із безліччю компонентів. Як видно із рис. 8.3, ці тести групують компоненти і тестують взаємодії між ними. Інші компоненти системи, як завжди, ізолюються за допомогою відповідних макетів та тест-дублерів.

Інтеграційні тести є *«хореографічними»*: вони не тестують бізнес-правила, а лише перевіряють, наскільки добре компоненти групи «танцюють» один з одним. Вони, як *техніки*, перевіряють правильність зв'язування компонентів та обміну інформацією між ними.

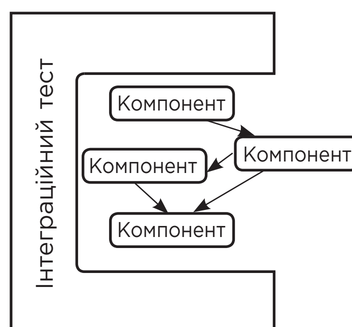


Рис. 8.3. Інтеграційний тест

Інтеграційні тести зазвичай пишуться системними архітекторами (або провідними проєктувальниками) системи. Вони перевіряють ґрунтовність архітектурної структури системи. Саме на цьому рівні зазвичай зустрічаються тести продуктивності і пропускної спроможності.

Інтеграційні тести в переважній більшості випадків пишуться тією ж мовою і в тому ж середовищі, що й компонентні тести. Зазвичай вони *не* виконуються в контексті безперервної інтеграції, бо забирають надто багато часу. Натомість ці тести виконуються періодично з інтервалами, що визначаються їх авторами.

Системні тести

Це автоматизовані випробування, що перевіряють роботу всієї інтегрованої системи. Фактично вони є граничним випадком інтеграційних тестів. Системні тести безпосередньо не перевіряють бізнес-правила. Натомість вони перевіряють, чи правильно пов'язані один з одним компоненти системи, і чи взаємодії між ними відбуваються за вихідним планом. Тести продуктивності та пропускної спроможності зазвичай відносять саме до цієї категорії.

Ці тести пишуться системними архітекторами і провідними технічними спеціалістами. Зазвичай вони пишуться тією ж мовою і в тому ж середовищі, що й інтеграційні тести інтерфейсу користувача. Системні тести виконуються відносно рідко (це пов'язане із тривалістю їх виконання), але що частіше — то краще.

Системні тести покривають 10 % системи. Це пояснюється тим, що вони призначені для перевірки правильності *конструкції*, а не поведінки системи. Правильність поведінки розташованого нижче коду й компонентів вже була перевірена на нижчих рівнях піраміди.

Ручні дослідницькі тести

У цій категорії розробникам доводиться попрацювати за клавіатурою і монітором. Дослідницькі випробування не автоматизуються і *не оформлюються у сценарії*. Вони призначені для дослідження системи з метою виявлення несподіваної поведінки і підтвердження наявності очікуваної поведінки. Тому для дослідження системи у цьому аспекті потрібен людський мозок із властивим йому творчим мисленням. Складання письмового плану для тестування такого роду суперечить його меті.

Деякі команди мають фахівців для виконання цієї роботи. В інших командах просто оголошується «день полювання на помилки», коли максимальна кількість людей, включно з керівниками, секретарями, розробниками, тестувальниками й авторами технічної документації, «знущаються» над системою, намагаючись її «зламати».

Покриття коду не входить до цілей дослідницького тестування. У цьому випадку ми взагалі не збираємося перевіряти всі бізнес-

правила і всі шляхи виконання. Метою є перевірка нормальності поведінки системи під час використання людиною, а також творче виявлення якнайбільшої кількості «дивних особливостей».

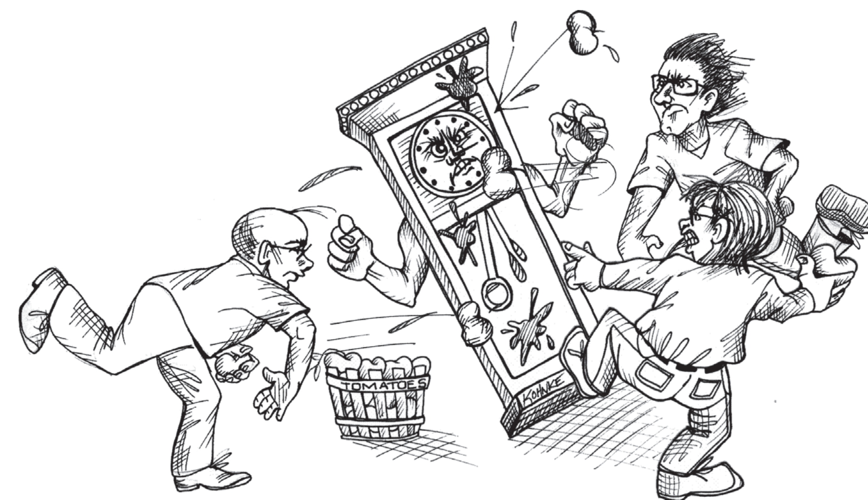
Висновки

TDD — потужна методологія, а приймальні тести — цінний засіб формулювання і перевірки дотримання вимог. Але вони є лише частиною загальної стратегії тестування. Для досягнення мети (коли служба контролю якості не в змозі виявити дефекти) команда розробників повинна працювати у тісній взаємодії із цією службою заради створення ієрархії модульних, компонентних, інтеграційних, системних і дослідницьких тестів. Тести слід виконувати якнайчастіше, щоби забезпечити максимальний зворотний зв'язок і гарантувати постійну «чистоту системи».

Бібліографія

Cohn M. *Succeeding with Agile*, Boston, MA: Addison-Wesley, 2009.

9 ТАЙМ-МЕНЕДЖМЕНТ



Вісім годин — напрочуд короткий проміжок часу. Це лише 480 хвилин, або 28 800 секунд. Ви як професіонал повинні використовувати ці дорогоцінні секунди якомога ефективніше. Яку стратегію ви оберете, щоби уникнути марнування свого часу? Як організувати ефективне управління часом?

У 1986 році я мешкав у Літл-Сендхерсті (Велика Британія), керуючи відділом розробки компанії *Teradyne*, у якому працювало 15 осіб. Мої дні здавалися суцільним хаосом, що складався з телефонних дзвінків, імпровізованих зустрічей, обговорення проблем обслуговування «на місцях» і непередбачених подій, які переривали мою основну роботу. Для того щоби моя робота хоч якось виконувалася, мені довелося встановити жорстку дисципліну тайм-менеджменту.

- Я прокидався о п'ятій ранку і їхав велосипедом до свого офісу у Брекнеллі до шостої години. Завдяки цьому я мав 2,5 спокійних годин до того, як починався щоденний хаос.
- Прибувши на місце, я записував на дошці порядок денний. Потім поділяв робочий час на 15-хвилинні інтервали й позначав, чим займатимусь у кожному з інтервалів.
- Перші три години мого розкладу були розплановані. Після дев'ятої ранку я залишав один вільний 15-хвилинний інтервал на годину, що дозволяло мені без зусиль перенести більшість відволікаючих подій в один із вільних інтервалів і продовжити роботу.
- Час після обіду залишався нерозпланованим, бо я знав: до цього моменту хаос сягне апогею, й останню частину дня мені доведеться працювати в реактивному режимі. У ті нечисленні денні періоди, коли ніщо не завадило моїй роботі, я просто працював над найважливішими завданнями, поки не траплялося щось непередбачене.

Ця схема не завжди працювала успішно. Прокидатися о п'ятій ранку вдавалося не щодня, а інколи непередбачені обставини порушували мою ретельно продуману стратегію і поглинали весь день. І все ж таки здебільшого мені вдавалося триматися на поверхні.

Зустрічі

Зустрічі обходяться приблизно у 200 доларів на годину на кожного учасника. Тут ураховані зарплатні, премії, витрати на використання приміщення тощо. Коли ви наступного разу проводитимете робочу зустріч, порахуйте витрати на неї. Результат вас вразить.

Дві істини про зустрічі:

1. Зустрічі потрібні.
2. Зустрічі часто виявляються марною тратою часу.

Часто ці дві істини стосуються тієї ж самої зустрічі. Одним учасникам зустріч здається безцінною, іншим — зайвою або марною.

Професіонали знають, що зустрічі коштують дорого. Вони також знають, що їх власний час дорогоцінний: їм потрібно писати код, дотримуючись графіку. Із цієї причини вони активно опираються участі у зустрічах, що не надають негайної та відчутної користі.

Відмова від участі

Ви не мусите відвідувати кожну зустріч, на яку вас запрошують. Ба більше, відвідувати забагато зустрічей — непрофесійно. Час треба витрачати розумно. Будьте вкрай обережними з вибором зустрічей, які слід відвідати або від яких варто чемно відмовитися.

Людина, яка запрошує вас на зустріч, не відповідає за ваш тайм-менеджмент. За нього відповідаєте лише *ви*. Отже, отримавши запрошення, не приймайте його, якщо ваша участь не обумовлена нагальною та істотною необхідністю для поточної роботи.

Іноді зустріч може бути присвячена темі, що представляє для вас інтерес, але не є необхідною. Вирішуйте самі, чи зможете приділити їй час. Будьте обережні — такі зустрічі здатні поглинути весь ваш робочий час.

В інших випадках ви можете знати, що ваша участь може принести користь, але сама по собі зустріч не є значущою для того, чим ви зараз займаєтеся. Вирішуйте, чи компенсуються витрати для вашого проєкту користю для інших проєктів. Можливо, це прозвучить

цинічно, але передусім ви повинні відповідати за *власні* проєкти. Однак взаємодопомога команд часто приносить користь, тож ви можете обговорити можливість своєї участі з командою і керівництвом.

Іноді з проханням про вашу присутність на зустрічі до вас звертається якась авторитетна особа — наприклад, технічний спеціаліст дуже високого рівня або керівник іншого проєкту. Вам доведеться вирішувати, чи переважає його авторитет порушення вашого робочого графіку. Як і в попередньому випадку, у прийнятті рішення може допомогти команда чи керівництво.

Один із найважливіших обов'язків вашого керівника — утримувати вас від участі у зайвих зустрічах. Хороший керівник охоче підтримає вашу відмову від зустрічі, бо ваш час представляє для нього таку саму цінність, як і для вас.

Як піти із зустрічі

Зустрічі не завжди проходять так, як планувалося спочатку. Іноді ви усвідомлюєте, що відмовилися би від участі у поточній зустрічі, якби знали про неї більше. Під час зустрічі можуть порушуватися нові теми або в обговоренні на передній план виходить чиясь улюблена тема. Із роками я виробив просте правило: якщо на зустрічі стало нудно — йди з неї.

Ще раз нагадую: ви повинні ефективно керувати своїм часом. Якщо ви застрягли на зустрічі і тільки гаєте час, знайдіть спосіб ввічливо залишити її.

Звісно, не слід обурено вибігати із приміщення, галасуючи: «Яка бісова нудьга!» Не треба демонструвати неповагу. Просто запитайте у відповідний момент, наскільки важливою є ваша присутність. Поясніть, що не можете приділити більше часу і запитайте, чи не можна пришвидшити обговорення або змінити порядок денний.

Важливо розуміти, що залишатися на зустрічі, що марнує ваш час, і в перебіг якої ви не можете зробити хоч якийсь помітний внесок, просто непрофесійно. Ви мусите розумно витрачати час і гроші свого роботодавця, тому в обговоренні можливості залишити нараду немає нічого шокуючого.

Мати порядок денний і мету

Причина, із якої ми змиряємося з витратами на проведення зустрічей, полягає в тому, що іноді просто необхідно зібрати всіх учасників проєкту в одній кімнаті заради досягнення конкретної мети. Але для того, щоби час учасників витрачався ефективно, зустріч повинна мати чіткий порядок денний із визначенням кінцевої мети і часу обговорення кожної теми.

Якщо вас просять відвідати зустріч, переконайтеся, що знаєте, які теми будуть обговорюватися, скільки часу їм приділятиметься і яка мета має бути досягнута. Якщо вам не вдасться отримати чіткі відповіді на ці питання, ввічливо відмовтеся від участі.

Якщо з'явившись на зустріч ви виявите, що порядок денний порушений або змінений, запропонуйте відкласти нову тему і повернутися до первісного порядку денного. Якщо це неможливо, ввічливо залиште зустріч за першої-ліпшої нагоди.

Стендапи

Стендап-зустрічі є частиною канону agile-методологій. Їхня назва пов'язана із тим, що учасники під час таких коротких зустрічей зазвичай навіть не сідають. Усі учасники по черзі відповідають на три питання:

1. Що я зробив учора?
2. Що я робитиму сьогодні?
3. Що заважає мені в реалізації плану?

От і все. Відповідь на кожне питання має тривати *не більше* 20 секунд, тому на кожного учасника витрачається не більше хвилини. Навіть у команді, що складається з десяти осіб зустріч буде завершена через 10 хвилин.

Зустрічі із планування ітерацій

Це найскладніший різновид зустрічей у каноні гнучких методологій. За поганого виконання вони забирають занадто багато часу. Для

ефективного проведення таких зустрічей потрібні відповідні навички, що варто засвоїти.

На зустрічах із планування ітерацій зі списку задач, що призначаються для реалізації, треба вибирати тільки ті елементи, які мають бути виконані під час наступної ітерації. Оцінки та прогнози комерційної цінності робляться заздалегідь. За вдалої організації роботи приймальні/компонентні тести вже повинні бути написані або існувати у чорнових варіантах.

Зустріч має відбуватися максимально швидко: стисле обговорення кожної позиції списку реалізації, після чого позиція або обирається, або відхиляється. На жоден елемент не повинно витрачатися більше 5–10 хвилин. Якщо існує потреба в докладнішому обговоренні, заплануйте його на інший час із частиною команди.

Я додержуюся простого емпіричного правила: зустріч не має потребувати понад 5 % загального часу ітерації. Отже, для тижневої ітерації (40 годин) зустріч має завершуватися приблизно за дві години.

Ретроспективні зустрічі щодо ітерацій та демо

Ці зустрічі проводяться наприкінці кожної ітерації. Члени команди обговорюють, що зроблене вдало, а що — погано. Власникам продукту демонструються нові функції застосунку. За хибної організації ці зустрічі можуть забирати багато часу; плануйте їх десь за 45 хвилин до завершення останньої ітерації. Виділяйте не більше 20 хвилин на ретроспективу і 25 хвилин на демонстрацію. Не забувайте, що минули лише один-два тижні, тож матеріалу для обговорення не так уже і багато.

Аргументи/розбіжності

Кент Бек одного разу сказав дуже важливу річ: «Будь-яка суперечка, яку не вдається завершити за п'ять хвилин, не може бути вирішена шляхом обговорення». Якщо суперечка забирає забагато часу, то це означає, що бракує чітких доказів на користь однієї зі сторін. У таких ситуаціях суперечка зазвичай має переважно метафізичне підґрунтя, а не базується на фактах.

Технічні розбіжності часом не мають меж. Кожна сторона висловлює численні обґрунтування своєї позиції, що рідко підкріплюються даними. Без конкретних даних будь-яка суперечка, що не завершується згодою за кілька хвилин (від 5 до 30), просто не здатна привести до згоди. Єдине, що можна зробити у цій ситуації — знайти потрібні дані.

Деякі люди намагаються отримати перевагу в суперечці за рахунок демонстрації характеру. Вони підвищують голос, намагаються тиснути або зображують поблажливість. Це не принципово, адже будь-яка сила волі не здатна тривало впливати на прийняття рішення. А дані — здатні.

Дехто постає пасивно-агресивну позицію. Ці люди на все погоджуються просто для того, щоби завершити суперечку, а потім саботують результат, відмовляючись брати участь у розв'язанні задачі. Вони кажуть собі: «Вони на цьому наполягали, от тепер нехай самі й розсьорбують». Це, схоже, найгірший із варіантів непрофесійної поведінки. Ніколи, ніколи не дійте таким чином. Якщо ви погодилися, то *зобов'язані* взяти участь.

Як отримати дані, необхідні для усування розбіжностей? Іноді можна влаштовувати експеримент, імітацію чи моделювання. А іноді краще просто кинути монетку, щоби обрати один із двох шляхів. Якщо все вийде — обраний шлях був працездатним. Якщо виникли проблеми — поверніться та спробуйте інший. Заздалегідь погодьте час та набір критеріїв, за якими будуть ухвалюватися рішення про відмову від обраного шляху.

Остерігайтеся зустрічей, що проводяться тільки заради силового усунення розбіжностей і підтримки однієї зі сторін. Також уникайте зустрічей, на яких присутня лише одна зі сторін, що сперечаються.

Якщо суперечку необхідно врегулювати, попросіть кожну сторону викласти свої аргументи команді протягом п'яти хвилин, а потім проведіть голосування. Тоді вся зустріч потребуватиме трохи більше чверті години.

Мана концентрації

Вибачте, якщо цей розділ трохи тхнутиме метафізикою «епохи Водолія» чи рольовою грою «Dungeons & Dragons». Річ у тім, що особисто я ставлюся до цього питання саме так.

Програмування — інтелектуальна діяльність, що вимагає тривалих періодів зосередження та концентрації. Концентрація — цінний ресурс, своєрідна *мана*[1]. Після того як ви витратите власну ману концентрації, вам доведеться відновлювати її протягом години і більше, займаючись нескладною роботою.

Я не знаю, що таке насправді мана концентрації (focus-mana), але мені здається, що це якась фізична субстанція (а можливо, її відсутність), що впливає на уважність і сприйнятливість. Та хай би чим вона була, ви *відчуваєте*, коли вона є, і так само відчуваєте, коли її нема.

Професійні розробники навчаються керувати своїм часом так, аби повною мірою використовувати свою ману концентрації. Ми пишемо код, коли маємо великий ресурс мани, а коли її лишається замало — займаємось іншими, менш творчими справами.

З іншого боку, мана концентрації — недовговічний ресурс. Якщо не використовувати її, поки вона є, то, найімовірніше, ви її втратите. Але якщо ви витратите всю ману концентрації на зустрічі, то у вас залишиться обмаль ресурсів для програмування.

Занепокоєння та роздратування також призводять до втрати мани концентрації. Учорашня сварка з дружиною, подряпина на крилі машини, рахунок, що ви забули оплатити минулого тижня,— усе це призводить до швидкого спадання кількості мани.

[1] Мана — стандартний ресурс у фентезійних та рольових іграх типу «Dungeons & Dragons». Кожен гравець має певну кількість мани — магічної субстанції, що витрачається на застосування заклинань. Що потужніше заклинання, то більше мани витрачається на нього. Відновлення мани відбувається повільно, із фіксованим щоденним збільшенням. Недосвідчені гравці здатні необережно витрачати всю ману за кілька застосувань.

Сон

Важливість сну неможливо переоцінити. Гарний нічний сон відновлює більшу частину мани концентрації. Семигодинний сон часто повертає мені повну восьмигодинну дозу мани. Професійні розробники керують графіком свого сну так, аби резерв мани концентрації досягав максимуму у той час, коли вранці вони розпочинають працювати.

Кофеїн

Безперечно, помірні дози кофеїну підвищують ефективність використання мани концентрації. Але будьте обережні! Кофеїн також призводить до дивної нестійкості вашої концентрації. Надлишок кофеїну зміщає вашу увагу в непередбачуваних напрямках. Суперпотужна кофеїнова стимуляція нерідко призводить до того, що ви витрачаєте весь день, концентруючись на малозначних речах.

Використання кофеїну й толерантність до нього — особиста справа. Я, наприклад, задовольняюся чашкою міцної кави вранці та дієтичною колою під час ланчу. Іноді ця доза подвоюється, але я перевищую її дуже рідко.

Перезаряджання

Ману концентрації можна частково поповнити за рахунок відволікаючих занять. Добра довга прогулянка, бесіда із друзями, звичайний погляд із вікна — все це допоможе відновити резерв мани.

Дехто медитує. Іншим до вподоби відновлювальний сон. Ще хтось слухає подкасти або гортає журнали.

Мій досвід свідчить, що після того, як мана скінчилася, зберігати концентрацію за рахунок вольових зусиль неможливо. Ви можете писати код, але наступного дня вам майже напевно доведеться його переписувати. Краще витратити пів години або навіть годину на перепочинок.

Фізичні вправи

У фізичних вправах на кшталт бойових мистецтв, тайцзицюань або йоги є щось особливе. Всі вони вимагають значної концентрації, але ця концентрація істотно відрізняється від необхідної для програмування, бо має не інтелектуальну, а фізіологічну природу. Інколи саме фізіологічна концентрація допомагає відновити розумову концентрацію, і це щось більше, ніж звичайна перезарядка. Я виявив, що регулярні фізичні вправи збільшують мій резерв інтелектуальної концентрації.

Моєю улюбленою формою фізичної концентрації є велоспорт. Я катаюсь на велосипеді годину чи дві, іноді долаючи по 20–30 кілометрів. Мій маршрут пролягає стежкою, паралельною річці Дес-Плейнс, тож мені не доводиться мати справу з автівками.

Під час поїздок я слухаю подкасти про астрономію і політику. Іноді я просто ставлю улюблену музику, а іноді знімаю навушники і слухаю голоси природи.

Дехто вважає за краще попрацювати руками. Одні захоплюються столярною справою, інші — створенням авіамоделей або садівництвом. Утім у будь-якій фізичній роботі, незалежно від її виду, теж присутнє щось таке, що допомагає нам працювати головою.

Ввід vs вивід

Також мені здається важливим, аби мої результати (вивід) підживлювалися відповідним надходженням (вводом). Створення програмного коду — *творча робота*. Зазвичай мої творчі здібності найбільше виявляються тоді, коли я стикаюся із творчою працею інших людей. Тому я читаю багато наукової фантастики. Творча уява авторів якимсь чином стимулює мої власні творчі здібності в галузі програмування.

Розподіл часу та помідори

Для управління своїм часом та концентрацією я також використовую надзвичайно ефективний *помідорний метод планування*[1] (Pomodoro Technique). Основна ідея надзвичайно проста: ви ставите стандартний кухонний таймер (зазвичай такі таймери мають вигляд помідора) на 25 хвилин. Протягом роботи таймера *ніщо* не повинно заважати вашій роботі. Якщо хтось телефонує, ви відповідаєте і ввічливо просите зателефонувати через 25 хвилин. Якщо хтось звертається до вас із питанням, ви чемно пропонуєте зайти через 25 хвилин. Незалежно від виду переривання воно відкладається до моменту спрацьовування таймера. Навряд чи існує так багато невідкладних справ, що не можуть почекати 25 хвилин!

[1] http://www.pomodorotechnique.com/

За сигналом таймера-помідора ви *негайно* припиняєте поточну роботу. Настав час розібратися з усіма проблемами, що виникли під час роботи таймера. Потім ви робите перерву приблизно на п’ять хвилин, знову ставите таймер на 25 хвилин і починаєте наступний цикл. Кожен четвертий цикл відпочинку стає триваліשим — 30 хвилин або близько того.

Цей метод управління часом добре описаний; я рекомендую ознайомитися з ним докладніше. Проте навіть цей короткий опис дає початкове уявлення про «помідорну методику».

При використанні цього методу ваш час ділиться на «помідорний» і «не-помідорний». «Помідорний» час продуктивний. Саме в ньому виконується справжня робота. Решта часу витрачається на відволікаючі фактори, зустрічі, перерви або іншу діяльність, не пов’язану безпосередньо з виконанням ваших завдань.

Скільки «помідорів» можна зробити за день? У хороші дні — 12, навіть 14. У погані може вийти лише 2 або 3. Якщо підрахувати «помідори» та побудувати графік, ви швидко отримаєте уявлення про те, яка частина робочого часу витрачається продуктивно, а яка марнується на «всілякі дрібниці».

Деякі професіонали настільки звикають до цієї методики, що оцінюють витрати часу на свої завдання в «помідорах» й обчислюють свою щотижневу «помідорну швидкість». Однак це лише додаткові плюси. Головна перевага «помідорної методики полягає в тому, що 25-хвилинне «вікно» продуктивного часу агресивно захищене від будь-яких втручань.

Уникання

Іноді у вас просто «душа не лежить» до роботи. Наприклад, доручене завдання вас відлякує, здається незручним або нудним. А може, вам здається, що воно призведе до конфліктів чи неминуче поставить вас у складну ситуацію. Або вам просто не хочеться його виконувати.

Інверсія пріоритетів

Незалежно від причини, ви завжди знайдете спосіб уникнути виконання небажаної роботи. Скажімо, переконаєте себе, що маєте більш термінові завдання, і звернетеся до них. Саме це й називається *інверсією пріоритетів*. Ви штучно підвищуєте пріоритет наявного завдання, аби відкласти виконання іншого завдання, що має справжній пріоритет. Інверсія пріоритетів — це брехня, звернена до себе. Нам не вистачає сміливості підступитися до необхідного і нагального, і ми переконуємо себе, що інше завдання важливіше. Насправді ми знаємо, що це не так, і все одно обманюємо себе.

Хоча точніше буде сказати, що дуримо ми зовсім не себе. Просто таким чином ми готуємо комплект брехні для тих, хто запитає нас, чим ми займаємося і чому займаємось саме цим. Ми наперед готуємо захист від осуду із боку колег. Звісно, така поведінка є непрофесійною.

Професіонал оцінює пріоритет кожного завдання незалежно від своїх особистих побоювань чи переваг і виконує ці завдання у пріоритетній послідовності.

Безвихідь

Безвихідь — явище, добре знайоме всім розробникам програмних продуктів. Іноді ви приймаєте рішення і рухаєтеся в напрямку, що приводить вас у нікуди. Що більше завзятості ви виявляєте, обстоюючи своє рішення, то далі зайдете на цьому шляху. А якщо на кону опинилася ваша професійна репутація, то зможете блукати довіку.

Досвід і розсудливість допоможуть уникнути деяких глухих кутів, але взагалі уникнути їх все одно не вдасться. Тож насправді вам треба якнайшвидше зрозуміти, що ваш шлях веде у глухий кут, і набратися сміливості для відступу. Дехто називає це «*правилом ями*»: якщо ви опинилися в ямі, насамперед припиніть копати далі.

Професіонал ніколи не захоплюється ідеєю настільки, щоби в нього не вистачило хисту відмовитися від неї і повернутися до вихідної точки. Він неупереджено ставиться до інших ідей, аби в нього залишалися додаткові варіанти на випадок, якщо він усе ж таки опиниться в безвиході.

Бруд, болото та інший безлад

Безлад ще гірший за глухий кут. Він уповільнює вас, але не зупиняє повністю. Безлад перешкоджає вашому просуванню, але ви все одно можете рухатися вперед, діючи методом «форсажу». Безлад небезпечніший, ніж тупик, бо ви постійно бачите шлях попереду, і він здається вам коротшим за шлях назад (хоча, насправді, це й не так).

Я бачив продукти і цілі компанії, знищені безладом у програмному коді. Я бачив, як продуктивність роботи команд знижувалася мало не до нуля за якихось кілька місяців. Ніщо не має більш руйнівного та довгострокового негативного впливу на продуктивність групи розробників, ніж безлад у програмному коді. Ніщо!

Проблема полягає в тому, що виникнення безладу, як і безвиході, є неминучим. Досвід і розсудливість допоможуть вам деякий час уникати його, але рано чи пізно ви ухвалите рішення, що заведе вас у безлад.

Виникнення безладу вкрай підступне. Ви розв’язуєте нескладну задачу, щосили прагнучи до того, щоби код залишався простим і чистим. Зі зростанням масштабу та складності завдання ви розширюєте його кодову базу, намагаючись зберігати його чистоту наскільки це можливо. Але якоїсь миті ви розумієте, що на самому початку прийняли хибне архітектурне рішення, тому ваш код погано масштабується в напрямку зсуву вимог.

Ось вона — критична точка! Ви й досі ще можете повернутися і виправити архітектуру. Але також можете продовжити рухатися вперед. Вам здається, що повернення обійдеться надто дорого, бо доведеться переробляти існуючий код, однак у майбутньому воно обійдеться ще дорожче. Якщо ви рухатиметеся вперед, то система сповзе в багно, і цілком можливо, що вже ніколи з нього не вибереться.

Професіонали бояться безладу значно більше, ніж глухих кутів. Вони завжди звертають увагу на безлад, що починає необмежено розростатися, і прикладають усі необхідні зусилля до його усунення — за можливістю, якомога раніше і швидше.

Рух уперед болотом (якщо вам відомо, що це болото) є найгіршим різновидом інверсії пріоритетів. Рухаючись уперед, ви обманюєте себе, обманюєте команду, вводите в оману свою компанію і замовників. Ви кажете їм, що все буде гаразд, хоча насправді ведете їх до загальної катастрофи.

Висновки

Професійні розробники серйозно ставляться до управління своїм часом і концентрацією. Вони розуміють спокусу інверсії пріоритетів і борються з нею, оскільки це є справою професійної честі. Вони неупереджено ставляться до альтернативних рішень. Вони ніколи не захоплюються якимсь рішенням настільки, що не здатні відмовитись від нього. А ще — вони завжди стежать за можливостями виникнення безладу й усувають їх за перших же ознак. Не можна уявити більш сумне видовище, ніж команда розробників, яка безцільно пробирається крізь трясовину, що постійно поглиблюється.

10 ОЦІНКИ

Оцінювання — один із найпростіших, але при цьому найризикованіших видів діяльності, із якими стикаються професійні розробники. Від нього безпосередньо залежить комерційна цінність проєкту. А також наші репутації. Невірне оцінювання стає причиною наших страхів і провалів. Воно також є першим клином, що вбивається поміж бізнесменами та розробниками, і джерелом майже всіх виявів недовіри, пов'язаних із цими відносинами.

У 1978 році я був провідним розробником 32-кілобайтної вбудованої програми для Z-80, написаної мовою асемблера. Програма «прошивалась» на 32 перепрограмованих мікросхемах. Ці мікросхеми вставлялися у три плати, по 12 мікросхем на кожну.

У нас були сотні пристроїв, установлених на центральних телефонних станціях по всіх Сполучених Штатах. І щоразу, як ми виправляли помилку або додавали нову функцію, нам доводилося відряджати техніків до кожного пристрою для заміни всіх 32 мікросхем!

Це був справжній жах. Мікросхеми і плати були крихкими, контакти мікросхем гнулися та ламалися. Ненавмисні деформації плат могли пошкодити місця пайки. Ризик помилок і поломок був величезний, а витрати компанії зависокі.

Саме тому Кен Файндер, мій бос, попросив мене щось зробити. Нам був потрібен спосіб заміни деяких мікросхем, який не вимагав би заміни решти. Якщо ви читали мої книжки або слухали мої лекції, то знаєте, що я багато уваги приділяю можливості незалежного розгортання. Саме тоді я вперше засвоїв цей урок.

Проблема полягала в тому, що програма являла собою єдиний скомпонований виконуваний файл. Додання нового рядка коду призводило до зміни адрес усіх наступних рядків. Оскільки в кожній мікросхемі зберігався лише один кілобайт адресного простору, то змінювався вміст практично всіх мікросхем.

Рішення здавалося досить простим. Мікросхеми потрібно було ізолювати одну від одної. Вміст кожної мікросхеми при цьому перетворювався на незалежну одиницю компіляції, яку можна було записувати незалежно від інших.

Я визначив розміри всіх функцій у застосунку й написав просту програму, що «укладала» їх, немов фрагменти пазла, на мікросхемах,

залишаючи по 100 байт вільного простору для розширення. На початку кожної мікросхеми розміщувалася таблиця вказівників на всі функції даної мікросхеми. Під час завантаження, ці вказівники переміщувалися в оперативну пам'ять. Весь код системи було змінено таким чином, аби функції ніколи не викликалися безпосередньо — лише через вектори, що зберігалися в пам'яті.

Так, ви правильно зрозуміли: ці чіпи перетворилися на аналоги об'єктів з v-таблицями, а функції викликалися поліморфно. Саме так я дізнався про деякі принципи об'єктно-орієнтованого програмування задовго до того, як познайомився з поняттям «об'єкт».

Вигода від такого рішення виявилася величезною. Ми отримали можливість обмежитися заміною окремих чіпів, а окрім цього спосіб вносити виправлення «у польових умовах», переописуючи вектори оперативної пам'яті. Це значно спростило відлагодження та оперативні виправлення.

Але я трохи відхилився від теми. Коли Кен прийшов до мене з пропозицією розв'язати проблему, він запропонував подумати про вказівники на функції. Я витратив день-два на формальний виклад ідеї, а потім подав докладний план. Він спитав, скільки часу займе робота; я відповів, що десь близько місяця.

Мені знадобилося *три місяці*.

Я напивався тільки двічі у житті і лише раз напився *ґрунтовно*. Це сталося на святкуванні Різдва в *Teradyne* 1978 року. Мені на той час було 26 років.

Святкування відбувалося в офісі *Teradyne*, що являв собою переважно відкритий рабочий простір. Усі співробітники прийшли рано, а потужна снігова буря не дозволила оркестру та кейтеринговій фірмі дістатися офісу. Натомість випивки виявилося більш ніж достатньо.

Я не дуже добре пам'ятаю той вечір, а те, що пам'ятаю, хотів би забути. І все ж таки, не можу не поділитися одним важливим моментом.

Я сидів по-турецьки на підлозі з Кеном (моєму босу тоді було 29 років, і він був тверезий), нарікаючи на те, скільки мого часу поглинула векторизація. Алкоголь вивільнив мої приховані страхи і відсутність упевненості щодо вихідного оцінювання. Сподіваюся, я не клав

голову йому на плече — але ці подробиці не дуже чітко зафіксувалися в моїй свідомості.

Пригадую, як спитав, чи не сердиться він на мене і чи не вважає, що робота забрала забагато часу. І хай би як смутно я пам'ятав той вечір, його відповідь чітко відклалася у мене в пам'яті, закарбувавшись на всі наступні десятиліття. Він сказав: «Так, я думаю, що витрачено багато часу, але бачу, що ти старанно працюєш, і робота не стоїть на місці. І нам це дійсно потрібно. Тому, ні, я не гніваюся».

Що таке оцінка?

Проблема в тому, що на оцінки можна дивитися по-різному. Бізнес зазвичай розглядає оцінки як зобов'язання. Розробники вважають їх припущеннями. Між цими точками зору існують глибокі відмінності.

Зобов'язання

Зобов'язання — це щось таке, що ви повинні зробити. Якщо ви зобов'язуєтесь зробити щось до певної дати, то до цієї дати «щось» має бути готове. Якщо для цього вам доведеться працювати по 12 годин на добу й у вихідні, нехтуючи сімейними святами,— то так тому і бути. Ви взяли зобов'язання, і його треба дотримуватися.

Професіонали не беруть зобов'язань, не маючи впевненості, що їх можна виконати вчасно. Все дуже просто: якщо вам пропонують взяти на себе зобов'язання, а ви не впевнені, що зможете впоратися із завданням, ваша професійна честь вимагає відповісти відмовою. Якщо вам пропонують зобов'язання, що, як на ваш погляд, здійсненне, але вимагатиме понаднормових, роботи у вихідні та відсутності сімейного відпочинку,— вирішуйте це питання самі. Але якщо вже пообіцяли — виконуйте.

Зобов'язанням притаманна *визначеність*. Інші люди беруть до уваги ваші зобов'язання і використовують їх як основу для створення власних планів. Порушення зобов'язань може нанести величезну шкоду вашій професійній репутації, бо само по собі є проявом непорядності, лише трохи кращим за відверту брехню.

Оцінка

Оцінка насамперед є припущенням. Вона не містить жодних зобов'язань. Ви нічого не обіцяєте. Порушення оцінки жодною мірою не шкодить вашій репутації. Ми робимо оцінювання насамперед тому, що *не знаємо*, скільки часу насправді забере робота.

На жаль, багато розробників дуже слабкі в оцінюванні. І річ не в тім, що оцінка вимагає якоїсь таємної майстерності, а в тім, що ми часто не розуміємо справжньої природи оцінки.

Оцінка — це не число, а *розподіл*.

Ось приклад:

> Майк: Скільки, за твоєю оцінкою, знадобиться для завершення роботи над Frazzle?
>
> Пітер: Три дні.

Чи справді Пітер упорається з роботою за три дні? Можливо, але наскільки можливо? Правильна відповідь: уявлення не маємо. Що мав на увазі Пітер, і чого конкретно дізнався Майк? Якщо Майк повернеться через три дні і робота виявиться невиконаною, чи мусить він здивуватися? А чому, власне? Пітер не брав на себе жодних зобов'язань. Пітер не сказав навіть того, наскільки ці три дні є більш імовірними, ніж чотири або п'ять.

А якби Майк поцікавився в Пітера, наскільки високою є ймовірність його оцінки?

> Майк: Яка ймовірність того, що ти впораєшся за три дні?
>
> Пітер: Мабуть, впораюся.
>
> Майк: Можеш назвати цифру?
>
> Пітер: П'ятдесят чи навіть шістдесят відсотків.
>
> Майк: Отже, є доволі висока ймовірність, що тобі знадобиться чотири дні?
>
> Пітер: Так. Може знадобитися навіть п'ять або шість, хоча я сумніваюся у цьому.
>
> Майк: Наскільки сумніваєшся?
>
> Пітер: Ну, я не знаю… Я на дев'яносто п'ять відсотків упевнений, що робота буде зроблена менше ніж за шість днів.

Майк: Тобто може бути і сім?

Пітер: Ну, якщо все піде шкереберть… Чорт, якщо все піде шкереберть, може бути і десять, і навіть одинадцять днів. Але ж імовірність такого збігу подій дуже мала, так?

Ми поступово починаємо наближатися до істини. Оцінка Пітера є *ймовірнісним розподілом*. У власній уяві він бачить можливість завершення завдання у такий спосіб (рис. 10.1).

Тепер ми розуміємо, чому Пітер видав вихідну оцінку у три дні. Це найвищий стовпець гістограми, і Пітерові він ввижається найбільш можливою тривалістю виконання завдання. Але Майк дивиться на цю справу інакше. Він звертає увагу на правий край гістограми, і в нього виникає занепокоєння те, що за певних несприятливих обставин Пітерові може знадобитися більше 11 днів.

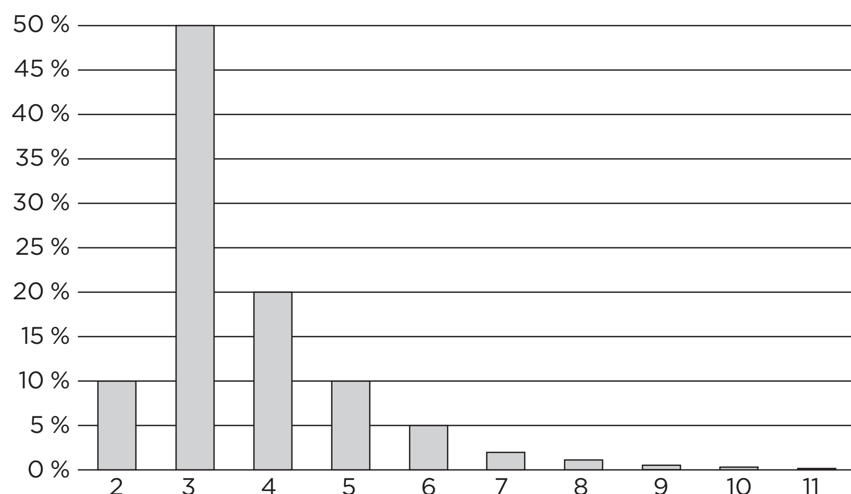


Рис. 10.1. Імовірнісний розподіл

Чи має Майк перейматися цим? Звісно! Закон Мерфі[1] ще ніхто не скасовував, тому завжди можуть виникнути непередбачені ускладнення.

[1] Закон Мерфі проголошує: якщо щось може піти шкереберть, то воно обов'язково піде шкереберть.

Передбачувані зобов'язання

Майк стикається із проблемою. Він не впевнений у тому, скільки часу знадобиться Пітерові для виконання роботи. Для того щоби звести до мінімуму невизначеність, він може спробувати домогтися від Пітера зобов'язань. Але Пітер не може щось обіцяти із цілковитою впевненістю.

Майк: Пітере, можеш назвати остаточну дату, коли задача буде виконана?

Пітер: Ні, Майку. Як я вже казав, роботу буде виконано за три, а може, за чотири дні.

Майк: Тоді пишемо чотири?

Пітер: Ні, теоретично *може бути* і п'ять, і шість.

Поки що все відбувається чесно. Майк попросив розробника взяти на себе зобов'язання, а Пітер ввічливо відмовився його давати. Тоді Майк пробує застосувати іншу тактику:

Майк: Добре, Пітере, але ти можеш хоча би спробувати вкластися у шість днів?

Прохання Майка звучить доволі безневинно, до того ж він керується найкращими намірами. Але чого саме Майк вимагає від Пітера? Що означає «спробувати»?

Ми вже обговорювали це в Розділі 2. У це слово вкладається різний зміст. Якщо Пітер погодиться, то фактично візьме на себе зобов'язання вкластися у шість днів.

Які ще інтерпретації тут можливі? Що саме Пітер збирається зробити, щоби спробувати? Він збирається працювати понад 8 годин? Очевидно, це мається на увазі, якщо він дасть згоду. Він збирається працювати у вихідні? Так, це також мається на увазі. Втратити години спілкування із сім'єю? І це також. Усе перелічене міститься у слові «спробувати». І якщо Пітер чогось не зробить, Майк зможе звинуватити його в тому, що він доклав недостатньо зусиль.

Професіонали проводять чітку межу між оцінками та зобов'язаннями. Вони намагаються не брати на себе зобов'язань, аж поки не будуть твердо впевнені в успіху. Також вони намагаються уникати

неочевидних зобов'язань і якомога чіткіше здійснюють імовірнісний розподіл своїх оцінок, аби керівники могли спиратися на них, будуючи відповідні плани.

PERT

Програма PERT (Program Evaluation and Review Technique) була створена 1957 року ВМС США для проєктування підводних човнів «Polaris». Одним з елементів PERT був спосіб обчислення оцінок. Схема PERT надає дуже простий, але винятково ефективний спосіб перетворення оцінок у ймовірні розподіли, придатні для менеджерів.

Оцінюючи завдання, ви визначаєте три числа. Це так званий *аналіз за трьома змінними*:

- *O*: оптимістична оцінка. Це значення вибирається *гранично* оптимістично. Завдання може бути виконано протягом цього часу лише в тому разі, якщо все без винятку піде гладко. Ба більше, аби математична теорія спрацювала, ймовірність такого результату має бути меншою за 1 %[1]. Як випливає з рис. 10.1, у ситуації Пітера це один день;
- *N*: номінальна оцінка (найбільш імовірна). На гістограмі вона представлена найвищим стовпцем. На рис. 10.1 номінальна оцінка складає три дні;
- *P*: песимістична оцінка. Ця оцінка має бути *гранично* песимістичною. У ній слід врахувати всі можливі неприємності, окрім ураганів, ядерної війни, блукаючих «чорних дір» та інших катастроф. Математична база також працюватиме лише в тому разі, якщо ймовірність цього результату набагато менша за 1 %. У ситуації Пітера песимістична оцінка представлена крайнім правим стовпцем. Отже, це дванадцять днів.

За цими трьома оцінками ймовірнісний розподіл описується такою формулою:

- $\mu = \dfrac{O + 4N + P}{6}$,

[1] Точне число для нормального розподілу становить 1:769, або 0,13 %, або 3 σ. Імовірно, можна безпечно говорити про можливість 1:1000.

де μ — очікувана тривалість завдання. У випадку Пітера вона становитиме (1 + 12 + 12)/6, або близько 4,2 дні. Для більшості завдань оцінка є трохи завищеною, бо права частина розподілу довша за ліву[1].

- $\sigma = \dfrac{P - O}{6}$,

де σ — середньоквадратичне відхилення[2] розподілу часу виконання завдання. Фактично це міра невизначеності завдання: якщо це число велике, то й невизначеність також велика. У нашому прикладі воно дорівнює (12 – 1)/6, або близько 1,8 днів.

Виходячи з оцінки Пітера (4,2/1,8) Майк розуміє, що завдання, найвірогідніше, буде завершено за п'ять днів, але може забрати шість і навіть дев'ять днів.

Але Майк керує не одним завданням — він адмініструє проєкт із безліччю завдань. Пітерові доручені три завдання, над якими він має працювати послідовно. Оцінки тривалості виконання цих завдань, представлені Пітером, наводяться в табл. 10.1.

Таблиця 10.1. Завдання Пітера

Завдання	Оптимістична оцінка	Номінальна оцінка	Песимістична оцінка	μ	σ
Альфа	1	3	12	4,2	1,8
Бета	1	1,5	14	3,5	2,2
Гама	3	6,25	11	6,5	1,3

Що відбувається із завданням «Бета»? Схоже, Пітер доволі впевнений у його оцінці, але непередбачені чинники можуть серйозно загальмувати його роботу. Як Майкові інтерпретувати ці результа-

[1] Передбачається, що модель PERT використовується для апроксимації β-розподілу. Це розумне припущення, бо мінімальна тривалість виконання завдання зазвичай визначається набагато точніше за максимальне. Див.: McConnell S. *Software Estimation: Demystifying the Black Art*. Redmond, WA: Microsoft Press, 2006. Fig. 1–3.

[2] Якщо ви не знаєте, що це таке, знайдіть хороший підручник з теорії ймовірностей і математичної статистики. Зрозуміти цю концепцію неважко, але вона вам знадобиться.

ти? Скільки часу потрібно запланувати для завершення всіх трьох завдань? Виявляється, що шляхом простих обчислень Майк може об'єднати всі завдання Пітера й отримати ймовірнісну оцінку для всього набору завдань.

Обчислення досить тривіальні:

- $\mu_{sequence} = \sum \mu_{task}$.

Для будь-якої послідовності задач передбачувана тривалість виконання обчислюється простим підсумовуванням тривалостей усіх завдань у послідовності. Отже, якщо Пітерові треба виконати три завдання з оцінками 4,2/1,8, 3,5/2,2 і 6,5/1,3, то найімовірніше, що на їх виконання йому знадобиться близько 14 днів: 4,2 + 3,5 + 6,5.

- $\sigma_{sequence} = \sqrt{\sum \sigma_{task}^2}$.

Середньоквадратичне відхилення послідовності дорівнює квадратному кореню із суми квадратів середньоквадратичних відхилень завдань. Отже, стандартне відхилення всіх трьох завдань Пітера дорівнює приблизно 3.

$$(1{,}8^2 + 2{,}2^2 + 1{,}3^2)^{1/2} =$$
$$= (3{,}24 + 2{,}48 + 1{,}69)^{1/2} =$$
$$= 9{,}77^{1/2} = \sim 3{,}13.$$

Із цього результату Майк дізнається, що Пітерові для виконання його завдань, імовірно, знадобиться 14 днів, але з досить великою ймовірністю може знадобитися і 17 (1s), і навіть 20 днів (2s). Виконання завдань може затягнутися і на триваліший термін, але це досить малоймовірно.

Повернемося до таблиці оцінок. Хіба вам не хочеться припустити, що всі три завдання будуть виконані за 5 днів? Адже зрештою оптимістичні оцінки дорівнюють 1,1 і 3. Навіть номінальні оцінки в сумі дають лише 10 днів. Звідки ж узялися 14 днів із можливими 17-ма і навіть 20-ма? Річ у тім, що підсумовування невизначеності у серії завдань додає *реалістичності* вихідному плану.

Будь-якому розробнику зі значним досвідом роботи відомі проєкти, що спочатку оцінювалися оптимістично, а в підсумку потребували

у 3–5 разів більше часу. Проста схема PERT — один із розумних способів запобігання таким надмірно оптимістичним очікуванням. Професіонали дуже ретельно ставляться до вибору розумних термінів, незважаючи на тиск та вмовляння *спробувати* працювати швидше.

Оцінювання завдань

Майк і Пітер припустилися жахливої помилки. Майк запитує в Пітера, скільки часу знадобиться на виконання роботи. Пітер дає чесну відповідь із трьома змінними, але як щодо думки його колег? Може, вони мають особливу думку із цього приводу?

Найважливіший ресурс оцінки — це люди, які вас оточують. Вони можуть бачити те, чого ви не помічаєте. Саме вони допоможуть вам оцінити ваші завдання точніше, ніж якби ви робили це самотужки.

Широкосмуговий дельфійський метод

У 1970-х роках Баррі Бем представив метод експертної оцінки, названий «широкосмуговим дельфійським методом» (wideband Delphi)[1]. За минулі роки з'явилося багато його різновидів — як формальних, так і неформальних. Але всі вони мають щось спільне: принцип консенсусу.

Стратегія проста. Група експертів збирається, обговорює завдання й оцінює його складність. Обговорення та оцінка повторюються доти, доки не буде досягнуто згоди. Вихідний метод, описаний Бемом, вимагає проведення кількох зустрічей і складання документів, що, як на мене, обертається зайвими зволіканнями і непродуктивними витратами. Я віддаю перевагу більш економним методам — на кшталт описаних далі.

Метод швидкого голосування

Усі учасники сидять за столом. Завдання розглядаються послідовно. Для кожного завдання проводиться коротке обговорення, описую-

[1] Boehm B. W. *Software Engineering Economics*. Upper Saddle River, NJ: Prentice Hall, 1981. P. 81.

ться можливі ускладнюючі фактори і можлива реалізація. Потім учасники опускають руку під стіл. Модератор рахує до трьох, після чого всі учасники одночасно показують від 0 до 5 пальців — залежно від того, скільки, на їхню думку, потребує розв'язання завдання.

Якщо оцінки узгоджуються, учасники переходять до наступного завдання. У протилежному випадку обговорення продовжується, щоби визначити причини розбіжностей. Цикл повторюється доти, доки оцінки не будуть узгоджені.

Згода має бути абсолютною — якщо оцінки доволі близькі, це теж добре. Наприклад, набір змішаних оцінок (3–4) вважається згодою. Але якщо всі підняли чотири пальці, крім однієї людини, яка підняла один палець, в учасників оцінювання з'являється привід для обговорення.

Масштаб оцінки визначається на початку зустрічі. Трудомісткість завдання може визначатися як кількістю пальців, так і більш складними метриками на кшталт кількості пальців, помноженої на 3, або кількості пальців у квадраті.

Одночасність у демонстрації оцінок дуже важлива. Ніхто не повинен змінювати свої оцінки під впливом оцінок інших учасників.

«Покер планування»

У 2002 році Джеймс Греннінг написав чудову статтю з описом «покеру планування»[1]. Цей різновид широкосмугового дельфійського методу став настільки популярним, що кілька компаній вдалися до створення маркетингових сувенірів у вигляді карткових колод для нього[2]. Існує навіть спеціалізований сайт planningpoker.com, що може використовуватися розподіленими групами для проведення сеансів «покеру планування» в Інтернеті.

Ідея дуже проста. Всім учасникам експертної групи видають картки з різними числами. Числа від 0 до 5 працюють досить добре; така система логічно еквівалентна системі із демонстрацією пальців.

Виберіть завдання та обговоріть його. Якоїсь миті модератор просить усіх учасників вибрати картку. Учасники групи обирають картку, що відповідає їхній внутрішній оцінці, і тримають її закритою, щоби інші не побачили номінал. Потім модератор подає сигнал показати картки.

[1] Grenning J. Planning Poker or How to Avoid Analysis Paralysis while Release Planning.— http://renaissancesoftware.net/papers/14-papers/44-planing-poker.html.

[2] http://store.mountaingoatsoftware.com/products/planning-poker-cards.

Решта — як у методі швидкого голосування. Якщо оцінки учасників узгоджуються, оцінка приймається. У протилежному разі картки повертаються в руку, а учасники продовжують обговорення завдання.

Щодо найкращого вибору номіналів карток існують цілі наукові теорії. Деякі спеціалісти використовують картки, чиї номінали визначаються числами Фібоначчі. Інші додають до колоди картки зі знаком нескінченності та знаком питання. Як на мене, п'яти карток із цифрами 0, 1, 3, 5, 10 цілком достатньо.

Споріднене оцінювання

Кілька років тому Ловелл Ліндстром показав мені незвичайний різновид широкосмугового дельфійського методу — споріднене оцінювання (affinity estimation). Я досить успішно застосовував його з різними замовниками та командами.

Усі завдання записуються на картках без жодних оцінок. Експертна група стоїть біля столу або дошки, де картки розподілені випадковим чином. Учасники групи не розмовляють між собою — лише сортують картки. Картки завдань, що забирають більше часу, переміщуються праворуч, а картки менш тривалих завдань — ліворуч.

Будь-який учасник групи може будь-якої миті перемістити будь-яку картку, навіть, якщо вона була переміщена іншим учасником. Картки, переміщені понад певну кількість разів, відкладаються вбік для обговорення.

Згодом мовчазне сортування завершується і починається обговорення. Учасники досліджують всі розбіжності щодо послідовності карток. Інколи для досягнення згоди доводиться намалювати кілька нескладних схем.

На наступному етапі між картками малюються лінії, що визначають трудомісткість завдання у днях, тижнях або умовних пунктах. Традиційно застосовуються п'ять значень, що утворюють послідовність Фібоначчі (1, 2, 3, 5, 8).

Аналіз за трьома змінними

Широкосмуговий дельфійський метод добре підходить для вибору однієї номінальної оцінки. Але, як уже зазначалося, краще мати три оцінки для створення ймовірнісного розподілу. Оптимістичну та песимістичну оцінку для кожного завдання можна дуже швидко отримати за допомогою будь-якого з різновидів широкосмугового дельфійського методу. Наприклад, якщо ви використовуєте «покер планування», попросіть групу підняти карти для песимістичної оцінки і виберіть найбільшу. Потім зробіть те саме для оптимістичної оцінки, але цього разу візьміть найменше значення.

Закон великих чисел

У будь-яких методах оцінки закладена можливість помилки. Власне, тому вони і називаються *оцінками*. Один зі способів виявлення помилок ґрунтується на *законі великих чисел*[1]. Зокрема, із цього закону випливає, що при розбитті великого завдання на кілька менших, сума незалежних оцінок менших завдань виявиться більш точною, ніж загальна оцінка великого завдання. Зростання точності пояснюється тим, що похибки оцінки менших завдань взаємно компенсуються.

Щиро кажучи, це твердження занадто оптимістичне. Помилки в оцінці зазвичай пов'язані з недооцінкою, а не з переоцінкою, тому така компенсація навряд чи може вважатися ідеальною. Проте розбиття великих завдань на менші із подальшою незалежною оцінкою все одно корисне. Деякі помилки взаємно компенсуються, а розбиття завдання допомагає краще усвідомити його суть і виявити можливі несподіванки.

Висновок

Професійні розробники ПЗ знають, як надати бізнес-стороні практичну оцінку, придатну для планування. Вони не дають обіцянок,

[1] http://en.wikipedia.org/wiki/Law_of_large_numbers.

яких не в змозі дотриматись, і не приймають зобов'язань, у дотриманні яких не впевнені.

Коли професіонал бере на себе зобов'язання, він визначає *конкретні* терміни, а потім орієнтується на них. Проте частіше професіонали все ж таки не надають таких зобов'язань. Натомість вони дають імовірнісні оцінки, що визначають передбачуваний час завершення і дисперсію.

Професійні розробники спілкуються з іншими членами команди з метою узгодження оцінок, що передаються керівництву.

У цьому розділі описані лише окремі *приклади* методів, що використовуються професійними розробниками для створення практичних оцінок. Перелік не повний, а представлені методи не обов'язково є найкращими. Це лише деякі методи, що добре працювали, коли я ними користувався.

Бібліографія

McConnell S. *Software Estimation: Demystifying the Black Art*. Redmond, WA: Microsoft Press, 2006.

Boehm B. W. *Software Engineering Economics*. Upper Saddle River, NJ: Prentice Hall, 1981.

Grenning J. Planning Poker or How to Avoid Analysis Paralysis while Release Planning.— http://renaissancesoftware.net/papers/14-papers/44-planing-poker.html

11 ТИСК

Уявіть, що опинилися у стані «позатілесних переживань»: спостерігаєте за операційним столом, на якому хірург робить операцію на вашому серці. Хірург намагається врятувати вам життя, але час обмежений — операція має бути закінчена впродовж певного терміну, інакше пацієнт помре. Це у *буквальному сенсі* дедлайн («лінія смерті» англійською).

Якої поведінки ви очікуєте від лікаря? Хочете бачити його спокійним і зібраним? Щоби він віддавав ясні, чіткі накази помічникам? Щоби пам'ятав усе, чому його вчили, і дотримувався чинних медичних норм?

А якщо лікар потіє та нервує? Жбурляє інструменти? Звинувачує керівництво клініки в нереалістичних вимогах та безперервно скаржиться на брак часу? Як він поводиться — як досвідчений професіонал чи як типовий розробник?

Професіонал залишається спокійним і рішучим навіть тоді, коли перебуває під тиском. Навіть зі зростанням тиску він вправно використовує отримані знання і методи, знаючи, що це найкращий шлях до виконання зобов'язань у встановлені терміни.

У 1988 році я працював у компанії *Clear Communications*. Це був стартап, якому так і не судилося «піднятися». Ми здійснили перший цикл фінансування, потім другий і третій.

Початкова ідея виглядала непогано, але проєктування так і лишилося на папері. Спочатку продукт складався з програмних та апаратних компонентів. Потім він перетворився на суто програмний. Програмна платформа змінилася — від ПК ми перейшли на Sparc. Замовники змінили вимоги: із високопродуктивних робочих станцій зажадали перейти на низькопродуктивні. Зрештою змінилася навіть вихідна концепція продукту — компанія відчайдушно намагалася знайти можливість отримання прибутку. Не думаю, що за ті чотири роки, коли я працював у *Clear Communications*, вони отримали хоча би цент.

Не варто й казати, що ми, розробники, постійно перебували під значним тиском. Було проведено чимало довгих ночей та ще довших вихідних за терміналами в офісі. Писалися страховидні функції на мові C у 3000 рядків. Постійно точилися запеклі суперечки із криками та переходами на особистості. Було все, що завгодно: інтриги і саботаж, удари кулаками у стіну, жбурляння маркерів, карикатури на колег і нескінченні хвилі гніву та стресів.

Терміни визначалися подіями. Нові функції нашвидкуруч готувалися до торгово-промислової виставки або випуску демоверсії для замовника. Хоч би яку дурість зажадав замовник, вона мала неодмінно опинитися в наступній демоверсії. Часу постійно бракувало, ми завжди відставали від графіка. Плани набагато перевищували наші можливості.

Працюючи по 80 годин на тиждень, можна було вважатися героєм. Зліпивши нашвидкуруч демоверсію для замовника, ви отримували заохочення. Якщо ви докладали купу зусиль, вас могли підвищити. А якщо ні — звільнити. Це ж стартап, і тут потрібно викладатися «на повну»…

Тільки уявіть — 1988 року, вже маючи двадцятирічний досвід розробника, я «купився» на це.

Я був керівником відділу розробки, і саме я говорив розробникам, які працювали в мене, що працювати потрібно ще більше і швидше. Це я був одним із тих хлопців з 80-годинним тижнем, я писав функції по 3000 рядків о другій годині ночі, поки мої діти спали вдома. Я жбурляв ручку і розлючено волав. І я звільняв людей, якщо вони не впоралися. Це було жахливо!

Якось дружина змусила мене як слід придивитися до власного зображення у дзеркалі. І те, що я там побачив, мені не сподобалося. Також дружина сказала, що бути поряд зі мною стало не дуже приємно. Мені довелося погодитись. Але все це настільки мені не сподобалося, що я в гніві вибіг із дому і подався невідомо куди. Так я простував хвилин зо тридцять, охоплений нестримним хвилюванням; а потім почався дощ.

І в моїй голові щось клацнуло. Я почав сміятися. Я сміявся із власних переживань. Я сміявся над стресами. Я сміявся з людини у дзеркалі — нещасного бовдура, який отруює життя собі та іншим в ім'я… В ім'я чого?

Відтоді все змінилося. Я припинив божевільні понаднормові, припинив роботу в умовах постійного стресу. Я перестав жбурляти ручки і писати функції на C у 3000 рядків. Я вирішив, що отримуватиму задоволення від роботи, виконуючи її добре, а не займаючись дурницями.

Я пішов зі своєї посади настільки професійно, наскільки це було можливо, і став консультантом. Відтоді я вже ніколи не називав іншу людину «начальником».

Уникання тиску

Найкращий спосіб зберігати спокій під тиском — уникати ситуацій, що *створюють* тиск. Можливо, це не вирішить проблему в цілому, але принаймні зведе до мінімуму і скоротить тривалість напружених періодів.

Зобов'язання

Як ми дізналися з Розділу 10, вкрай важливо уникати прийняття зобов'язань на терміни, у дотриманні яких ви не впевнені. Бізнес завжди намагатиметься отримати від вас такі зобов'язання, бо прагне повністю виключити ризики. А ми мусимо подбати про те, щоби ризики об'єктивно оцінювалися і представлялися бізнесу таким чином, аби той міг ефективно керувати ними. Прийняття нереалістичних зобов'язань суперечить цій меті та робить погану послугу як бізнесу, так і нам особисто.

Іноді зобов'язання приймають замість нас. Іноді ми раптово дізнаємося, що начальство щось пообіцяло замовнику, не проконсультувавшись із нами. У цьому разі професійна честь вимагає допомогти бізнесу відшукати можливість виконання цих зобов'язань. Водночас та сама професійна честь *не вимагає* від нас *прийняття* цих зобов'язань.

Відмінність досить глибока. Професіонал завжди допомагає бізнесу знайти спосіб досягнення його цілей, але він не завжди має виконувати зобов'язання, прийняті замість нього. Зрештою, якщо нам не вдасться виконати зобов'язання, прийняті замість нас, то відповідальність за це повинен нести той, хто ці зобов'язання прийняв.

Легко сказати… Коли ваш бізнес розвалюється, а зарплата відтерміновується через порушення зобов'язань, важко встояти під таким тиском. Утім, якщо ви поводилися професійно, то принаймні зможете шукати нову роботу з гідністю і чистим сумлінням.

Дотримання чистоти

Для того щоби рухатися швидко і не порушувати термінів, у коді необхідно підтримувати чистоту. Професіонал не піддається спокусі влаштувати безлад у коді, щоби швидко рухатися вперед. Професіонал розуміє, що «швидко і брудно» — це оксюморон. Брудно завжди означає *повільно*!

Підтримання чистоти в системі, коді та архітектурі допомагає уникнути тиску. Це зовсім не означає, що ви повинні до нескінченності відшліфовувати код. Ідеться про інше: про непримиренне ставлення до безладу. Ми знаємо, що безлад уповільнить нашу роботу, призведе до недодержання термінів і порушення зобов'язань. І, відповідно, намагаємося зберегти результати нашої роботи в настільки чистому стані, наскільки це можливо.

Дисципліна у кризових ситуаціях

Щоби зрозуміти, у що ви насправді вірите, спостерігайте за собою у кризовій ситуації. Якщо під час кризи ви не відхиляєтеся від своїх робочих методів, ви дійсно переконані в їх ефективності.

Якщо ви зазвичай дотримуєтеся методології розробки через тестування (TDD), але відмовляєтеся від неї під час кризи, значить ви не вірите в корисність TDD. Якщо ваш код залишається чистим за сприятливих обставин, а в кризу ви розводите в ньому бруд, значить ви не вірите, що бруд уповільнює роботу. Якщо ви використовуєте парне програмування під час кризи, але не в звичайній ситуації, ви вважаєте, що парне програмування є ефективнішим за індивідуальне.

Оберіть ті методи, із якими ви комфортно почуваєтеся у кризовій ситуації. *А потім застосовуйте їх постійно!* Використання цих методів — найкращий спосіб уникнути кризи.

Не змінюйте свою поведінку в напруженій ситуації. Якщо ваші методи насправді оптимальні, то їх треба дотримуватися навіть у найважчі часи.

Як скористатися тиском

Запобігання та усунення криз — справа, звісно, складна, але іноді ви опиняєтеся під тиском незалежно від свого бажання. Іноді проєкт просто потребує більше часу, ніж планувалося спочатку. Іноді вихідна архітектура виявляється хибною і її доводиться переробляти. Іноді ви несподівано втрачаєте цінного працівника або замовника. А трапляється і так, що ви приймаєте зобов'язання, що не вдається виконати.

Що робити в таких випадках?

Без паніки

Візьміть свій стрес під контроль. Безсонні ночі не допоможуть виконати роботу швидше. Лайка і сперечання теж не допоможуть. Але найгірше, що ви можете зробити,— це поспіх! Боріться із цією спокусою за всяку ціну. Поспіх здатен затягнути вас ще глибше на дно.

Навпаки, пригальмуйте та продумайте завдання. Прокладіть курс на краще з можливих рішень, а потім тримайтеся цього курсу, зберігаючи стабільний і розумний темп.

Взаємодія

Повідомте свою команду й керівництво про неприємності. Запропонуйте свій план щодо виходу із кризи. Зверніться до них по інформацію та поради. Уникайте сюрпризів. Ніщо так не сердить людей і не робить їх менш раціональними, як сюрпризи. Сюрпризи підвищують рівень стресу вдесятеро.

Покладайтеся на власні методи

Якщо ви потрапили у скрутну ситуацію, *довіряйте своїм методам*. Вони для того й потрібні, щоби допомагати вам витримати тиск. У такий час слід особливо ретельно стежити за дотриманням правил, а не ставити їх під сумнів або навіть відмовлятися від них.

Замість того щоби в паніці шукати хоч щось здатне допомогти вам прискорити роботу, ви маєте свідомо і цілеспрямовано дотримуватись вимог обраних методів. Якщо ви дотримуєтеся методології розробки через тестування, пишіть ще більше тестів, ніж зазвичай. Якщо ви обрали безкомпромісний рефакторинг, дійте ще більш безкомпромісно. Якщо ви обмежуєте розміри функцій, робіть їх ще меншими. Для того щоби вибратися з важкої ситуації, необхідно покластися на те, в ефективності чого ви вже впевнені,— вашим методам.

Не відмовляйтеся від допомоги

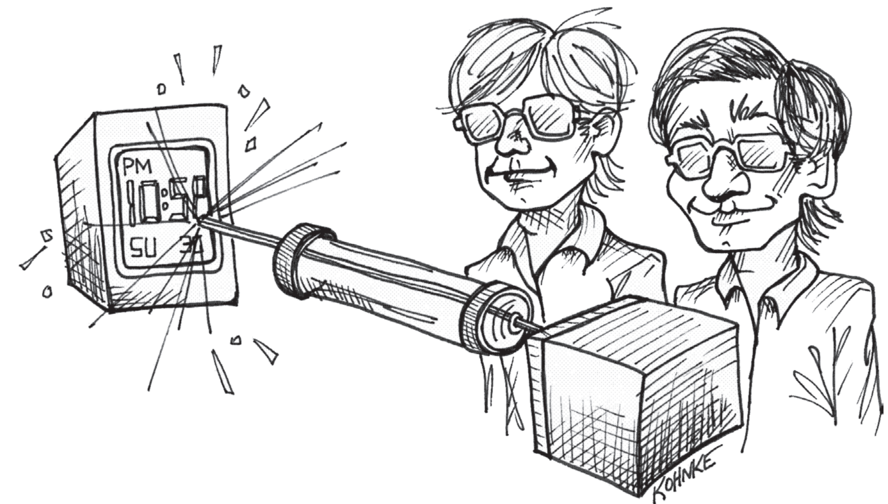
Парне програмування! Коли стає по-справжньому гаряче, знайдіть партнера, який захоче програмувати в парі з вами. Так ви виконаєте свою роботу швидше і з меншою кількістю дефектів. Партнер допоможе вам триматися в рамках методологій і не впадати в паніку. Партнер побачить те, що ви прогавили, висуне корисну ідею і підхопить естафету, якщо ви втомилися.

Відповідно, коли ви бачите, що хтось із ваших колег потрапив у напружену ситуацію, запропонуйте йому попрацювати в парі. Допоможіть колезі вибратися з ями, у якій він опинився.

Висновки

Тримайте напружені ситуації під контролем. Насамперед намагайтеся не потрапляти в них, а якщо це неможливо — тримайтесь. Аби уникнути проблемної ситуації, контролюйте свої зобов'язання, дотримуйтесь методологій і підтримуйте чистоту коду. А щоби витримати тиск, зберігайте спокій, спілкуйтеся з колегами, дотримуйтесь правил і частіше звертайтеся по допомогу.

12 СПІВПРАЦЯ



Більшість програм зазвичай створюється командами розробників. Однією з умов ефективної роботи команди є професійна взаємодія учасників. Бути відлюдником або пустельником у команді — непрофесійно.

У 1974 році мені було 22. Минуло пів року з того моменту, як ми одружилися з моєю чудовою Енн-Мері. До народження нашої першої дитини залишався ще рік. Тоді я працював в одному з підрозділів *Teradyne*, що мав назву *Chicago Laser Systems*.

Поруч зі мною працював мій приятель по середній школі Тім Конрад. Свого часу ми з ним займалися цікавими речами: збирали комп'ютери в його підвалі, майстрували «сходи Якова»[1] в моєму, вчили один одного програмувати для PDP-8 і збирали працездатні калькулятори з мікросхем і транзисторів.

На роботі ми програмували систему, що керувала лазерами для високоточного вирізання електронних компонентів мікросхем на кшталт резисторів, конденсаторів і т. п. Зокрема, ми нарізали кристали для перших цифрових годинників «Motorola Pulsar».

Програмування здійснювалося на комп'ютері M365 — клоні моделі PDP-8 від *Teradyne*. Система писалася на асемблері, а вихідні файли зберігалися на магнітних стрічках. Хоча ми могли змінювати код в екранному редакторі, цей процес був доволі складним, тому для вичитки коду і попередньої правки переважно застосовувалися роздруковані лістинги.

У нас не було жодних засобів пошуку за кодовою базою. Неможливо було знайти всі місця викликів заданої функції або використання заданої константи. І не важко уявити, як це гальмувало нашу роботу.

Тому ми з Тімом вирішили написати генератор перехресних посилань. Програма мала читати стрічки з вихідним кодом і виводити на принтер списки всіх символічних імен із зазначенням файлу та номера рядка, у якому це ім'я використовувалося.

Вихідна програма була доволі простою. Вона читала стрічку, розбирала асемблерний код, будувала таблицю символічних імен і додавала до неї посилання. Програма чудово працювала, але була дуже повільною. Обробка головної операційної програми (MOP) забирала близько години.

[1] «Сходи Якова» — назва фізичного експерименту, що демонструє певні властивості електричного дугового розряду.

Така низька швидкість пояснювалася тим, що зростаюча таблиця символічних імен зберігалася в одному буфері. Щоразу, коли програма знаходила нове посилання, воно додавалося в буфер, а решта змісту буфера зміщалася на кілька байт, аби звільнити потрібне місце.

Ми з Тімом, звісно, не були експертами зі структур даних та алгоритмів. Ми ніколи не чули про хеш-таблиці та бінарний пошук. Ми навіть гадки не мали, як прискорити алгоритм. Ми просто знали: наша програма працює надто повільно.

Тоді ми почали випробовувати одне рішення за іншим. Ми об'єднували посилання у зв'язаний список. Ми залишали пропуски в масиві і збільшували буфер лише після його заповнення. Ми намагалися створювати зв'язані списки пропусків в масивах. Ми випробували купу шалених варіантів.

Ми стирчали біля дошки в офісі, малювали діаграми структур даних і займалися обчисленнями для прогнозування швидкодії. І щодня приходили на роботу з новими ідеями. Постійно відбувався інтенсивний творчий обмін.

Зрештою ми скоротили час роботи програми до 15 хвилин, що було досить близько до часу звичайного читання стрічки. І лише тоді заспокоїлися.

Програмісти проти людей

Ми стали розробниками не тому, що не любимо працювати з людьми. Просто ми вважаємо, що міжособистісні стосунки надто безладні та непередбачувані, і тому віддаємо перевагу чіткій, передбачуваній поведінці комп'ютерів. Ми щасливі, коли годинами просиджуємо в кімнаті, глибоко занурившись у розв'язання якоїсь цікавої задачі.

Гаразд, це занадто огульне узагальнення, але із правил існує безліч винятків. Багато розробників успішно співпрацюють із людьми, і їм це подобається. Однак у середньому тенденція виглядає саме так, як я її описав. Ми, розробники, насолоджуємося помірною сенсорною деприваціею і глибоким зануренням у роботу — своєрідний ізольований кокон.

Програмісти і роботодавці

У 1970–1980-х роках під час роботи в *Teradyne* я добре освоїв процес відлагодження. Мені подобалася ця справа, я кидався на складні завдання із запалом та ентузіазмом. Жодна помилка не могла приховатися від мене!

Усунувши чергову помилку, я почувався переможцем. Згодом вирушав до свого боса Кена Файндера і з ентузіазмом розповідав, яка *цікава* помилка мені трапилася. Але одного разу Кен у розпачі вигукнув: «Помилки не цікаві! Їх просто треба виправляти!»

Цього дня я отримав важливий урок. З ентузіазмом ставитися до того, чим ви займаєтеся,— це добре. Але при цьому варто пам'ятати про цілі людей, які вам платять.

Найперший обов'язок професійного розробника — дбати про інтереси його роботодавців. Це означає, що ви маєте інтенсивно спілкуватися зі своїми безпосередніми керівниками, бізнес-аналітиками, тестерами та іншими членами команди, щоби *глибоко усвідомити* бізнес-цілі проєкту. Ніхто не змушує вас ставати експертом у галузі бізнесу. Просто *треба* розуміти, навіщо ви пишете свій код і яку користь він надасть вашій фірмі.

Найгірше, що може зробити професійний розробник,— сховатися у блаженному невіданні у своєму технологічному притулку, коли бізнес навколо палає і руйнується. Ваша *робота* — утримувати бізнес на плаву!

Отже, професійний розробник завжди приділяє час дослідженню комерційного боку справи. Він обговорює з користувачами програмний продукт, із яким вони працюють. Він спілкується із представниками відділу продажів та маркетингу щодо проблем, які виникають. Він контактує з керівництвом, аби глибше зрозуміти короткострокові і довгострокові цілі команди.

Професійний розробник переймається справами корабля, на якому пливе!

Протягом усієї кар'єри мене звільнили з посади розробника лише один раз. Це сталося 1976 року, коли я працював на *Outboard Marine Corp*. Я допомагав писати систему автоматизації виробництва, що

використовувала комп'ютери IBM System/7 для контролю за десятками автоматів для алюмінієвого лиття в головному цеху підприємства.

Із технічного погляду це була цікава і складна робота. Архітектура System/7 подобалася мені, а система автоматизації виробництва теж виглядала вельми захопливою.

У нас була гарна команда. Керівник групи Джон був компетентним і цілеспрямованим спеціалістом. Двоє моїх колег-розробників були приємними людьми, завжди готовими прийти на допомогу. Під наш проєкт було виділено спеціальну лабораторію, у якій ми працювали. Наш бізнес-партнер брав участь у роботі і майже постійно перебував поруч із нами. Наш проєкт-менеджер Ральф був компетентним і відповідальним.

Словом, усе виглядало чудово. Проблема полягала в мені. Я з ентузіазмом ставився до проєкту та технології, але у віці 24 років я не міг змусити себе думати про бізнес чи внутрішню політику компанії.

Початкової помилки я припустився першого ж дня. Я прийшов на роботу без краватки. Я одягав краватку, вирушаючи на співбесіду з прийому на роботу, і бачив, що всі співробітники носять краватки, але не зробив висновків. Отже, першого ж дня Ральф підійшов до мене й повідомив: «Ми всі тут носимо краватки».

Не можу описати, наскільки це мене дратувало. Відтоді я начіпляв краватку щодня і з кожним днем усе дужче ненавидів її. Але чому? Адже я знав, куди влаштовуюсь на роботу. Я знав правила. Чому мене це дратувало? Відповідь проста: тому що я був егоїстичним і самозакоханим типом.

Я не міг змусити себе приходити на роботу вчасно. Мені здавалося, що це взагалі не має значення. Зрештою, я «добре справлявся». І це було правдою: я справді добре справлявся із написанням програм. Безперечно, я був найкращим розробником у команді. Я вмів писати код швидше і краще за будь-кого. Я швидко знаходив й усував проблеми. Я *знав*, що я «цінний фахівець», тому час і дати для мене майже нічого не значили.

Рішення про моє звільнення було ухвалено тоді, коли я спізнився до однієї з контрольних точок проєкту. Очевидно, Джон казав нам,

що наступного понеділка йому потрібна демоверсія робочих функцій системи. Напевно, я знав про це, але просто не звернув уваги на його слова.

Система перебувала у стадії активної розробки. До справжньої експлуатації було ще далеко. Не було жодних причин залишати систему в робочому стані, коли в лабораторії нікого не було. Я останнім ішов з роботи у п'ятницю, і, мабуть, залишив систему в непрацездатному стані. Той факт, що понеділок є важливою для проєкта датою, просто не зафіксувався в моїй пам'яті.

У понеділок я з'явився в лабораторії на годину пізніше й побачив, що всі похмуро зібралися довкола неробочої системи. Джон запитав: «Чому система сьогодні не працює, Бобе?» Я відповів: «Не знаю»,— і зайнявся відлагодженням. Я навіть не згадав про демоверсію, заплановану на понеділок, але з поведінки моїх колег явно випливало, що сталося щось погане. Нарешті Джон підійшов до мене й прошепотів на вухо: «А якби нагрянув Стенберг?» — і відійшов з обуреним виглядом.

Стенберг був віцепрезидентом компанії з питань автоматизації. Сьогодні цю посаду назвали би «керівник технічної служби» (CIO). Питання Джона здалося мені безглуздим. «Ну і що? — подумав я.— Система ж не в режимі експлуатації, то в чому проблема?»

Цього дня я отримав перший попереджувальний лист. У ньому мені пропонувалося негайно змінити ставлення до роботи, інакше «станеться негайне звільнення».

Так, я був наляканий!

Мені довелося проаналізувати свою поведінку і зрозуміти, що я робив не так. Я поговорив із Джоном і Ральфом, повідомивши обом, що маю твердий намір змінити себе і свій стиль роботи.

І я це зробив! Я припинив спізнюватися, почав звертати увагу на внутрішню політику компанії. Я нарешті зрозумів, чому Джон непокоївся стосовно появи Стенберга. Я усвідомив, як підвів його, залишивши до понеділка непрацездатну систему.

Надто пізно. Жереб був кинутий. Через місяць я отримав друге попередження із приводу якоїсь тривіальної помилки. Цієї миті мені

варто було зрозуміти, що листи були звичайною формальністю, а рішення про звільнення вже було ухвалене. Але я твердо вирішив усе виправити і почав працювати ще старанніше.

Ще за кілька тижнів мені повідомили про звільнення.

Мені довелося повернутися додому до своєї вагітної 22-річної дружини і розповісти їй, що я втратив роботу. Не хотів би я, щоби щось подібне трапилося зі мною знову.

Програмісти проти програмістів

У розробників часто виникають труднощі під час роботи в тісному контакті з іншими розробниками. Іноді це створює жахливі проблеми.

Приналежність коду

Одна з найгірших ознак неправильно функціонуючої команди — коли кожен розробник зводить стіну навколо *свого* коду і забороняє іншим розробникам торкатися його. Я бачив компанії, де розробники забороняли іншим навіть *дивитися* на їх код. Це вірний шлях до катастрофи.

Якось я консультував компанію, що виготовляла висококласні принтери. Ці пристрої складалися з безлічі різних компонентів: систем подачі паперу, друку, укладання паперу, зшивання листів, різаків тощо. Із погляду бізнесу ці системи мали різну цінність. Система подачі паперу була важливішою за систему укладання, але жоден елемент не міг зрівнятися за важливістю із системою друку.

Кожен розробник працював зі *своєю* окремою системою. Один писав код системи подачі аркушів, інший — код системи зшивання і т. д. Усі вони пильно охороняли свою технологію і не дозволяли нікому торкатися свого коду. Професійна вага розробників була пов'язана з комерційною цінністю пристрою. Розробник, який працював над пристроєм друку, мав незаперечний авторитет.

Для технології такий стан справ мав катастрофічні наслідки. Мені, консультантові, було добре видно масове дублювання коду та різнобій в інтерфейсах між модулями. Але жодні аргументи з мого боку

не могли переконати розробників (або представників бізнесу) змінити підхід до роботи. Адже їхня зарплатня була щільно пов'язана із важливістю пристроїв, які вони розробляли.

Колективна приналежність коду

Зруйнуйте стіни приналежності коду — код має належати всій команді! Я завжди намагаюся працювати з групами, у яких кожен учасник може перевірити будь-який модуль і внести зміни, що вважатиме за необхідні. Я прибічник того, щоби код належав *команді*, а не окремим її представникам.

Професійні розробники не забороняють колегам працювати зі своїм кодом. Вони не зводять навколо свого коду «стіни приналежності». Навпаки — намагаються працювати разом над якомога більшою частиною коду. А працюючи над іншими частинами системи, вони навчаються один в одного.

Парне програмування

Багато хто з розробників не є прихильниками ідеї парного програмування. Мені це здається дивним, бо в напружених ситуаціях більшість розробників об'єднуються в пари. Чому? Тому що парне програмування є найефективнішим способом розв'язання задачі. Згадайте стару приказку: «Дві голови краще за одну». Але якщо парне програмування є найефективнішим способом розв'язання задач у напружених ситуаціях, чого б йому не бути найефективнішим способом розв'язання задач у більш спокійні часи?

Я не збираюся цитувати наукові дослідження, хоча у мене в запасі знайдеться чимало цитат. Я не розповідатиму «випадків із життя», хоча і їх у мене теж вистачає. Я також не маю наміру радити, яку частину часу слід проводити за парним програмуванням. Скажу тільки одне — *професіонали працюють у парах*. Але чому? Тому що (принаймні для деяких завдань) ця методологія є найефективнішою. Втім, це не єдина причина.

Професіонали поєднуються в пари, бо це найкращий спосіб обміну знаннями. Професіонали не створюють «банки знань» — натомість вони вивчають різні аспекти системи і бізнесу, об'єднуючись у пари. Вони розуміють просту річ: хоча кожен член команди обіймає певну посаду, всі вони мають бути готові моментально перемкнутися в разі потреби.

Професіонали поєднуються в пари також із тієї причини, що це найкращий спосіб рецензування коду. Жодна система не повинна містити код, що не був прорецензований іншими розробниками. Існує багато механізмів рецензування; здебільшого вони жахливо неефективні. Проте найефективніший і найдієвіший із них — персональна участь у написанні коду.

Мозочок

Якось 2000 року, на самому піку буму електронної комерції, я приїхав потягом до Чикаго. І не встиг ступити на платформу, як в очі мені кинувся величезний плакат над виходом. Широко відома фірма-розробник запрошувала до співпраці програмістів. На плакаті значилося: «*Попрацюйте мозочком поруч із найкращими!*».

Така видатна дурість мене вразила. Жалюгідні, геть безглузді рекламники намагалися справити враження на розумних, ерудованих, технічно грамотних розробників — людей, які погано переносять невігластво. Рекламники намагалися повідомити, що на новій роботі ви, начебто, обмінюватиметеся цінними знаннями з іншими, не менш розумними людьми. Насправді ж, згадана в рекламі частина мозку, тобто мозочок, відповідає за координацію рухів, а не за інтелект. І ті люди, яких намагалися залучити цією рекламою, лише посміхалися із цієї дурної помилки.

Але в цій рекламі мене зацікавило дещо інше. Я уявив групу людей, які намагаються «попрацювати мозочком». Оскільки мозочок міститься в задній частині головного мозку, я уявив собі групу розробників, що розсілися по офісних кабінках потилицями один до одного, втупилися в монітори і повністю відокремилися від зовнішніх подразників навушниками. Хіба це можна вважати командою?

Професіонали працюють *разом*. Неможливо працювати разом, розсівшись по кутках і затуливши вуха навушниками. Члени команди мають сидіти за спільним столом, обличчями один до одного. Вони повинні відчувати побоювання своїх колег, чути навіть їхнє роздра-

товане бурмотіння. Між ними має тривати постійне спілкування — як вербальне, так і на рівні «мови тіла». Вони мають взаємодіяти як єдине ціле.

Можливо, вам здається, що наодинці ви працюватимете ефективніше. Навіть якщо це й так, звідси зовсім не випливає, що ефективніше працюватиме вся команда. І насправді дуже малоймовірно, що наодинці ви працюєте краще.

У деяких ситуаціях робота поодинці — саме те, що потрібно. Іноді вам потрібно довго й напружено поміркувати над завданням. В інших випадках завдання може виявитися настільки тривіальним, що залучати до його виконання іншу людину — просто марнотратство. Але, якщо взяти до уваги більшість ситуацій, набагато краще віддавати значну частину робочого часу роботі в тісному контакті з колегами та парному програмуванню.

Висновки

Можливо, хтось із нас прийшов у програмування не для того, щоби працювати разом з іншими людьми. Йому не пощастило. Програмування нерозривно пов'язане *із співпрацею*. Нам доводиться спілкуватися з представниками бізнесу та менеджменту, а також із колегами-розробниками. Знаю, знаю: будь наша воля, ми віддали би перевагу праці в ізольованому приміщенні з шістьма великими моніторами, каналом T3, паралельним масивом надшвидких процесорів, необмеженим обсягом пам'яті та дискового простору, а також нескінченним запасом дієтичної коли і чипсів.

На жаль, так не буде. Якщо ви хочете займатися програмуванням сьогодні, доведеться навчитись спілкуватися[1].

[1] Посилання на останню репліку із фіналу стрічки «Soylent Green» — антиутопії за романом Гаррі Гаррісона «Посуньтесь! Посуньтесь!», знятої 1973 року режисером Річардом Флейшером.

13 КОМАНДИ І ПРОЄКТИ

Уявіть, що вам потрібно виконати багато дрібних проєктів. Як розподілити їх між розробниками? А якщо проєкт лише один, але дуже великий?

Чи можна змішувати?

За минулі роки я консультував чимало банків і страхових компаній. Єдине, що вони мали спільного,— якийсь дивний підхід до розподілу проєктів.

Банківський проєкт часто є відносно невеликим завданням, що може бути виконане одним або двома розробниками за кілька тижнів. До нього зазвичай включаються керівник, який водночас веде й інші проєкти, а також бізнес-аналітик, який формулює вимоги для інших проєктів. Далі додаються розробники, котрі також працюють над іншими проєктами, і один-два тестери, які теж мають інші завдання.

Помітили закономірність? Проєкт настільки малий, що немає сенсу займатися лише ним одним. Усі учасники приділяють йому від 25 до 50 відсотків робочого часу.

А ось вам і правило: половини людини не існує.

Безглуздо наказувати розробнику присвятити половину часу проєкту A, а іншу половину — проєкту B, особливо якщо у цих двох проєктах беруть участь різні керівники, бізнес-аналітики, розробники і тестувальники. Хіба такого монстра можна назвати командою? Це не команда, а якась мішанина, що зазвичай виходить із блендера!

Згуртована команда

Команди формуються не одразу. Між учасниками поступово налагоджуються стосунки. Вони вчаться працювати один з одним, знайомляться з особливостями, сильними та слабкими сторонами колег. Згодом учасники починають *згуртовуватися*.

У згуртованій команді є щось чарівне: вона здатна творити дива. Її члени розуміють одне одного з півслова, підтримують та вимагають від колег максимальної віддачі. Завдяки цій взаємодії досягаються видатні результати.

Згуртована команда зазвичай налічує близько дюжини учасників. Їх може бути і більше (до 20) або менше (до 3), але оптимальний розмір зазвичай близько дюжини. Команда має включати розробників, тестувальників та аналітиків. І в неї має бути проєкт-менеджер.

Співвідношення чисельності розробників і тестувальників/аналітиків може змінюватися у певних межах, але пропорція 2:1 є цілком доречною. Отже, добре згуртована команда з 12 осіб може складатися із семи розробників, двох тестувальників, двох аналітиків та менеджера проєкту.

Аналітики розробляють вимоги і пишуть для них автоматизовані приймальні тести. Тестувальники теж пишуть автоматизовані приймальні тести, але від аналітиків відрізняються спрямованістю. Ті й інші створюють тести, але аналітики орієнтуються на комерційну цінність, а тестувальники — на правильність роботи коду. Аналітики пишуть «оптимістичні» тести, а тестувальники турбуються про те, що може зашкодити виконанню проєкту, і пишуть тести для виявлення збоїв та граничних випадків.

Менеджер проєкту стежить за прогресом і вживає заходів, аби команда чітко розуміла терміни і пріоритети.

Той самий член команди може поєднувати виконання своїх безпосередніх обов'язків із місією наставника (коуча), відповідального за дотримання технологічних процесів та методів. Він має ставати своєрідною «колективною совістю», коли в команди під тиском виникає спокуса порушити правила.

Дозрівання

Для того щоби члени команди розібралися зі своїми відмінностями і домовилися між собою, а сама команда по-справжньому згуртувалася, потрібен час. Процес може тривати пів року або навіть рік. Однак коли це трапляється, відбувається диво. Згуртована команда разом планує, разом розв'язує завдання, разом розбирається із проблемами і *досягає результатів*.

Розформовувати такі команди тільки тому, що проєкт завершений, надто марнотратно. Краще зберегти команду в тому самому складі і доручати їй роботу над новими проєктами.

Що спочатку — команда чи проєкт?

Банки та страхові компанії намагаються формувати команди на базі проєктів. Такий підхід принципово помилковий: команди просто не встигають згуртуватися. Учасники працюють над проєктом протягом досить короткого терміну, до того ж, присвячуючи йому лише частину робочого часу, а отже не мають можливостей спрацюватися.

Професійні фірми-розробники доручають проєкти вже існуючим згуртованим командам, а не формують групи під конкретний проєкт. Згуртована команда може здійснювати відразу кілька проєктів і розподіляти роботу відповідно до вподобань, кваліфікації та здібностей її членів. Згуртованій команді під силу виконати навіть найскладніший проєкт.

Але як керувати такою командою?

Кожна команда має певну швидкість роботи[1]. Швидкість, за визначенням,— це обсяг роботи, що виконується за фіксований проміжок часу. Деякі команди вимірюють свою швидкість у *пунктах* на тиждень (пункт — одиниця складності частини завдання). Функціональність кожного проєкту, над яким працює команда, розбивається на блоки, складність яких оцінюється в пунктах. А далі продуктивність групи оцінюється за кількістю пунктів, реалізованих протягом тижня.

Швидкість — статистична метрика. Команда може реалізувати 38 пунктів за один тиждень, 42 пункти за другий і 25 — упродовж третього. У міру накопичення даних обчислюється усереднений показник.

[1] Martin R. C. *Agile Software Development: Principles, Patterns, and Practices*, Upper Saddle River, NJ: Prentice Hall, 2003; Cohn M. *Agile Estimating and Planning*. Upper Saddle River, NJ: Prentice Hall, 2006, p. 20–22.— У покажчику до цієї книжки ви знайдете багато корисних посилань до теми швидкості роботи.

Керівництво може задавати цілі для кожного окремого проєкту. Наприклад, якщо середня швидкість команди дорівнює 50 і вона працює над трьома проєктами, керівництво може попросити команду розподілити зусилля у пропорції 15, 15 і 20.

Окрім того, що над проєктами працює досвідчена і згуртована команда, ця схема має додаткову перевагу. У напруженій ситуації бізнес-сторона може сказати: проєкт B у критичному стані, тому в наступні три тижні спрямуйте на його реалізацію всі сто відсотків зусиль.

Такий швидкий перерозподіл пріоритетів практично неможливий із групами, сформованими з випадкових людей. Тоді як згуртована команда, що працює над двома або трьома проєктами водночас, легко виконує подібні «розвороти на місці».

Дилема власника проєкту

Один з аргументів проти методу, який я обстоюю, полягає в тому, що він частково позбавляє власників проєкту впевненості та влади. Власники проєкту, для реалізації якого було створено спеціальну команду, можуть розраховувати на повну віддачу членів команди. Вони знають, що формування і розпуск команди обходяться недешево, тому бізнес не лишатиме їх команди, виходячи лише з короткострокових тактичних міркувань.

З іншого боку, якщо проєкти передаються згуртованим командам і ці команди ведуть одразу кілька проєктів, бізнес може змінювати їхні пріоритети на власний розсуд. Завдяки цьому власник проєкту втрачає частину впевненості в майбутньому. У нього можуть зненацька відібрати ресурси, на які він покладається.

Чесно кажучи, я віддаю перевагу другій ситуації. «Руки» бізнесу не мають бути зв'язані штучними складнощами формування та розпуску команд. Якщо бізнес вирішить, що якийсь проєкт має більш високу пріоритетність, то цей проєкт повинен отримати можливість швидко перерозподілити ресурси. А сам власник проєкту має навести вагомі аргументи на його користь.

Висновки

Створити команду складніше, ніж проєкт. Ось чому краще формувати команди з постійним складом, які переходять від одного проєкту до іншого і здатні займатися кількома проєктами одночасно. Під час формування команда мусить отримати достатньо часу, щоби її члени спрацювалися, і лише після цього її можна використовувати як інструмент для успішного виконання багатьох складних проєктів.

Бібліографія

Martin R. C. *Agile Software Development: Principles, Patterns, and Practices*, Upper Saddle River, NJ: Prentice Hall, 2003.

Cohn M. *Agile Estimating and Planning*. Upper Saddle River, NJ: Prentice Hall, 2006.

14 НАСТАВНИЦТВО, НАВЧАННЯ І МАЙСТЕРНІСТЬ

Рівень випускників у галузі комп'ютерних технологій мене постійно розчаровує. Річ не в тім, що вони недостатньо розумні чи талановиті — просто їх не навчають тому, що повинен знати справжній програміст.

Диплом — і нічого більше

Якось я проводив співбесіду з дівчиною, яка працювала над дипломною роботою в галузі комп'ютерних технологій. Ця студентка відомого університету подала документи на літнє стажування. Я попросив її написати частину коду для мене, а вона відповіла: «Але я взагалі не пишу код!»

Прошу ще раз перечитати попередній абзац, а вже потім перейти до наступного.

Я спитав, які курси із програмування вона відвідувала для отримання ступеня магістра. Вона відповіла, що не відвідувала жодних.

Можливо, вам варто прочитати розділ від самого початку. Ви хоча би переконаєтеся в тому, що не потрапили в альтернативний Всесвіт і вам не наснився страшний сон.

Ви запитаєте, як студент, що здолав навчальну програму магістра в галузі комп'ютерних технологій, міг обійтися без єдиного курсу програмування? Я поставив собі те саме питання ще тоді — і досі не знаходжу відповіді.

Звісно, цей випадок — чи не найвидатніший серед розчарувань, які мені довелося зазнати під час співбесід із випускниками. Не всі випускники в галузі комп'ютерних технологій настільки неосвічені, однак я помітив у них щось спільне: майже всі вони *вивчали програмування самостійно* до вступу до університету і продовжували вивчати його, *незважаючи на навчання в університеті*.

Зрозумійте мене правильно. Насправді я впевнений, що в університеті можна здобути відмінну освіту. Водночас можна якось прослизнути через цю систему й отримати диплом — і не отримати нічого більше.

Інша проблема: навіть найкращі курси з комп'ютерних наук зазвичай не готують молодого випускника до того, із чим він зіткнеться на практиці. І річ тут не стільки в недоліках навчальних програм, як у реаліях майже всіх дисциплін. Те, що ви дізнаєтесь під час навчання, часто дуже, дуже відрізняється від того, із чим вам доведеться зіткнутися на роботі.

Наставництво

Як ми вчимося програмувати? Дозвольте розповісти вам історію про наставників та учнів.

«Digi-comp I», мій перший комп'ютер

У 1964 році мама подарувала мені до дванадцятиріччя маленький механічний комп'ютер. Він називався «Digi-Comp I»[1] і складався з трьох пластикових тригерів та шести пластикових кон'юнкторів. Виходи тригерів можна було поєднувати із входами кон'юнкторів чи навпаки — виходи кон'юнкторів із входами тригерів. Коротше кажучи, пристрій дозволяв створити трирозрядний кінцевий автомат.

До комплекту входила інструкція із кількома програмами. Для того аби запрограмувати пристрій, треба було проштовхнути пластикову трубочку (короткий відтинок соломинки для коктейлю) у штифт на тригері. У посібнику було точно зазначено, куди вставляти трубочки, але жодного слова не говорилося про те, що ці трубочки насправді *роблять*. Знали би ви, як це мене дратувало!

Я годинами роздивлявся той пристрій, намагаючись зрозуміти, як він працює на найнижчому рівні, але не міг осягнути, як змусити його робити те, що мені потрібно. На останній сторінці інструкції було надруковано: якщо я заплачу долар, то мені надішлють посібник із програмування «Digi-Comp I».

Я послав долар і почав з нетерпінням чекати. Коли ж посібник надійшов, я проковтнув його за день. Це був спрощений виклад булевої алгебри з азами булевої логіки, законів асоціативності і дистрибутивності, а також законів де Моргана. У посібнику був наведений приклад, як записувати завдання у вигляді послідовності логічних формул і як скорочувати ці формули для шести кон'юнкторів.

Ось коли я замислив свою першу програму. Я досі пам'ятаю її назву: «Комп'ютеризовані ворота містера Паттернсона». Я написав рівнян-

[1] «Digi-Comp I» являв собою функціонуючий механічний цифровий комп'ютер, що, починаючи з 1963 року, продавався у США у вигляді комплекту для збирання. Деталі цієї освітньої іграшки вироблялися з полістиролу, а ціна складала 4,99 долара США.

ня, скоротив їх і відтворив на «Digi-Comp I», використовуючи трубки та штифти машини. *І програма запрацювала!*

Навіть від спогадів про це в мене й досі мурахи біжать по спині. Півстоліття тому ті враження визначили долю дванадцятирічного підлітка. Програмування захопило мене, і моє життя вже ніколи не було таким, як раніше.

Чи пам'ятаєте ви момент, коли запрацювала ваша перша програма? Чи змінила вона ваше життя? Чи відкрила перед вами шлях, із якого ви вже не змогли звернути?

Звісно, я не міг у всьому розібратися сам. Я *мав наставників*. Дуже добрі та обдаровані люди (яким я багато чим зобов'язаний) не пошкодували часу, щоби написати виклад булевої алгебри, доступний підліткові. Вони пов'язали математичну теорію із практичними прийомами програмування маленького механічного комп'ютера; завдяки їм я змусив свій комп'ютер зробити те, що було мені потрібно.

Я щойно зняв із полиці свій екземпляр цього доленосного посібника. Я зберігаю його у пластиковому пакеті із застібкою. Час бере своє: сторінки пожовтіли і стали крихкими. Але сила слів сяє, як і колись. Елегантний виклад булевої алгебри займає лише три сторінки нещільного тексту. Покроковий аналіз виразів кожної з вихідних програм досі справляє враження. То справжній маленький шедевр, праця, що змінила життя принаймні одного парубка. А я, найімовірніше, навіть ніколи не дізнаюся імен його творців…

ECP-18 у середній школі

У п'ятнадцятирічному віці, коли я навчався у старших класах, мені подобалося проводити час у математичній лабораторії (саме так!). Якось туди привезли пристрій розміром із токарний верстат. Це був навчальний комп'ютер для середніх шкіл, він мав назву ECP-18. Нашій школі він був наданий на двотижневий випробний період.

Я стояв осторонь і дослухався до розмов учителів і техніків. Пристрій мав 15-розрядні слова (що таке «слово»?) і барабанний магнітний накопичувач на 1024 слова (тоді я вже знав, що це таке, але суто теоретично).

Коли машину увімкнули, вона видала звук на кшталт того, що видає реактивний літак під час зльоту. Я припустив, що саме так набирає обертів барабан. Коли пристрій виходив на робочий режим, то працював відносно тихо.

Машина була *чарівною*. Вона нагадувала офісний стіл, над яким височіла гарна панель управління — як на капітанському містку бойового корабля. Панель була прикрашена рядами лампочок, що також можна було натискати як кнопки. Сидячи за таким столом, людина почувалася немов у кріслі капітана Кірка[1].

Я спостерігав, як техніки натискають кнопки. Коли кнопку натискали, лампочка спалахувала, а при повторному натисканні — згасала. Також вони натискали інші кнопки, під якими значилося «*Завантажити*» або «*Виконати*».

Кнопки в кожному ряду були об'єднані у п'ять груп по три в кожній. Мій «Digi-Comp» теж був трирозрядним, тому я міг читати вісімкові цифри у двійковій формі. Завдяки цьому мені легко вдалося усвідомити, що кожен рядок становить п'ять вісімкових цифр.

Я чув, як техніки, що натискали кнопки, щось бурмотять. Вони натискали 1, 5, 2, 0, 4 в рядку «*Буфер пам'яті*», кажучи при цьому: «Зберегти у 204». Вони натискали 1, 0, 2, 1, 3 і бурмотіли: «Завантажити 213 в *акумулятор*». Там, звісно ж, була низка кнопок із підписом «*Акумулятор*»!

Через десять хвилин моєму п'ятнадцятирічному розуму було абсолютно ясно, що 15 означає «*Зберегти*», а 10 — «*Завантажити*», що в акумуляторі перебувають дані, які зберігаються або завантажуються, а решта чисел є номерами одного з 1024 слів на барабані. *(Так ось що таке «слово»!)*

Слово за словом (це справді ненавмисний каламбур), мій нетерплячий розум усе глибше проникав у коди та концепції. На той момент, коли техніки пішли, я вже розумів основні принципи роботи комп'ютера. Цього ж дня під час годин, відведених для самостійної роботи, я пробрався в математичну лабораторію і почав експери-

[1] Джеймс Тіберій Кірк (англ. James Tiberius Kirk) — персонаж науково-фантастичного телевізійного серіалу «Зоряний шлях», мультиплікаційного серіалу «Зоряний шлях» і супутніх повнометражних фільмів.

ментувати з комп’ютером. На той час я вже добре знав, що набагато простіше вибачитися, ніж отримати дозвіл! Я ввів програму, що збільшувала вміст акумулятора на 2 і додавала 1. Я ввів в акумулятор 5, запустив програму — і побачив в акумуляторі 138! Програма працювала!

Я ввів ще кілька таких простих програм, і вони теж працювали так, як належить. Я почувався володарем Всесвіту!

За добу я зрозумів, який я був дурний — і як мені пощастило. Я знайшов у лабораторії пам’ятку, у якій містилися всі інструкції та коди операцій. Серед них були і ті, про які я не міг дізнатися, спостерігаючи за техніками. Я з радістю виявив, що знайомі коди були інтерпретовані правильно, і це викликало в мене приплив ентузіазму. Однак серед численних пунктів я помітив інструкцію для зупинки — *HLT*. Так збіглося, що інструкція для *зупинки* була словом з одних нулів. Збіг був і в тому, що я включав у кінець кожної зі своїх програм слово із самих нулів, аби стерти вміст акумулятора. Концепція зупинки просто не спадала мені на думку, бо я вважав, що програма сама зупиниться, коли все зробить!

Пам’ятаю, якось я стирчав у математичній лабораторії, спостерігаючи за тим, як один із учителів боровся зі своєю програмою. Він намагався ввести два числа в десятковій формі з підключеного телетайпа, а потім роздрукувати їхню суму. Кожен, хто намагався програмувати подібні завдання машинною мовою мінікомп’ютерів, знає, що вони аж ніяк не тривіальні. Потрібно прочитати символи, розбити їх на цифри, потім перетворити у двійкову форму, підсумувати, перетворити назад у десяткову систему і перекодувати на символи. Можете повірити, що коли програма вводиться у двійковому вигляді з передньої панелі, все має *набагато* гірший вигляд!

Я спостерігав за тим, як учитель вставив у програму команду зупинки і запустив її. (О! Гарна думка!) Примітивна точка переривання дозволила йому проаналізувати вміст регістрів і зрозуміти, що зробила програма. Пам’ятаю, як він промовив: «Ого! Як швидко!»

Гадки не маю, який алгоритм він використав. Таке програмування для мене ще залишалося чимось подібним до магії. І він ані слова не сказав мені, доки я дивився йому через плече. Так, *ніхто* не говорив зі мною про цей комп’ютер. Думаю, мене вважали завадою, на

яку не варто звертати уваги,— чимось на зразок мошки, що пурхає лабораторією.

Зрештою його програма запрацювала. Виглядало це приголомшливо: вчитель обережно вводив два числа, бо, незважаючи на його вигуки, комп’ютер працював досить повільно (досить згадати, скільки часу займало читання послідовних слів із барабана 1967 року). Коли ж він натискав *«Введення»* після другого числа, комп’ютер, люто поблимавши всіма лампочками, починав виводити результат. На кожну цифру витрачалося близько секунди. Вивівши всі цифри, крім останньої, пристрій секунд п’ять блимав ще лютіше, тоді виводив останню цифру і зупинявся.

Що то була за пауза перед останньою цифрою? Цього я так і не дізнався. Однак зрозумів, що вибір шляху виконання завдання може мати істотні наслідки для користувача. Незважаючи на те що програма виводила правильну відповідь, із нею *все одно* щось було не так.

Це також було навчанням. Звісно, не таким, на яке я сподівався. Було би набагато краще, якби один із тих учителів узяв мене під опіку і почав працювати зі мною. Але навіть *спостереження* за ними дозволяло мені швидко отримувати нові знання.

Нетрадиційне навчання

Я розповів ці дві історії, бо вони описують два зовсім різні типи навчання, жоден із яких не підпадає під визначення цього поняття. У першому випадку я навчався в авторів дуже добре написаного посібника. У другому — навчався, спостерігаючи за людьми, які намагалися не звертати на мене уваги. Однак в обох випадках набуті знання виявилися глибокими та ґрунтовними.

Звісно, були в мене й інші вчителі. Добрий сусіда, який працював у *Teletype*, подарував мені коробку із тридцятьма телефонними реле. Ось що я вам скажу: дайте хлопцю реле і трансформатор від іграшкової залізниці — і він підкорить світ!

Інший добрий сусіда, який захоплювався аматорським радіо, навчив мене користуватися авометром (котрий я негайно зламав). Власник магазину офісного приладдя дозволяв мені «погратися» з його стра-

шенно дорогим програмованим калькулятором. А ще поряд був розташований відділ продажів *Digital Equipment Corporation*, куди мені інколи дозволялося зайти і повозитися із PDP-8 і PDP-10.

Також був великий Джим Карлін — BAL-програміст, який урятував мене від звільнення з першого місця роботи. Він допоміг відлагодити програму на Cobol, що виходила за межі мого розуміння. Він навчив мене читати дампи ядра, форматувати код за допомогою порожніх рядків, зірочок і коментарів. Він дав мені перший поштовх на шляху до майстерності. Прикро, що я не мав змоги відплатити послугою за послугу, коли гнів начальства обрушився на нього через рік після цього.

Але, правду кажучи, список майже вичерпано. На початку 1970-х років було не так багато досвідчених програмістів. У всіх інших місцях, де мені доводилося працювати, я виявлявся найдосвідченішим. Не було нікого, хто допоміг би мені розібратися, що таке справжнє професійне програмування. Не було зразка для наслідування, людини, яка навчила би мене як треба поводитися і на що передусім звертати увагу. Усе це мені довелося дізнаватися на власному досвіді, і цей процес виявився досить складним.

Гіркий досвід

Як я вже згадував, мене таки звільнили з тієї компанії, де 1976 року я займався автоматизацією виробництва. Хоча з технічного погляду я виглядав цілком компетентним, однак іще не навчився звертати увагу на бізнес та його цілі. Дати і терміни для мене не мали значення. Саме тому я геть забув про важливу демонстрацію програми в понеділок вранці, залишив систему в непрацездатному стані у п’ятницю, а на додаток спізнився на роботу в понеділок.

Мій керівник надіслав мені лист із вимогою негайно змінити ставлення до роботи інакше мене звільнять. Це був «тривожний дзвінок», завдяки якому я переглянув свої погляди на життя і кар’єру та вніс суттєві зміни у власну поведінку. Але було пізно, надто пізно. Маховик був запущений, і дрібниці, на які раніше ніхто не звернув би уваги, раптом перетворилися на обтяжуючі обставини. Отже, хай як я намагався, зрештою мені довелося піти із фірми.

Не варто й казати, як прикро було повідомляти таку новину вагітній дружині. Але я зібрався з духом і скористався отриманими уроками на наступному місці роботи, де я пропрацював цілих 15 років, заклавши міцний фундамент для подальшої кар’єри.

Зрештою я вижив і досяг певного успіху. Але повинен існувати інший, більш ефективний шлях. Було би набагато простіше, якби я мав справжнього наставника — того, хто пояснив би мені «що до чого», фахівця, за яким я міг би постійно спостерігати, допомагаючи з виконанням дрібних завдань, людину, яка би рецензувала і спрямовувала мої перші професійні кроки. І як же я потребував його — того, хто став би для мене взірцем для наслідування і навчив головним професійним цінностям.

Сенсея. Наставника. Ментора.

Учнівство

А що відбувається у сфері медицини? Невже ви думаєте, що лікарні беруть на роботу випускників і з першого дня відправляють їх робити операції на серці? Звісно, ні.

Професійна медицина розробила методологію інтенсивного навчання, щільно огорнену ритуалами та освячену традицією. Професійна медицина спостерігає за університетами і стежить за тим, аби випускники здобували кращу освіту. У процесі навчання час приблизно *порівну* розподіляється між заняттями у класах та роботою у клініках поруч із професіоналами.

Після випуску, але ще до отримання ліцензії, молоді лікарі повинні пройти річну практику під керівництвом фахівців (інтернатуру). Відбувається інтенсивне навчання на робочому місці: інтерна буквально оточують зразки для наслідування і ментори.

Після завершення інтернатури різні медичні спеціальності вимагають від трьох до п’яти років подальшої практики (ординатури). Лікар-стажер поступово набуває впевненості, виконуючи все більш відповідальні завдання і залишаючись у середовищі (і під наглядом) більш досвідчених лікарів.

Багато спеціальностей вимагають від одного до трьох років аспірантури, протягом яких продовжується навчання за спеціальністю та накопичення практичного досвіду.

І *лише після цього* молодий фахівець допускається до іспитів й атестації.

Звісно, цей опис стану справ у професійній медицині дещо ідеалізований і, мабуть, не зовсім точний. Але факт залишається фактом: коли ризик великий, ніхто не відправляє зелених новачків до операційної, підкидаючи їм час від часу пацієнтів та очікуючи, що з цього вийде щось путнє. То чому ж саме це відбувається у галузі програмування?

Звісно, кількість смертей через помилки у програмах відносно невелика. З іншого боку, економічні втрати дуже значні. Через недостатню підготовку розробників багато компаній втрачають величезні суми.

Із якоїсь причини в галузі розробки програмного забезпечення утвердилася думка, що програміст одразу ж після отримання диплому може братися за написання коду. Деякі фірми наймають хлопців мало не зі шкільної лави, збирають із них «команди» і доручають розробляти критично важливі системи. Справжнє безумство!

Художники так не роблять. Сантехніки так не роблять. Електрики так не роблять. Напевно, навіть кухарі у фаст-фудах не роблять нічого подібного! Мені здається, що компанії, котрі наймають випускників у галузі комп'ютерних технологій, повинні витрачати на їхнє навчання трохи більше, ніж «McDonald's» витрачає на підготовку своїх працівників.

І не варто дурити себе, стверджуючи, ніби це не важливо. Ставки дуже високі. Наша цивілізація працює на ПЗ. Саме програми займаються переміщенням, обробкою і поширенням інформації, що наповнює наше повсякденне життя. Програми керують двигунами, передачами та гальмами наших машин. Вони підтримують баланс наших банківських вкладів, розсилають рахунки та отримують оплату. ПЗ пере наш одяг і повідомляє час. Вони виводять зображення на екрани телевізорів, відправляють текстові повідомлення, роблять телефонні дзвінки і розважають нас, коли нам нудно. Вони всюди.

Оскільки ж ми довіряємо розробникам всі аспекти нашого життя, від дрібниць до найважливіших сфер, то, на мій погляд, розумний період навчання і практики під керівництвом фахівців є цілком доречним.

Навчання розробці програмного забезпечення

Але як саме випускники університетів *мусять* вливатися до лав професійних програмістів? Який шлях вони повинні проходити? Із якими перешкодами стикатимуться? Яких цілей вони мають досягати?

Розгляньмо професійні рівні розробників, так би мовити, по низхідній.

Майстри

До цієї категорії належать розробники, які очолювали більше ніж один серйозний програмний проєкт. Зазвичай вони мають десятирічний стаж роботи з різними системами, мовами та операційними системами. Вони вміють керувати та координувати роботу кількох команд, є кваліфікованими проєктувальниками, розробниками архітектур і можуть із легкістю закодувати будь-що. Їм пропонували керівні посади, але вони або відхилили пропозицію, або повернулися до попередньої діяльності після цього, або інтегрували керівництво у свою основну технічну роль. Для підтримки кваліфікації у цій ролі вони постійно читають профільну літературу, навчаються, тренуються, працюють і *навчають*. Саме майстри відповідають за технічний бік реалізації проєкту.

Підмайстри

Пересічні розробники — навчені, компетентні та енергійні. У цей період своєї кар'єри вони вчаться працювати в командах і виконувати функції керівників. Вони добре знаються на сучасній технології, але зазвичай їм не вистачає досвіду роботи з різноманітними системами. Як правило, підмайстер володіє однією мовою, знає одну систему та одну платформу; разом із тим він намагається дізнатися більше. Стаж роботи у цій категорії дуже відрізняється, але середнє значення становить близько п'яти років. На ближньому кінці цього

діапазону перебувають нещодавні учні, а на віддаленому — потенційні майстри.

Наставниками підмайстрів є майстри чи найдосвідченіші підмайстри. Молодим підмайстрам рідко надається самостійність. За їхньою роботою щільно спостерігають, а їхній код ретельно перевіряється. У міру накопичення досвіду, ступень самостійності зростає, контроль стає менш щільним і зрештою перетворюється на звичайне рецензування коду.

Учні/стажери

Кар'єра випускника розпочинається з позиції учня. Учні не мають самостійності — їх ретельно контролюють підмайстри. Спочатку учні взагалі не виконують жодних задач, лише допомагають підмайстрам. Це час для надзвичайно інтенсивного парного програмування. Саме тоді вивчаються та фіксуються методи і прийоми, закладається фундамент системи професійних цінностей.

Підмайстри стають наставниками. Вони стежать за тим, аби учні засвоїли принципи і патерни проєктування, методи і ритуали. Підмайстри навчають їх TDD, рефакторингу, мистецтву оцінки тощо. Вони рекомендують учням необхідні видання і публікації, призначають вправи та навчальні завдання, стежать за їхнім прогресом.

Учнівство має тривати не менше року. За цей час, якщо підмайстри забажають інтегрувати новачка у своє середовище, вони звертаються до майстрів по рекомендацію. Майстри перевіряють новачка — як в особистій співбесіді, так і шляхом аналізу їхніх досягнень. Якщо майстри дають згоду, учень перетворюється на підмайстра.

Реальність

І знову мушу зазначити, що цей опис ідеалізований і гіпотетичний. Але, якщо трохи змінити термінологію, ви зрозумієте, що він не так уже відчутно відрізняється від сьогоднішньої ситуації. Випускниками опікуються молоді керівники команд, а тими опікуються керівники проєктів. Проблема в тому, що здебільшого ця опіка має *нетехнічну* природу! Більшість компаній взагалі не має технічного контролю. Розробники отримують надбавки і підвищення… як би то краще сказати? Тому, що приблизно так і повинно бути.

Відмінність між сьогоднішнім станом справ і моєю ідеалізованою схемою полягає в орієнтованості останньої на технічне навчання, опіку та контроль. Ця відмінність виявляється і в уявленні про те, що професійні цінності та технічну кмітливість належить культивувати, плекати і вирощувати. У нашому сучасному «стерильному» підході геть відсутня відповідальність старших за навчання молоді.

Майстерність

Отже, тепер маємо можливість визначити саме поняття *майстерності*. Що це таке насправді? Для того щоби це зрозуміти, почнемо з поняття *«майстер»*. Воно зазвичай асоціюється із майстерністю та якістю, залишає враження досвідченості та компетентності. Майстрами ми називаємо тих, хто працює швидко, але без поспіху, хто розумно оцінює ситуацію і виконує свої зобов'язання. Майстер знає, коли треба сказати «ні», але чесно намагається сказати «так». Майстер — це професіонал.

Майстерність — образ мислення майстра. Це стиль життя з певними уявленнями про цінності, методології, прийоми, підходи та відповіді на питання.

Але як професіонали досягають такого стилю життя? Як формується подібний склад розуму?

Майстерність передається від однієї людини до іншої. Старші навчають молодших. Колеги обмінюються досвідом між собою. Старші, спостерігаючи за молодшими, бачать себе, так би мовити, збоку і заново навчаються. Майстерність поширюється ніби своєрідний інтелектуальний вірус. Ви «заражаєтесь» майстерністю, спостерігаючи за іншими, і дозволяючи їй укорінитися у вашій свідомості.

Як переконати людей

Не можна переконати людей бути майстрами. Не можна переконати їх професійно ставитися до своєї справи. Будь-які аргументи тут не дієві. Ситуаційні дослідження нічого не доводять. Прийняття такого образу мислення — не стільки раціональне, скільки емоційне рішення. Це дуже *людське* рішення.

Як підвести іншу людину до майстерного ставлення до справи? Так, воно заразливе, але лише в тому разі, якщо вона може спостерігати його прояви. Отже, ви мусите *демонструвати* його. Ставати зразком для наслідування. Ви стаєте професіоналом самі і демонструєте свій професіоналізм іншим. А потім сама концепція зробить усю роботу за вас.

Висновки

Навчальний заклад може навчити теорії програмування. Але там не навчають і не можуть навчити практиці, методам і професіоналізму. Усе це набувається роками особистої практики — як у ролі учня, так і в ролі наставника. Настав час усім нам, працівникам галузі програмування, визнати, що завдання виховання наступного покоління розробників доведеться вирішувати нам, а не університетам. І найкраще, що ми можемо зробити зараз,— це створити програму навчання, стажування і довгострокової опіки.

A
НАЛАШТУВАННЯ

1978 року я працював у *Teradyne* над телефонною тестовою системою, про яку вже згадував раніше. Система складалася приблизно з 80 тисяч рядків коду асемблера M365. Початковий код зберігався на магнітних стрічках.

Ті стрічки нагадували восьмидоріжечні стерео-касети, що були шалено популярні в 1970-х роках. Кінець і початок стрічки були склеєні, а стрічковий накопичувач міг перемотувати її лише в одному напрямку. Стрічки в касетах мали довжину 10, 25, 50 та 100 футів. Що довша була стрічка, то більше часу забирало перемотування, бо накопичувачу доводилося просто рухатися вперед до «точки завантаження». На перемотування стофутової стрічки до точки завантаження витрачалося близько п'яти хвилин, тому ми дуже обачно підходили до вибору довжини стрічок[1].

На логічному рівні стрічки поділялися на файли. На одну стрічку можна було записати стільки файлів, скільки вона могла вмістити. Для того щоби знайти потрібний файл, належало завантажити стрічку, а потім послідовно читати файли, поки потрібний не буде знайдений. На дошці в лабораторії завжди висіла роздруківка каталогу з вихідним кодом — за нею ми визначали скільки файлів потрібно пропустити, щоби знайти потрібний.

На полиці в лабораторії також лежала стофутова еталонна копія стрічки з вихідним кодом і позначкою «Майстер». Аби відредагувати файл, ми встановлювали в один накопичувач еталонний екземпляр, а в інший — десятифутову порожню (робочу) стрічку. Еталонна стрічка промотувалася до потрібного файлу, після чого файл копіювався на робочу стрічку. Тоді ми перемотували обидві стрічки на початок, й еталонна стрічка поверталася на полицю.

[1] Стрічки, як я вже казав, можна було перемотувати лише в одному напрямку. Якщо виникала помилка читання, стрічку не можна було перемотати назад і прочитати відповідну ділянку заново. Доводилося припиняти те, що ви робили, перемотувати стрічку до точки завантаження і починати все спочатку. Це відбувалося двічі-тричі на день. Помилки запису теж відбувалися досить часто, і накопичувач не міг їх виявити. Із цієї причини, ми завжди записували стрічки парами, а потім перевіряли кожну після завершення запису. Якщо в одній зі стрічок виявлялася помилка, ми негайно робили копію. Якщо були зіпсовані обидві стрічки (що траплялося дуже рідко), то вся операція повторювалася. Ось так виглядала реальність у 1970-х роках.

Окрім цього, на дошці в лабораторії була присутня спеціальна роздруківка еталонної стрічки «Майстер». Коли ми створювали копії файлів, котрі належало відредагувати, то встромляли в роздруківку кольорову кнопку поруч з ім'ям цього файлу. Ось так і здійснювався контроль за внесенням змін!

Редагування проводилося в екранному режимі. Ми користувалися дуже вдалим текстовим редактором ED-402 — близьким аналогом vi. Сторінка читалася зі стрічки, ми редагували її вміст, записували назад і бралися за наступну. Сторінка зазвичай складалася із приблизно 50 рядків коду. Ми не мали можливості зазирнути вперед, аби побачити наступні сторінки, і не могли повернутися назад — до вже відредагованих сторінок. Тому ми використовували лістинги.

У лістингах позначалися всі зміни, після чого файли редагувалися відповідно до позначок. *Ніхто* не писав і не змінював код спонтанно! Це було би чимось на кшталт самогубства. Після внесення змін до всіх файлів, що потребували редагування, ми поєднували їх із вмістом еталонного екземпляра. Отриману стрічку використовували для здійснення компіляції і тестування.

Завершивши тестування і переконавшись, що зміни працюють, ми дивилися на роздруківку. Якщо на ній не було нових кнопок, робоча стрічка просто перейменовувалась на еталонну, а всі «відпрацьовані» кнопки знімалися з дошки. Якщо ж на дошці з'являлися нові кнопки, ми прибирали всі попередні і передавали робочу стрічку тому, хто їх виставив, для злиття результатів.

У команді було лише троє розробників, і кожен із них мав кнопки відповідного кольору. Тому ми завжди могли визначити, хто взяв на редагування ті чи інші файли. А оскільки всі ми працювали в одній лабораторії і постійно спілкувалися один з одним, поточний стан кодової бази постійно зберігався в наших головах. Найчастіше навіть роздруківка виявлялася зайвою, і ми обходилися без неї.

Інструменти

Сучасним розробникам доступні найрізноманітніші інструменти програмування. Від деяких, на мою думку, варто триматися якнай-

далі, але є серед них і такі, якими повинен упевнено володіти будь-який розробник. У цьому розділі описаний мій особистий поточний інструментарій. Я не наводжу повного опису всіх представлених інструментів, тому огляд не є вичерпним, і розповідаю лише про те, що використовую сам.

Управління вихідним кодом

Що стосується управління вихідним кодом, то, як правило, краще використовувати програми з відкритим кодом. Чому? Бо вони пишуться розробниками для розробників. Інакше кажучи, розробники пишуть програми з відкритим кодом для себе, коли їм необхідне працездатне рішення.

Існує багато дорогих комерційних («корпоративних») систем контролю за версіями. На мою думку, ними спокушаються не стільки розробники, скільки керівники, менеджери та «інструментальні групи». Список функцій таких програм завжди виглядає вражаюче. Але, на жаль, у ньому часто не виявляється того, що дійсно потрібне розробникам. До того ж, головною проблемою для користувачів зазвичай стає їх *швидкість*.

«Корпоративні» системи керування вихідним кодом

Цілком можливо, що ваша компанія вже вклала цілий статок у «корпоративну» систему керування вихідним кодом. У такому разі — прийміть мої співчуття. Звісно, із політичних міркувань ви не зможете просто бовкнути: «Дядечко Боб каже, що цим лайном не треба користуватися». Однак існує простий вихід.

Ви можете реєструвати вихідний код у «корпоративній» системі наприкінці кожної ітерації (кожні два тижні або близько того), а в середині ітерацій використовувати систему з відкритим кодом. Усі будуть задоволені, ви не порушите корпоративних настанов, а ваша продуктивність залишиться високою.

Песимістичне та оптимістичне блокування

У 1980 роках песимістичне блокування здавалося гарною ідеєю. Зрештою, найпростіший шлях до розв'язання проблем паралельного оновлення — «розпаралелювання». Якщо я редагую файл, то вам краще його не чіпати. Система кольорових кнопок, яку ми використовували наприкінці 1970-х, була своєрідним механізмом песимістичного блокування. Якщо файл був позначений певною кнопкою, інші не повинні були його редагувати.

Звісно, песимістичне блокування має свої недоліки. Якщо заблокувати файл і піти у відпустку, то решта користувачів, які забажають працювати із цим файлом, опиняться в безвиході. Навіть якщо файл залишиться заблокованим на день-два, це затримає роботу інших людей.

Сучасні інструменти значно краще справляються зі злиттям паралельно редагованих вихідних файлів. Адже це нетривіальне завдання: програма аналізує два різні файли разом із попередниками цих двох файлів, а потім застосовує відповідні стратегії для визначення способу інтеграції паралельних змін. І мушу сказати, вона робить це добре.

Отже, час песимістичного блокування минув. Тепер нам не потрібно робити блокування файлів під час редагування. Ба більше, нам узагалі не потрібно турбуватися про блокування окремих файлів — ми запитуємо відразу всю систему й редагуємо потрібні файли.

Коли все буде готове до реєстрації змін, виконується операція оновлення. Вона повідомляє нам, чи не було більш ранньої реєстрації змін, здійсненої іншими користувачами, виконує автоматичне злиття більшості змін, знаходить конфлікти і допомагає вирішити їх. Після цього об'єднаний код додається в кодову базу.

Далі я більш докладно висвітлю роль автоматизованих тестів і безперервної інтеграції у цьому процесі. А тут варто підкреслити ось що: код, який не пройшов всіх тестів, за жодних умов не повинен додаватися в кодову базу. *Ніколи*.

CVS/SVN

Одна із традиційних систем управління вихідним кодом — CVS — була гарною для свого часу, але в сучасних проєктах уже починає відставати. Хоча CVS відмінно працює з окремими файлами і теками, вона не дуже добре справляється з перейменуванням файлів та видаленням тек. Що вже казати про підтеки… І на цьому — завіса. Що менше говорити про це, то краще.

Із іншого боку, система Subversion працює дуже добре. Вона дозволяє отримати доступ до всієї системи за одну операцію. У ній легко виконуються операції оновлення, злиття та закріплення змін. За відсутності розгалужених змін, системи SVN доволі прості в управлінні.

Розгалуження

До 2008 року я уникав будь-яких форм розгалуження (branching), окрім найпростіших. Якщо розробник створює гілку, то ця гілка має бути повернена до основної лінії ще до кінця ітерації. При цьому я так суворо ставився до розгалуження, що воно дуже рідко застосовувалося у тих проєктах, у яких я брав участь.

Якщо ви використовуєте SVN, то я, як і раніше, вважаю, що це вдала політика. Однак останнім часом з'явилися нові інструменти, що цілковито змінили ситуацію. Я маю на увазі *розподілені* системи управління вихідним кодом. У цій категорії моїм улюбленим інструментом є `git`. Зараз я розповім про цю систему докладніше.

git

Я почав використовувати `git` наприкінці 2008 року. Ця система цілковито змінила мій підхід до управління вихідним кодом. Пояснення того, чому ця програма так відчутно змінила правила гри, виходять за межі цієї книжки. Проте порівняння рис. Д.1 із рис. Д.2 набагато красномовніше будь-яких слів.

На рис. Д.1 показаний перебіг розробки проєкту FitNesse за кілька тижнів під керуванням SVN. Ви бачите наслідки моєї жорсткої політики відмови від розгалуження. Ми просто його не використовували. Натомість у головній гілці часто виконувалися операції оновлення, злиття та закріплення.


```
More bug fixes
Docs now say that Java 1.5 is required.
Bug fix
Many usability and behaviorial improvements.
Clean up
Added PAGE_NAME and PAGE_PATH to pre-defined variables.
Added ** to !path widget.
link to the fixture gallery
fixture gallery release 2.0 (2008-06-09) copied into the trunk wiki at
Firefox compatability for invisible collapsible sections; removed .ce
Updated documentation suite for all changes since last release.
Enhancement to handle nulls in saved and recalled symbols.  Adde
Added a "Prune" Properties attribute to exclude a page and its chil
Fixed type-o
Added check for existing child page on rename.
Added "Rename" link to Symbolic Links property section; renamed
Adjusted page properties on recently added pages such that they c
Enhanced Symbolic Links to allow all relative and absolute path for
Cleaned up renamPageReponder a bit more.
Cleaned Up PathParser names a bit.  Pop -> RemoveNameFromE
Cleaned up RenamePageResponder a bit.  Fixed TestContentsHel
updated usage message
Fixed a bug wherein variables defined in a parent's preformatted bl
Added explicit responder "getPage" to render a page in case query
Tweaks to TOC help text.
New property: Help text; TOCWidget has rollover balloon with new
Redundant to the JUnit tests and elemental acceptance tests.
Removed the last of the [acd] tags.
!contents -f option enhancement to show suite filters in TOC list; fix
TOC enhancements for properties (-p and PROPERTY_TOC and F
1) Render the tags on non-WikiWord links;
Added http:// prefix to google.com for firewall transparency.
Isolate query action from additional query arguments.  For example
Accommodate query strings like "?suite&suiteFilter=X"; prior logic v
Cleaned up AliasLinkWidget a bit.
```

Рис. Д.1. Проєкт FitNesse під керівництвом subversion

На рис. Д.2 показана структура кількох тижнів розробки того самого проєкту з використанням `git`. Як бачите, операції розгалуження та злиття відбуваються постійно. І річ не в тім, що я чомусь змінив своє ставлення до розгалуження. Просто такий робочий процес став очевидним та ефективним. Окремі розробники створюють гілки

з дуже коротким терміном життя, а потім поєднують їх із результатами колег тоді, коли вважають за потрібне.

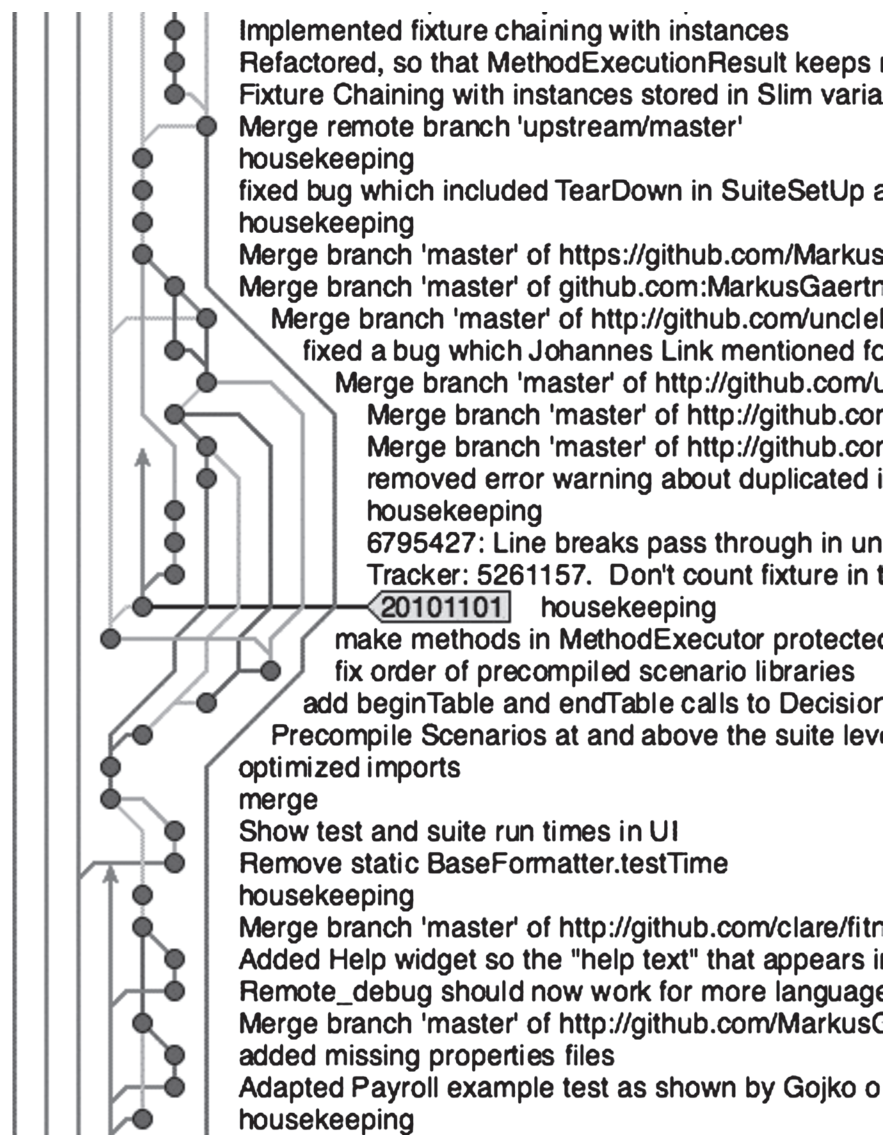


Рис. Д.2. Проєкт FitNesse під керівництвом `git`

Також зауважте, що на рисунку не видно «справжньої» головної гілки. Річ у тім, що її *просто немає*. Розробники зберігають на своїх локальних комп'ютерах копії *всієї* історії проєкту. Вони вносять зміни і реєструють їх у своїй локальній копії, а потім синхронізують із копіями колег у разі потреби.

Щоправда, я підтримую спеціальний «золотий репозиторій», у якому зберігаються всі опубліковані версії та внутрішні збірки. Але називати його «головною гілкою» означало би не розуміти суті справи. Насправді це лише зручна «світлина» всієї історії, що зберігається локально кожним розробником.

Якщо ви не зрозуміли щось зі сказаного, це нормально. Спочатку `git` викликає чимало труднощів. До того, як працює ця система, потрібно звикнути. Але запевняю вас: `git` та інші подібні системи — це майбутнє керування вихідним кодом.

IDE/Редактор

Ми, розробники, проводимо більшу частину часу за читанням та редагуванням коду. Інструменти, що ми використовуємо для цього, значно змінилися за минулі роки. Деякі з них мають неймовірну міць, а деякі майже не змінилися з 1970-х років.

VI

Здається, що доба використання vi як основного редактора для розробки давно минула. У наші дні з'явилися інструменти, що значно перевершують vi за своїми можливостями, та інші нескладні текстові редактори того самого типу. Однак останнім часом спостерігається помітний сплеск популярності vi. Це пояснюється його простотою, зручністю використання, швидкістю та гнучкістю. Хоча vi й поступається Emacs або Eclipse за широтою можливостей, він залишається швидким та потужним редактором.

Водночас, я вже не досвідчений користувач vi. Колись мене називали «богом» vi, але цей час давно минув. Я використовую vi час від часу, коли потрібно швидко відредагувати текстовий файл. Нещодавно я застосовував його для швидкої зміни вихідного файлу Java у віддаленому режимі. Але обсяг повноцінного коду, написаного мною у vi за останні 10 років, істотно зменшився.

Emacs

Emacs досі залишається одним із найпотужніших редакторів і, ймовірно, залишиться ним ще кілька найближчих десятиліть. Його потужність забезпечується внутрішньою моделлю з урахуванням lisp. У категорії інструментів редагування загального призначення жоден інший редактор навіть віддалено не може з ним конкурувати. З іншого боку, на мою думку, Emacs не може повноцінно конкурувати зі спеціалізованими інтегрованими середовищами розробки (IDE), що зараз займають провідні позиції. Редагування коду не є редагуванням загального призначення.

У 1990 роках я був фанатом Emacs. Я просто не розглядав інших варіантів. Тодішні редактори з керуванням мишею були просто кумедними іграшками, до яких жоден розробник не ставився серйозно. Але на початку 2000-х я познайомився з IntelliJ — моїм фаворитом серед IDE, і вже не озирався назад.

Eclipse/IntelliJ

Я — відданий користувач IntelliJ. Я люблю цю систему й використовую її для написання коду на Java, Ruby, Clojure, Scala, Javascript та багатьох інших мовах. Вона була створена розробниками, які розуміють, що саме потрібно колегам під час написання коду. За минулі роки IntelliJ майже ніколи не підводила мене, а враження від неї завжди залишалися позитивними.

Середовище Eclipse за своєю потужністю і масштабом можна порівняти з IntelliJ. Ці два середовища якісно перевершують Emacs за можливостями редагування Java-коду. У цій категорії існують інші IDE, але я не згадую їх тут, бо в мене немає безпосереднього досвіду їх використання.

Ці інтегровані середовища відрізняються від таких інструментів, як Emacs, насамперед надзвичайно багатими можливостями маніпулювання кодом. Наприклад, IntelliJ дозволяє отримати суперклас із класу всього однією командою. Ви можете перейменовувати змінні, отримувати методи, перетворювати наслідування на композицію — і це далеко не весь перелік.

Із цими інструментами редагування коду переходить від рівня рядків і символів до більш складних маніпуляцій. Розробник при цьому думає не про кілька наступних символів і рядків, що він збирається ввести, а про наступні перетворення. Коротше кажучи, модель програмування у цих нових системах значно змінюється і стає більш продуктивною.

Звісно, за широту можливостей доводиться платити. Освоєння інтегрованих середовищ вимагає багато часу і зусиль, а фаза початкової підготовки проєкту стає досить помітною. Крім того, ці інструменти дуже вимогливі — для їхньої роботи необхідні значні обчислювальні потужності.

TextMate

Редактор TextMate потужний і невимогливий до ресурсів. Щоправда, він не вміє виконувати такі приголомшливі маніпуляції, на які здатні IntelliJ й Eclipse. У нього немає потужного lisp-ядра та бібліотеки Emacs. Він не має швидкості та гнучкості vi. З іншого боку, його можна швидко освоїти, а всі операції є інтуїтивно зрозумілими.

Я використовую TextMate лише іноді, особливо для спонтанного програмування на C++. У великому проєкті я би скористався Emacs, але для невеликих завдань на C++ мені просто лінькі з ним возитися.

Відстеження завдань

Зараз я використовую Pivotal Tracker. Ця система проста і елегантна, вона добре інтегрується з Agile/ітеративними методологіями і дозволяє всім зацікавленим сторонам та розробникам швидко спілкуватися один з одним. Я дуже задоволений.

У дуже маленьких проєктах я іноді використовую Lighthouse. Ця система проста і зручна в налаштуванні та використанні, але за широтою можливостей навіть не наближається до Tracker.

Також у деяких випадках я просто використовую wiki. Таке рішення добре підходить для внутрішніх проєктів. Wiki дозволяє застосувати

будь-яку організаційну схему на ваш розсуд. Ніхто не змушує вас використовувати конкретний процес чи жорстку структуру. Цей варіант теж надзвичайно простий для розуміння та використання.

Іноді найкращою системою відстеження завдань стає звичайна настінна дошка з набором карток. Дошка ділиться на стовпці на кшталт «Слід зробити», «У розробці» та «Готово». Розробники просто переносять картки з одного стовпця до іншого під час роботи над завданнями. Мабуть це одна з найпоширеніших систем відстеження завдань у сучасних командах, які використовують Agile-методології.

Я рекомендую своїм клієнтам починати з подібної «ручної» системи, перш ніж купувати спеціалізовані програми. Освоївши «ручне» відстеження, клієнт отримує знання, що дозволяють йому зробити розумний вибір — іноді на користь тієї ж дошки з картками.

Лічильники дефектів

Команді розробників напевно потрібен перелік поточних завдань. До них належать як завдання щодо реалізації нових можливостей і функцій, так і виправлення помилок. Для команди звичайного розміру (від 5 до 12 розробників) подібний список має містити від кількох десятків до сотень позицій. *Але не тисяч!*

Якщо у вашій системі тисячі помилок, із нею явно щось не так. Якщо ваш список налічує тисячі завдань та/або функцій, із ним теж щось не так. У загальному випадку список повинен бути відносно невеликим, тому для роботи з ним має вистачити таких нескладних інструментів, як wiki, Lighthouse або Tracker.

На ринку присутні комерційні програми, що мають непоганий вигляд. Я бачив, як мої клієнти користуються ними, але поки що не мав можливості попрацювати особисто. Не маю нічого проти таких інструментів, але за умови, що список завдань залишається невеликим та легко керованим. Коли засоби відстеження завдань вимушені працювати зі списками, що налічують тисячі позицій, саме слово «відстеження» втрачає сенс. Список поточних завдань перетворюється на «звалище поточних завдань» (і часто тхне як будь-яке звалище).

Безперервне складання

Останнім часом, для забезпечення безперервного складання я використовую Jenkins. Система невибаглива, проста, а робота з нею не вимагає тривалої підготовки. Ви завантажуєте програму, запускаєте її, проводите нескладне налаштування конфігурації — і далі все працює. Дуже зручно.

Мій підхід до безперервного складання простий: зв'яжіть його із системою управління вихідним кодом. Щоразу коли в базі реєструється змінений код, має виконуватися автоматичне безперервне складання, звіт про результати якого повинен видаватися команді.

Команда зобов'язана стежити, щоби складання завжди виконувалося успішно. Якщо складання не проходить, це повинно стати тривожним сигналом, а команда має негайно зібратися для розв'язання проблеми. За жодних умов збій, що стався, не повинен зберігатися протягом доби або довше.

У проєкті FitNesse я вимагаю, щоби кожен розробник виконував сценарій безперервного складання перед реєстрацією внесених змін. Складання забирає менше п'яти хвилин, що не надто обтяжливо. Якщо під час складання виявляються проблеми, розробники усувають їх до реєстрації. Отже, автоматичне складання лише зрідка стикається із будь-якими проблемами. Найпоширенішим джерелом збоїв автоматичного складання стають проблеми, пов'язані із налаштуваннями робочого середовища, оскільки моє середовище автоматичного складання значно відрізняється від середовищ розробників.

Інструменти модульного тестування

Для кожної мови існують свої спеціалізовані засоби модульного тестування. Я особисто використовую JUnit для Java, rspec для Ruby, NUnit для .Net, Midje для Clojure і CppUTest для C та C++.

Утім, хай би який інструмент модульного тестування ви обрали, всі вони повинні мати набір базових властивостей і функцій.

1. Тести повинні запускатися легко і швидко. Не має значення, як саме це робиться — за допомогою плагінів IDE або простих ути-

літ командного рядка; головне полягає в тому, щоби розробники могли запускати ці тести на власний розсуд. Команда запуску має бути тривіально простою. Наприклад, я запускаю свої тести CppUTest у TextMate простим натисканням клавіш *command+M*. Я зв'язав цю комбінацію із запуском makefile, що автоматично виконує тести і виводить короткий однорядковий звіт у тому випадку, якщо всі тести пройдені успішно. IntelliJ підтримує JUnit та rspec, тому мені потрібно лише натиснути кнопку. Для виконання NUnit я використовую плагін Resharper.

2. Інструмент має видавати чітку візуальну ознаку проходження/непроходження тесту — хай це буде зелена смуга на графічному екрані або консольне повідомлення «Усі тести пройдено». Суть у тому, щоби ви могли швидко й однозначно визначити, що всі тести пройдені. Якщо для визначення результату тестування доводиться читати багаторядковий звіт чи (що ще гірше) порівнювати вихідні дані двох файлів,— умову не виконано.
3. Програма має видавати чітку візуальну ознаку прогресу. Не важливо, чи буде це графічний індикатор або рядок точок, що поступово з'являються,— головне, щоби користувач бачив, що процес триває, а тестування не зупинилося і не перервалося.
4. Програма має перешкоджати взаємодії між окремими тестовими сценаріями. JUnit вирішує цю проблему, створюючи новий екземпляр тестового класу кожного тестового методу; таким чином цей інструмент запобігає використанню змінних екземплярів у взаємодії між тестами. Інші інструменти запускають тестові методи у випадковій послідовності, щоб унеможливити будь-яку залежність від конкретної послідовності виконання тестів. Незалежно від вибору конкретного механізму, програма має вжити заходів для забезпечення незалежності тестів. Залежні тести — підступна пастка, якої слід уникати.
5. Інструмент має за можливості спрощувати написання тестів. Наприклад, JUnit надає зручний API для перевірки умов (assertions), а також використовує рефлексію та атрибути Java, щоби відрізняти тестові функції від звичайних. Зрештою вдалі IDE можуть автоматично ідентифікувати тести, позбавляючи вас від клопоту із підготовкою.

Інструменти тестування компонентів

Інструменти цієї категорії призначені для тестування компонентів на рівні API. Їхня роль — визначення поведінки компонента мовою, зрозумілою для бізнесу і спеціалістів із контролю якості. В ідеалі бізнес-аналітики та фахівці з контролю якості мусять писати специфікації з використанням саме цих інструментів.

Визначення «Виконано»

Інструменти компонентного тестування більшою мірою, ніж будь-які інші, виражають наше уявлення про те, що розуміється під «виконаною» роботою. Коли бізнес-аналітики і спеціалісти з контролю якості створюють специфікацію, що визначає поведінку компонента, і коли ця специфікація виконується як набір тестів, які або проходять, або ні,— поняття «done» набуває абсолютно чіткого значення: «Всі тести пройдено».

FitNesse

Мій улюблений інструмент компонентного тестування — FitNesse. Я написав більшу частину його коду, і я його основний автор. І взагалі це моє дітище.

FitNesse — система на базі wiki, що дозволяє бізнес-аналітикам і спеціалістам із контролю якості писати тести в дуже простому табличному форматі. Ці таблиці за формою і призначенням мають багато спільного з таблицями Парнаса. Тести легко поєднуються в пакети, які можна виконати будь-якої миті.

Система FitNesse написана на Java, однак може використовуватися для тестування систем на будь-якій мові, бо взаємодіє із базовою системою тестування, що може бути написана будь-якою мовою. Нині підтримуються мови Java, C#/.NET, C, C++, Python, Ruby, PHP, Delphi тощо.

В основі FitNesse лежать дві системи: Fit і Slim. Систему Fit написав Ворд Каннінгем, вона стала джерелом натхнення для FitNesse і є її

близьким родичем. Slim — простіша і більш портативна система тестування, якою сьогодні користуються багато прихильників FitNesse.

Інші інструменти

Мені відомі ще кілька систем, які можна зарахувати до категорії інструментів для тестування компонентів.

- Система RobotFX була розроблена інженерами *Nokia*. Як і у FitNesse, у ній використовується табличний формат, але не на базі wiki. RobotFX просто працює з неструктурованими файлами, створеними в Excel або іншій аналогічній програмі. Програма написана на Python, але, за використання відповідних мостів, може тестувати системи, написані будь-якою мовою.
- Green Pepper — комерційна програма, що в деяких аспектах схожа на FitNesse. Базується на популярній confluence wiki.
- Cucumber — текстова програма, що працює на ядрі Ruby, але підходить для тестування на багатьох різних платформах. У мові Cucumber використовується популярний стиль Given/When/Then.
- JBehave — аналог та «ідеологічний батько» Cucumber. Програма написана на Java.

Інструменти інтеграційного тестування

Інструменти компонентного тестування можуть використовуватися для виконання багатьох інтеграційних тестів, але погано підходять для тестів, керованих через інтерфейс користувача.

В ідеальному випадку кількість тестів, керованих через інтерфейс користувача, повинна бути зведена до мінімуму, бо інтерфейси користувача славляться своєю нестабільністю. Завдяки цій нестабільності тести, що виконуються через інтерфейс користувача, стають дуже крихкими.

Утім, деякі тести мусять виконуватися через інтерфейс користувача — і насамперед йдеться про тести самого інтерфейсу. Крім того,

деякі комплексні тести *мають* проходити через усю зібрану систему, включно з інтерфейсом користувача.

Для тестування інтерфейсу користувача я зазвичай використовую програми Selenium і Watir.

UML/MDA

На початку 1990 років я дуже сподівався, що галузь автоматизованої розробки ПЗ кардинально змінить характер діяльності розробника. Зазираючи в майбутнє із тих бурхливих днів, я сподівався, що через пару десятиріч усе вже буде кодуватися у вигляді діаграм на вищому рівні абстракції, а текстовий код залишиться в минулому.

Як же я помилявся! Мої мрії не лише не справдилися, а й незліченні спроби руху в цьому напрямку зазнали цілковитого краху. Річ навіть не в тім, що не існує інструментів і систем, що мають потенціал для такого підходу; просто ці інструменти по-справжньому не реалізують цю мрію і навряд чи хтось забажає ними скористатися.

Так, я мріяв, що розробники позбавляться подробиць текстового коду й почнуть будувати системи на більш високому діаграмному рівні. У цих зухвалих мріях розробники взагалі здавалися зайвими. Архітектори могли будувати цілі системи за діаграмами UML. Ядро системи — величезне, безпристрасне і байдуже до турбот пересічних розробників — самотужки перетворюватиме діаграми на виконуваний код. Ось як виглядала велична мрія про модельно-орієнтовану архітектуру (MDA, Model Driven Architecture).

На жаль, у величної мрії була і є одна крихітна вада. MDA передбачає, що проблема в коді. Але проблема не в коді. Це ніколи не було проблемою. Проблема в *деталізації*.

Деталізація

Розробники керують деталями. Так, саме цим ми займаємося. Ми описуємо поведінку системи в найдрібніших подробицях. Задля цього ми використовуємо текстові мови, бо вони надзвичайно зручні (візьміть, наприклад, хоча б англійську).

Але якими деталями ми керуємо?

Чи відомо вам, чим відрізняються два символи \n і \r? Перший із них — це перехід рядка (line feed), а другий — повернення каретки (carriage return).

Що таке каретка? У 1960-х і на початку 1970 років основним пристроєм виведення для комп'ютерів був телетайп. Модель ASR33[1] була найпоширенішою.

Цей пристрій складався із друкувальної голівки, що могла друкувати до 10 символів за секунду. Друкувальна голівка складалася із циліндра, на який були нанесені рельєфні символи. Циліндр повертався так, аби до паперу був звернений правильний символ, після чого молоточок ударяв по циліндру і притискав його до паперу. Між циліндром і папером містилася просочена фарбою стрічка, фарба з якої переносилася на папір, повторюючи обриси символу.

Друкувальна голівка переміщалася на каретці. Із кожним символом каретка зміщувалася на одну позицію праворуч, разом із нею зміщувалась і друкуюча головка. Коли каретка досягала кінця 72-символьного рядка, її доводилося повертати на початок рядка, вводячи символ повернення каретки (\r = 0 × 0D); без цього друкуюча голівка продовжувала би друкувати символи у сімдесят другому стовпці, і від багаторазового друку там виникав би чорний прямокутник.

Звісно, цього було замало. Повернення каретки не призводило до зсуву паперу до наступного рядка. Якби після повернення каретки не передавався символ переходу рядка (\n = 0 × 0A), то новий рядок друкувався би поверх старого.

[1] http://en.wikipedia.org/wiki/ASR-33_Teletype.

Отже, для телетайпа ASR33 всі рядки повинні були завершуватися послідовністю \r\n. Однак і тут була необхідна обережність, бо повернення каретки могло забрати більше ста мілісекунд. Якщо обмежитися лише відправкою \n\r, наступний символ міг бути надрукований під час зворотного ходу каретки, а в середині рядка з'явився би змащений символ. Для надійності, символи кінця рядка часто доповнювалися одним або двома символами «забій»[1] (0 × FF).

У 1970 роках, коли телетайпи почали поступово відходити в минуле, в операційних системах UNIX послідовність кінця рядка скоротилася до n. Однак інші операційні системи на кшталт DOS продовжували використовувати позначку \r\n.

Коли до вас потрапляв текстовий файл, що використовував «неправильне» позначення? Я стикаюся із цією проблемою не рідше одного разу на рік. Два ідентичні файли з вихідним кодом не порівнюються, а їхні контрольні суми різняться, бо в них використовуються різні завершувачі рядків. Текстові редактори некоректно переносять слова або поділяють рядки подвійним інтервалом, бо інтерпретують \r\n як два рядки. Деякі програми розуміють \r\n, але не розпізнають \n\r… І так далі.

Ось що я маю на увазі під *деталізацією*. Спробуйте закодувати логіку обробки завершувачів рядків на UML!

Без змін та надій

Рух MDA обіцяв, що використання діаграм замість коду дозволить виключити із програмування велику кількість другорядних подробиць. Досі ці обіцянки не виконані. Як виявилося, у коді не так багато зайвих деталей, що їх можна було би виключити за допомогою діаграм. Ба більше: діаграми також містять власні другорядні

[1] Ці символи виявилися надзвичайно корисними при редагуванні перфострічок. За правилами, символи «забій» ігнорувалися при введенні. Їх код 0 x FF означав, що у стрічці пробиті отвори у всіх позиціях. Отже, будь-який символ можна було перетворити на «забій» — достатньо пробити «забій» поверх попереднього символу. Отже, якщо ви припускалися помилки під час введення програми, можна було повернутися до попереднього символу, натиснути клавішу «забій», а потім продовжити введення.

деталі. Вони мають свою граматику, синтаксис, правила та обмеження. Зрештою розбіжності в ступені деталізації виявляються незначними.

Рух MDA також обіцяв, що робота з діаграмами буде вестися на більш високому рівні абстракції, ніж робота з кодом,— подібно до того, як Java стоїть на вищому рівні абстракції порівняно з асемблером. Але і ці сподівання також не виправдалися. Відмінності в рівні абстракції виявилися, у кращому разі, незначними.

Наостанок уявіть, що одного прекрасного дня буде винайдено дійсно працездатну й ефективну діаграмну мову. Але ж малювати діаграми будуть не архітектори, а ті самі розробники! Діаграми перетворяться на новий різновид коду, і розробникам доведеться «*малювати*» цей код — адже все зводиться до подробиць, а керувати ними доводиться саме розробникам.

Висновки

Відтоді як я зайнявся програмуванням, з'явилися безліч нових і надзвичайно потужних інструментів. Мій поточний інструментарій обмежується невеликою підмножиною цього арсеналу. Я використовую git для керування вихідним кодом, Tracker для відстеження поточних завдань, Jenkins для безперервного складання, інтегроване середовище IntelliJ, XUnit для тестування і FitNesse для компонентного тестування.

Я працюю на комп'ютері Macbook Pro із процесором Intel Core i7 із тактовою частотою 2,8 ГГц, із 17-дюймовим монітором, 8 Гб пам'яті, 512 Гб SSD та двома додатковими екранами.

В ПОКАЖЧИК

А

Автоматизовані приймальні тести 109, 139, 142–143, 151, 155, 211, 233

Б

Безжальний рефакторинг 46
Безперервна інтеграція (CI) 49, 129, 150, 155–156, 158, 233
Безперервне
  навчання 50
  програмування 94
  складання 241, 248
Бізнес-цілі 202
Блокування 99, 102, 233
Боссавіт, Лорен 128

В

Вадза 130
Вибачення 42
Вимоги
  комунікації до 134–138
  невизначеність і 136
  передчасна точність в 136
  пізня неоднозначність в 137–138
  тривожність із приводу оцінки 136
Відволікання 105
Відкритий код 44, 131, 143, 232
Відлагодження 101–104
Відстеження завдань 239–240
Водіння 105
Впевненість 117–118, 120
Втома 105, 197

Г

Галло, Еммануель 128
Гіркий досвід 222–223
Гнучкість 45–46, 237, 239
Готовність 86–89, 99, 103, 108–110, 203, 207
Гра в кеглі 127–129
Графічний інтерфейс користувача 148–150
Греннінг, Джеймс 186

Д

Демонстраційні зустрічі 166–167
Деталі 246–247
Додзьо кодування 127–130
Документація 86–88, 90, 118–120, 159
Допомога 69, 86, 98–99, 109–111, 146, 164, 197, 203

З

Закон великих чисел 188
Запізнення 106–109
Зарозумілість 53
Згуртована команда 69, 210–214
Зниження щільності дефектів 117
Зона потоку 96–99
«Зроблено», визначення 81–86, 90
Зустрічі із планування ітерацій 165–166

І

Ідентифікація з роботодавцем/замовником 47
Імовірність 179
Інверсія пріоритетів 172, 174
Інструменти 231
Інструменти, аналіз за трьома змінними 182, 188
Інтеграційне тестування 156, 158–160, 244, 245
Інтеграція, безперервна 49, 129, 150, 155–156, 158, 233

К

Ката 51, 127–130
Клієнт, ідентифікація з 52
Код
  приналежність 205–206
  третя година ночі 94
  управління 232–237
Колективна приналежність коду 206
Команди та командна робота 63–69
  дилема власника проєкту з 213
  згуртована 210–212
  керування 212–213
  намагається і 59–61
  пасивно-агресивна поведінка і 67–69
  швидкість 212–213
Компонентні тести
  інструменти для 243–244
  у стратегії тестування 154
Комунікація
  вимоги 134–138
  приймальні тести і 139, 142
  тиск і 194
Контроль
  вихідного коду 232–237
  зобов’язання і 84
  якості
    автоматизований 44–45
    ідеальний варіант як відсутність проблем 149–151, 154–155
    проблеми, знайдені 43
    як ловці помилок 42–43
    як специфікатори 155
    як частина команди 154–155
Конфлікт під час зустрічей 166–167
Концентрація 168–169
Кофеїн 169

Л

Ліндстром, Ловелл 24, 187
Лічильники дефектів 240

М

Майстер 225, 227
Майстерність 227
Мана 168
Методи 49
Методологія розробки через тестування 49, 104, 114–115, 155, 195, 197
  архітектура і 119–120
  визначення 44
  дебют 114–115
  документація і 118–119
  зниження щільності дефектів і 117–118
  переваги 117–120
  перерви і 105
  перешкоди і 98–99
  три закони 116–120
  упевненість і 117
  час циклу в 114–115
  чим це не є 120–121
Метод швидкого голосування 185–186
Модельно-орієнтована архітектура 245–248
Модульні тести
  інструменти для 241–242
  приймальні тести 139–151
  у стратегії тестування 156
Музика 97–98

Н

Навчання трудова етика і 47–49
Наставництво 217–223
Невизначеність вимоги і 136
Неоднозначність у вимогах 137–138
Неочевидні зобов’язання 182
Нетрадиційне наставництво 221–222
Номінальна оцінка 182

О

Оптимістичне блокування 233
Оцінка
  визначення 178–180
  завдання 185–188
  закон великих чисел і 188
  за трьома змінними 188
  зобов’язання і 178
  імовірність і 179

номінальна 182
оптимістична 182
песимістична 182
тривожність 136
Очікування зобов'язання і 85

П

Паніка 196–197
Парне програмування 49, 97, 99–100, 195, 197, 206–208, 226
Пасивно-агресивна позиція 60, 68, 146–147, 167
Патерни проєктування 45, 49, 70, 226
Переговори приймальні тести і 146–147
Передбачувані зобов'язання 181
Перезаряджання 169–170
Перерви 105
Песимістична оцінка 182
Песимістичне блокування 233
Підмайстер 225–226
«Покер планування» 186–187
Поспіх 75, 107, 196, 227
«Потрібно/повинен» 82
Правило «Не зашкодь» 41–46
Практика
досвід і 131–132
етика 132
трудова етика і 47–53
час виконання і 126–127
Приймальні тести
автоматизовані 142–143
безперервна інтеграція та 150–151
взаємодія сторін і 139, 142
визначення 139
графічні інтерфейси і 149–150
додаткова робота і 144
модульні тести і 148
написання 144–145
пасивно-агресивна позиція і 146–147
переговори і 146–147
роль розробника в 145–146
терміни 144–145
Принципи проєктування 49
Програмісти
люди та 201–205
програмісти і 205–207
роботодавці і 202–205
Простота 44–45, 237
Протилежні ролі 58–62

Р

Рандорі 130–131
Репутація 41, 107, 172, 176, 178–179
Ретроспективні зустрічі 166
Роботодавці ідентифікація з 47
Роботодавці і розробники 202–205

Розбіжності 95–96, 137, 166–167, 186–187, 248
Розгалуження 234–237
Ролі, протилежні 58–62
Ручні дослідницькі тести 159–160

С

Сантана, Карлос 127
Системні тести у стратегії тестування 159
Сон 169
Співпраця 12, 110–111, 144, 199, 201, 207–208
Сподівання, терміни і 106–107
Споріднене оцінювання 187
Спостерігання за собою 105–106
Ставки 62–63
Стажери 226
Стендап-зустрічі 165
Структура
важливість 45
гнучкість і 45
«не зашкодь» 45–46

Т

Творчий внесок 100
Терміни
надія і 106
понаднормова праця і 108
поспіх і 107
хибна готовність і 108–109
Тестування
GUI і 148–150
автоматизоване приймання 155–159
безперервна інтеграція і 155–156, 158
визначення 139
зв'язок і 155
інструменти для 241–245
інтеграція інструменти для 244–245
модульні тести і 148
пасивно-агресивна поведінка 146–147
переговори і 146–147
піраміда автоматизації 155–159
додаткова робота і 144
роль розробника в 145–146
терміни 144–145
Тиск
взаємодія і 196–197
дисципліна і 195
допомога і 197
зобов'язання і 194
кризові ситуації і 195
паніка і 196
уникання 194–195
чистота і 195
Томас, Дейв 128
Точність
вимоги 142
оцінки 185, 188
передчасна 136–137

Трудова етика 47
- безперервне навчання і 50
- знання і 48–49
- менторство і 52
- розуміння і 52
- спільна робота і 51
- тренування і 50

Тупики 172–174

У

Уникання 163, 172
Управління часом
- безлад і 173–174
- зустрічі і 163–167
- інверсія пріоритетів і 172
- перезарядка і 169
- «помідорний» метод планування 170–171
- приклади 162
- тупики і 172–173
- уникання і 172

Учні 217, 223, 226
Учнівство 223–227

Ф

Фізична активність 170
Функціональність у правилі «Не зашкодь» 41–42

Х

Хибна готовність 108–109

Ц

Цілі 41, 59–61, 94, 136, 142, 159, 173, 194, 202, 213, 222, 225

Ч

Час виконання практика і 126, 127
Читання 100

Ш

Широкосмуговий дельфійський метод 185

C

Coding Dojo 128
Cucumber 109, 143, 157, 244
CVS 234

E

Emacs 237–239

F

FitNesse 44, 109, 117, 126, 141, 143, 157, 234–236, 241, 243–244, 248

G

git 234–237, 248
Green Pepper 244

I

IDE/редактор 237–238, 241–242
IntelliJ 238–239, 242, 248

J

JBehave 157, 244

M

MDA 245, 247–248

P

PERT 182, 185

R

RobotFX 109, 244

S

SVN 234

T

TDD 44, 49, 97, 99, 104, 114–118, 120, 125, 127–129, 155, 160, 195, 226
TextMate 239, 242

U

UML 49, 245, 247

V

Vi 231, 237, 239

Переклад з англійської
ГАННИ ЯКУБОВСЬКОЇ

Видавництво «Фабула» є складовою
видавничої групи «Ранок»

Науково-популярне видання
Роберт Мартін
ЧИСТИЙ КОДЕР:
Кодекс поведінки для професійних розробників

Robert C. Martin
THE CLEAN CODER:
A Code of Conduct for Professional Programmers

Дизайн обкладинки *І. І. Нестеренко*
Головний редактор *Т. О. Попова*
Науковий редактор *К. Ю. Горбушко*
Редактор *А. А. Клімов*
Технічний редактор *Т. Г. Орел*
Коректор *Н. В. Красна*
ФБ1335006У. Підписано до друку 10.01.2023.
Формат 70 × 100/16. Папір офсетний.
Гарнітура Minion. Друк офсетний.
Ум. друк. арк. 20,64.
ТОВ ВИДАВНИЧИЙ ДІМ «ФАБУЛА»
Свідоцтво ДК № 7005 від 12.12.2019.
вул. Котельниківська, 5, м. Харків, 61071.
Для листів: вул. Космічна, 21а, м. Харків, 61145.
e-mail: info@fabulabook.com
Тел. (057) 717-61-80,
тел./факс (057) 719-58-67.
Надруковано у ПП «Юнісофт»
UNISOFT
вул. Морозова, 13 б, м. Харків, 61036
www.unisoft.ua
Свідоцтво ДК №5747 від 06.11.2017 р.
Наклад 2100 прим. Замовлення № 303/09.

З питань реалізації звертайтесь: trade@fabulabook.com.

Замовити книжку або залишити видгук

Інтернет-книгарня видавництва «ФАБУЛА»