РОБЕРТ МАРТІН

Чистий код

СТВОРЕННЯ І РЕФАКТОРИНГ ЗА ДОПОМОГОЮ AGILE

ВИДАВНИЦТВО
ФАБУЛА
#PRO

УДК 004.41
М25

ВИДАВНИЦТВО
ФАБУЛА
#PRO

Авторизований переклад з англомовного видання під назвою
CLEAN CODE: A HANDBOOK OF AGILE SOFTWARE CRAFTSMANSHIP,
1-ше видання, автора Роберта Мартіна, опублікованого Pearson Education, Inc,
що виступає як Prentice Hall

Мартін Роберт

М25 Чистий код: створення і рефакторинг за допомогою Agile / пер. з англ. І. Бондар-Терещенко. — Харків : Вид-во «Ранок» : Фабула, 2019. — 448 с.

ISBN 978-617-09-5285-1

Роберт Мартін, також відомий як дядечко Боб,— знакова постать у світі розробки ПЗ, блискучий професіонал, міжнародний консультант, один із тих, хто створив у 2001 році всесвітньо відомий Agile-маніфест. «Чистий код» — мабуть, найвідоміша й найпопулярніша книжка цього автора.

Навіть поганий програмний код може працювати. Однак якщо він не є «чистим», це заважатиме розвитку проєкту, відтягуючи значні ресурси на його підтримку та «приборкання». Книжка Р. Мартіна присвячена саме гарному програмуванню і насичена реальними прикладами коду. Із нею ви навчитеся відрізняти хороший код від поганого, дізнаєтеся, як писати гарний код і як перетворити поганий код на по-справжньому чистий. Автор наводить принципи, патерни і прийоми написання чистого коду, практичні сценарії зростаючої складності, перелік евристичних правил і «запахів коду». Загалом це першокласна база знань і прикладів, що описують хід нашої думки в процесі читання, написання та чищення коду.

УДК 004.41

ISBN 978-617-09-5285-1

Зміст

Присвячується Анні-Марії —
вічному коханню всього мого життя

ПЕРЕДМОВА

У Данії дуже популярні льодяники *Ga-Jol*. Їх сильний лакричний смак добре пасує до нашої сирої і зазвичай холодної погоди. Однак нас, данців, льодяники *Ga-Jol* приваблюють ще й мудрими або дотепними висловами, надрукованими на кришці кожної коробочки. Сьогодні вранці я придбав дві такі коробочки і виявив на них стару данську приказку: «*Ærlighed i små ting er ikke nogen lille ting*», тобто «Чесність у дрібницях — зовсім не дрібничка». Це було добрим знаком, який цілком відповідав тому, про що я збирався написати в передмові. Дрібниці важливі, оскільки ця книжка присвячена речам простим, але зовсім не малозначущим.

Бог ховається в дрібницях, зауважив архітектор Людвіг Міс ван дер Рое. Ця цитата нагадує про нещодавні дебати про роль архітектури в розробленні програмного забезпечення — і особливо у світі *Agile*. Ми з Бобом час від часу захоплено приєднуємося до цього діалогу. Насправді, Міс ван дер Рое виявляв увагу і до зручності, і до непідвладних часу будівельних форм, що лежать в основі великої архітектури. З іншого боку, він також особисто вибирав кожну дверну ручку для кожного спроєктованого ним будинку. Чому? Та тому ж, що дрібниці важливі.

У наших з Бобом постійних «дебатах» про *TDD* з'ясувалося, що ми згодні з тим, що архітектура відіграє важливу роль під час розроблення ПЗ, хоча ми порізному дивимося на те, яке значення має це твердження. Утім, ці розбіжності доволі несуттєві, оскільки ми вважаємо цілком очевидним, що відповідальні професіонали приділяють деякий час обдумуванню й плануванню проєкту. Оприлюднені наприкінці 1990-х концепції проєктування, що залежать тільки від тестів і коду, давно відкинуті. Проте увага до дрібниць є більш важливим аспектом професіоналізму, ніж будь-які грандіозні плани. По-перше, завдяки вправлянню з дрібницями професіонали здобувають кваліфікацію та репутацію для участі в масштабних проєктах. По-друге, навіть найдрібніший прояв недбалості під час будівництва — наприклад, двері, які нещільно зачиняються, криво покладена плитка на підлозі або навіть захаращений стіл,— геть знищує чарівність усієї споруди. Щоб цього не відбувалося з вашими програмами, код має бути чистим.

Утім, архітектура — всього лише одна з метафор для розроблення програмних продуктів. Вона найкраще підходить для проєктів, у яких продукт «зводиться» в тому ж сенсі, у якому архітектор зводить будову. В епоху *Scrum* і *Agile*

основну увагу приділяють швидкому виведенню продукту на ринок. Фабрики, що виробляють програмні продукти, повинні працювати на максимальній швидкості. Однак цими «фабриками» є живі люди — програмісти, які працюють над користувацькими історіями або беклогом для створення нових продуктів. Метафора виробництва зараз як ніколи актуальна для їхнього світогляду. Скажімо, фреймворк *Scrum* багато в чому надихали виробничі аспекти японського автобудування з його конвеєрами.

Але навіть в автобудуванні основна частина роботи пов'язана не з виробництвом, а із супроводом продуктів — або його відсутністю. У програмуванні 80 % того, що ми робимо, теж витончено називається «супроводом». Насправді мова йде про полагодження. Наша робота ближче до роботи домашніх майстрів-умільців або автомеханіків. А що японська теорія управління каже з цього приводу?

У 1951 році в японській промисловості з'явився підхід до підвищення якості, що називалася *TPM* (*Total Productive Maintenance*). Вона була орієнтована насамперед на супровід, а не на виробництво. Доктрина *TPM* базувалася на так званих «принципах 5*S*». По суті, ті 5*S* були звичайним набором життєвих правил. До речі, вони також покладені в основу концепції *Lean* — іншої модної на Заході течії ощадливого виробництва, що набирає ваги і в програмних колах. Як зазначає дядечко Боб у своєму вступі, гарна практика програмування вимагає таких якостей, як зосередженість, наявність духу і дисципліна мислення. Проблеми не завжди розв'язують за допомогою максимального завантаження обладнання з метою підтримання оптимальних темпів.

Отже, загалом філософія 5*S* базується на таких засадах:

- **Сейрі, або організація.** Абсолютно необхідно знати, де що перебуває — і в цьому допомагають такі методи, як грамотний вибір імен. Думаєте, вибір імен ідентифікаторів не важливий? Зазирніть у наступні глави.
- **Сейтон, або акуратність.** Стара американська приказка каже: всьому своє місце — і все опиняється на своїх місцях. Фрагмент коду має бути там, де читач коду очікує його знайти. Якщо ж він десь в іншому місці, переробіть свій код і розмістіть його там, де йому належить бути.
- **Сейсо, або чищення.** Робоче місце має бути вільним від обвислих дротів, бруду, сміття і мотлоху. А що кажуть автори цієї книжки про захаращення коду коментарями й закоментованими рядками коду? Вони радять їх негайно позбутися.
- **Сейкецу, або стандартизація:** команда має досягти згоди щодо того, як підтримувати чистоту на робочому місці. А що в цій книжці сказано про наявність єдиного стилю кодування й набору командних правил? Звідки беруться ці стандарти? Прочитайте — дізнаєтеся.
- **Сюцуке, або дисципліна.** Програміст має бути достатньо дисциплінованим, щоб підкорятися правилам; він має постійно розмірковувати про свою роботу й бути готовим до змін.

Якщо ви не пошкодуєте зусиль — так, саме зусиль! — щоб прочитати і засвоїти цю книжку, то зрозумієте вагомість останнього пункту.

Нарешті ми наблизилися до підвалин відповідального професіоналізму в професії, що неодмінно змушена цікавитися життєвим циклом створеного продукту. У випадку супроводу автомобілів та інших виробів, за правилами *TPM*, аварійний ремонт (аналог прояву помилки, якої припустилися на стадії виробництва) вважають винятком. Замість цього треба щоденно оглядати машини й замінювати зношені частини ще до того, як вони зламаються, або виконувати правила, аналогічні «заміні масла кожні 10 тисяч миль» задля запобігання спрацюванню. Тобто маємо безжально переробляти створений нами код. А ще можна зробити наступний крок, який вважався новаторським в *TPM* півстоліття тому: почати будувати машини, заздалегідь орієнтовані на зручний супровід. Отже, ваш код повинен не тільки працювати, але й добре читатися. Як вчить нас Фред Брукс[1], великі блоки програмного коду варто переписувати «з нуля» щосім років або близько того, щоб вони «не обростали мохом». Але, можливо, цю давню константу Брукса варто звести до тижнів, днів і навіть годин — замість років. Саме на цьому рівні «живуть» дрібниці.

У дрібницях ховається величезна сила, але сам по собі такий підхід до життя здається скромним й ґрунтовним, як зазвичай очікують від будь-якого методу з японським корінням. Утім, такий погляд на життя не є суто східним; у західному фольклорі можна знайти чимало подібних настанов. Так, принцип, наведений в описі сейтон, належить політичному діячу з Огайо, який завжди вважав: «акуратність є засобом від будь-якого зла». А як щодо сейсо? Чистота — атрибут божественності. Хоч би яким гарним був будинок, захаращений стіл зіпсує будь-яке враження. А що кажуть про сюцуке? Той, хто вірний у дрібницях, вірний у всьому. А прагнення до перероблення коду, зміцнення власних позицій для подальших серйозних рішень замість того, щоб відкладати перероблення «на потім»? Хто рано встає — у того й є. Або: не відкладай на завтра те, що можна зробити сьогодні. (Фраза «останній відповідальний момент» в методології *Lean* мала саме такий сенс, аж поки не потрапила до консультантів із розроблення ПО). То що ж нам відомо про роль індивідуальних зусиль у загальній картині? Із маленьких жолудів виростають величезні дуби. І дещо про інтеграцію простої праці, що має профілактичне значення, у повсякденне життя: хто яблуко щодня з'їдає, у того лікар не буває. Або: дорога ложка до обіду. Чистий код із повагою ставиться до глибинного коріння мудрості, що лежить в основі нашої культури — такої, якою вона колись була або мала бути і може стати за належної уваги до дрібниць.

Навіть в літературі з архітектури ми знаходимо фрази, що повертають нас до важливої ролі дрібниць. Згадайте дверні ручки Міса ван дер Рое — це ж сейрі в чистому вигляді. Увага до імені кожної змінної. Ім'я змінної мають вибирати так само ретельно, як ім'я новонародженого.

---

[1] Фредерік Філіпс Брукс (1931) — учений-інформатик, найбільш відомий через управління розробленням ОС *System*/360 для *IBM*. Нагороджений Національною медаллю технологій та інновацій США (1985) та премією Тюрінга (1999).

Як відомо будь-якому домовласникові, турбота і безперервне прагнення до поліпшення ніколи не добігають кінця. Архітектор Крістофер Александер — батько патернів і мови їх опису, що істотно вплинула на екстремальне програмування — розглядав кожен акт проєктування як маленький локальний акт відновлення. Із його точки зору, майстерність на рівні тонкої структури є єдиним сенсом архітектури; більші форми можна залишити на відкуп патернам, а їхнє застосування — на відкуп мешканцям. Проєктування триває не тільки під час прибудови до будівлі нових кімнат, але й разом із фарбуванням стін, заміною старих килимів або кухонної раковини. Аналогічні принципи діють у багатьох видах мистецтва. У пошуках інших майстрів, які вважали, що Бог живе в дрібницях, ми опиняємося у славетній компанії французького письменника XIX століття Гюстава Флобера. Французький поет і теоретик мистецтва Поль Валері також казав, що вірш ніколи не буває закінченим, оскільки вимагає постійного переробляння, а припинити роботу над ним — означає відкинути його. Така зосереджена увага до дрібниць характерна для всіх справжніх творців. Можливо, принципово нового тут не так вже й багато, але ця книжка нагадає вам про необхідність дотримання життєвих правил, що їх давно закинули через байдужість або прагнення до стихійності і банальної «готовності до змін».

На жаль, згадані тут аспекти рідко розглядають як наріжні камені у мистецтві програмування. Ми швидко залишаємо свій код — і не тому, що він ідеальний, а тому, що наша система цінностей зосереджена на зовнішньому вигляді, а не на внутрішній сутності того, що ми робимо. Недбалість зрештою обходиться недешево: фальшива монета завжди повертається до свого власника. Дослідження — як галузеві, так і академічні — зазвичай оминають скромну галузь підтримування чистоти коду. У ті часи, коли я працював в дослідницькій організації з виробництва ПЗ *Bell Labs*, у ході досліджень з'ясувалося, що послідовний стиль застосування відступів є однією з найбільш статистично значущих ознак невеликої кількості помилок. Ми ж хочемо, щоб ознакою якості була архітектура, мова програмування або щось інше, настільки ж важливе й вагоме. Нас, як людей, чий професіоналізм обумовлений майстерним володінням інструментами і методами створення ПЗ, ображає сама ідея, що просте послідовне застосування відступів може мати таку цінність. Посилаючись на свою книжку, що побачила світ майже два десятиріччя тому, скажу, що саме такий стиль відрізняє досконалість від звичайної компетентності. Японський світогляд недарма усвідомив критичну важливість кожного рядового співробітника, а головне — систем розроблення, які існують завдяки простим повсякденним діям цих співробітників. Якість виникає внаслідок мільйонів проявів небайдужого ставлення до справи, а не завдяки застосуванню якогось величного методу, що спустився з небес. Простота цих дій не означає їх примітивності й не свідчить про їхню легкість. Проте з них виникає справжня велич і краса будь-якого людського починання. Забути про це — значить не бути людиною повною мірою.

Звичайно, я, як і раніше, виступаю за широту мислення й цінність архітектурних підходів, що мають коріння в глибокому знанні предметної галузі та зручності використання програмних продуктів. Ця книжка написана не про це,

або принаймні у ній цю тему безпосередньо не розглянуто. Але її глибину не варто недооцінювати, оскільки вона відповідає поточному світогляду справжніх програмістів — таких як Пітер Соммерлад, Кевлін Хенні і Джованні Аспроні. «Код є архітектура» і «простий код» — ось як звучать їхні головні мантри. Хоча ми не маємо забувати, що інтерфейс і є програма і що його структурні елементи несуть багато інформації щодо структури програми, дуже важливо пам'ятати, що архітектура живе в коді. І якщо перероблення у виробничій метафорі призводить до витрат, перероблення в архітектурній метафорі веде до підвищення цінності. Розглядайте свій код як гарне втілення благородних зусиль із проєктування як процесу, а не як статичної кінцевої точки. Архітектурні метрики прив'язки і зв'язності проявляються саме в коді. Якщо ви послухаєте, як Ларрі Константайн описує прив'язку та зв'язність, то переконаєтеся, що він говорить про них в контексті коду, а не якихось величних, але абстрактних концепцій, що зазвичай трапляються в *UML* — уніфікованій мові моделювання. Річард Гебріел в своєму есе *Abstraction Descant* стверджував, що абстракція — зло. Так ось, код — це антизло, а чистий код, найімовірніше, має божественну природу.

Повертаючись до свого прикладу з коробочкою льодяників *Ga-Jol*, хочу підкреслити один важливий момент: данська народна мудрість рекомендує нам не тільки звертати увагу на дрібниці, але й бути чесними в дрібницях. Це означає чесність у коді, чесність із колегами і, що найважливіше,— чесність перед самим собою з приводу стану вашого коду. Чи справді ми зробили все можливе для того, щоб «залишити місце табору чистішим, ніж воно було до нас»? Чи переробили свій код перед тим, як здати його? Ці речі — справжнє осердя системи цінностей *Agile*. Одна із рекомендованих практик в *Scrum* — включати рефакторинг в критерії готовності. Ні архітектура, ні чистий код не вимагають від нас досконалості — просто будьте чесними і робіть все, на що здатні. Людині властиво помилятися; небесам властиво прощати. У *Scrum* все таємне стає явним. Ми виставляємо напоказ свою брудну білизну. Ми чесно демонструємо стан нашого коду, але ж код ніколи не буває ідеальним. І через це стаємо більш людяними й наближаємося до величі в дрібницях.

У нашій професії нам потрібна вся допомога, яку ми зможемо дістати. Якщо чиста підлога в магазині скорочує вірогідність нещасних випадків, а ретельно розташовані інструменти підвищують продуктивність, то я обома руками «за». Що ж стосується цієї книжки, то вона є найкращим практичним застосуванням принципів *Lean* у галузі розроблення ПЗ, яке я коли-небудь бачив у друкованому вигляді. Утім, іншого я й не очікував від цієї невеличкої групи мислячих особистостей, які протягом багатьох років прагнуть не тільки дізнатися щось нове, але й діляться своїми знаннями в книжках, одну з яких ви зараз тримаєте в руках. Світ став досконалішим, ніж був до того моменту, коли дядечко Боб надіслав мені рукопис.

Завершуючи ці пишномовні міркування, я вирушаю наводити лад на своєму робочому столі.

Джеймс О. Коплін

Мьоруп, Данія

ВСТУП

Відтворено з люб'язного дозволу Тома Голверда (http://www.osnews.com/story/19266/WTFs_m).

Які з двох дверей відповідають вашому коду? Які двері найчастіше трапляються у вашій команді або компанії? Чому ви потрапили саме в цю кімнату? Там триває нормальний перегляд коду або одразу ж після випуску програми зійшла лавина жахливих помилок? Ви панічно переглядаєте код, який, як вважалося, вже добре працює? Клієнти йдуть від вас юрбою, а начальство дихає в потилицю? Як опинитися за правильними дверима, коли все так погано?

Відповідь: *потрібна майстерність*.

Майстерність має дві складові: знання й практичний досвід. Ви повинні мати уявлення про принципи, патерни, прийоми та евристичні правила, відомі

кожному професіоналу, а також «втерти» ці знання у свої пальці, очі й мозкові звивини шляхом ретельної практики.

Я можу пояснити вам фізику їзди на велосипеді. Справді, класична фізика доволі прямолінійна. Сила тяжіння, сила тертя, ротаційний момент, центр ваги і т. ін.— усе це можна описати рівняннями менш ніж на одній сторінці. Цими формулами я доведу вам, що їзда на велосипеді є можливою, і надам всю необхідну для цього інформацію Але коли ви вперше сядете на велосипед, ви все одно неминуче впадете.

Із програмуванням точнісінько те саме. Звичайно, ми могли б записати всі «гарні» принципи чистого коду, а потім довірити вам всю практичну роботу (іншими словами, дозволити вам гепнутися, видершись на велосипед), але які б з нас тоді були вчителі?

Ні, у цій книжці ми підемо іншим шляхом.

Уміння писати чистий код — *важка праця*. Вона не обмежується знанням патернів і принципів. Над кодом необхідно попрацювати. Необхідно робити спроби й зазнавати поразок. Необхідно спостерігати за тим, як інші намагаються і теж зазнають невдач. Необхідно бачити, як вони спотикаються й повертаються до початку; як болісно приймають рішення та яку ціну доводиться платити за хибний вибір.

Приготуйтеся ґрунтовно попрацювати під час читання. Перед вами не «легке чтиво», яке можна проковтнути в літаку й перегорнути останню сторінку перед посадкою. Книжка змусить вас працювати, а головне — *робити це старанно*. Яка робота на вас чекає? Ви будете читати код — багато коду. І вам доведеться як слід поміркувати над тим, що в цьому коді правильно, а що — ні. Ви будете спостерігати за тим, як ми розбираємо ці модулі, а потім збираємо наново. Це потребуватиме чимало часу і зусиль, але ми вважаємо, що результат того вартий.

Книжка розділена на три частини. У кількох початкових розділах викладено принципи, патерни й прийоми написання чистого коду. Також у них наведений досить солідний обсяг коду, і читати їх буде непросто. Весь цей матеріал підготує вас до другої частини. Якщо ви відкладете книжку після першої частини — бажаємо всього найкращого!

У другій частині книжки доведеться докласти ще більших зусиль. Вона містить декілька практичних сценаріїв, складність яких поступово зростає. Кожен сценарій являє собою вправу з чищення коду, тобто перетворення проблемного коду в код зі значно меншою кількістю проблем. Щоб засвоїти матеріал цієї частини, слід *ґрунтовно попрацювати*. Вам доведеться постійно перемикатися між текстом і лістингами. Вам доведеться аналізувати й розбирати код, з яким ми працюємо, і нарешті усвідомити причину кожної внесеної зміни. Приділіть цьому час, тому що ця робота *забере не один день*.

Третя частина книжки — концентроване вираження її суті. Вона містить один розділ, у якому подано перелік евристичних правил і «запахів коду», зібраних під час аналізу. Під час очищення коду за допомогою практичних сценаріїв ми задокументували причину кожної виконаної дії у вигляді евристичного правила або

«запаху». Ми намагалися зрозуміти нашу власну реакцію на код під час його читання і зміни, а також прагнули пояснити, чому ми відчували те, що відчували, або робили те, що робили. Результат являє собою базу знань, що описує шлях нашого мислення в процесі читання, написання та очищення коду.

Утім, без глибокого засвоєння всіх практичних сценаріїв із другої частини книжки користі від цієї бази знань буде небагато. У цих сценаріях ми ретельно позначили кожну внесену зміну з допомогою посилань на відповідне евристичне правило. Посилання подані у квадратних дужках і мають приблизно такий вигляд: [H22]. Це дає змогу читачеві бачити контекст, у якому застосовують евристики. Але головна цінність полягає навіть не у власне евристиках, а *у зв'язках між ними і конкретними рішеннями, прийнятими в ході чищення коду за практичними сценаріями*.

Щоб допомогти вам відстежувати ці зв'язки, ми розмістили наприкінці книжки список перехресних посилань. У ньому наведено номери сторінок усіх посилань. За допомогою цього списку можна знайти всі контексти, де застосовували ту чи іншу евристику.

Якщо ви прочитаєте першу й третю частину, пропустивши аналіз практичних сценаріїв,— вважайте, що ви прочитали ще одну «легку» книжку про написання якісного коду. Але якщо ви витратите час на опрацювання всіх сценаріїв, простежите за кожним крихітним кроком, за кожним рішенням, якщо ви поставите себе на наше місце й змусите себе думати в тому ж напрямку, то ваше розуміння цих принципів, патернів, прийомів та евристик значно поглибиться. Знання вже не будуть «поверховими». Вони проникнуть у ваші пальці, очі і мозок. Вони стануть частиною вашої особистості — як велосипед стає продовженням вашого тіла, коли ви добре навчилися на ньому їздити.

Подяки

Я дякую двом художникам — Дженніфер Конке й Анджелі Брукс. Дженніфер створила чудові дотепні рисунки на початку кожного розділу, а також намалювала портрети Кента Бека, Ворда Каннінгема, Б'ярна Страуструпа, Рона Джеффріса, Греді Буча, Дейва Томаса, Майкла Фізерса — і мене.

Анджела робила рисунки, що пояснюють матеріал розділів. У минулі роки вона підготувала чимало ілюстрацій для моїх книжок, зокрема для книжки *Agile Software Develpment*: *Principles*, *Patterns*, *and Practices* (2002). Крім того, вона мій первісток, і я нею пишаюся.

Розділ 1

ЧИСТИЙ КОД

Є дві причини, чому ви читаєте цю книжку. По-перше, ви програміст. По-друге, ви хочете підвищити свою кваліфікацію. Чудово. Гарних програмістів завжди не вистачає.

Ця книжка присвячена гарному програмуванню. Вона сповнена реальних прикладів коду. Ми будемо розглядати код з усіх можливих напрямків: згори вниз, знизу вгору і навіть ізсередини. Ви дізнаєтеся багато нового про код та навчитеся не тільки відрізняти гарний код від поганого, але й, зрештою, як писати гарний код і перетворювати поганий код на гарний.

Хай живе код

Можливо, хтось скаже, що книжка про код відстала від часу — код зараз вже не такий актуальний; замість нього увагу слід було б звернути на моделі та вимоги. Нам навіть доводилося чути думку, що код як такий незабаром взагалі перестане існувати, його будуть генерувати, а не писати вручну, а програмісти стануть не потрібні, тому що бізнесмени будуть генерувати програми за специфікаціями.

Дурниці! Код ніколи не зникне, тому що він репрезентує деталі вимог. На певному рівні ці деталі неможливо ігнорувати або абстрагуватися від них; їх доводиться визначати. А коли вимоги визначають настільки детально, щоб їх міг виконати комп'ютер, це і є *програмування*. І їхнім визначенням є *код*.

Імовірно, рівень абстракції наших мов продовжить зростати. Я також очікую, що кількість предметно-орієнтованих мов зростатиме. І це добре. Але код через це не перестане існувати. Усі визначення, написані цими високорівневими предметно-орієнтованими мовами, *стануть кодом*! І цей код має бути досить компактним, точним, формальним і докладним, щоб комп'ютер міг зрозуміти і виконати його.

Люди, які вважають, що код колись зникне, нагадують математиків, які сподіваються створити неформальну математичну дисципліну. Вони сподіваються, що колись будуть побудовані машини, які робитимуть те, чого ми хочемо, а не те, що ми наказуємо їм зробити. Такі машини повинні розуміти нас настільки добре, щоб перетворювати набір нечітких побажань на програми, які ідеально виконують і точно відповідають цим побажанням.

Але цього ніколи не станеться. Навіть люди з усією їхньою інтуїцією та винахідливістю не здатні створювати успішні системи на основі туманних уявлень своїх клієнтів. Дисципліна у визначенні вимог до ПЗ нас навчила, що максимально чітко визначені вимоги так само формальні, як сам код, і їх можна використовувати як виконувані тести цього коду!

За своєю суттю код являє собою мову, якою, зрештою, виражають ті чи інші потреби. Ми можемо створювати мови, близькі до реальних потреб. Ми можемо створювати інструменти, що допомагають нам обробляти й збирати ці потреби у формальні структури. Але необхідність в точності ніколи не зникне, а отже, і код залишиться назавжди.

Поганий код

Нещодавно я читав передмову до книжки Кента Бека «*Implementation Patterns*» [*Beck*, 2007]. Автор каже: «...Ця книжка базується на доволі хиткій передумові, що гарний код важливий...» *Хитка* передумова? Не згоден! На мій погляд, ця передумова є однією з найпотужніших, основоположних і багатогранних підвалин нашого ремесла (і я думаю, що Кенту це відомо). Ми знаємо, що гарний код важливий, тому що нам доводилося довго миритися з його відсутністю.

Одна компанія наприкінці 1980-х років написала застосунок-*бестселер*. Застосунок став надзвичайно популярним, багато професіоналів купували й використовували його, але потім випуск нових версій почав відтерміновуватися. Помилки у проміжку між версіями не виправляли. Час завантаження зростав, а збої відбувалися частіше. Пам'ятаю той день, коли я закрив цей продукт і більше ніколи його не запускав. Незабаром ця компанія збанкрутувала.

Через два десятиліття я зустрів одного з працівників тієї компанії і запитав його, що ж сталося. Відповідь підтвердила мої побоювання. Вони поспішали з випуском продукту на ринок і не звертали уваги на якість коду. Із додаванням нових можливостей код дедалі погіршувався, аж поки в якийсь момент не вийшов з-під контролю. *Поганий код призвів до краху компанії.*

Чи заважав колись поганий код *вашій* роботі? Будь-який досвідчений програміст не раз потрапляв у подібну ситуацію. Ми продираємося крізь поганий код. Ми грузнемо в хитросплетінні гілок, потрапляємо в приховані пастки. Ми ледь прокладаємо шлях, сподіваючись отримати хоч якусь підказку щодо того, що ж насправді відбувається в коді, але не бачимо нічого, окрім нових скупчень незрозумілого коду.

Звичайно, поганий код заважав вашій роботі. Чому ж тоді ви його писали? Намагалися якнайшвидше розв'язати задачу? Поспішали? Можливо. А може, вам здавалося, що у вас немає часу, щоб якісно виконати свою роботу, що ваше начальство буде невдоволене, якщо ви витратите час на чищення свого коду. Можливо, ви втомилися працювати над програмою і вам хотілося скоріше позбутися її. Або ви подивилися на список запланованих змін і зрозуміли, що вам необхідно якнайшвидше «прикрутити» той чи інший модуль, щоб перейти до наступного. Таке відбувається скрізь.

Кожен із нас із жахом дивився на хаос, який він щойно створив, і вирішував залишити його на завтра. Кожен із полегшенням бачив, що недолуга програма все ж таки працює, і вирішував, що робоча мішанина — це краще, ніж нічого. І кожен обіцяв собі повернутися й почистити код... колись. Звичайно, у ті дні ми ще не знали закону Леблана: *потім означає ніколи.*

Розплата за хаос

Якщо ви програмуєте більше двох-трьох років, вам напевно доводилося грузнути в чужому — або своєму власному — безладному коді. Уповільнення може виявитися доволі значним.

За якихось рік-два команди, що дуже швидко рухалися вперед на початку проєкту, починають повзти зі швидкістю равлика. Кожна зміна, яку вносять у код, порушує його роботу у двох-трьох інших місцях. Жодна зміна не минає тривіально. Для кожного доповнення або модифікації системи необхідно «тримати в голові» всі хитросплетіння коду, щоб у програмі їх стало ще більше. Згодом плутанина шириться настільки, що впоратися з нею вже не вдається. Виходу просто немає.

У міру накопичення хаосу в коді продуктивність команди починає знижуватися, асимптотично наближаючись до нуля. Дивлячись на зниження продуктивності керівництво робить єдине, що може зробити: підключає до проєкту нових працівників, сподіваючись підвищити продуктивність. Але новачки нічого не розуміють в архітектурі системи. Вони не знають, які зміни узгоджуються з намірами проєктувальника, а які їм суперечать. Більше за це, вони — і всі інші члени команди — опиняються під жахливим тиском із боку начальства. Через поспіх вони працюють все більш недбало, а продуктивність тільки продовжує знижуватися (*рис. 1.1*).

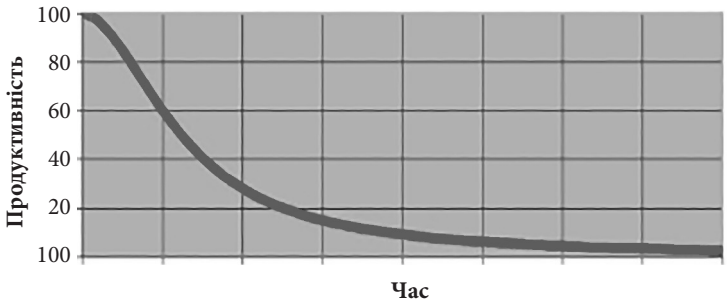

Рис. 1.1. Залежність продуктивності від часу

Грандіозне перероблення

Урешті-решт команда влаштовує бунт. Вона повідомляє керівництву, що не може продовжувати розробляти огидний код, і вимагає перероблення архітектури. Керівництво ж, зі свого боку, не бажає витрачати ресурси на повне перероблення проєкту, але не може заперечувати, що продуктивність просто жахлива. Із часом воно все ж таки піддається на вимоги розробників і дає дозвіл на проведення грандіозного перероблення.

Набирають нову «ударну команду». Усі прагнуть брати в ній участь, тому що проєкт розпочинається «з нуля». Розробники будуватимуть «на порожньому місці», і створять щось воістину прекрасне. Але в «ударну команду» відбирають тільки найкращих і найрозумніших. Усім іншим доводиться супроводжувати попередню систему.

Між двома командами починаються перегони. «Ударна команда» має вибудувати нову систему, яка робитиме те саме, що робила стара. Більше того, вона повинна своєчасно враховувати зміни, що безперервно вносять у стару систему. Адже керівництво не відмовляється від старої системи, допоки нова не буде повністю відтворювати її функціональність.

Такі перегони можуть тривати дуже довго. Мені відомі випадки, коли вони тривали близько десяти років. І до моменту їх завершення виявлялося, що початковий склад давно залишив «ударну команду», а ті, хто зараз у її складі, вимагають переробити нову систему, тому що в ній знову-таки панує справжній хаос.

Якщо ви мали справу хоча б з деякими аспектами історії, яку я зараз розповів, то ви вже знаєте, що підтримання чистоти коду не тільки компенсує витрачений час — воно є питанням професійного виживання.

Відносини

Вам доводилося продиратися крізь настільки заплутаний код, що ви витрачали тижні на те, що мало б тривати кілька годин? Ви бачили, як зміни, що начебто мають вносити в один рядок, доводиться вносити в сотні різних модулів? Ці симптоми стали занадто звичними.

Чому це відбувається з кодом? Чому гарний код так швидко «загниває» й перетворюється на поганий? Зазвичай цьому є купа пояснень. Ми скаржимося на зміни у вимогах, що суперечать вихідній архітектурі. Ми стогнемо через графіки, занадто жорсткі, щоб зробити все, як годиться. Ми брешемо про дурне керівництво, нетерпимість клієнтів і безглуздих типів з відділу маркетингу. Однак вина лежить не на них, а на нас самих. Справа в нашому непрофесіоналізмі.

Можливо, проковтнути цю гірку пігулку непросто. Хіба це ми винні в цьому хаосі? А як же вимоги? Графік? Поганий менеджмент і безглузді типи з відділу маркетингу? Хіба принаймні частина провини не на них?

Ні. Керівництво й маркетологи звертаються до нас за інформацією, на підставі якої вони формують свої обіцянки і зобов'язання; але навіть якщо вони до нас не звертаються, ми повинні відверто казати їм те, що думаємо. Користувачі звертаються до нас, щоб ми висловили свою думку щодо того, наскільки доречні вимоги до системи. Менеджери проєктів звертаються до нас по допомогу в складанні графіка. Ми беремо діяльну участь у плануванні проєкту й несемо значну частку відповідальності за будь-які провали, особливо якщо ці провали обумовлені поганим кодом!

«Але стривайте! — скажете ви.— Якщо я не зроблю те, що каже мій керівник, мене звільнять». Вірогідніше, ні. Зазвичай начальство бажає знати правду, навіть якщо за його поведінкою цього не видно. Керівники хочуть отримати гарний код, навіть якщо вони схиблені на робочому графіку. Вони можуть пристрасно обстоювати графік і вимоги, але це їхня робота. А *ваша* робота — так само пристрасно захищати програмний код.

Заради наочності уявіть, що ви — лікар, а ваш пацієнт вимагає від вас припинити безглузде миття рук під час підготовки до операції, тому що на це йде надто багато часу[1]! Природно, пацієнт тут — це ваш начальник; і все ж лікар повинен навідсіч відмовитися підкорятися його вимогам. Чому? Тому що лікар знає про небезпеку інфікування більше, ніж пацієнт. Було б непрофесійно (а то й злочинно) підкоритися у цьому випадку волі пацієнта.

[1] Коли Ігнац Земмельвейс (1818–1865), видатний угорський акушер-гінеколог, основоположник асептики в медицині, уперше порекомендував лікарям мити руки перед оглядом пацієнтів, його поради були відкинуті на тій підставі, що у лікарів занадто багато роботи й на миття рук бракує часу.

Так, програміст, який підкорюється волі керівника, розуміючи при цьому загрозу створення неякісного коду, проявляє непрофесіоналізм.

Головний парадокс

Програмісти постійно натрапляють на фундаментальний парадокс. Кожен розробник, який має хоч скільки-небудь значний досвід роботи, знає, що попередній безлад уповільнює його роботу. Але при цьому всі розробники, відчуваючи тиск «згори», створюють безлад у своєму коді заради дотримання графіка виконання робіт. Коротше кажучи, їм бракує часу, щоб працювати швидко!

Справжні професіонали знають, що неможливо дотримувати графіка, влаштовуючи безлад. Насправді цей безлад негайно уповільнить вашу роботу, і графік буде зірвано. Єдиний спосіб дотримувати графіка — і єдиний спосіб працювати швидко — полягає в тому, щоби постійно підтримувати чистоту в коді.

Мистецтво чистого коду?

Припустімо, ви погодилися з тим, що безлад в коді уповільнює вашу роботу. Припустімо, ви погодилися, що для швидкої роботи необхідно дотримувати чистоти. Тоді ви повинні запитати себе: «А як мені написати чистий код?» Марно намагатися написати чистий код, якщо ви не знаєте, що це таке!

На жаль, написання чистого коду має багато спільного із живописом. Зазвичай ми здатні відрізнити хорошу картину від поганої, але це ще не означає, що ми вміємо малювати. Так, уміння відрізняти чистий код від брудного ще не означає, що ви вмієте писати чистий код!

Щоб написати чистий код, необхідно свідомо застосовувати безліч прийомів, керуючись набутим ретельною працею почуттям «чистоти». Ключову роль тут відіграє «відчуття коду». Одні з цим відчуттям народжуються. Інші працюють, щоб розвинути його. Це відчуття не тільки дозволяє відрізнити гарний код від поганого, але й формує стратегію застосування наших навичок для перетворення поганого коду в чистий.

Програміст без «відчуття коду» дивиться на брудний модуль і розпізнає безлад, але й гадки не має, що з ним робити. Програміст *із* «відчуттям коду» дивиться на брудний модуль — і бачить різні варіанти та можливості.

«Відчуття коду» допоможе йому вибрати кращий варіант і спланувати послідовність перетворень, що зберігають поведінку програми і приводять до потрібного результату.

Інакше кажучи, програміст, який пише чистий код,— це справжній художник, що проводить порожній екран крізь серію перетворень, аж поки той не перетвориться на елегантно запрограмовану систему.

Що таке «чистий код»?

Імовірно, скільки існує програмістів, стільки знайдеться й визначень. Тому я спитав деяких відомих і надзвичайно досвідчених програмістів, що вони думають з цього приводу.

Бйорн Страуструп, творець C++ і автор книжки «The C++ Programming Language»:
«Я люблю, щоб мій код був елегантним та ефективним. Логіка має бути достатньо прямолінійною, щоб помилкам було важко сховатися; залежності — мінімальними, щоб спростити супровід; оброблення помилок — повним відповідно до виробленої стратегії; а продуктивність — близькою до оптимальної, щоб не спокушати людей забруднювати код безпринципними оптимізаціями. Чистий код добре розв'язує одну задачу».

Бйорн використовує слово «елегантний». Гарне слово! Словник в моєму *MacBook* видає такі визначення: *той, що дарує задоволення своєю витонченістю та стилем; поєднує простоту з винахідливістю*. Зверніть увагу на зворот «дарує задоволення».

Очевидно, Бйорн вважає, що чистий код *приємно* читати. Під час читання чистого коду ви посміхаєтеся, немовби милуєтесь майстерно зробленою музичною скринькою або добре сконструйованою машиною.

Бйорн також згадує про ефективність, до того ж *двічі*. Напевно, нікого не здивують ці слова, сказані винахідником C++, але я думаю, що тут криється щось більше, ніж звичайне прагнення швидкості. Марні витрати процесорного часу неелегантні, вони не радують око. Також зверніть увагу на дієслово «спокушати», яким Бйорн описує наслідки неелегантності. У цьому криється глибока істина. Поганий код справді *спокушає*, збільшуючи безлад! Коли інші програмісти змінюють поганий код, вони зазвичай роблять його ще гіршим.

Прагматичні Дейв Томас і Енді Хант висловили таку ж думку трохи інакше. Вони порівняли поганий код із розбитими вікнами[1]. Будівля з розбитими вікнами має такий вигляд, наче вона нікого не обходить. Тому люди теж перестають звертати на неї увагу. Вони байдуже дивляться, як з'являються нові розбиті вікна, а з часом починають і самі їх бити. Вони спотворюють фасад написами і влаштовують усередині сміттєзвалище. Одна розбита шибка стає початком занепаду.

Бйорн також згадує про необхідність повного оброблення помилок. Це один із проявів уваги до дрібниць. Спрощене оброблення помилок — всього лише одна з галузей, у яких програмісти нехтують дрібницями. Витоки пам'яті — друга галузь, стан перегонів — третя, непослідовний вибір імен — четверта... Суть у тому, що чистий код приділяє пильну увагу всім дрібницям.

На завершення Бйорн говорить про те, що чистий код добре розв'язує одну задачу. Багато підходів до створення програмного забезпечення не випадково зводяться до цієї простої настанови. Автори книжок із цієї тематики один за одним також прагнуть донести цю думку. Поганий код намагається зробити

[1] http://www.pragmaticprogrammer.com/booksellers/2004-12.html.

занадто багато всього, для нього характерні неясні наміри і неоднозначність цілей. Для чистого коду характерна *цілеспрямованість*. Кожна функція, кожен клас, кожен модуль фокусуються на конкретній меті, не відвертаються від неї і не забруднюються зайвими подробицями.

Греді Буч, автор книжки «Object Oriented Analysis and Design with Applications»:
«Чистий код простий і прямолінійний. Чистий код читається, як добре написана проза. Чистий код ніколи не затьмарює наміри проєктувальника; він сповнений чітких абстракцій і простих ліній передання управління».

Греді каже майже те саме, що й Бйорн, але з точки зору *зручності читання*. Мені особливо подобається його зауваження про те, що чистий код має читатися, як добре написана проза. Згадайте якусь гарну книжку. Згадайте, як слова під час читання начебто зникали, замінюючись зоровими образами! Справжнє кіно, так? Ні, краще! Ви немовби бачили персонажів, чули звуки, відчували збентеження і співпереживали героям.

Звичайно, читання чистого коду аж ніяк не порівнянне з читанням «Володаря кілець». І все ж літературна метафора в цьому випадку цілком доречна. Чистий код, як і гарна повість, має наочно розкрити інтригу розв'язуваної задачі. Він повинен довести цю інтригу до вищої точки, щоб читач нарешті вигукнув: «Ага! Ну звичайно!» — коли всі питання і суперечності нарешті дадуть у фіналі в очевидне рішення.

На мій погляд, вислів «чітка абстракція», використаний Греді, являє собою чарівний оксюморон! Зрештою, слово «чіткий» майже завжди є синонімом для слова «конкретний». У словнику мого *MacBook* наведено таке визначення слова «чіткий»: *короткий, рішучий, фактичний, без вагань чи зайвих подробиць*. Незважаючи на удавану смислову суперечність, ці слова несуть потужне інформаційне повідомлення. Наш код має бути фактичним, а не умоглядним. Він повинен містити тільки те, що необхідно. Читач має побачити за кодом нашу рішучість.

«Великий» Дейв Томас, засновник OTI, хрещений батько стратегії ECLIPSE:
«Чистий код можуть читати і удосконалювати інші розробники, крім його вихідного автора. Для цього написані модульні та приймальні тести. У чистому коді використовують змістовні імена. Для виконання однієї операції в ньому використовують один шлях (замість декількох різних). Чистий код має мінімальні залежності,

що явно визначені, і чіткий, мінімальний API. Код повинен бути грамотним, тому що, залежно від мови, не вся необхідна інформація може бути чітко виражена в самому коді».

«Великий» Дейв поділяє прагнення Греді до зручного читання, але з однією важливою особливістю. Дейв стверджує, що чистота коду спрощує його доопрацювання *іншими* людьми.

На позір це твердження здається очевидним, але його важливість важко переоцінити. Зрештою код, який легко читається, і код, який легко змінюється,— не одне й те саме.

Дейв пов'язує чистоту з тестами! Десять років тому це спричинило б безліч здивованих поглядів. Однак дисципліна розроблення через тестування справила величезний вплив на нашу галузь і стала однією з найбільш фундаментальних дисциплін. Дейв має рацію. Код без тестів не можна вважати чистим, хоч би яким елегантним він був і хоч би як добре читався.

Дейв використовує слово «*мінімальний*» двічі. Очевидно, він віддає перевагу компактному коду перед об'ємним кодом. Направді це положення постійно повторюється в літературі з програмування від початку її існування. Що менше — то краще.

Дейв також каже, що код має бути *грамотним*. Це ненав'язливе посилання на концепцію «*грамотного програмування*» Дональда Кнута [*Knuth*, 1992]. Отже, код має бути написаний у такій формі, щоб його було легко читати фахівцям.

Майкл Фізерс, автор книжки «Working Effectively with Legacy Code»:

«Я міг би перерахувати всі ознаки, властиві чистому коду, але існує одна найважливіша ознака, з якої випливають усі інші. Чистий код завжди має такий вигляд, наче його автор ретельно над ним попрацював. Ви не знайдете жодних очевидних можливостей для його поліпшення. Усі вони вже враховані автором коду. Спробувавши уявити можливі вдосконалення, ви знову прийдете до того, із чого все почалося: ви розглядаєте код, ретельно продуманий і написаний майстром, небайдужим до свого ремесла».

Лише одне слово: ретельно. Насправді воно є головною темою цієї книжки. Можливо, варто було додати до її назви підзаголовок: «*Як дбайливо працювати над кодом*».

Майкл влучив в десятку. Чистий код — це код, над яким ретельно попрацювали. Хтось не пошкодував часу, щоб зробити його простим і чітким. Хтось приділив належну увагу всім дрібницям і поставився до коду з душею.

Рон Джеффріс, автор книжок «Extreme Programming Installed» та «Extreme Programming Adventures In C#».

Кар'єра Рона почалася з програмування мовою *Fortran*. Відтоді він писав код практично всіма мовами і на всіх комп'ютерах. До його слів варто дослухатися.

«Останніми роками я постійно керуюся "правилами простого коду", сформульованими Беком.

Перш за все, простий код:

- проходить всі тести;
- не містить дублікатів;
- висловлює всі концепції проєктування, закладені в систему;
- містить мінімальну кількість сутностей: класів, методів, функцій і т. ін.

Із усіх правил я приділяю головну увагу дублюванню. Якщо щось роблять у програмі знов і знов, це свідчить про те, що якась концепція не здобула втілення в коді. Я намагаюся зрозуміти, що це таке, а потім намагаюся висловити ідею більш чітко.

Виразність для мене, перш за все, означає змістовність імен. Зазвичай я виконую перейменування по кілька разів, аж поки не зупинюся на остаточному варіанті. У сучасних середовищах програмування — таких як *Eclipse* — перейменування здійснювати легко, тому зміни мене не турбують. Утім, виразність не обмежується самими лише іменами. Я також дивлюся, чи не виконує об'єкт або метод більше однієї операції. Якщо це об'єкт, то його, ймовірно, варто розбити на два й більше об'єктів. Якщо це метод, я завжди застосовую до нього прийом "вилучення методу"; у результаті в мене залишається основний метод, який більш чітко пояснює, що саме він робить, і декілька підметодів, що пояснюють, як він це робить.

Відсутність дублювання й виразності є найважливішими складовими чистого коду в моєму розумінні. Навіть якщо під час поліпшення брудного коду ви будете керуватися тільки цими двома цілями, різниця в якості коду може виявитися величезною. Однак існує ще одна мета, про яку я також постійно пам'ятаю, хоча пояснити її буде трохи складніше.

Внаслідок багаторічної праці мені здається, що всі програми містять схожі елементи. Для прикладу візьмімо операцію "знайти елемент в колекції". Незалежно від того, чи працюємо ми з базою даних, що містить інформацію про працівників, чи з хеш-таблицею з парами "ключ—значення", чи з масивом з однотипними об'єктами, на практиці часто виникає завдання витягти конкретний елемент із цієї колекції. У таких ситуаціях я часто інкапсулюю конкретну реалізацію в більш абстрактному методі або класі. Це надає кілька цікавих можливостей.