РОБЕРТ МАРТІН

Чиста архітектура

МИСТЕЦТВО РОЗРОБЛЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

ВИДАВНИЦТВО
ФАБУЛА
#PRO

УДК 004.41
М29

ВИДАВНИЦТВО
ФАБУЛА
#PRO

Авторизований переклад з англомовного видання під назвою CLEAN ARCHITECTURE: A CRAFTSMAN'S GUIDE TO SOFTWARE STRUCTURE AND DESIGN, 1-ше видання, автора Роберта Мартіна, опублікованого Pearson Education, Inc, що виступає як Prentice Hall.

Мартін Роберт

М29 Чиста архітектура: Мистецтво розроблення програмного забезпечення / пер. з англ. І. Бондар-Терещенко. — Харків : Вид-во «Ранок» : Фабула, 2019. — 368 с.

ISBN 978-617-09-5286-8

Роберт Мартін — визнаний авторитет у галузі комп'ютерної архітектури, без перебільшення геній і чудовий оповідач. Спираючись на багаторічний досвід і використовуючи цікаві приклади з власної (і не лише) практики, автор розповідає про базові принципи створення програмного забезпечення.

Книжка в першу чергу адресована програмістам і програмним архітекторам, але буде корисною будь-кому, хто цікавиться сучасними комп'ютерними технологіями.

УДК 004.41

ISBN 978-617-09-5286-8

Присвячується моїй коханій дружині
та моїм чотирьом чудовим дітям,
їхнім сім'ям, включаючи п'ятьох онуків,—
радостям мого життя.

Зміст

Частина V. АРХІТЕКТУРА

Частина VI. ДЕТАЛІ

Частина VII. ДОДАТОК

Передмова

Про що йдеться, коли ми обговорюємо архітектуру?

Подібно до того, як будь-яка інша метафора, опис програмного забезпечення з точки зору архітектури може щось приховати, а щось, навпаки, проявити; може обіцяти більше, ніж давати, і давати більше, ніж обіцяти.

Очевидна привабливість архітектури — це структура. А структура — це те, що домінує над парадигмами і суперечками у світі розробки програмного забезпечення — компонентами, класами, функціями, модулями, шарами і службами, мікро- або макро-. Але макроструктура багатьох програмних систем часто нехтує переконаннями або розумінням — організація радянських підприємств, неймовірні вежі Дженга, що сягають хмар, археологічні шари, що залягають у гірській породі. Структура програмного забезпечення не завжди інтуїтивно очевидна, як структура будівель.

Будинки мають очевидну фізичну структуру, що не залежить від матеріалу, з якого вони побудовані, від їхньої висоти або ширини, від їхнього призначення, від наявності або відсутності архітектурних прикрас. Їхня структура загалом має небагато відмінностей — значною мірою вона обумовлена законом тяжіння і фізичними властивостями матеріалів. Програмне забезпечення, навпаки, жодним чином не пов'язане з вагою, крім почуття серйозності. З чого ж зроблене програмне забезпечення? На відміну від будівель, які можуть бути побудовані з цегли, бетону, дерева, сталі й скла, програмне забезпечення складається з програмних компонентів. Великі програмні конструкції створюються з менших програмних компонентів, які, своєю чергою, побудовані зі ще дрібніших програмних компонентів, і так аж до основи.

Говорячи про архітектуру, можна сказати, що програмне забезпечення за своєю природою є фрактальним і рекурсивним, вигравіруваним і окресленим у коді. Тут важливі всі деталі. Переплетення рівнів деталізації також робить свій внесок в архітектуру, але безглуздо говорити про програмне забезпечення в фізичному сенсі. Програмне забезпечення має

структуру — безліч структур і силу-силенну їхніх різновидів,— але ця різноманітність затьмарює діапазон фізичних структур, які можна побачити на прикладі будівель. Можна навіть із упевненістю стверджувати, що у процесі проєктування програмного забезпечення архітектурі приділяють значно більше уваги, ніж під час проєктування будинків,— у цьому сенсі архітектура програмного забезпечення є більш різноманітною, ніж архітектура будівель!

Але фізичний масштаб звичніший для людей, і вони часто шукають його у довколишньому світі. Незважаючи на привабливість і візуальну очевидність, прямокутники на діаграмах PowerPoint не є архітектурою програмного забезпечення. Так, вони представляють певний погляд на архітектуру, але приймати прямокутники за загальну картину, що відображає архітектуру, значить не отримати ані загальної картини, ані будь-якої гадки про архітектуру: архітектура програмного забезпечення ні на що не схожа. Конкретний спосіб візуалізації — не більше ніж приватний вибір. Цей вибір заснований на наступному наборі варіантів: що ввімкнути; що вимкнути; що підкреслити формою або кольором; що, навпаки, затінити. Жоден погляд не має ніяких переваг над будь-яким іншим.

Можливо, немає сенсу говорити про закони фізики і фізичні масштаби стосовно архітектури програмного забезпечення, але ми справді враховуємо певні фізичні обмеження. Швидкість роботи процесора і пропускна здатність мережі можуть винести суворий присуд продуктивності. Обсяг пам'яті і дискового простору може обмежити амбіції будь-якого програмного коду. Програмне забезпечення можна порівняти з такою матерією, як мрії, але йому доводиться працювати в реальному, фізичному світі.

> *Така лише потворність є в коханні, моя любове: жадання безмежне, а дійсність має певні рамки; бажання неокраєне, а здійснення — раб певної межі.*
>
> Вільям Шекспір[1]

Фізичний світ — це світ, у якому ми живемо, у якому існують і діють наші компанії і економіка. Це дозволяє нам по-іншому підходити до розуміння архітектури програмного забезпечення, що дає змогу розмірковувати і говорити не тільки у визначеннях фізичних законів і понять.

> *Архітектура відображає важливі проєктні рішення відносно формування системи, де важливість визначають вартістю змін.*
>
> Граді Буч

[1] Переклад М. Лукаша.

Час, гроші, трудовитрати — ось іще одна система координат, що допомагає нам розрізняти велике і мале та відокремлювати те, що стосується архітектури, від усього іншого. Ця система також допомагає дати якісну оцінку архітектурі — гарна та чи ні: гарна архітектура відповідає потребам користувачів, розробників і власників не лише зараз, але й продовжить відповідати їм у майбутньому.

Якщо ви вважаєте, що гарна архітектура коштує дорого, спробуйте погану архітектуру.

Браян Фут і Джозеф Йодер

Типові зміни, що відбуваються під час розробки системи, не повинні бути дорогими або складними в реалізації; вони повинні вкладатися у графік розвитку проєкту і в рамки денних або тижневих завдань.

Це веде нас до чималої проблеми, пов'язаної з фізикою: подорожей у часі. Як дізнатися, які типові зміни будуть відбуватися, щоб на основі цього знання прийняти важливі рішення? Як зменшити трудовитрати і вартість розробки без машини часу і ворожіння на кавовій гущі?

Архітектура — це набір правильних рішень, які хотілося б прийняти на ранніх етапах роботи над проєктом і які на момент прийняття цього рішення вірогідні так само, як і всі інші можливі рішення.

Ральф Джонсон

Аналіз минулого складний; розуміння дійсності в кращому разі мінливе; пророкування майбутнього нетривіальне.

Мети можна досягти багатьма шляхами.

На найскладнішому шляху підстерігає думка, що міцність і стабільність архітектури залежать від суворості і жорсткості. Якщо зміни здаються надто дорогими, їх відкидають, а їхні причини скасовують вольовим рішенням. Архітектор має повну і беззастережну владу, натомість архітектура перетворюється на антиутопію для розробників і постійне джерело невдоволень.

Другий шлях просто-таки тхне спекулятивною узагальненістю. Він повний здогадок, числених параметрів і могильників із «мертвим» кодом, на ньому підстерігає безліч випадкових складнощів, здатних похитнути бюджет, виділений на обслуговування.

Але найцікавіший — третій, чистий шлях. Він враховує природну гнучкість програмного забезпечення і прагне зберегти її як основну властивість системи. Він враховує, що ми оперуємо неповними знаннями

і, будучи людьми, непогано пристосувалися до цього. Він грає більше на наших сильних сторонах, ніж на слабкостях. Ми створюємо щось і робимо відкриття. Ми ставимо питання і проводимо експерименти. Гарна архітектура ґрунтується швидше на розумінні руху до мети як безперервного процесу досліджень, аніж на розумінні самої мети як зафіксованого артефакту.

> *Архітектура — це гіпотеза, яку треба довести реалізацією та оцінкою.*
>
> Том Гілб

Щоб піти цим шляхом, потрібно бути старанним і уважним, потрібно вміти думати і спостерігати, набувати практичних навичок і засвоювати принципи. Спочатку здається, що це довгий шлях, але насправді все залежить від темпу вашої ходьби.

> *Єдиний спосіб крокувати швидко — це крокувати правильно.*
>
> Роберт С. Мартін

Отримуйте задоволення від подорожі.

Кевлін Хенні, травень 2017

Передмова автора

Ця книжка називається «Чиста архітектура». Смілива назва. Хтось вважатиме її самовпевненою. Чому я вирішив написати цю книжку і обрав таку назву?

Свій перший рядок коду я написав 1964-го, у 12 років. Зараз на календарі 2016-й, тобто я пишу код уже понад півстоліття. За цей час я дещо дізнався про структурування програмних систем, і мої знання, як мені здається, багато кому могли б стати у пригоді.

Я придбав ці знання, створивши безліч систем, великих і маленьких. Я створював маленькі вбудовані системи і великі системи пакетної обробки, системи реального часу і веб-системи, консольні застосунки, застосунки з графічним інтерфейсом, застосунки управління процесами, ігри, системи обліку, телекомунікаційні системи, інструменти для проєктування, графічні редактори і багато-багато іншого.

Я писав одно- і багатопотокові програми, застосунки з кількома важкими процесами і застосунки зі значною кількістю легковагих процесів, багатопроцесорні застосунки, застосунки баз даних, застосунки для математичних обчислень і обчислювальної геометрії і багато-багато іншого.

Я написав дуже багато застосунків, я створив дуже багато систем. І завдяки накопиченому досвіду я дійшов приголомшливого висновку:

Усі архітектури працюють за однаковісінькими правилами!

Приголомшливого, тому що всі системи, у створенні яких я брав участь, радикально відрізнялися одна від одної. Чому архітектури таких різних систем підкоряються тим самим правилам? Я дійшов висновку, що *правила створення архітектури не залежать від будь-яких змінних.*

Цей висновок вражає тим більше, якщо згадати, як змінилися комп'ютери за ті ж півстоліття. Я починав програмувати на машинах завбільшки з холодильник, які мали процесори з тактовою частотою пів мегагерца, 4 кілобайти оперативної пам'яті, 32 кілобайти дискової пам'яті, інтерфейс телетайпа зі швидкістю передачі даних 10 символів за секунду. Я пишу

цей вступ в автобусі, подорожуючи Південною Африкою. Переді мною MacBook із чотириядерним процесором i7 із тактовою частотою 2,8 ГГц, оперативною пам'яттю 16 Гбайт, дисковою пам'яттю об'ємом 1 Тбайт (на пристрої SSD) і дисплеєм із матрицею 2880 × 1800, що здатний відображати високоякісне відео. Різниця в обчислювальній потужності запаморочлива. Будь-який аналіз покаже, що цей MacBook щонайменше у 10^{22} разів потужніший за ті комп'ютери, на яких я починав пів століття тому.

Двадцять два порядки — дуже велике число. Це число ангстремів від Землі до альфи Центавра. Це кількість електронів у дрібних монетах у вашій кишені або гаманці. І ще це число описує, у скільки разів (щонайменше) зросла обчислювальна потужність комп'ютерів за моєї пам'яті.

А як впливало зростання обчислювальної потужності на програми, які мені доводилося писати? Безумовно, вони стали більші. Раніше я думав, що 2000 рядків — це велика програма. Зрештою, така програма займала повну коробку перфокарт і важила 4,5 кілограма. Однак тепер програму вважають по-справжньому великою, лише якщо обсяг коду перевищує 100 000 рядків.

Програмне забезпечення також стало набагато продуктивнішим. Сьогодні ми можемо швидко виконувати такі обчислення, про які навіть не мріяли у 1960-х. У «The Forbin Project», «The Moon Is a Harsh Mistress» і «2001: A Space Odyssey» було зроблено спробу зобразити наше поточне майбутнє, але їхні автори серйозно промахнулися. Усі вони зображують величезні машини, що мають почуття. Натомість у нас є неймовірно маленькі машини, які все одно залишаються машинами.

І ще одна важлива подібність сучасного програмного забезпечення і тогочасного програмного забезпечення: *і те й друге зроблене з одного й того ж матеріалу*. Воно складається з інструкцій `if`, інструкцій присвоювання і циклів `while`.

Так, ви можете заперечити, заявивши, що сучасні мови програмування набагато кращі й підтримують кращі парадигми. Урешті-решт, ми програмуємо на Java, C# або Ruby і використовуємо об'єктно-орієнтовану парадигму. І все ж програмний код складається з послідовностей операцій, умовних інструкцій та ітерацій, як і у 1950-х чи 1960-х роках.

Уважно розглянувши практику програмування комп'ютерів, ви помітите, що за 50 років змінилося не так і багато. Трохи удосконалилися мови. Інструменти стали просто фантастичними. Але основні будівельні блоки комп'ютерних програм залишилися незмінними.

Якщо взяти програмістку з 1966 року[1], перемістити у 2016-й, посадити за мій MacBook, запустити IntelliJ і показати Java, їй вистачить доби, щоб

[1] У ті роки програмістами були здебільшого жінки.

оговтатися від шоку. А потім вона зможе писати сучасні програми. Мова Java не надто відрізняється від C або навіть від Fortran.

Так само якщо вас перемістити назад, у 1966-й, і показати, як писати і правити код на PDP-8, пробиваючи перфокарти на телетайпі, що підтримує швидкість 10 символів за секунду, вам, напевне, теж вистачить доби, щоб оговтатися від розчарування. Але потім і ви будете здатні писати код. Сам код зазнав не таких уже й великих змін.

У цьому весь секрет: незмінність принципів програмування — ось причина спільності правил побудови програмних архітектур для систем найрізноманітніших типів. Правила визначають послідовність і порядок компонування програм із будівельних блоків. І оскільки власне будівельні блоки є універсальними і не змінилися з плином часу, правила їхнього компонування також є універсальними і повсякчасними.

Програмісти-початківці можуть подумати, що все це нісенітниця, що зараз усе зовсім інакше і краще, що правила минулого залишилися в минулому. Якщо вони справді так думають, вони, на жаль, дуже помиляються. Правила не змінилися. Незважаючи на появу нових мов, фреймворків і парадигм, правила залишилися тими ж, як за часів, коли у 1946-му Алан Тьюринг написав перший програмний код.

Змінилося лише одне: тоді, у минулому, ми не знали цих правил. Тому порушували їх знову і знову. Тепер, з півстолітнім досвідом за плечима, ми знаємо і розуміємо правила.

Саме про ці правила — що не старіють і не змінюються — розповідає ця книжка.

Подяки

Нижче поіменовані (у довільному порядку) всі, хто зіграв важливу роль у створенні цієї книжки:

Кріс Ґузіковскі (Chris Guzikowski)
Кріс Зан (Chris Zahn)
Метт Гойзер (Matt Heuser)
Джефф Овербі (Jeff Overbey)
Міка Мартін (Micah Martin)
Джастін Мартін (Justin Martin)
Карл Гікман (Carl Hickman)
Джеймс Ґреннінг (James Grenning)
Симон Браун (Simon Brown)
Кевлін Генні (Kevlin Henney)
Джейсон Ґорман (Jason Gorman)
Дуг Бредбері (Doug Bradbury)
Колін Джонс (Colin Jones)
Ґраді Буч (Grady Booch)
Кент Бек (Kent Beck)
Мартін Фаулер (Martin Fowler)
Алістер Кокберн (Alistair Cockburn)
Джеймс О. Коплін (James O. Coplien)
Тім Конрад (Tim Conrad)
Річард Ллойд (Richard Lloyd)
Кен Фіндер (Ken Finder)
Кріс Івер (Kris Iyer (CK))
Майк Карєв (Mike Carew)
Джеррі Фіцпатрік (Jerry Fitzpatrick)
Джим Ньюкірк (Jim Newkirk)
Ед Телень (Ed Thelen)

Джо Мейбл (Joe Mabel)
Білл Дегнан (Bill Degnan)

Та багато інших.

Коли я переглядав книжку в остаточному варіанті і перечитував розділ про кричущу архітектуру, переді мною стояв образ усміхненого Джима Веріча (Jim Weirich) і у вухах лунав його дзвінкий сміх. Хай щастить тобі, Джиме!

Про автора

Роберт С. Мартін (Robert C. Martin), відомий також як «дядечко Боб» (Uncle Bob), професійно займається програмуванням від 1970 року. Співзасновник компанії cleancoders.com, яка пропонує відеоуроки для розробників програмного забезпечення, і засновник компанії Uncle Bob Consulting LLC, що надає консультаційні послуги та послуги з навчання персоналу великим корпораціям. Працює в консалтинговій фірмі 8th Light, Inc. у місті Чикаго. Опубліковував десятки статей у спеціалізованих журналах і регулярно виступає на міжнародних конференціях і презентаціях. Три роки був головним редактором журналу «C++ Report» і першим головою «Agile Alliance».

Мартін написав і відредагував безліч книжок, зокрема «The Clean Coder», «Clean Code», «UML for Java Programmers», «Agile Software Development», «Extreme Programming in Practice», «More C++ Gems», «Pattern Languages of Program Design 3» і «Designing Object Oriented C++ Applications Using the Booch Method».

I Вступ

Щоб написати програму, яка працюватиме, не потрібно багато знати і вміти. Діти в школі роблять це постійно. Юнаки та дівчата в коледжах починають свій бізнес, що виростає до вартості у мільярди доларів, написавши кілька рядків коду на PHP або Ruby. Програмісти-початківці в офісах по всьому світі перелопачують гори вимог, що зберігаються в гігантських баг-трекерах, і вносять виправлення, щоб змусити свої системи «працювати». Можливо, вони пишуть і не найкращий код, але він *працює*. Він працює, тому що змусити щось працювати — один раз — не дуже складно.

Створити програму, яка буде працювати правильно,— зовсім інша справа. Написати правильну програму *складно*. Для цього необхідні знання і вміння, які молоді програмісти ще не встигли надбати. А щоб надбати їх, потрібно розмірковувати і аналізувати, на що у багатьох програмістів просто немає часу. Це вимагає такої самодисципліни і організованості, які не снилися більшості програмістів. А ще для цього потрібні пристрасть до професії і бажання стати професіоналом.

Проте коли створюється правильний програмний код, відбувається щось незвичайне: вам не потрібні юрби програмістів для підтримки його працездатності. Вам не потрібні об'ємна документація з вимогами і гігантські баг-трекери. Вам не потрібні величезні опенспейси, що працюють цілодобово без вихідних.

Правильний програмний код не вимагає великих трудовитрат на своє створення і супровід. Зміни вносяться легко і швидко. Помилок небагато. Трудовитрати мінімальні, а функціональність і гнучкість — максимальні.

Так, це здається утопічним. Але я бачив це на власні очі. Я брав участь у проєктах, де дизайн та архітектура системи спрощували їхнє створення і супровід. У мене є досвід роботи в проєктах, де знадобилося набагато менше учасників, ніж передбачалося. Мені доводилося працювати над системами, у яких помилки траплялися напрочуд рідко. Я бачив, який неймовірний та ефективний вплив має гарна архітектура на систему, проєкт і колектив розробників. Я бував на землі обітованій.

Але я не прошу вірити мені на слово. Озирніться на свій досвід. Чи траплялося у вас щось протилежне? Чи доводилося вам працювати над системами з такими заплутаними і тісними зв'язками, що будь-які зміни, незалежно від складності, вимагали тижнів праці та були пов'язані з величезним ризиком? Чи відчували ви опір поганого коду і невдалого дизайну? Чи доводилося вам бачити, як дизайн системи мав руйнівний вплив на моральний дух команди, довіру клієнтів і терпіння керівників? Чи доводилося вам опинятися в ситуації, коли відділи, підрозділи та цілі компанії ставали жертвами невдалої архітектури їхнього програмного забезпечення? Чи бували ви в пеклі програмування?

Я був, і багато хто з нас був там. У нашому середовищі частіше трапляється досвід боротьби з поганим дизайном, ніж отримання задоволення від втілення добре продуманої архітектури.

1 Що таке дизайн та архітектура?

За довгі роки навколо понять «дизайн» і «архітектура» накопичилося багато плутанини. Що таке дизайн? Що таке архітектура? Чим вони відрізняються?

Одна з цілей цієї книжки — усунути весь цей безлад і визначити раз і назавжди, що таке дизайн та архітектура. Перш за все я стверджую, що між цими поняттями немає жодної різниці. *Узагалі жодної.*

Слово «архітектура» часто використовують у контексті загальних міркувань, коли не йдеться про низькорівневі деталі, стосовно яких зазвичай вживають слово «дизайн». Але такий розподіл безглуздий, коли мова про те, що робить справжній архітектор.

Візьмемо для прикладу архітектора, що спроєктував мій новий будинок. Цей будинок має архітектуру? Звичайно! А в чому вона виражається? Ну… Це форма будинку, зовнішній вигляд, а також розташування кімнат і організація простору всередині. Але коли я розглядав креслення, створені архітектором, я побачив на них купу деталей. Я побачив розташування всіх розеток, вимикачів і світильників. Я побачив, які вимикачі будуть керувати тими чи іншими світильниками. Я побачив, де буде розміщений вузол опалення, а також місце розташування і розміри водогрійного котла і насоса. Я побачив докладний опис, як потрібно конструювати стіни, дах і фундамент.

Простіше кажучи, я побачив усі дрібні деталі, які підтримують усі високорівневі рішення. Я також побачив, що низькорівневі деталі і високорівневі рішення разом складають дизайн будинку.

Те саме стосується архітектури програмного забезпечення. Низькорівневі деталі і високорівнева структура є частинами одного цілого. Вони утворюють суцільне полотно, яке визначає форму системи. Одне без іншого неможливе; немає жодної чіткої лінії, яка розмежовувала б їх. Є просто сукупність рішень різного рівня деталізації.

Мета

У чому полягає мета таких рішень, мета гарного дизайну програмного забезпечення? Головна мета — не що інше, як моє утопічне визначення:

> *Мета архітектури програмного забезпечення — зменшити трудовитрати на створення і супровід системи.*

Мірою якості дизайну може слугувати проста міра трудовитрат, необхідних для задоволення потреб клієнта. Якщо трудовитрати невеликі

і залишаються невеликими протягом експлуатації системи, система має гарний дизайн. Якщо трудовитрати збільшуються з виходом кожної нової версії, система має поганий дизайн. Ось так все просто.

Приклад із практики

Для прикладу розглянемо результати досліджень із практики. Вони засновані на реальних даних, наданих реальною компанією, котра побажала не розголошувати своєї назви.

Спочатку розглянемо графік зростання чисельності інженерно-технічного персоналу. Ви, напевно, погодитеся, що тенденція вражає. Зростання чисельності, що показане на *рис. 1.1*, мало б слугувати ознакою успішного розвитку компанії!

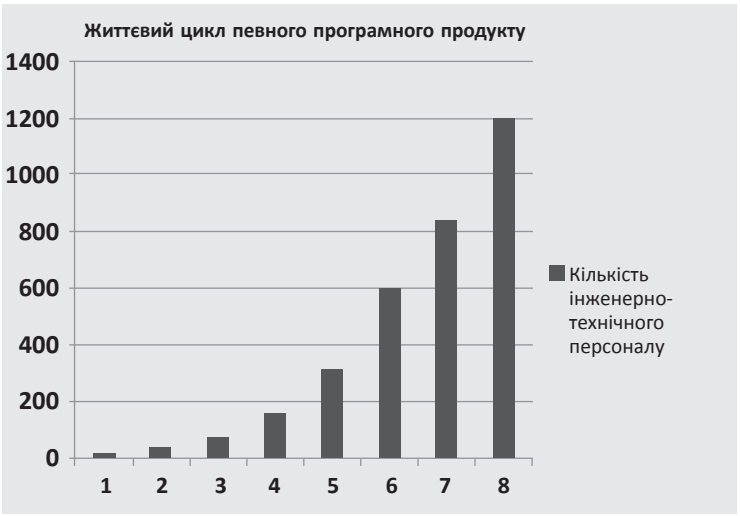

Рис. 1.1. Зростання чисельності інженерно-технічного персоналу

Відтворюється з дозволу автора презентації Джейсона Гормана (Jason Gorman)

Тепер погляньте на графік продуктивності компанії за той самий період, що вимірюється в кількості рядків коду (див. *рис. 1.2*).

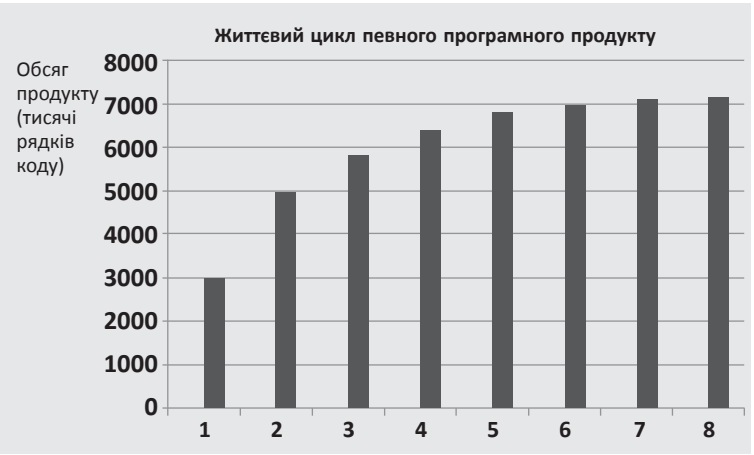

Рис. 1.2. Продуктивність за той самий період

Очевидно, тут щось не так. Навіть із урахуванням того, що випуск кожної версії підтримується зростаючою кількістю розробників, схоже, що кількість рядків коду наближається до своєї межі.

А тепер погляньте на по-справжньому гнітючий графік: на *рис. 1.3* показане зростання вартості рядка коду з плином часу.

Рис. 1.3. Зміна вартості рядка коду з плином часу

Ця тенденція свідчить про нежиттєздатність. Хоч якою б рентабельною наразі не була компанія, накладні витрати, які постійно зростають, поглинуть прибуток і призведуть до застою, а то й до краху.

Чим зумовлене таке жахливе зниження продуктивності? Чому рядок коду у версії 8 продукту коштує в 40 разів дорожче, ніж у версії 1?

Причини неприємностей

Причини неприємностей ви бачите й самі. Коли системи створюються поспіхом, коли збільшення штату програмістів — єдиний спосіб продовжувати випускати нові версії, коли чистоті коду або дизайну приділяють мінімум уваги або не приділяють уваги взагалі, можна навіть не сумніватися, що така тенденція рано чи пізно призведе до краху.

На *рис. 1.4* показано, який вигляд має ця тенденція у перекладі на мову продуктивності розробників. Спочатку розробники показують продуктивність, близьку до 100 %, але з виходом кожної нової версії вона знижується. Починаючи з четвертої версії, як неважко помітити, продуктивність розробників наближається до нижньої межі — до нуля.

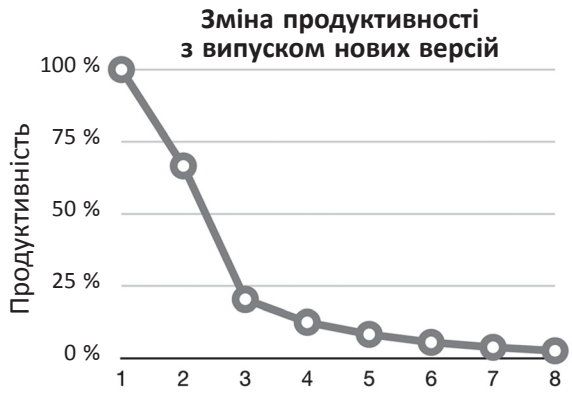

Рис. 1.4. Зміна продуктивності з випуском нових версій

З точки зору розробників така ситуація справляє неймовірно гнітюче враження, тому всі вони продовжують працювати *з повною віддачею*. Ніхто не ухиляється від роботи.

І все ж таки, незважаючи на понаднормову працю і самовідданість, вони просто не можуть зробити більше. Усі їхні зусилля тепер спрямовані не на реалізацію нових функцій, а на боротьбу з безладом. Більшу частину часу вони зайняті тим, що раз у раз переносять безлад із одного місця в інше, щоб отримати можливість додати ще якусь дрібничку.

Точка зору керівництва

Якщо ви гадаєте, *що ситуація* кепська, погляньте на неї з точки зору керівництва! На *рис. 1.5* зображено графік зміни місячного фонду оплати праці розробників за той самий період.

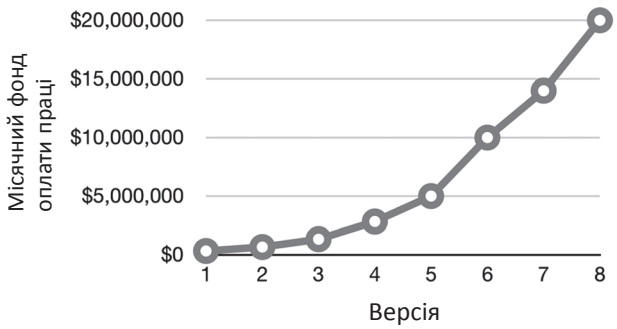

Рис. 1.5. Зміна фонду оплати праці розробників із випуском нових версій

Коли виходила версія 1, місячний фонд оплати праці становив кілька сотень тисяч доларів. До випуску другої версії цей фонд збільшився ще на кілька сотень тисяч. До випуску восьмої версії місячний фонд оплати праці досяг суми 20 мільйонів доларів, і тенденція до збільшення зберігається.

Цей графік не просто здається поганим, він лякає. Очевидно, що відбувається щось жахливе. Жевріє надія, що зростання прибутків випереджає зростання витрат, а отже, виправдовує витрати. Але з будь-якої точки зору ця крива породжує занепокоєння.

А тепер порівняйте криву на *рис. 1.5* із графіком зростання кількості рядків коду від версії до версії на *рис. 1.2*. Спочатку всього за кілька сотень тисяч доларів на місяць вдалося реалізувати величезний обсяг функціональності, а за 20 мільйонів в останню версію майже нічого не було додано! Будь-який менеджер, глянувши на ці два графіки, дійде висновку, що треба щось зробити, аби запобігти краху.

Але що можна зробити? Що пішло не так? Що спричинило таке неймовірне зниження продуктивності? Що можуть зробити керівники, окрім того, щоб тупнути ногою і насварити розробників?

Що не так?

Приблизно 2600 років тому Езоп склав байку про Зайця і Черепаху.

Мораль цієї байки можна висловити по-різному:

- «повільний і постійний перемагає у перегонах»;
- «у перегонах не завжди перемагає найшвидший, а в битві — найсильніший»;
- «що більше поспішаєш, то менше встигаєш».

Притча підкреслює безглуздість самовпевненості. Заєць був настільки впевнений у своїй швидкості, що не поставився серйозно до змагання, вирішив подрімати і проспав, коли Черепаха перетнула лінію фінішу.

Сучасні розробники також беруть участь у схожих перегонах і проявляють схожу самовпевненість. О ні, вони не дрімають! Багато хто з сучасних розробників працює як проклятий. Але частина їхнього мозку *дійсно* спить — та частина, яка знає, що гарний, чистий, добре розроблений код грає *важливу роль*.

Ці розробники вірять у відому брехню: «Ми зможемо навести лад потім, нам зараз потрібно просто вийти на ринок!» У результаті лад не наводиться, бо тиск конкуренції на ринку ніколи не слабшає. Вихід на ринок означає, що тепер у вас на хвості висять конкуренти і ви повинні прагнути залишатися попереду них і бігти вперед щосили.

Тому розробники ніколи не перемикають режим роботи. Вони не можуть повернутися і навести лад, тому що повинні реалізувати нову функцію, а потім ще одну, і ще, і ще. У результаті безладдя зростає, а продуктивність тяжіє до нульової позначки.

Так само як Заєць був надто впевнений у своїй швидкості, багато розробників надміру впевнені у своїй здатності залишатися продуктивними, але накоплюваний безлад у коді, що висушує їхню продуктивність, ніколи не спить і ніколи не байдикує. Якщо дати йому волю, він зменшить виробничі показники до нуля протягом лічених місяців.

Найбільша брехня, у яку вірить багато розробників,— що брудний код допоможе їм швидко вийти на ринок, але насправді він загальмує їхній рух у довгостроковій перспективі. Розробники, які увірували у цю брехню, проявляють самовпевненість Зайця, вважаючи, що в майбутньому зможуть перейти від створення безладу до наведення ладу, але вони припускаються простої помилки. Справа в тому, що *створюючи безлад, завжди рухатимешся повільніше, аніж неухильно дотримуючись чистоти*, незалежно від обраних одиниць часу.

Подивімося на *рис. 1.6* результати показового експерименту, що проводився Джейсоном Горманом протягом шести днів. Щодня він писав від початку до кінця просту програму перетворення цілих чисел із десяткової системи числення в римську. Роботу вважали закінченою, коли програма успішно проходила зумовлений комплект приймальних тестів. Щодня

на рішення поставленого завдання витрачалося трохи менше 30 хвилин. Першого, другого і третього дня Джейсон використовував добре відому методику розробки через тестування (Test-Driven Development, TDD). В інші дні він писав код, не обмежуючи себе рамками цієї методики.

Рис. 1.6. Час на виконання ітерації з використанням методики TDD і без нього

Перш за все зверніть увагу на криву навчання, зображену на *рис. 1.6*. Щоразу на рішення задачі витрачалося менше часу. Відзначте також, що в дні, коли була застосована методика TDD, вправа виконувалась приблизно на 10 % швидше, ніж у дні без застосування TDD, і що навіть найгірший результат, отриманий із TDD, виявився кращим від найкращого результату, отриманого без TDD.

Хтось, глянувши на цей результат, може назвати його дивним. Але для того, хто не піддався на оману самовпевненості Зайця, результат буде цілком очікуваним, тому що він знає просту істину розробки програмного забезпечення:

Єдиний спосіб крокувати швидко — це крокувати правильно.

Також ця істина є відповіддю на дилему, що стоїть перед керівництвом. Єдиний спосіб повернути назад зниження продуктивності і збільшення вартості — змусити розробників перестати думати, як самовпевнений Заєць, і почати нести відповідальність за безлад, який вони вчинили.

Розробники можуть подумати, що проблему можна виправити, лише почавши все від самого початку і перепроєктувавши всю систему,— але це

в них промовляє той самий Заєць. Та ж самовпевненість, яка колись уже привела до безладдя, тепер запевняє, що вони зможуть побудувати кращу систему, варто лише ще раз взяти участь у перегонах. Однак насправді все не так райдужно:

Самовпевненість, що керує перепроєктуванням, призведе до того ж безладдя, що й раніше.

Висновки

Будь-якій організації, що займається розробкою, найкраще уникати самовпевнених рішень і від самого початку з усією серйозністю поставитися до якості архітектури кінцевого продукту.

Серйозне ставлення до архітектури програмного забезпечення передбачає знання про те, що таке гарна архітектура. Щоб створити систему, дизайн та архітектура якої сприяють зменшенню трудовитрат і збільшенню продуктивності, потрібно знати, які елементи архітектури сприяють цьому.

Саме про це розповідається у книжці, що зараз перед вами. У ній розповідається, який вигляд має добротна, чиста архітектура і дизайн, щоб розробники могли створювати системи, здатні тривалий час приносити прибуток.