Від чого розвалюються повітряні замки?

Введення у керування складністю при створенні програмних систем

Руслан Дмитракович

Руслан Дмитракович

Від чого розвалюються повітряні замки? Введення у керування складністю при створенні програмних систем

2025

Від чого розвалюються повітряні замки? Введення у керування складністю при створенні програмних систем

Руслан Дмитракович

Мова видання: українська
Ілюстрації: Юлія Дмитракович. Частина ілюстрацій згенерована за допомогою AI (Midjourney).

Анотація

Програмні системи є одними з найскладніших витворів людства. Саме через це їх створення пов'язане з великими ризиками. За даними різних досліджень, лише близько 30% проектів вважаються успішними, тобто вони завершуються вчасно, в межах бюджету та з усіма запланованими функціями.

Дана книга присвячена саме тому, щоб зменшити ці ризики. Погляд на процес розробки програмного забезпечення як на ланцюжок комунікації від замовника до комп'ютера дозволить зрозуміти ключові чинники, що впливають на успіх проекту, а також свідомо керувати ними.

Книга розрахована на широке коло читачів, чия діяльність пов'язана з розробкою програмного забезпечення: як професійних розробників, які бажають підвищити свою кваліфікацію, так і тих, хто робить перші кроки у цій сфері.

Зміст

Поглиблюємо розуміння, корисні принципи

Передмова

Передмова

Програміст, подібно до поета, працює майже безпосередньо з чистою думкою. Він будує свої замки в повітрі і з повітря, творячи силою уяви.
Ф. Брукс, Міфічний людино-місяць, або як створюються програмні системи

Починаючи програмний проєкт, ви, швидше за все, хотіли б його завершити, і завершити в строк. Однак, чим більший масштаб проекту, тим вищий ризик - існує багато причин, які можуть завадити його завершенню. Ця книга написана для того, щоб значно збільшити ваші шанси на успіх. Якщо ви **замовник,** то отримані знання допоможуть зменшити витрати на проект. Якщо ви **розробник,** то ваша робота стане більш зрозумілою та приємною, а результат - прогнозованим. Якщо ви **менеджер проекту,** ця книга допоможе вкластися в строк та зменшити кількість конфліктів.

У сучасному світі важко уявити інженера-будівельника, який би не знав законів фізики та не вивчав властивості матеріалів. Для нього ключовим фактором є сила тяжіння - вона визначає, чи витримає споруда власну вагу, чи зруйнується. У програмній інженерії існує аналогічна «сила», але, на жаль, багато хто не усвідомлює її наявності та суттєвого впливу, пояснюючи свої невдачі лише нестачею ресурсів.

Розробники програмного забезпечення концентруються на знаннях про алгоритми, мови програмування, бібліотеки та інші робочі інструменти. Однак цих знань недостатньо для реалізації великих проектів. Так само недостатньо вміння класти цеглу, якщо ви будуєте багатоповерхову будівлю. Як ви думаєте, що було б, якби інженерів-будівельників навчали управлінню баштовим краном замість вивчення опору матеріалів?

Складність - це суттєвий чинник, який впливає на процес розробки ПЗ. Розробники програмного забезпечення повинні знати про нього. Це так само важливо, як знання законів фізики для інженера-будівельника.

Стів Макконнелл, автор однієї з найкращих книг у галузі розробки програмного забезпечення «Досконалий код», вважає: «Управління складністю настільки важливе,

що воно має бути Головним Технічним Імперативом Розробки ПЗ».

На жаль, у літературі з програмування рідко зустрічається чіткий опис того, що таке складність і як саме нею керувати. Але як можна керувати чимось, не знаючи, що це взагалі таке? Ця книга покликана подолати наявну прогалину в розумінні питань пов'язаних зі складністю.

Подяки

Створення цієї книги було б неможливим без підтримки та внеску багатьох чудових людей. Я хотів би висловити свою щиру вдячність усім, хто допоміг цьому проекту стати реальністю.

Насамперед, я безмежно вдячний моїй дочці Юлії за графічні матеріали та допомогу в оформленні книги.

Особлива подяка моєму багаторічному колезі Дмитру Бурмістренку. Його унікальний погляд на речі, який часто відрізнявся від мого власного, був неоціненним у пошуку оптимальних рішень. Наші дискусії та обмін думками збагатили зміст цієї книги, а його професійний досвід допоміг зробити матеріал більш практичним та корисним для читачів.

Велика вдячність Степану Мозирі за його прискіпливість до тексту та конструктивні зауваження. Його уважне до деталей око допомогло зробити книгу більш зрозумілою.

Я також хочу подякувати Максиму Бондаренку, талановитому розробнику та ентузіасту біології, який став першим читачем багатьох розділів. Його слушні зауваження та надані наукові джерела допомогли забезпечити точність викладення матеріалу, особливо в розділах, пов'язаних з нейробіологією.

Окрема подяка всім колегам, з якими мені пощастило працювати протягом багатьох років. Саме щоденна спів-

праця з вами, обговорення технічних рішень, суперечки стали тим фундаментом, на якому зросли ідеї цієї книги. Ваш досвід, помилки та успіхи, які ми разом переживали, допомогли мені краще зрозуміти природу складності та знайти способи її подолання. Дякую за те, що кожен з вас додав свою частинку до мого професійного досвіду.

Розбираємося в проблемі

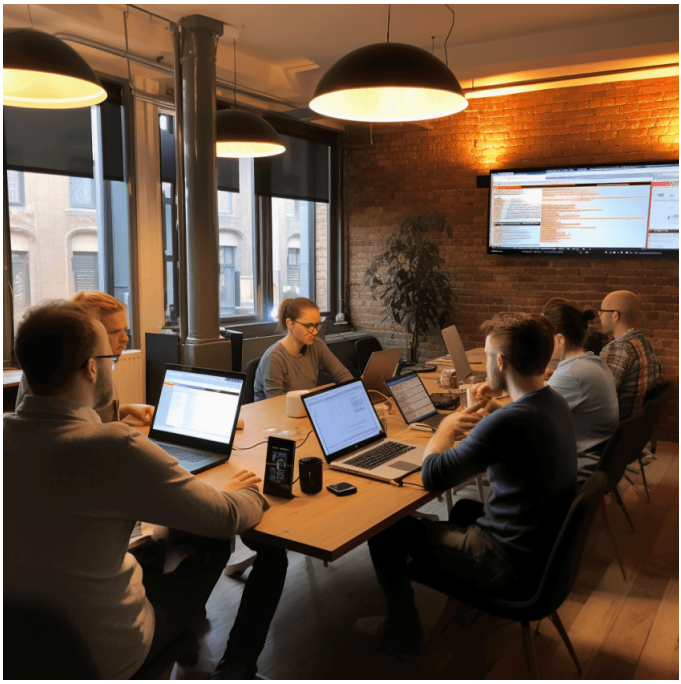
Чим займаються розробники програмного забезпечення

Для машин природні мови не є природними.
Алан Перліс

Як ви думаєте, чим саме займається розробник ПЗ? Якщо зайти у відділ книжкового магазину, присвячений комп'ютерній тематиці, і подивитися, які книги там продаються, може скластися враження, що розробник в першу чергу є технічним спеціалістом. Мови програмування, технології, архітектурні шаблони та тому подібне.

А що ж із тематичними форумами та конференціями? І тут ми бачимо ту ж картину. Більшість доповідей присвячені технічним темам – прикладам вирішення складних завдань, підвищенню продуктивності обчислень, практикам застосування мов та фреймворків.

Однак дослідження показують зовсім іншу картину того, чим наповнений робочий день розробника. І як не дивно, відсоток часу, коли програміст пише код, складає меншу частину його робочого часу. [1]



Давайте поставимо питання. Звідки програмісту відомо, що саме він повинен запрограмувати? Часи, коли одна людина могла займатися розробкою програмного забезпечення, давно минули. Сьогодні програмне забезпечення створюють команди спеціалістів, у яких кожен відповідає за свою область. І всі вони повинні спілкуватися один з одним, щоб отримати працюючий, цілісний продукт.

Розробники ПЗ не були б потрібні, якби замовники ПЗ могли чітко формулювати свої вимоги мовою, зрозумілою комп'ютеру. Однак часто Замовник сам не знає, що саме йому потрібно. Найчастіше Замовник таки може сформулювати проблему, але для того, щоб її рішення втілити у вигляді програмного продукту, йому потрібні розробники ПЗ — посередники між ним і комп'ютером.

Важливо глибоко зрозуміти суть проблеми, отримати від замовника необхідну інформацію та структурувати її. Першим посередником між замовником і комп'ютером є Аналітик. Робота аналітика багато в чому схожа на роботу психолога, який у ході спілкування з клієнтом намагається заглибитись в суть проблеми. В результаті цього спілкування знання та потреби клієнта будуть описані у вигляді вимог до програмного продукту.

Наступним ланцюгом є Архітектор ПЗ, який розуміє, що має представляти собою система в цілому. Він проєктує великі блоки, з яких складається програмна система. Але якщо подивитися під іншим кутом, то Архітектор перетворює знання, отримані від Аналітика, а також інших зацікавлених осіб (stakeholders), в опис моделі предметної галузі. Аналізуючи різні вимоги, архітектор проєктує модулі системи та потоки даних, якими ці модулі будуть обмінюватися.

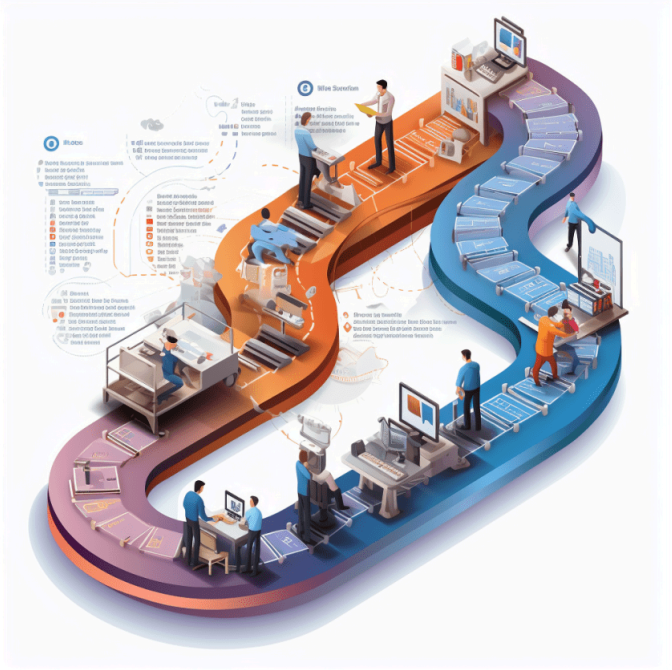
На цьому етапі особливості функціонування комп'ютерів вже беруться до уваги, з'являється більше технічної термінології. Але все ж, людська мова зберігається у вигляді коментарів, описів та уточнень.

Останнім ланцюгом у створенні ПЗ є програміст. Це саме той спеціаліст, який знаходиться на межі між формальними та неформальними мовами. Усі попередні учасники процесу могли передавати свої думки, використовуючи слова звичайної мови. Але програміст вже не може цього робити - машина його не зрозуміє. Будь-яка неточність або недомовленість або буде відхилена комп'ютером, або буде інтерпретована згідно з установленими правилами.

Наприклад, у мовах із нечіткою типізацією можна написати 3 + "яблуко", але результат буде сильно залежати від правил використовуваної мови. Наприклад, у PHP ви отримаєте число 3, а у JavaScript рядок "3яблуко".

Саме програміст повинен мати повне розуміння всіх тонкощів і деталей завдання. На додаток до написання коду, програміст повинен вивчити задачу і вимоги, створені аналітиком, спілкуватися з архітектором, щоб його код відповідав як бізнес-вимогам, так і технічним. Комунікація з колегами-програмістами також необхідна, раптом хтось вже робив щось подібне… І список можна продовжувати. Результатом цієї діяльності стає ментальна модель, яку програміст реалізує у вигляді програмного коду.

Таким чином, процес створення програмного забезпечення представляє собою конвеєр зі збору, аналізу і перетворення інформації. Кожен спеціаліст отримує інформацію із зовнішніх джерел, обробляє її і передає далі.



Два емпіричних закони, що виникли на зорі промислової розробки ПЗ, підтверджують цю тезу. Перший із

них, сформульований Мелвіном Конвеєм у 1967 році, говорить: “Організації проєктують системи, які копіюють структуру комунікацій в цій організації.” Другий, сформульований Фредеріком Бруксом, звучить так: “Якщо проект не вкладається у строки, то додавання робочої сили затримає його ще більше”. Причина криється в необхідності передачі знань новим учасникам, що відбирає час у тих, хто вже працює над проектом. Обидва ці твердження явно вказують на суттєвий вплив комунікації на різні аспекти реалізації програмного проекту.

Робота розробників дуже схожа на роботу журналіста: для написання статті потрібно зібрати матеріал і спілкуватися з багатьма людьми. Читачі очікують, що стаття буде правдиво відображати обрану тему.

Як при написанні статті, так і при розробці програми важливо, щоб вихідна інформація не губилася і не спотворювалася. Однак, чим більше і складніше інформація, тим вірогідніше поява помилок. Людська природа передбачає можливість помилок, і складність інформації суттєво впливає на ймовірність їх виникнення.

Складність також впливає і на швидкість обробки інформації: складна інформація обробляється повільніше.

Існують механізми зворотного зв’язку, які допомагають коригувати спотворення вихідних вимог, що виникають у ході розробки. Деякі з них можуть бути інтегровані у процес розробки. Серед них:

- Перегляд коду (code review)
- Автоматизоване та ручне тестування
- Статичний аналіз коду
- Зворотна комунікація між учасниками проекту, наприклад, уточнення задачі
- Зворотний зв’язок від користувачів ПЗ

Складність, таким чином, впливає на ключові характеристики успішності проекту: строки розробки, точність і повноту реалізації вимог. Неконтрольована складність веде до втрати часу та грошей, і, як наслідок, до провалу програмних проектів. Дивно, але більшість розробників не приділяє належної уваги управлінню складністю, концентруючись виключно на технічних аспектах своєї діяльності.

Висновки

Процес розробки програмного забезпечення представляє собою конвеєр спеціалістів, що збирають, аналізують і перетворюють інформацію. Складність цього процесу є ключовим фактором, що впливає на його якість. Більшість спеціалістів не надають належного значення цьому фактору, що призводить до збільшення витрат і невдач проектів. Важливо, щоб спеціалісти у галузі розробки ПЗ розуміли, що таке складність, вміли ідентифікувати її і управляти нею.

---

[1] Today was a Good Day: The Daily Life of Software Developers. Andre N. Meyer, Earl T. Barr, Christian Bird and Thomas Zimmermann. https://www.microsoft.com/en-us/research/uploads/prod/2019/04/devtime-preprint-TSE19.pdf

Що таке складність?

Знаєш, все набагато складніше. Олівер Квін,
«Стріла» (серіал)

То що таке складність? Спілкування з колегами показує, що кожен сприймає це всім відоме з дитинства поняття по-своєму. Спробуємо чітко сформулювати, про що далі йтиметься і що ми будемо розуміти під словом «складний».

Для початку заглянемо в тлумачний словник [2], який дає такі значення:

- складається з кількох частин, елементів;
- відрізняється взаємопов'язаністю багатьох частин;
- той, що складається з багатьох взаємопов'язаних явищ, ознак, відносин, процесів і т. д.;
- щось з різноманітними і суперечливими якостями, властивостями, особливостями (про людину, її характер, почуття тощо);
- важкий для розуміння, вирішення, здійснення.

Як видно, слово «складність» у різних контекстах може мати своє значення.

Розглянемо, що складність означає в різних областях науки.

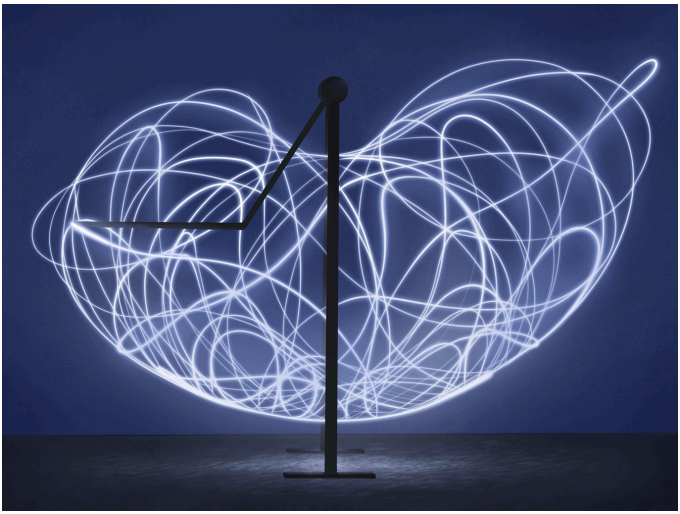
У теорії обчислень це поняття пов'язане з алгоритмами. Воно впливає на кількість обчислювальних ресурсів і часу виконання залежно від кількості вхідних даних. Ці

знання потрібні програмісту для того, щоб оцінити працездатність своєї програми з реальними обсягами даних. Цю складність називають обчислювальною складністю.

У теорії інформації також є своя складність. Візьмемо певний рядок. Довжина найкоротшої програми, яка виводить цей рядок, називається описовою складністю. Або, іншими словами, у даному контексті складність - це кількість інформації, необхідна для опису певного об'єкта.

У контексті вивчення систем складність часто інтерпретується як передбачуваність поведінки. Систему вважають складною, коли її поведінка слабо передбачувана. Система може складатися з окремих частин, які самі по собі досить прості для розуміння і мають передбачувану поведінку. Але поведінка системи, що складається з цих простих компонентів, може бути складною і слабо передбачуваною. Таку властивість системи називають «емерджентністю». Система - це більше, ніж просто набір своїх складових, і має властивості, яких немає у окремих компонентів.

Прикладом може бути подвійний маятник. Поведінка маятника передбачувана і описується простою формулою. Однак рух подвійного маятника може бути хаотичним.



Отже, у різних контекстах «складність» може бути синонімом інших слів: розмір або передбачуваність. Однак описане вище слабо пов'язане з реальним процесом розробки ПЗ. Усі ці «складності» пов'язані з об'єктом – алгоритмом, системою, інформацією.

Є ще одне слово, яке часто плутають зі складністю - «трудність». Трудність (трудомісткість), кількість роботи, яку потрібно виконати для отримання результату.

Часто задачу називають складною, якщо для її вирішення потрібно витратити багато зусиль. Але необхідно розрізняти складність і трудність (важкість). Задача може бути важкою, але не складною.

Уявімо, що вам дали лопату і попросили викопати яму. Якщо це неглибока, але довга траншея, то задача буде важкою, але здійсненною. Вам просто потрібно буде витратити певну кількість часу, відкидаючи лопатою ґрунт. Але якщо вам скажуть, що необхідний колодязь, задача з важкої перетвориться на складну. Вам доведеться враховувати безліч факторів – стінки колодязя можуть обвали-

тися, вам потрібно буде вирішити, як виймати ґрунт, що робити, якщо натрапите на водоносний шар і т. д.

Як видно з визначення у тлумачному словнику, навіть на побутовому рівні «складність» може бути як об'єктивною характеристикою чого-небудь (складається з багатьох частин і зв'язків), так і чимось відносним, залежним від конкретної людини. Складне для одного може бути простим для іншого.

Складність характеризує міру розумового напруження людини, яка зіткнулася з об'єктом або завданням.

Саме розумове напруження ми будемо називати «складністю» у контексті цієї книги. А також розглядатимемо фактори, що впливають на це.

Отже, складність відносна. Те, що є складним для одного, може бути елементарним для іншого: наприклад, розв'язання квадратного рівняння. Як для дитини, так і для людини, яка давно закінчила навчання, це може бути складним завданням.

Висновки

Когнітивна складність (об'єкта, системи, проблеми тощо) - це суб'єктивна характеристика, що дозволяє оцінити навантаження, розумове напруження, яке виникає у людини при спробі зрозуміти, передбачити поведінку, внести бажану зміну. Складним для людини є те, що виходить за межі її когнітивних здібностей і знань.

Когнітивну складність часто плутають з об'єктивними характеристиками – розміром, кількістю, передбачуваністю, а також трудовитратами. Оскільки розробники ПЗ частіше мають технічну освіту, вони думають про складність як про щось абсолютне, не пов'язане з конкретною

людиною. Хоча саме при розробці ПЗ слід враховувати те, як інформація буде сприйнята різними людьми.

---

[1] Серіал «Стріла» https://www.imdb.com/title/tt4862540/characters/nm4703025
[2] Одинадцятитомний «Словник української мови»
[3] Double pendilum https://youtube.com/shorts/CgYGyRt5kVI

Сила, яка діє в уявному світі

Проведення аналогій часто призводить до важливих відкриттів. Порівнюючи не зовсім зрозуміле явище з чимось схожим, але більш зрозумілим, ви можете здогадатися, як впоратися з проблемою.
С. Макконнелл, «Досконалий код»

Спробуємо звернутися до аналогій, щоб пояснити особливу важливість складності в розробці ПЗ.

Розробка програмних систем часто порівнюється з будівництвом. Шалаш може звести практично хто завгодно за досить короткий час.

Будівництво одноповерхового будинку – це завдання, яке не викликає суттєвих технічних проблем. Однак чим вище будівля, тим більша сила тисне на нижні поверхи. Для того щоб будівля не розвалилася, будівельнику необхідно проектувати будівлю так, щоб сила, що діє на кожен окремий елемент, не перевищувала допустимого значення.

Таким чином, сила тяжіння – це той головний фактор, який необхідно враховувати архітектору при проектуванні висотної будівлі.

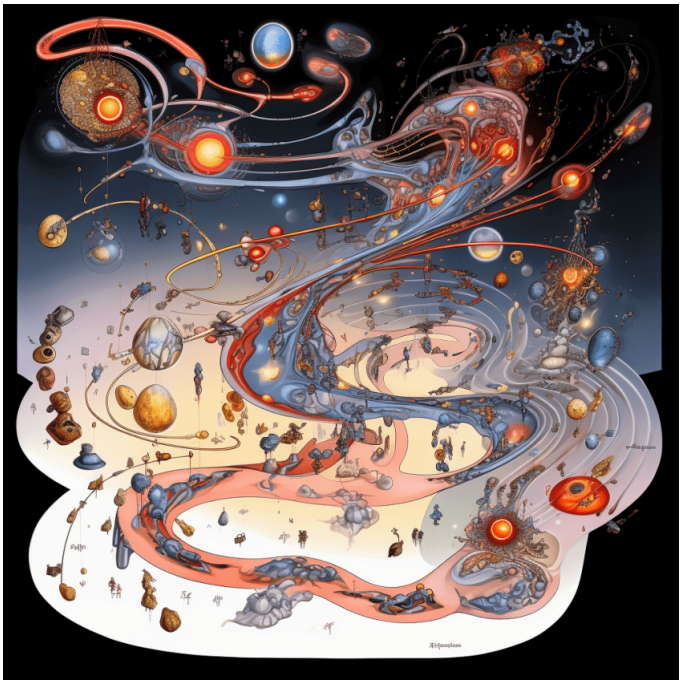
Застосування типових рішень спрощує будівництво, але чим вище будівля, тим більша сила тяжіння і як наслідок потрібно докладати більше зусиль для того, щоб зробити будівлю вищою.

Якби сила тяжіння була відсутня, то побудувати будівлю висотою в кілька кілометрів не було б серйозною технічною проблемою. Як вказувалося в попередньому розділі, це було б важке, але не складне завдання.

А як бути з програмами? Чому розробляти великі програмні комплекси не менш складно, ніж будувати хмарочоси?

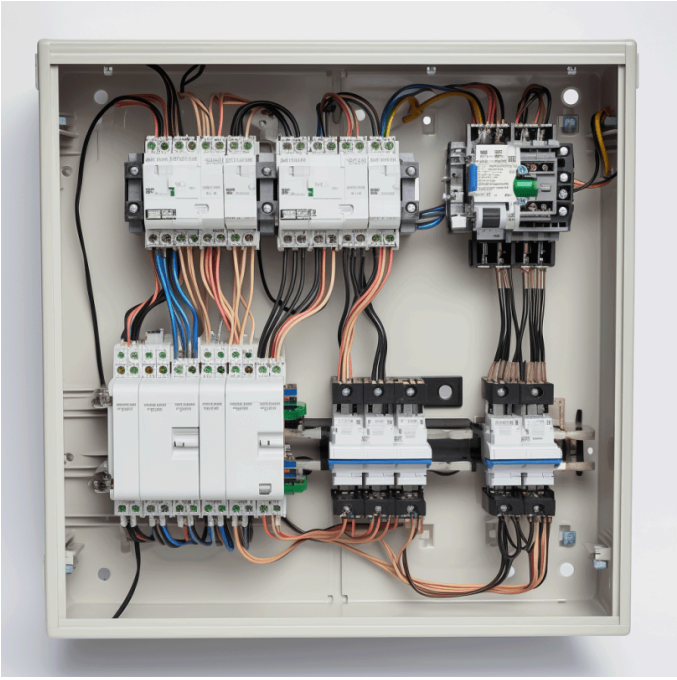
У віртуальному світі програмного забезпечення аналогом сили тяжіння виступає складність.

Складна система створюється ітераційно, великий програмний продукт постійно розвивається і змінюється. Однак для того, щоб щось змінити або додати в програмному продукті, потрібно розуміти, що він робить у поточний момент, а також мати можливість оцінити вплив цих змін.



Саме в цьому контексті важливо враховувати обмежені розумові здібності людини. Якщо зміни впливають на велику кількість інших компонентів, то, швидше за все, внесені зміни спричинять непередбачувану поведінку програми, помилки в її роботі.

Продовжимо аналогію з області будівництва. У вас є будинок, і ви вирішили розширити його, зробити прибудову. Ви додали приміщення, завели туди електроживлення, встановили освітлення, підключили обігрівач, холодильник і телевізор.



Все добре працює до того моменту, поки ви на кухні не ввімкнули електрочайник. Це призвело до того, що відключилося електропостачання у всьому будинку, оскільки ви не врахували обмеження джерела живлення, до якого підключений будинок.

Або, наприклад, ви вирішили повісити полицю на стіну, але забувши про товщину стіни, пробили картину, що висіла на протилежному боці…

Такі випадки при розробці ПЗ трапляються постійно. Розробнику необхідно тримати в голові те, з чим він має справу. І якщо кількість інформації перевищує його розумові можливості, то щось точно піде не так.

Необхідно враховувати не тільки зв’язки між компонентами, але й їх стан, який може змінюватися (як у випадку з чайником). А якщо це відбувається в різних

місцях програмної системи, та ще й одночасно, то отримаємо гримучу суміш, з якою людський мозок впоратися не в силах.

Складність, яка накопичується в ході розробки, в якийсь момент обрушує програмний продукт — відмови в роботі, помилки в даних і дуже важке внесення змін. Отже, складність — це саме та сила, яка подібно до сили всесвітнього тяжіння діє на програмний продукт.

Повітряний замок програмного продукту може розвалитися під вагою складності, створеної в процесі розробки.

Висновки

У процесі розробки програмного забезпечення складність відіграє роль, порівнянну з дією сили тяжіння при зведенні будівель. Якщо складність певного компонента програмного продукту досягає критичного рівня, то розробники не зможуть зрозуміти, як він працює, і це може викликати проблеми в роботі всього програмного продукту. Таким чином, необхідно відстежувати будь-яке локальне збільшення складності і намагатися його зменшити, щоб складність не досягла критичної позначки, після якої внесення змін стане вкрай важким завданням.

Багатоликий бог — ворога треба знати в обличчя

Скільком богам не молилися б люди, за всіма ними стоїть смерть.
Історії та фольклор, «Гра престолів» [1]

У розділі “Що таке складність?” ми визначили складність як міру розумового напруження людини, яка зіткнулася з об’єктом або проблемою. Якщо використовувати аналогію сили тяжіння, складність – це те, наскільки легко людині підняти певну вагу. Чим сильніша, чим більш тренована людина, тим з більшою вагою вони може впоратись. Чим вищі розумові здібності і знання, тим складнішу проблему може вирішити людина.

Якщо вагу предмета можна виміряти за допомогою вагів, то як бути зі складністю? І що саме ми будемо вимірювати?

Складність – Багатоликий бог

У творі «Гра престолів» Бога смерті називають Багатоликим богом. Те саме можна сказати й про складність. Складність проявляється у найрізноманітніших обличчях і видах. Вона починається з вимог до програмного продукту, які потрібно описати, проаналізувати і знайти протиріччя, та закінчується тестуванням. Складність може проявлятися як на рівні інтерфейсу взаємодії з користувачем, так і в загальній архітектурі системи.

“На жаль, але складність, про яку ми говоримо, ймовірно, притаманна всім великим програмним системам. Говорячи «притаманна», ми маємо на увазі, що ця складність тут неминуча: з нею можна впоратися, але позбутися її не можна.” [2]

Якщо ви в якомусь місці зменшуєте складність, то вона збільшується в іншому. Наприклад, розбиваючи модуль на простіші складові, ви збільшуєте кількість зв'язків між модулями та кількість модулів.

Отже, ми визначили, що складність може проявлятися по-різному і варіантів цих проявів може бути безліч. Кожен з цих аспектів впливає на сприйняття людиною програмної системи.

У випадку складності зазвичай працює правило БІЛЬШЕ → СКЛАДНІШЕ. Чим більше файлів, тим важче їх усі переглянути і запам'ятати. Чим більше бізнес-вимог, тим складніше з ними розібратися, чим більше станів може мати система, тим складніше передбачити її поведінку.

Кількість об'єктів системи, кількість зв'язків, кількість станів і т. д. [3].

Додаючи нові функції, ви принципово не можете не збільшувати складність програмної системи. Тут можна згадати про Колмогоровську (описову) складність, про яку йшла мова в попередньому розділі. Грубо кажучи, це розмір тексту, яким описується програмна система. І чим більше функцій у системі, тим довшим буде текст, що їх описує.

Але повернемося до питання вимірювання. Для цього можна використовувати деякі характеристики програмної системи, які мають числове значення, або просто метрики.

Метрика – це кількісна оцінка певного аспекту системи, який може впливати на її складність. Кількість файлів проекту, кількість методів класу, кількість параметрів, що приймаються функцією, кількість таблиць у базі даних, кількість зв'язків, станів… Це лише кілька прикладів з багатьох можливих метрик, які використовуються для оцінки складності.

Велику кількість метрик можуть розрахувати статичні аналізатори коду. Вони можуть допомогти автоматизувати процес контролю складності і включати його в процес розробки. Ви самі можете придумати певну метрику і використовувати її для контролю складності програмного продукту.

Але є багато факторів, які також серйозно впливають на процес розробки і не стосуються безпосередньо програмного продукту.

Відсутність необхідних знань. Програмні системи не створюються на пустому місці. Розробник повинен мати певні знання та навички: мови програмування, шаблони проєктування, алгоритми, інструменти розробки тощо. Перелічувати можна довго. Простіший приклад: визначення площі трикутника для студента не становить труднощів. Для дитини 5 років це, швидше за все, нерозв'язна проблема.

Складність предметної галузі. Крім технічних знань, розробник повинен розуміти предметну область, у якій він працює. Дуже часто розробники спеціалізуються не тільки на певній мові програмування чи фреймворку, але й на певній предметній області. Мені розповідали про розробника, який займається виключно інтеграцією платіжних систем. Не розуміючи області, яку автоматизує ваша програмна система, ви не зможете створити правильно працюючий і надійний програмний продукт.

Привнесена складність

Привнесена складність. Одні й ті самі завдання можна вирішити по-різному, одну й ту саму програму можна написати багатьма способами.

Порівняйте ці вирази:

```
if(isCartEmpty != false)
if(isCartEmpty)
if( isCartEmpty )
```

У першому випадку ми маємо справу з привнесеною складністю. Вираз 1 важче зрозуміти, але він робить те ж саме, що й вирази 2 і 3. Я постійно стикаюсь з випадками, коли значну кількість програмного коду можна видалити, не завдаючи шкоди працездатності програми. Усі люди мислять по-різному і для деяких абсолютно нормально писати штучно ускладнені конструкції, які інші

люди розуміють з труднощами. Існує також інтелектуальний снобізм, коли розробник свідомо використовує більш складну конструкцію, не переймаючись тим, наскільки вона зрозуміла його колегам. Але про це в наступних розділах книги.

Вираз 3 сприйняти ще легше, тому що в ньому простіше виділити окремі частини. Саме тому одноманітність коду, застосування стандартів і форматування дозволяє зменшити складність. Ви, швидше за все, захочете подивитися знову на вирази, щоб краще зрозуміти, про що йдеться. І це також приклад привнесеної складності, але вже для цього тексту.

Висновки

Когнітивне напруження, що виникає у людини, яка має справу з програмною системою, залежить від багатьох факторів, як об'єктивних, так і суб'єктивних, що залежать від знань та інтелектуальних здібностей конкретної людини. Тим не менш, деякі фактори, що впливають на складність, можна оцінювати, використовуючи кількісні характеристики системи що розробляється — метрики.

---

[1] Гра престолів https://uk.wikipedia.org/wiki/%D0%93%D1%80%D0%B0_%D0%BF%D1%80%D0%B5%D1%81%D1%82%D0%BE%D0%BB%D1%96%D0%B2

[2] Grady Booch, Object-Oriented Analysis and Design with Applications

[3] George_M._Whitesides, Toward a science of simplicity https://www.ted.com/talks/george_whitesides_toward_a_science_of_simplicity

Комп'ютер у нашій голові

Два сорти мила? Це занадто складно для мене!
Приписується Альберту Ейнштейну

Конкуренція операційних систем і технологій «подарувала» необхідність створювати один і той самий застосунок під різні платформи. Звісно, це не є обов'язковим, але дає певні переваги. Як би там не було, розробнику в таких умовах доводиться враховувати наявність різних середовищ, у яких виконуватиметься написаний ним код.

Проте така ситуація була завжди. Код, написаний для комп'ютера, «виконується» в мозку програміста. Якщо не враховувати цю особливість, розробляти ПЗ стає вкрай важко. На цій ідеї побудована обфускація коду — приведення його до вигляду, коли при збереженні початкової функціональності код перетворюється на нечитабельний для людини вигляд.

Ось два приклади коду, які роблять одне й те саме. Для комп'ютера вони еквівалентні. Для людини ж зрозуміти, що робить другий код, набагато складніше, ніж перший.

Приклад 1:

```
if(number > 0) {
number++;
console.log(number);
}
```

Приклад 2:

```
   eval(function(p,a,c,k,e,r){e=String;if(!''.replace(/
^/,String)){while(c--)r[c]=k[c]||c;k=[function(e){return
r[e]}];e=function(){return'\
\w+'};c=1};while(c--)if(k[c])p=p.replace(new   RegExp('\
\b'+e(c)+'\\b','g'),k[c]);return         p}('2($1>0){$1+
+3.4($1)}',5,5,'|number|if|console|log'.split('|'),0,
{}))
```

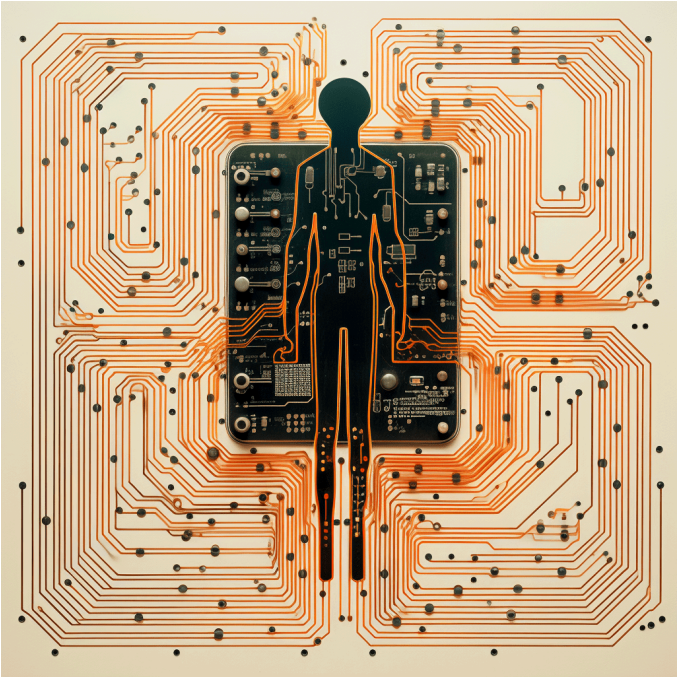
По суті, комп'ютеру байдуже, що напише програміст, головне — щоб виконувалися формальні вимоги мови програмування.

Будь-який дурень може написати код, зрозумілий комп'ютеру. Хороший програміст пише код, зрозумілий людині.

Мартін Фаулер

Людина, яка читає код, подумки виконує його. І зрозумілий людині код — це такий код, результат роботи якого буде ясний без значних розумових зусиль. Код, який важко зрозуміти, називають **нечитабельним**.

Але що означає нечитабельно? Давайте подивимося на здатність людини сприймати, запам'ятовувати і обробляти інформацію. Ось кілька фактів, що дають уявлення про людину як обчислювальну машину:



Невелика оперативна пам’ять. Максимальний обсяг уваги, кількість об’єктів, які людина може утримувати в короткочасній пам’яті, становить не більше чотирьох [2]. У фокусі уваги вона може зосередитися не більше ніж на двох об’єктах, за умови, що вони об’єднані в один блок [3].

Одноядерний процесор, повільне перемикання контекстів. Нормальна людина мислить однопоточно, і перемикання від однієї діяльності до іншої часто призводить до помилок, а також знижує темп роботи.

Низька тактова частота. Кількість логічних і арифметичних операцій за секунду сильно обмежена. Наприклад, скажіть, чому дорівнює цей логічний вираз:

((true && !false) && (!true || false))

Невелика пропускна здатність комунікаційного інтерфейсу. Обсяг інформації, що сприймається за одиницю часу, сильно обмежений.

Є цікаве дослідження, що стосується швидкості кодування інформації різними природними мовами [4]. Мови мають різну інформаційну насиченість складу. Однак і кількість складів, що вимовляються за одиницю часу, також різна. Дослідження показало приблизно однакову швидкість передачі інформації для різних мов, і вона становить 39 біт/сек. Виходить, що інформаційна пропускна здатність є характеристикою відділів людського мозку, відповідальних за обробку мови.

«…мінімальний час, необхідний свідомості для того, щоб відрізнити один сигнал від іншого, становить приблизно 1/18 секунди. Легко підрахувати, що за одну секунду свідомістю обробляється близько 126 біт інформації, за хвилину — 7560, а за годину — майже півмільйона. Якщо вважати, що людина щодня не спить шістнадцять годин на добу, то за життя тривалістю 70 років ця кількість складе близько 185 мільярдів біт інформації, куди входить кожна наша думка, спогад, почуття і дія. Це число здається величезним, але насправді це не зовсім так. Межі можливостей людської свідомості можна продемонструвати на такому прикладі. Для засвоєння змісту усної мови людині необхідно обробляти близько 40 біт інформації в секунду. Враховуючи, що наш межа становить 126 біт в секунду, можна зробити висновок, що зрозуміти мову трьох людей, які говорять одночасно, теоретично можливо, але тільки в тому випадку, якщо вдасться позбутися всіх інших думок і відчуттів. Ми вже не зможемо визначити, з яким виразом говорять ці люди, зрозуміти, чому вони говорять саме це, і помітити, у що вони одягнені.» [5]


ЭЛЕКТРОНИКА МК 61
ГРД
Вкл
F
ШГ
ШГ
В/О
С/П
К
БП
ПП
7
8
9
4
5
6
1
2
3
0
ВП
СХ

Якщо порівнювати людину за цими параметрами з ЕОМ, то людина програватиме навіть програмованому калькулятору, створеному багато десятиліть тому.

Однак не все так просто. Навряд чи б людина вижила багато тисяч років тому, якби всю інформацію, що надходить до неї, вона обробляла б із описаною вище швидкістю.

У книзі «Думай повільно, вирішуй швидко» Д. Канемана описано два основних механізми роботи мозку: автоматичний (Система 1) і такий, що вимагає мислення (Система 2). Описуючи обмеження вище, мова йшла якраз про Систему 2, яка відповідає за логічне мислення. Саме вона працює в першу чергу, коли програміст пише код.

Що ж із Системою 1? Ця система з вражаючою швидкістю розпізнає образи і дозволяє виконувати засвоєні дії. Її величезна перевага — це малі витрати енергії в процесі мислення. Весь мозок при своїй вазі приблизно 1,5 кг споживає 25% отримуваної з їжі енергії. Тому, щоб зменшити енергетичні витрати, наш мозок прагне більшість завдань віддати Системі 1. Система 2 включається тоді, коли завдання для мозку нестандартне.

Однак Система 1 не дуже акуратна, у випадку коли вона не може дати точну відповідь на поставлене завдання, вона намагається підібрати щось найбільш схоже. Як наслідок, якщо при цьому не включається Система 2, виникає помилка.

Ось приклад з книги Канемана. Спробуйте швидко вирішити наступне завдання. Ви купили в магазині биту і м'яч, заплативши при цьому 1 долар 10 центів. Бита на 1 долар коштує більше м'яча. Скільки коштує бита? Запишіть відповідь, яка першою спала вам на думку.

З великою ймовірністю ви помилилися, і відповідальність за цю помилку несе Система 1, підібравши найбільш

підходящу з її точки зору відповідь. Якщо ви спокійно подумаєте над рішенням, включивши Систему 2, то відповідь швидше за все буде правильною.

Незважаючи на те, що в програмуванні в першу чергу бере участь Система 2, досить багато допоміжної роботи виконує Система 1. Наприклад, вона відповідальна за те, щоб виділити блоки програмного коду. Якщо код відформатований звично для вас, ви легко можете зосередитися на тому, що саме він робить. В іншому випадку, Система 1 відключається, оскільки не може знайти знайомий шаблон. Зростає навантаження на Систему 2, і, як наслідок, неформатований код стає набагато важчим для розуміння (складність зростає). Саме тому так важливі домовленості про стиль написання коду, коли ви працюєте в команді.

Багато психологічних експериментів показали, що будь-яке додаткове навантаження на мозок зменшує швидкість і погіршує якість розумової діяльності людини. Спробуйте задати складне логічне завдання людині, яка йде. Швидше за все, вона зупиниться, щоб обдумати відповідь.

Психологічні експерименти показали, що навантажена Система 2 стає більш «довірливою», критичність мислення знижується, і ми готові повірити у що завгодно. [1] Така властивість нашого мозку призводить до того, що при роботі зі складним кодом або діаграмою нам набагато важче побачити помилку або логічну неточність.

Мистецтво розробника полягає в тому, щоб мінімізувати цю складність. Простий приклад — назви змінних. Використовуючи більш довгі імена, ви додаєте навантаження при розпізнаванні, але маєте можливість точніше описати дані. Короткі назви сприймаються легше, але мозок повинен робити додаткові перетворення, щоб відновити сенс. Тому короткі імена змінних найчастіше зу-

стрічаються у вигляді i, j — це звичні для розуміння назви індексів, які прийшли з математики.

Саме тому короткі імена допустимі в обмеженому контексті, коли змінна існує в 2-3 рядках коду. Але якщо це 20-30 рядків, ви зіткнетеся з тим, що вам важче розуміти програму.

Назви також мають велике значення. Наш мозок повинен співвіднести назву з сутністю, до якої вона належить. Скорочення, якщо вони для вас незвичні, значно уповільнюють розуміння тексту. Те ж саме можна сказати і про різні назви однієї і тієї ж сутності; ваш мозок замість однозначної відповідності розпізнавання повинен задіяти механізм ідентифікації. Людині властиво мати якусь одну асоціацію, пов'язану з назвою.

Цією властивістю активно користуються маркетологи, намагаючись «завоювати» ваш мозок, асоціювавши назву продукту з виробником. Думаю, багатьом відома гра у вгадування. Коли на прохання швидко назвати предмет меблів, люди частіше назвуть стіл або стілець.

Ще гірше справа, коли за однією й тією ж назвою ховаються різні сутності. Найяскравіше це можна відчути при написанні SQL-запиту, що містить кілька таблиць, у кожній з яких є поле з однаковою назвою (наприклад, популярне поле ID).

Зустрічаючи слова, які легко переплутати між собою, Система 1 вибере для вас не найправильніше, а найзнайоміше. Це пояснює, чому люди дуже часто чують і бачать не те, що їм кажуть, а те, що вони готові сприйняти.

> *«Дух свободи», Олег Шупляк [6]*

https://shupliak.art/uk/gallery/hidden-images/spirit-of-freedom

Висновки

Тисячі років еволюції налаштували наш мозок на сприйняття візуальної інформації та швидкість реакції, проте абстрактне і логічне мислення — це відносно нова для мозку діяльність, з якою він справляється не дуже ефективно. Усе це слід пам'ятати при створенні програмного забезпечення, намагаючись звести до мінімуму фактори, що створюють додаткове когнітивне навантаження.

---

[1] D. Kanneman. Thinking fast and slow.
[2] Mind's Limit Found: 4 Things at Once. https://www.livescience.com/2493-mind-limit-4.html
[3] Can the Focus of Attention Accommodate Multiple, Separate Items? https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3197943/
[4] Different languages, similar encoding efficiency: Comparable information rates across the human communicative niche. https://www.science.org/doi/10.1126/sciadv.aaw2594

[5] Mihaly Csikszentmihalyi. Flow: The Psychology of Optimal Experience
[6] Сайт Олега Шупляка https://shupliak.art/uk/gallery/hidden-images/spirit-of-freedom

Трепанація черепа

Думка про те, що навіть наші найвищі душевні прояви можуть залежати від таких «низьких» речей, як емоційний фон і фізіологічний стан організму, одним здається абсолютно очевидною, іншим — безглуздою, обурливою та аморальною.

Олександр Марков, «Еволюція людини»

Описані Д. Канеманом «системи» — це певна спрощена модель, яка дозволяє краще пояснити роботу мислення, але не відображає реальну будову мозку. Давайте спробуємо краще розглянути, що саме знаходиться у нас в голові (і не тільки в ній).

Мозок — найенерговитратніший орган людського організму, і для того, щоб виправдати свою прожерливість, він повинен давати своєму власнику значні переваги.

Мозок може бути хорошим прикладом еволюційного розвитку — нові частини мозку формувалися з часом, доповнюючи існуючі відділи новою функціональністю і дозволяючи краще адаптуватися в різних ситуаціях.

Розвиток мозку — відповідь на складне навколишнє середовище. Наші предки знаходилися в обставинах, коли з одного боку потрібно було виживати в оточенні хижаків, а з іншого — знаходити їжу і відтворювати потомство. Існує гіпотеза, що поштовхом до збільшення розміру мозку було життя на деревах. Для того, щоб вирішувати проблему пересування по гілках, потрібен був серйозний «обчислю-

вач». Цікаво, що нашими морськими «братами по розуму» є восьминоги, у яких для виживання в складних умовах також сильно розвинулися інтелектуальні здібності.

Уявімо, що ми змогли поглянути на те, що знаходиться в людському черепі. Перед нами постане однорідна субстанція, віддалено схожа на голівку цвітної капусти або грецький горіх. Те, що ми побачимо перш за все, — це неокортекс (нова кора). Назва говорить сама за себе; ця частина мозку лише намічена у нижчих ссавців, а ось у людини складає основну частину (90%) кори головного мозку. Саме ця частина мозку відповідає за вищу нервову діяльність, до якої також можна віднести і розробку ПЗ. Товщина нової кори становить 2–4 міліметри, а площа — приблизно 2200 квадратних сантиметрів. Химерний зовнішній вигляд обумовлений тим, що природі було необхідно помістити її в черепну коробку, і таким чином довелося цю область мозку трохи «скомкати».

Вивчаючи мозок, вчені дійшли висновку, що в неокортексі можна виділити зони, відповідальні за різну діяльність. Наприклад, у лобній частині мозку знаходяться зони, відповідальні за особистість, судження, плануванння, вирішення проблем, говоріння та письмо (область Брока) тощо. В потиличній частині знаходиться область, відповідальна за зорове сприйняття, а в скроневій — за сприйняття звуків тощо.

Поля Бродмана — відділи кори великих півкуль, що відрізняються своєю будовою і функціями.

Така структура може давати цікаві побічні ефекти, наприклад, синестезію або змішання почуттів. Синестети, так називають людей з цими особливостями сприйняття, можуть відчувати кольори на смак, бачити звуки або чути форму. Відбувається це тому, що зони, відповідальні за сприйняття, знаходяться поруч і можуть частково перетинатися. [1]

Мозок існував і до появи ссавців, але був значно простішим. Збільшення і ускладнення мозку додало дуже важливу **прогностичну** функцію. Сприймаючи вхідну інформацію, ми не просто реагуємо на неї, ми співвідносимо її з тією моделлю світу, яка сформувалася у нас з народження, і можемо передбачити те, що станеться в майбутньому.

Цей механізм працює як в автоматичному режимі, наприклад, йдучи сходами, ми точно знаємо, де уповільнити

ногу, щоб поставити її на сходинку, так і на більш високому рівні, знаючи набір фактів, ми можемо прогнозувати можливі варіанти майбутнього.

«Кора головного мозку людини особливо велика, тому має велику запам'ятовуючу здатність. Вона постійно складає прогнози того, що ви відчуєте, почуєте, побачите, причому ви цього не усвідомлюєте.» [2]

Поспостерігайте, як намагається ловити предмет дворічна дитина. Її дії на частки секунди запізнюються, тому що її мозок ще недостатньо навчений, щоб створити правильне передбачення про політ предмета і ловити його не в тому місці, де предмет знаходиться зараз, а в тому, де він буде через деякий час.

Структура неокортекса однорідна, з чого випливає, що мозок використовує універсальний алгоритм для обробки інформації, що надходить ззовні. Людям, які працюють з обчислювальною технікою, не потрібно пояснювати, що універсальний процесор завжди поступатиметься спеціалізованому в рішенні конкретного завдання. Якщо дуже сильно спрощувати, то неокортекс — це універсальний прогнозист, який постійно вибирає найбільш ймовірний варіант інтерпретації інформації з різноманіття можливих. Чим більше варіантів вибору, тим більша ймовірність того, що людина помилиться.

Проте неокортекс — це тільки частина мозку, хоч і найважливіша. Існують й інші частини мозку, які суттєво впливають на прийняття рішень.

Базальні ядра. Ця частина мозку може бути хорошою ілюстрацією механізмів роботи мозку, які проявляються на більш високому рівні у вигляді систем 1 і 2 Каннемана. Якщо ви навчалися водити машину, то, ймовірно, пам'ятаєте, наскільки напруженими були перші заняття. Ваші дії були повільними, і ви помилялися, але з часом ви

перестали замислюватися над тим, що натискати або повертати. Ця ділянка мозку, відповідальна за автоматизм дій, взяла управління на себе, і ви почали витрачати набагато менше енергії на дії по управлінню.

Схожий механізм лежить в основі багатьох когнітивних спотворень, коли замість того, щоб витратити більше енергії на пошук правильного рішення, мозок пропонує найбільш ймовірний варіант з набору наявних у пам'яті, таким чином заощаджуючи енергію.

Лімбічна система. Ця система включає кілька відділів мозку, однак важливо її згадати, тому що вона відповідає за емоції, а емоції, у свою чергу, суттєво впливають на прийняття рішень, а також на функціонування всього мозку.

Під впливом емоцій ми можемо робити вкрай ірраціональні речі. І це стосується дуже багатьох сфер нашого життя, включно з такою на вигляд неемоційною сферою, як розробка програмного забезпечення.

Привабливість тієї чи іншої технології на рівні відчуттів може зробити її предметом вибору, коли раціональні аргументи будуть говорити про зворотне.

Одному моєму знайомому дуже подобався брокер повідомлень; причину такого ставлення до програмного продукту я не знаю, однак це суттєво впливало на його роботу. У його технічних рішеннях брокер повідомлень був присутній навіть там, де це було не дуже доречно і раціонально необґрунтовано. Таку ж історію я часто спостерігаю щодо мікросервісів, ставлення до яких формується маркетологами провайдерів хмарних послуг.

Емоційний і фізичний стани тісно пов'язані, одне може впливати на інше, і, як наслідок, ваші, здавалося б, раціональні рішення будуть залежати від того, що з вами відбувається і що ви відчуваєте.

Вчені провели дослідження, підрахувавши статистику виправдувальних вироків у суді до і після обіду. Виявилося, що голодні судді виносили більш суворі рішення, ніж ті ж судді, але після прийому їжі. [3]

Емоційний стан безпосередньо пов'язаний з гормонами, які виробляє наш організм. Гормони ж, у свою чергу, впливають на роботу нашої нервової системи і мозку. Наприклад, під дією адреналіну ви краще запам'ятовуєте інформацію, але погіршується логічне мислення.

Може здатися дивним, але на прийняття рішення впливає не тільки мозок. У нашому організмі існує ентеральна нервова система, яка розташована в шлунково-кишковому тракті. Ця система не тільки продукує гормони, що впливають на наші рішення, але й безпосередньо пов'язана з мозком за допомогою блукаючого нерва. Цікавий факт: наші харчові вподобання залежать від мікрофлори нашого кишечника. Тобто на вибір того, що нам з'їсти, суттєво впливають мікроорганізми, що живуть в нашому череві. [4]

Висновки

Нервова система людини, включно з мозком, є надзвичайно заплутаним механізмом, на який впливають різні фактори. Його універсальність щодо оброблюваної інформації дає величезні можливості, однак за все потрібно платити, і ця плата – швидкість і точність вирішення специфічних завдань (наприклад, математичних обчислень).

З точки зору будови, людський мозок дуже далекий від комп'ютера. Якщо робота комп'ютера в першу чергу заснована на однозначних вхідних даних і обчисленнях, то мозок має справу з імовірностями і передбаченнями.

Одну і ту ж інформацію він може інтерпретувати порізному і, як наслідок, давати різноманітні відповіді.

«Дві пташки», Олег Шупляк [5]

https://shupliak.art/gallery/hidden-images/birds-of-a-feather-2009

Однак, якщо ми маємо справу фактично з постійним вгадуванням, то вгадувати набагато легше, коли обмежений набір варіантів. А краще за все, коли існує тільки один варіант. У цьому і проявляється феномен складності. Складність інформації збільшує набір її можливих інтерпретацій і тим самим знижує ймовірність прийняття «правильного» (найбільш відповідного в конкретних умовах) рішення.

[1] Вілейанур Рамачандран. Мозок розповідає. Що робить нас людьми.
[2] Джефф Гокінс, Сандра Блейкслей. Про інтелект.
[3] Вплив сторонніх факторів на судові рішення. https://www.pnas.org/doi/full/10.1073/pnas.1018033108
[4] Джулія Ендерс. Чарівний кишечник. Як наймогутніший орган керує нами.

[5] Сайт Олега Шупляка https://shupliak.art/gallery/hidden-images/birds-of-a-feather-2009

Крах вавилонської вежі

Як серцю виказать себе?
Як іншим зрозуміть тебе?
Ти думку висловиш — і вмить
Уже неправда в ній дзвенить.
Федір Тютчев, Silentium

Біблійна притча про те, чому не вдалося завершити найвеличніший будівельний проект, розповідає про те, що Бог, бажаючи зупинити роботи, позбавив учасників проекту можливості спілкуватися. В результаті вежа не була побудована, а у світі з'явилися сотні різних мов.

Напевно, у вас у житті були моменти, коли, пояснюючи щось іншій людині, ви були впевнені, що вона вас зрозуміла. Через деякий час виявлялося, що вас зрозуміли зовсім не так, як ви очікували. І розмовляли ви однією і тією ж мовою…

У чому ж справа? Чому так складно буває передати свою думку, як кажуть, «не вистачає слів», або зрозуміти, що говорить співрозмовник? Якщо задуматися, то у кожної людини своя власна «мова» - набір слів, до яких прив'язані конкретні образи і асоціації.

Можете провести простий експеримент: попросіть знайомих пояснити, що означає, наприклад, слово «коса», і порівняйте зі своїми думками, пов'язаними з цим словом. Хтось назве косою зачіску. Інший розкаже про намивну смугу суходолу на річці або озері. А від когось

ви почуєте щось про сільськогосподарський інструмент, тому що ваш знайомий любить порпатись на городі.

Отже, слово — це просто код для образу у вашому мозку. У кожного ця асоціація може бути своя, і якщо ви не узгодили розуміння слова, то ви вже розмовляєте різними мовами.

Те ж саме можна сказати й про абстракцію. Абстракція — це виділення найважливіших ознак об'єкта. Залежно від контексту, абстракція може бути зовсім різною. Наприклад, для бухгалтера автомобіль може представлятися як пробіг, залишкова вартість і рік випуску, для автомеханіка автомобіль — це набір деталей.

При розробці програм ми маємо справу саме з абстракціями — об'єкт у програмному коді має набір певних властивостей. Таблиця в БД має певні колонки. У бухгалтерській програмі ці властивості у об'єкта «автомобіль» будуть сильно відрізнятися від властивостей того ж автомобіля у програмі проектування.

Даючи назву чомусь, ми фактично визначаємо абстракцію. Ми явно чи неявно визначаємо умови, за якими об'єкт реального світу може називатися обраним нами словом. Наприклад, кажучи «мотоцикл», ми маємо на увазі два колеса і наявність двигуна.

До вибору назв необхідно підходити дуже серйозно. Справа в тому, що невдала назва може суттєво ускладнити комунікацію між учасниками проекту.

Ось проблеми, що виникають при виборі назви:

- використання загальних термінів.У проекті на початковому етапі розробки автоматизувався один бізнес-процес.Його реалізацію так і назвали «бізнес-процес».Коли згодом виникла потреба автоматизувати інші процеси, виникла проблема, тому що використання цього терміна щодо них викликало плутанину.

- використання одного терміна для позначення різних абстракцій.У цьому випадку може бути або неправильне розуміння сказаного, або доводиться уточнювати, що співрозмовник має на увазі.
- використання слів у значенні, відмінному від загальноприйнятого.Одного разу мені довелося зіткнутися з кодом, у якому терміном Controller позначався об'єкт, відповідальний за вибір даних із сховища.Оскільки у загальноприйнятому розумінні цей об'єкт відповідальний за прийом запитів до системи, довелося витратити додатковий час, щоб розібратися з кодом.
- використання абревіатур.Скорочення дозволяють швидше спілкуватися, але це тільки в тому випадку, якщо всі учасники проекту розуміють їхнє значення.В одному проекті я зіткнувся з тим, що дуже велика кількість термінів у проекті були абревіатурами.Справа в тому, що в попередній версії системи використовувалася БД, у якій було обмеження на довжину імен полів таблиць (4 символи).Ці назви використовувалися в програмному коді системи, їх було десятки.Створюючи нову версію ПЗ, використовуючи БД, яка вже не мала таких обмежень, розробники продовжували використовувати назви з трьох-чотирьох букв.Це було сущим пеклом для нових розробників, які прийшли на проект.

Не дивно, що у своїй книзі «DDD» Ерік Еванс окремо виніс розділ «Єдина мова» як ключовий момент для побудови складних систем. Створення такої мови для проекту дає кілька можливостей — покращує комунікацію між членами проекту, що прискорює і покращує її якість і зменшує кількість перетворень інформації від замовника до програміста. Якщо ви використовуєте цей підхід, то код, написаний програмістом, можна без особливих про-

блем показати представнику бізнесу і обговорити логіку його роботи. Код виглядатиме майже як текст звичайною мовою: [1]

```
public boolean accountIsDelinquent(Customer customer) {
    Date today = new Date();
    Specification delinquentSpec = new DelinquentInvoiceSpecification(today);
    Iterator it = customer.getInvoices().iterator();

    while (it.hasNext()) {
        Invoice candidate = (Invoice) it.next();
        if (delinquentSpec.isSatisfiedBy(candidate)) {
            return true;
        }
    }

    return false;
}
```

Однак про важливість термінології говорили ще до створення DDD: «...словник даних - центральне сховище абстракцій, що стосуються системи. Сама робота з ним допомагає виробити загальноприйняту і вичерпну термінологію, яку можна використовувати протягом усього проекту.» [2]

Таким чином, відсутність чіткої і зрозумілої всім мови проекту збільшує складність і тим самим уповільнює його реалізацію. І навпаки, створення вдалої термінології (яка відображає абстракції предметної області) може суттєво сприяти успіху проекту.

Проте найчастіше в проекті присутні кілька мов. Представники бізнесу мають свою термінологію, технічні спеціалісти – свою. У такому випадку постійно необхідно здійснювати переклад. І чим більший проект, тим більше «мов» може виникнути. Тому необхідно намагатися знаходити інваріанти, однаково зрозумілі всім учасникам комунікації.

В одному проекті спеціалісти аналітичного відділу в технічному завданні описували алгоритм розрахунку, використовуючи терміни з галузі розробки. Їм здавалося, що таке пояснення буде зрозумілішим для програміста. На жаль, терміни використовувалися неправильно, програмісти розуміли ТЗ по-своєму, і завдання кілька разів поверталося для виправлення. Проблему було вирішено після того, як алгоритм було написано із використанням мови SQL. І аналітики, і програмісти володіли цим інструментом, і в результаті вдалося однозначно висловити та зрозуміти інформацію.

Чути — це більше, ніж розуміти слова.

Приписується Карелу Чапеку

Як було описано в попередньому розділі, мозок розвивався як інструмент прогнозування. «У розмові ви часто можете передбачити (або, принаймні, думаєте, що можете) те, що ваш співрозмовник скаже далі. А часом ми слухаємо, але не чуємо: не вникаємо в сенс слів іншої людини, просто чуємо те, що нам хочеться почути. … Частково це відбувається тому, що люди часто використовують у розмові загальні фрази та вирази.» [3]

Скоріш за все, ви неодноразово стикалися з такою ситуацією. Людина, сказавши вам фразу, впевнена, що ви сприйняли її саме так, як їй хотілося б. Але ви цю фразу сприйняли зовсім по-іншому.

Тому, передаючи інформацію, намагайтеся отримати зворотний зв'язок. Попросіть людину переказати те, що вона почула від вас. І навпаки, почувши щось, переказуйте співрозмовнику іншими словами. Не повторюйте, а саме переказуйте, саме так ви можете створити у вашому спілкуванні зворотний зв'язок, що коригує спотворення інформації.

Висновки

Розробляючи проект, одним із ключових моментів, необхідних для досягнення успіху, є створення однозначної, зрозумілої всім учасникам проекту термінології. Це підвищує швидкість комунікації, а також зменшує ймовірність того, що інформація буде зрозуміла неправильно.

Потрібно розуміти, що кожне спотворення інформації при передачі її між учасниками проекту призводить до того, що через деякий час виникне необхідність вносити правки в проект. В результаті це може досить сильно впливати на терміни реалізації.

Просіть співрозмовника переказати своїми словами те, що він почув від вас. І навпаки, повторюйте почуту інформацію, але іншими словами. Ви будете здивовані, наскільки часто виявляється, що сказане вами чи співрозмовником було зрозуміло зовсім не так, як очікувалося.

---

[1] Ерік Еванс, Предметно-орієнтоване проектування (DDD)
[2] Граді Буч, Об’єктно-орієнтований аналіз і проектування
[3] Джефф Гокінс, Сандра Блейкслей. Про інтелект.

Чи проста простота?

Будь-який розумний дурень може зробити речі більшими, складнішими та жорстокішими. Потрібна геніальність — і багато мужності, щоб рухатися у протилежному напрямку. Ернст Ф. Шумахер

Чому ж досягнення простоти вимагає такої мужності та геніальності? Щоб зрозуміти це, давайте розглянемо звичайне побутове завдання, з яким стикався кожен із нас.

Кожен із нас робив прибирання в домі. Потрібно розсортувати розкидані речі, вирішити, в яку шафку або на яку поличку що покласти. Якщо речей багато, а шафок мало, то це звичне заняття змушує задуматися. Суть проблеми в тому, що ми стикаємося з певними обмеженнями — всі речі повинні бути розсортовані, щоб їх можна було легко знайти, але водночас помістилися в наявні шафки.

І якщо розкласти це завдання на складові, то ми стикаємося з необхідністю аналізу, класифікації та розстановки пріоритетів (ранжування).

Що таке класифікація? Це коли ви визначаєте якусь суттєву ознаку у групи об'єктів і групуєте або ділите їх відповідно до цієї ознаки. Наприклад, ви можете скласти разом усі шкарпетки. Або зібрати разом речі, в яких ви будете займатися спортом — штани, футболку та ті ж шкарпетки. І якщо ви визначили обидві ці ознаки, то у вас точно виникне питання: до чого відносяться спортивні шкарпетки? Не очевидно, правда?

Ранжування дозволяє впорядкувати предмети всередині групи. Наприклад, більш використовувані речі ви вирішите покласти зверху, а менш використовувані — засунути на нижню полицю.

І якщо підходити до питання прибирання з усією формальною строгістю, то виявиться, що потрібно:

- провести аналіз, виділити суттєві ознаки;
- класифікувати речі згідно з результатами аналізу;
- розставити пріоритети;
- вирішити протиріччя що виникли (наприклад, усі речі деякої групи не вміщуються у вибрану шафку).

По суті прибирання стане настільки складним, що ви можете відмовитися від нього і вирішити, що можна непогано жити і в речовому хаосі.

Розглянемо ще один важливий момент для розуміння, що таке простота. Простота — це антипод складності, і якщо для складності було сформульовано правило БІЛЬШЕ → СКЛАДНІШЕ, то для простоти можна написати протилежне твердження: МЕНШЕ → ПРОСТІШЕ.

Прибираючи, ви не зменшили кількість речей, ви просто згрупували їх так, щоб ними було легше керувати і знаходити. Вам не потрібно пам'ятати про всі конкретні речі, тепер вам достатньо пам'ятати, що шкарпетки лежать на певній поличці. Тобто все різноманіття ваших шкарпеток ви сховали за абстракцією і помістили у певну шафку з написом «Шкарпетки». Ви зменшили кількість інформації, яку потрібно тримати в голові.

Однак, як було написано вище, цьому передувала аналітична робота, основа наукової діяльності. І це важка розумова робота, за яку не кожен готовий взятися і ще менше людей здатні якісно її виконати.

Ви, звичайно, можете взяти очевидну ознаку, наприклад, колір, і відсортувати речі за кольором. Для деяких випадків це може працювати — червона сумочка і туфлі опиняться поруч, але загалом таке сортування навряд чи буде зручним у повсякденному житті.

А до чого тут розробка ПЗ, спитаєте ви? Справа в тому, що ще в далекому 1970 році доктор Вінстон Ройс виділив 2 суттєві стадії створення ПЗ: аналіз та кодування. [1]

Інакше кажучи, перед тим як писати код, вам потрібно гарненько подумати. З одного боку, частину аналітичної роботи повинні виконати аналітики, що пишуть вимоги і формалізують вимоги замовника, але це не звільняє програміста від аналітичної роботи. І якщо програміст не розуміє або не знає, як це робити, то через деякий час ми отримаємо складну систему. Те саме, як навряд чи можна назвати прибиранням сортування речей виключно за кольором. Я не дарма взяв для прикладу **колір**, як ми знаємо, однією з властивостей мозку є когнітивна лінь [2]. А відрізняти кольори — це природна функція мозку, що вимагає найменших зусиль.

Досить часто я задаю програмістам питання: "Що таке абстракція?" і, на жаль, дуже рідко отримую зрозумілу відповідь. Адже це основа програмування. Фактично, програміст створює і оперує абстракціями: об'єкти реального світу представлені у вигляді своїх спрощень (абстракцій) у програмному коді.

Користувач програмної системи представлений ім'ям, прізвищем та електронною поштою. У цьому випадку не важливий колір очей, політичні переконання і те, що він шульга. З усього різноманіття ознак ми взяли саме ці, тому що саме вони будуть використані при реалізації функцій системи. Отже, основа простоти — це вдало підібрані абстракції — ідеалізовані об'єкти, з обмеженим набором важливих для нас властивостей.

Правильно назвати – значить правильно зрозуміти.

Невідомий автор

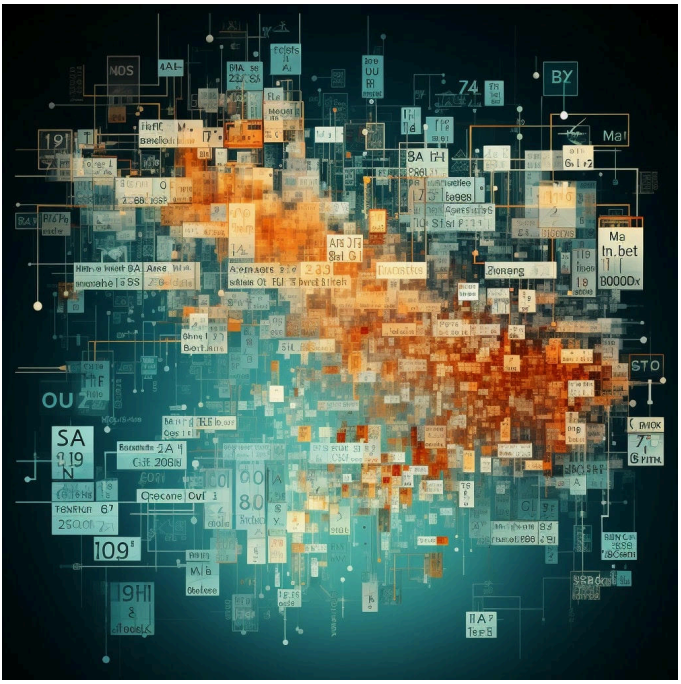
Край важливим для абстракції є її ім'я, назва. Майже як у детективному оповіданні Айзека Азімова «Що означає

ім'я?» [3]: прізвище відвідувача бібліотеки мало особливе значення для працівників бібліотеки і стало ключем до розкриття злочину.

У кожної людини своє уявлення про навколишній світ, але для того, щоб передати свою думку іншим людям, ми використовуємо мову. Саме частина мозку, відповідальна за мову, відрізняє нас від інших приматів, і саме ця частина мозку відповідальна за абстракції. [4] Фактично, використовуючи слова, ми використовуємо загальноприйняті абстракції, маючи на увазі саме ті суттєві властивості об'єкта, які ми хочемо передати словом. Тому імена та назви так важливі при розробці ПЗ, наприклад, створення Єдиної мови є частиною методології DDD [5], призначеної для побудови складних програмних систем.

Наявність слів, мови, що використовується розробниками (навіть якщо ви не використовуєте DDD), це частина процесу створення будь-якої програмної системи. Назва — це і є спрощення абстракції до одного слова. Є і зворотний бік медалі: якщо ви не можете дати чітке визначення слову, то, швидше за все, ви не розумієте, що цей термін означає.

Звісно, простота досягається не лише абстракціями, а й розбиттям системи на шари та частини. Кількість частин системи на кожному рівні також має бути обмежена, щоб не бути надто складною для сприйняття. Але знову-таки, вдале розбиття є результатом аналізу.



Хорошим прикладом описаного вище може служити періодична таблиця елементів – результат напруженої розумової роботи видатного вченого-хіміка Дмитра Менделєєва. Її створення значно спростило вивчення хімічних елементів і навіть дозволило припустити існування ще не відкритих на момент її створення елементів.

Висновки

Якщо поняття «складність» передбачає розумове напруження на межі можливостей людини, то «простота» передбачає очевидність і відсутність цих зусиль. Однак останнє досягається напруженою попередньою роботою: аналізом, класифікацією, створенням абстракцій та назв, побудовою вдалих моделей. Якщо не приділяти постійну

увагу перерахованим вище моментам, то кількість інформації та її різноманітність у програмній системі може збільшитися настільки, що розробники не зможуть оперувати цією інформацією.

---

[1] Dr. Winston W. Royce. Managing the Development of Large Software Systems
[2] Деніел Канеман. Думай повільно… вирішуй швидко.
[3] Вілейанур Рамачандран. Мозок розповідає.
[4] Айзек Азімов. Що означає ім'я?
[5] Eric Evans. Domain-Driven Design: Tackling Complexity in the Heart of Software.

Любителі кастомів

Різноманітність — мати насолоди.
Бенджамін Дізраелі

Думаю, з кожним траплялася ситуація, коли ви вдягали незвичайний наряд, і оточуючі оберталися в вашу сторону, проводили поглядом або якимось іншим чином проявляли свій інтерес. Така ж історія повторюється, коли на дорозі з'являється рідкісний спортивний автомобіль або мотоцикл, зібраний спеціально на замовлення власника. Багато хто бажає привернути до себе увагу, користується цим.

У чому ж причина такого ефекту? Справа в тому, що у людини присутні емоції страху та інтересу, які активуються, коли вона стикається з чимось новим або незвичайним.

Новий предмет може бути небезпечним, і тому на нього варто звернути увагу. Або навпаки – це нова можливість для отримання ресурсів. Наш мозок автоматично реагує на об'єкт і продукує гормони що відповідають ситуації – допамін, кортизол, адреналін. Концентрація гормонів і визначає наш емоційний стан, який, як описувалося в розділі про мозок (Трепанація черепа), суттєво впливає на прийняття, як нам здається, раціональних рішень.

Але згадаємо про тему нашої книги. У програмній індустрії існує феномен під назвою «велосипед». Це випадок, коли завдання, яке має загальноприйняте і перевірене рішення, реалізується новим, незвичайним способом. Замість того, щоб взяти готову бібліотеку або скористатися рішенням колег, програміст пише все сам. Цей термін став називним і найчастіше асоціюється з програмістами-початківцями.

Таку поведінку можна пояснити браком знань та досвіду. Якщо порівнювати з більш кваліфікованим колегою, то розробник високої кваліфікації в першу чергу буде шукати готове, перевірене рішення.

Індивідуальні речі, речі, зроблені на замовлення, коштують дорожче аналогів, вироблених у масових кількостях. Ситуація у сфері розробки ПЗ особливо не відрізняється. Справа в тому, що крім витраченого на написання коду часу, ми можемо зіткнутися з непродуманістю рішення, необхідністю тестування, а також помилками на реальному робочому середовищі. Але крім перерахованого вище, «індивідуальні» рішення суттєво збільшують складність. Адже інший фахівець, зіткнувшись з таким кодом, буде змушений розбиратися з ним, тим самим витрачаючи час і свої розумові ресурси. І навіть фахівець, який написав код, з часом забуває тонкощі свого рішення. Таким чином, бажання створити щось своє, привнести

елемент новизни у свою роботу, може вилитися у суттєві проблеми.

Мотивація фахівця може бути заснована саме на емоції цікавості. Приймаючи рішення, розробник буде керуватися бажанням спробувати щось нове, а не найефективніше вирішити завдання.

Крім цікавості, існує ще одна причина виникнення нових рішень – недостатньо добре аналітичне мислення розробника, а саме неспроможність побачити загальні властивості завдань, що вирішуються, і запропонувати не просто рішення для цього завдання, а рішення для класу завдань.

Якось я спостерігав розробку модуля реєстрації клієнта. Особливістю було те, що система мала працювати в різних країнах, а процес реєстрації для різних країн мав деякі відмінності. Розробники не придумали нічого кращого, як скопіювати код, створивши кілька окремих сервісів – по сервісу для кожної країни (!). Рівень дублювання коду зашкалював. Нічого дивного, що через деякий час команду попросили залишити проект, а нові розробники зіткнулися з великими проблемами при супроводі цього коду. Варто зазначити, що це були не новачки.

Існує й інша крайність, яка має назву “overengineering”. Ця проблема виникає, коли розробники навпаки надмірно абстрагуються від рішення одного конкретного завдання. Замість простого рішення вони створюють код, який вирішує клас завдань. Програмісти вигадують гіпотетичні ситуації, які можуть виникнути в майбутньому, тим самим значно ускладнюють код і уповільнюють час розробки. Багато стартапів зазнали краху саме з цієї причини. Коли замість простого, але не універсального рішення розробники намагалися створити цілу платформу, і в результаті не змогли вчасно реалізувати продукт, необхідний для бізнесу.

Висновки

Розробка ПЗ побудована на компромісах. Уміння балансувати між використанням існуючого коду та написанням нового значною мірою впливає на успіх проекту.

Писати новий код чи повторно використовувати існуючий є частиною кваліфікації розробника. Часто на рішення впливають емоційні фактори, брак аналітичних здібностей, брак знань та вміння прогнозувати шлях розвитку проекту.

Кожне з таких рішень має свої наслідки, що впливають на вартість та успішність проекту. Важливим фактором у прийнятті рішень є досвід: фахівець, який уже вирішував подібні завдання, з більшою ймовірністю зможе запропонувати оптимальне рішення.

Можна запропонувати такі правила. Якщо обсяг коду для вирішення завдання відносно невеликий, а саме завдання є очевидним – пріоритет слід віддавати написанню коду. Для завдань вищої складності слід пошукати аналогічні рішення, поспілкуватися з колегами, звернутися до

бібліотек та фреймворків, створених для вирішення подібних завдань.

Маючи у руках існуючі варіанти, порівняти набір факторів: відповідність даному завданню, продуктивність, обсяг коду, наявність документації, час реалізації, тощо. Результат порівняння повинен дати відповідь, що буде ефективніше в даній ситуації.

А що якщо?

Не дай вам бог жити в епоху змін!
Приказка[1]

При розробці програми завжди потрібно пам'ятати про те, що існує багато шляхів, якими може пройти логіка, а також про кількість станів, які може приймати програмна система.

Якщо ви займалися програмуванням, то, швидше за все, стикалися з відладчиком. Одне з завдань цього інструменту – показати стан вашої програми в момент виконання. Знаходячись у певній точці виконання вашої програми, вам необхідно розуміти, які значення знаходяться у змінних. Найчастіше відладчик використовують для пошуку помилок у програмі. Тобто ми маємо таку ситуацію – програміст написав код, який має працювати, але щось йде не так, і для того, щоб зрозуміти, що саме, програміст використовує відладчик. Виходить, що можливостей розробника недостатньо для того, щоб спрогнозувати поведінку програми, яку він сам же і написав.

Ми не будемо враховувати складність алгоритму і сконцентруємося на змінних. Уявімо, що ми точно знаємо, яке саме значення знаходиться в змінній у будь-якому місці (і в будь-який час) програми. Це означає, що ми легко зможемо сказати, як поведе себе код. Найпростіший випадок – це константи. Константи оголошуються один раз і не змінюються протягом усього часу виконання програми.

Деякі мови програмування (наприклад, Erlang) взагалі не містять змінних. У програміста є можливість лише створювати константи. Винятковим випадком є програмування паралельних потоків виконання.

Якщо дані можуть змінюватися одночасно з кількох місць, то не обійтися без примітивів синхронізації, блокувань і навантажувального тестування. Саме тому

розробники Erlang вирішили повністю відмовитися від можливості зміни даних у змінних. У разі паралельного програмування відладчика вже недостатньо для того, щоб розібратися, як працює (або чому не працює) програма.

У моїй практиці були випадки, коли програма переставала працювати саме під навантаженням – виникали такі збіги обставин і комбінації роботи кількох потоків виконання, які дуже важко було передбачити або знайти в процесі звичайної розробки.

Взагалі паралельне програмування, на думку багатьох розробників, – одна з найскладніших тем у розробці. Завданням даної книги не є розбір цієї теми. Головне розуміння, що найпростіший спосіб розібратися зі складністю в цій області – проектувати застосунок так, щоб виключати (наскільки це можливо) конкурентний доступ до даних і тим самим усувати проблеми, які при цьому можуть виникнути. Саме так вчинили розробники Erlang.

Проте більшість мов програмування дають можливість модифікувати змінні. У багатьох випадках це зручніше і лаконічніше, ніж робота з константами. Для вирішення проблем, які можуть виникнути при паралельній розробці, збільшена складність самої мови. Наприклад, у мові Java існують слова final і volatile: перше для того, щоб обмежити можливість змін, друге для того, щоб вказати, що змінна може змінюватися в різних потоках.

Чим більша область видимості змінної, тим на більшу частину програми змінна може вплинути. Це одна з причин, чому використання глобальних змінних небажане і вважається поганою практикою.

Змінні можуть містити не лише конкретні значення, але і посилання на більш складні структури, наприклад, об'єкти. При цьому значення даної змінної (посилання на об'єкт) може залишатися незмінним, але сам об'єкт

зміниться. Для того щоб виключати подібні ситуації, був введений термін «іммутабельність». Іммутабельним називається об'єкт, який не може бути змінений після його створення. Використання таких об'єктів допомагає зменшити кількість станів, і ми можемо бути впевнені, що об'єкт у кожній точці програми буде саме таким, яким він був при створенні.

Ще один приклад, коли проявляється складність, пов'язана зі зміною стану, це програмування рекурсії. Рекурсія часто дозволяє створювати елегантний і ефективний код, проте написання рекурсивного коду – непросте завдання саме через те, що вхідні дані для коду з кожним викликом можуть змінюватися, і тим самим необхідно враховувати, як поведе себе такий код при різних вхідних даних і які саме дії слід зробити, щоб отримати потрібний результат.

Існує таке поняття, як «чиста функція». Це функція, яка при однакових значеннях аргументів повертає один і той самий результат, а також не має побічних ефектів. Це також про зменшення можливих станів програми.

З проблемами зміни стану програміст стикається і на більш високих рівнях, наприклад, при розробці або використанні мережевих сервісів. Уявіть, що ви інтегруєте в свій програмний продукт платіжну систему, яка дозволяє платити за товар або послугу. Ви вказуєте номер платіжної карти і суму і відправляєте запит до системи. Якщо вам пощастить, то ви отримаєте відповідь від системи, що операція пройшла успішно. Однак можлива втрата зв'язку, і відповідь про успішність операції отримана не буде.

У цій ситуації ви не будете знати, перераховані гроші чи ні, чи потрібно повторювати операцію чи ні. Ви знову стикаєтеся з набором можливих станів. Для того щоб спростити життя, було введено поняття «ідемпотентність». Дія є ідемпотентною, якщо її багаторазове повторення еквівалентно одноразовому. У нашому випадку з платіжною системою, якщо система є ідемпотентною, ми можемо просто повторювати запит, поки не дочекаємося отримання відповіді. Правда, до даних по карті і сумі необхідно буде додати ідентифікатор запиту, щоб можна було розрізнити повторення операції і новий платіж.

З проблемою станів ви зіткнетеся і при роботі з базою даних (БД). І мова не про багатокористувацький доступ, який частіше за все доводиться враховувати. Йдеться про зміну структури БД (якщо БД має структуру) і про

зміни самих даних. Програма може додавати, змінювати і видаляти дані. Саме операції зміни і видалення можуть викликати багато головного болю у розробника. У разі зміни структури даних в БД на великому обсязі даних, таке завдання може стати нетривіальною. Операції зміни і видалення можуть призвести як до порушення цілісності даних, так і до неможливості реалізувати вимоги замовника, які виникають в міру того, як розвивається система. У моїй практиці був випадок, коли замовник вирішив видаляти деякі дані, які на його думку були тимчасовими і не були потрібні для роботи. Через кілька місяців виникла задача зібрати статистику роботи системи, але виявилося, що завдання не може бути вирішене через відсутність даних, які видалялися протягом всього попереднього часу роботи.

Висновки

Крім алгоритмів, структур даних, знань предметної області розробнику ПЗ доводиться стикатися з різноманіттям станів програмного продукту. Таке різноманіття впливає на розуміння того, як функціонує програмний продукт і може створювати значну складність при розробці.

[1] Щоб ти жив у часи змін

Герой-одинак

Нещасна країна, яка потребує героїв.
Бертольт Брехт

Комп'ютерний геній-одинак широко розрекламований кіноіндустрією. І в житті схожі персонажі зустрічаються досить часто, настільки часто, що їм варто присвятити окремий розділ. Розвиток програмної індустрії принципово не відрізняється від розвитку інших галузей.

Спочатку галузь розвивають і рухають вперед ентузіасти, продукт може бути створений однією людиною.

З часом галузь ускладнюється, технології, знання, вміння та час однієї людини вже не вистачає. Перемагати починають команди, які з часом стають великими компаніями.

Такий же цикл найчастіше проходить і окремий програмний продукт. Поширена ситуація, коли перша версія успішного продукту створюється однією людиною.

Ось деякі прояви цього типажу:

Вільний художник. Працюючи в невеликій, але успішній компанії, ми розширювали свій штат, шукали програмістів. До нас прийшов хлопець без резюме та якоїсь внятної історії роботи. Він чудово виконав тестове завдання і був прийнятий на роботу. Це був чудовий технічний фахівець, проте через деякий час виявилися «особливості». Він займався виключно тим, що було йому цікаво. Він міг сидіти цілодобово, розбираючись у проблемі, яка була йому цікава. Але було дуже важко вмовити його виконати необхідну роботу, яка на його думку була нудною.

Вундеркінд. Це молодий чоловік, який має обширні технічні знання та гострий розум, на додачу до великого самолюбства. Зазвичай у компанії такий працює пару місяців, добре виконуючи індивідуальні завдання, але

важко вписуючись у команду. Через деякий час йому стає не цікаво, він вважає себе недооціненим і йде.

Батько-засновник. Це програміст, який стояв біля витоків програмної системи. Його початкова реалізація виявилася досить успішною, система розвивалася, на проект приходили нові розробники. Його головна цінність для продукту в тому, що тільки він знає, як все працює. Однак цими знаннями він ділиться неохоче.

“ **Я знаю, як краще”.** Це люди, переконані у своїй правоті. Вони не вміють і не хочуть слухати опонента. “Є моя думка і неправильна”. Такі люди не будуть спілкуватися з колегами з приводу обраного рішення. Швидше за все, ви будете бачити вже готовий результат їхньої роботи, а внести зміни в рішення задачі буде досить складно.

Всіх цих людей об’єднує невміння бути частиною колективу розробки. Вони самовпевнені і погано спілкуються з оточуючими. Періодично вони можуть виконувати корисну роботу, але частіше в перспективі приносять більше витрат, ніж користі: код, написаний цими програмістами, доводиться з часом переписувати; рішення зрозумілі тільки їхнім творцям, а те, що зроблено, далеко не завжди відповідає очікуванням замовника.

Ось що про таких людей написано в книзі “Ідеальний керівник” [1]:

“Що робити з блискучим фахівцем, наприклад, незамінним інженером, який нікому не довіряє і нікого не поважає? Він вимагає довіри і поваги до себе, але не рахується з іншими. (Таких геніїв чимало.) Ставтеся до нього як до мавпи. Нехай він сидить у своїй клітці, а коли вам знадобиться інформація, дайте йому банан і отримаєте те, що вам потрібно.”

Важливо розуміти, що такі люди при роботі в колективі часто приносять більше шкоди, ніж користі, і від них або

потрібно позбавлятися, або знаходити роботу, що вимагає мінімальної кількості комунікацій. Наприклад, пошук причини проблем що виникли або створення прототипів.

Чим раніше ви зрозумієте, що перед вами така людина, тим менше проблем у вас виникне в майбутньому.

Але чому ми заговорили про цю проблему? Справа в тому, що подібні учасники процесу розробки можуть зробити істотний внесок у складність проекту. Важливо розуміти це на початковому етапі розробки, поки наш герой не наробив лиха.

Як було сказано вище, процес розробки ПЗ — це процес передачі інформації між його учасниками. Інформація змінюється за формою, але повинна залишатися незмінною за змістом. Наявність у ланцюзі передачі ланки, яка внесе велику кількість "шуму" або серйозно спотворить сенс, може внести додаткову складність і суттєво вплинути на успіх проекту. Більш детально ці моменти будуть розглянуті в розділах "Як виміряти програміста" та "Абстрактна модель процесу розробки ПЗ".

Висновки

Колектив, який займається розробкою, суттєво впливає на складність і, як наслідок, на успіх проекту. Підбираючи учасників, слід звертати особливу увагу на вміння спілкуватися, прислухатися до оточуючих і вміння змінювати свою точку зору залежно від наведених аргументів, продуктивно взаємодіяти в команді.

---

[1] Іцхак Адізес. Ідеальний керівник. Чому їм не можна стати і що з цього випливає.

Лакмусові папірці

Якщо у вашій процедурі 10 параметрів, ймовірно, ви щось упускаєте.
Алан Перліс

За роки розробки програм у мене з'явився цілий список ознак, які вказують на наявність якоїсь проблеми. Якої саме – потрібно розбиратися, але попередження про те, що щось пішло не так, вже корисно саме по собі.

Це все одно як червоний лакмусовий папір показує наявність кислоти, але яка саме кислота, потрібно з'ясовувати додатково. Нижче приведені деякі з цих ознак. Вони стосуються як загальних моментів розробки, так і безпосередньо написання коду.

Не знаю, як назвати

Мова — один з ключових засобів боротьби зі складністю. Тільки правильно назвавши якусь річ, ми розуміємо, що це є насправді. Якщо в програмі зустрічаються загальні слова, які не є доповненням до слів, що мають конкретне значення, знайте, той, хто писав цей код, не найкращим чином розібрався в проблемі.

Навіть якщо як такої проблеми немає, то погано підібрані слова дають когнітивного навантаження – ви повинні тримати в голові, до чого саме загальний термін відноситься. А тепер давайте подивимося приклади.

Helper – це слово найчастіше вказує на погане проєктування, особливо якщо ви використовуєте ООП. Воно

означає: «я не знаю, як назвати цей набір дій, тому я зберу їх в один клас і назву його допоміжним». Чесно кажучи, важко пригадати випадок, коли використання такої назви було дійсно виправданим. Ось яку думку з цього приводу можна знайти на Stackexchange:

“Клас Helper — це менш відомий запах коду, який виникає, коли програміст виділяє деякі розрізнені операції, що часто використовуються та намагається зробити їх такими що використовуються повторно, об’єднавши їх в одну неприродню групу.” [1]

Слова parameter, config, container, data доречні тільки в тому випадку, якщо вони використовуються локально і максимально компактно, коли ви чітко розумієте, що за ними стоїть.

Наявність цифр у назвах

У програмному коді все, що більше одного – багато. Найчастіше використання цифр можна замінити структурою даних, наприклад масивом. У цьому випадку ви чітко розумієте, що маєте справу з кількома схожими об’єктами, і ваш код легко розширюється. Якщо ж ви розумієте, що масив використовувати не можна, то тоді ви повинні замінити цифри на значущі назви. Уявімо, що в коді ви зустріли змінні user1 та user2. Варіант 1: ми маємо справу з кількома однотипними об’єктами, і це можна замінити на щось на зразок users = new Array(). Якщо ж так зробити не можна, то швидше за все змінні можна перейменувати. Наприклад, employee і manager.

Короткі, малозначущі імена (x, j, tmp і т.д.) з великою областю видимості

Чим менша область, в якій зустрічається змінна, тим коротшим може бути її ім'я. Довжина імені – це також внесок у складність; чим довше ім'я, тим важче його сприймати. Але суміш малозначущих імен, які легко сплутати (наприклад і та j), призводить до додаткового розумового навантаження і, відповідно, до помилок.

Коментарі

Наявність коментарів означає, що в коді присутня неявна інформація, яку потрібно розшифрувати, прокоментувавши код. Часто коментар можна видалити, зробивши код більш зрозумілим. Якщо ви бачите коментар, спробуйте переписати код так, щоб коментар став непотрібним або перетворився на назву функції чи змінної.

Наприклад:

```
   // If user is inactive for 30 minutes, log out
   if                    (Duration.between(lastActivity,
Instant.now()).toMinutes() > 30) {
   session.logout();
   }
```

Код можна перемістити у функцію, ім'я якої збігається з коментарем:

```
   private static final long SESSION_TIMEOUT_MIN = 30;
   if (isSessionExpired(lastActivity)) {
   session.logout();
   }
   private boolean isSessionExpired(Instant lastActivity)
{
   return Duration.between(lastActivity, Instant.now())
   .toMinutes() > SESSION_TIMEOUT_MIN;
   }
```

Відладчик

Якщо при роботі з кодом ви активно використовуєте відладчик, це означає, що вам не вистачає можливостей вашого мозку для розуміння того, що робить код. Якщо код написали ви самі, це однозначна ознака наявності неконтрольованої складності і проблем у майбутньому.

Ваші колеги, працюючи з таким кодом, матимуть ще більше проблем.

Виправданням використання відладчика може бути робота з реверс-інжинірингом і спроби зрозуміти код, написаний не вами. Слід розуміти, що це не агітація відмовитися від корисного інструменту. Просто ви повинні усвідомлювати, що необхідність використання відладчика є серйозним сигналом, що ви недостатньо добре розумієте, що саме робить код. Існує кореляція між кваліфікацією розробника і тим, як часто він використовує відладчик. Чим вища кваліфікація, тим менша потреба у відладчику. Перефразовуючи відомий вислів Сунь Цзи, можна сказати: «Початківець спочатку пише код, а потім намагається зрозуміти, як він працює. Досвідчений розробник спочатку продумує, що буде робити код, і тільки потім його пише.» [2]

Заміна процесу мислення технологією

Ця ознака занадто загальна, тому спробую розшифрувати її на конкретних прикладах. На поточний момент технології не можуть бути панацеєю від проблем у розробці, пов’язаних із поганим проектуванням.

Тести

Використання тестів – досить тонкий момент. З одного боку, у складних системах, які часто модифікуються, наявність тестів обов’язкова. Але не можна перекладати

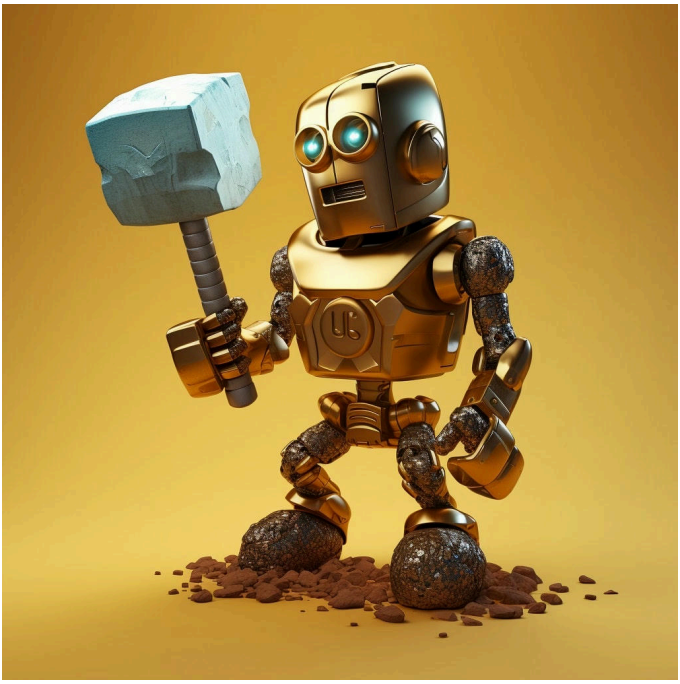
відповідальність за те, що має робити програма, з програміста на тести.

Приклад: розробник, використовуючи методологію TDD, пише тест, а потім пише код. Все йде чудово, поки розробник не каже: «Ура! У мене є тест, я можу писати будь-яку дурницю, тест мене виручить і покаже, що я десь помилився».

З цього моменту відповідальність за код програміст переклав на тест. Якщо виникає помилка, в коді робиться правка, яка спрямована не на виправлення логіки роботи, а для того, щоб задовольнити тест.

У майбутньому може статися ситуація, не передбачена тестом, або зміняться вимоги до системи. Код, написаний цим програмістом, швидше за все доведеться викинути, тому що він не вирішує бізнес-задачу, а задовольняє логіку тесту. Модель предметної області, реалізована в коді, тільки зовні схожа на те, що потрібно. Тест стає фетишем, підміною реальних завдань, які має вирішувати код.

На жаль, з подібною ситуацією мені неодноразово доводилося стикатися у реальних проєктах.



ООП, NoSQL, мікросервіси та інші золоті молотки.

Досить часто з'являються нові технології, застосування яких обіцяє великі виграші: швидкість розробки, гнучкість, масштабованість. Але технології – це лише інструменти, і як у кожного інструмента, у них є переваги та недоліки. Все залежить від контексту їх використання. Бездумне застосування технології майже завжди призводить до проблем.

Якщо ви чуєте: «У нас зараз не дуже хороший код. Але ми все перепишемо з використанням нової мови, мікросервісів, нового фреймворку» (можете підставити те, що ви чули від своїх колег) – не вірте. Люди, які створили програмну систему з проблемами, створять не менш проблемну систему в майбутньому, але використовуючи інші

технології. Технологія поки ще не може замінити розумову діяльність людини при вирішенні складних завдань.

Як приклад можу згадати проект, де було прийнято рішення перейти на новий фреймворк для вирішення проблем, що виникли через надмірну складність у поточному проекті. Через півроку проект на новій платформі став ще складнішим і мав не менше проблем, ніж попереднє рішення.

Скарги на нестачу ресурсів

Дуже часто можна почути: «У мене не було достатньо часу для якісного вирішення цього завдання». Але, думаю, ви навряд чи чули, щоб хтось сказав: «Я недостатньо кваліфікований фахівець і тому створив погану систему». Те саме можна сказати, коли поганий код виправдовують недостатньою кількістю розробників. Магія розробки програмного забезпечення полягає в тому, що навіть за нестачі ресурсів, але добре подумавши, ви можете створити відмінне ПЗ. Іноді навпаки, нестача ресурсів підштовхує до більш якісного рішення.

Однак, якщо ви чуєте такі скарги, то, швидше за все, вам доведеться зіткнутися з проблемами в коді. А цей «лакмусовий папірець» вказує на рівень розробника, з яким ви маєте справу.

Проєктування класів

По цій темі написано багато літератури, і завдання книги не включає її глибоке обговорення. Я просто окреслю деякі ознаки, які часто вказують на проблемний код.

Наявність set, get в інтерфейсі класу — ця ознака поширюється на класи, що реалізують логіку застосунку. І мова

не йде про класи, які представляють фактично структури даних для обміну між модулями, DTO.

Наявність методів, які дозволяють отримати або встановити стан об'єкта, говорять про його зв'язки з оточенням. Чим більше таких зв'язків, тим складніше рішення. Логіка, за яку повинен відповідати обраний клас, розмазується по вашому застосунку. Якщо в інтерфейсі класу ви бачите ці методи, то у вас є гарний привід замислитися про те щоб перепроєктувати такий клас та його оточення.

Відсутність властивостей у класі. Якщо ви зустрічаєте клас, в якому присутні методи, але відсутні властивості, це досить чітка ознака того, що цей клас є просто набором процедур і функцій. Так само, як і попередній, це ознака процедурного підходу, незважаючи на використання мови, що підтримує парадигму ООП. Або вказівка на те, що реалізовано антипатерн — анемічна модель предметної області.

Виклик методу більше одного разу. Реалізуючи логіку застосунку, я намагаюся перевіряти, де викликаються як приватні методи, так і методи інтерфейсу класу. Хорошою ознакою є лише один виклик методу. Наявність кількох викликів може свідчити про дублювання логіки, яку можна виявити цією простою перевіркою.

Ви можете сказати, що необхідно застосовувати статичні аналізатори коду, однак у разі, якщо логіка дублюється неявно, то, на жаль, такий аналіз проблему не покаже. Під час написання коду слід прагнути до деревоподібної організації логіки програми. Такий підхід дозволяє дуже швидко визначити наявність помилки, а також значно спрощує сприйняття коду.

Отже, якщо метод викликається більше одного разу, то це може вказувати на проблеми проєктування.

Складність написання unit-тесту. Якщо під час написання unit-тесту виникають проблеми, то, швидше за все, ви маєте справу з погано спроєктованим модулем, який потребує переробки. Річ у тім, що під час написання такого тесту ви штучно створюєте оточення модулю. Якщо

у модуля велика кількість зовнішніх залежностей, то ви зіткнетеся з цією проблемою під час створення тесту.

Висновки

Людський мозок прагне мінімізувати розумову роботу. Ви не можете глибоко аналізувати кожен артефакт свого застосунку. Ви можете просто не помічати існування проблеми, поки вона не проявить себе на робочому оточенні. Існує безліч інструментів, що розширюють ваші можливості з аналізу коду, однак наявність простих маркерів проблем, які часто зустрічаються дозволить поліпшити якість цього аналізу. У цій главі наведено деякі з них, які я використовую сам. Бажаю і вам мати у своєму арсеналі подібний інструмент.

---

[1] StackExchange https://softwareengineering.stackexchange.com/questions/247267/what-is-a-helper-is-it-a-design-pattern-is-it-an-algorithm

[2] „Воїни-переможці спершу перемагають і лише потім вступають у битву; ті ж, хто терплять поразку, спершу вступають у битву і лише потім намагаються перемогти.“. Сунь Цзи

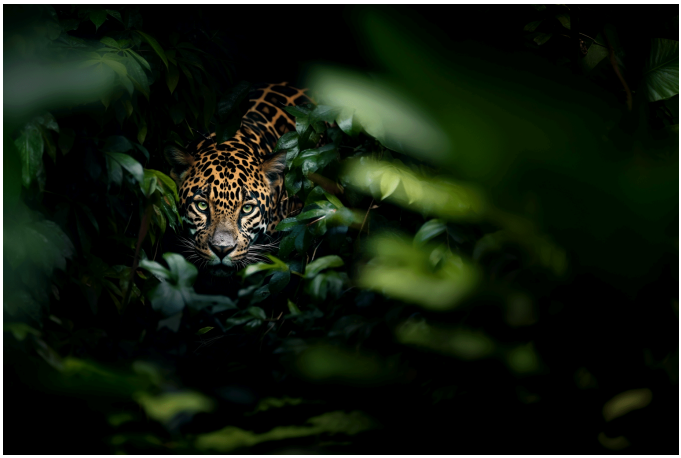
Так що ж робити? Способи зменшення складності

Про красу або трохи нейробіології

Краса є у всьому, однак не всім дано її побачити.
Конфуцій

Французький авіаконструктор і засновник Dassault Aviation Марсель Дассо любив повторювати:«Pour qu'un avion vole bien, il faut qu'il soit beau» — «Щоб літак добре літав, він має бути красивим». Фраза стала неофіційним девізом конструкторського бюро під час створення винищувачів Mirage та бізнес-джетів Falcon. [1]Подібне зауваження видається несподіваним від конструктора: він говорить про красу — тобто про емоції, а не про суху математику.

Емоції з'явилися у ссавців в ході еволюції як реакція на зовнішнє середовище для підвищення можливостей виживання виду. Емоції є несвідомою реакцією мозку на отриману інформацію. Якби наші предки свідомо аналізували інформацію, намагаючись зрозуміти, хто саме дивиться на них з лісу, замість того щоб миттєво відчути страх і залізти на дерево, вони б навряд чи вижили. Наш мозок несвідомо оцінює вхідні дані і дає миттєву пораду у вигляді викиду гормонів, який ми відчуваємо як емоцію.



Емоції запасені для різних випадків, у тому числі здивування і радість, коли людина стикається з чимось новим і не настільки страшним, як хижак. У потоці інформації наш мозок намагається розпізнати залежності, знайти зрозумілі йому структури. Реакцією на розпізнавання, яке не збігається з попереднім досвідом, є здивування. Коли ж розпізнавання не є тривіальним, ми відчуваємо задоволення.

“Ми так влаштовані, що любимо розгадувати загадки, і сприйняття набагато більше схоже на розв’язання задач, ніж думає більшість. Кожного разу, коли ми успішно вирішуємо задачу, ми отримуємо нагороду у вигляді вибуху задоволення. Навіть сам акт пошуку розв’язання задачі — будь то суто інтелектуальне завдання, як кросворд, або логічна задача, або суто зорове — приносить задоволення ще до того, як знайдено рішення.”“Кожного разу, коли мозок знаходить часткову відповідність, у ньому народжується маленьке Ага! (емоція впізнавання). Сигнал посилається лімбічним структурам винагороди, які, у свою чергу, викликають пошук додаткових більших Ага!” [2]

У всі часи рука об руку з поняттям краси йшли такі поняття, як гармонія, симетрія, пропорція частин, порядок [3]. Усі ці поняття тісно пов'язані з можливістю прогнозування, застосування попереднього досвіду на практиці.

Завданням наукової діяльності, діяльності інженера-розробника ПЗ є класифікація сутностей, пошук залежностей, виділення структурних компонентів, пошук порядку. Підсумком є створення теорії, побудова моделі, в яку вкладається спочатку отримана інформація. Чим простіша модель, що дозволяє описати складну предметну галузь, тим вона «красивіша», більш естетично значуща.

Чим складніше завдання, тим більші емоції ми відчуваємо від його вирішення.

У статті «Краса науки» [4] пропонується формула краси наукового відкриття. Думаю, що її можна застосувати й для оцінки технічних рішень, адже будь-яке технічне рішення є наслідком аналізу і опрацювання предметної області.

Естетична значущість (краса) = Спостережувана складність / Обсяг моделі

Або, якщо говорити іншими словами, то краса — це впорядкована, видима складність (якщо тлумачити складність як кількість і заплутаність інформації). У студентські роки у мене була подруга, студентка театрального вишу. Якось вона запропонувала разом сходити в театр. Для мене, що цікавився переважно фізикою та математикою, театр був річчю далекою і не дуже зрозумілою. Проте можливість провести час із дівчиною переважувала байдужість до цього виду мистецтва. Наші місця були недалеко від сцени, можна було легко розглянути всі деталі того, що відбувається. Спочатку було відверто нудно —

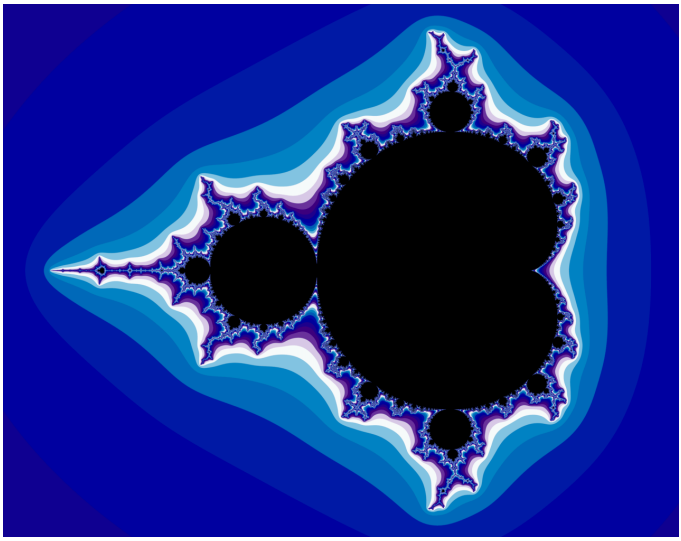
сюжет був відомий, а в усьому іншому я не розбирався. Проте під час вистави дівчина почала вказувати на деталі, які я до цього не помічав. Вона показала мені актора, що сидів на краю сцени, на якого я до цього не звертав уваги, пояснила тонкощі його гри, жести і міміку. Виявилося досить цікаво спостерігати за ним, хоча те, що він робив, просто створювало фон вистави. І коли, маючи нові знання, я почав помічати деталі гри інших акторів, вся вистава засяяла для мене новими барвами. Я побачив картину того, що відбувається, набагато глибше, актори діяли як частини єдиної складної, впорядкованої системи. Відкриту красу того, що відбувалося, і яскраві естетичні враження я добре пам'ятаю через десятки років. У квітні 2005 року аспірант Каліфорнійського технологічного інституту Вільям Коттрелл був засуджений до восьми років тюремного ув'язнення за пошкодження кількох автомобілів. Одним із доказів, що привели слідство до Коттрелла, стала ретельно нацарапана на позашляховику формула тотожності Ейлера. На суді він сказав: «Я знав теорему Ейлера з п'яти років, і її мають знати всі». Чим же чудова ця формула? У 1988 році математик Девід Веллс (David Wells) попросив своїх читачів назвати найкрасивішу математичну формулу. Більшість обрала все те ж рівняння, а в 2018 році вийшла книга математика Робіна Уїлсона під назвою «Новаторська тотожність Ейлера: найкрасивіша теорема математики» («Euler's Pioneering Equation: The Most Beautiful Theorem in Mathematics»).

$$e^{i\pi}+1=0$$

Але що ж такого унікального у цих семи простих символах? Річ у тому, що дана формула пов'язує фундаментальні математичні константи: основу натурального логарифма e, уявну одиницю i (корінь з −1) і число π. За кожним із цих чисел стоять цілі області математики, і такий зв'язок між різними областями викликає естетичні почуття у людей, які це розуміють. Більшість людей відчувають естетичні почуття при вигляді фракталів – нескінченних, але таких, що мають впізнавану людиною структуру об'єктів. Фрактальним є все що існує у природі – гори, хвилі в морі і берегова лінія, дерева і ліс, хмари, річки і кровоносна система організму. З розвитком комп'ютерів люди почали малювати математичні множини, що отримуються рекурсивним викликом досить простих функцій, наприклад:

$$z_{n+1} = z_n^2 + c$$

Малюнок нижче відповідає множині значень цієї функції яку називають множиною Мандельброта.



Множина Мандельброта з неперервним кольоровим градієнтом [5]

Фрактальну структуру має і код програмного забезпечення. Одні функції викликають інші; модулі програми складаються з менших модулів. І чим масштабніший програмний продукт, тим більше рівнів і взаємопов'язаних компонентів. Для того щоб створювати і розвивати такий продукт, розробник повинен усвідомлювати його структуру. Якщо таке розуміння відбувається на інтуїтивному рівні, то воно викликає емоційний відгук. Один з моїх перших учителів у сфері розробки ПЗ казав, що блок-схема програми, схема модулів і зв'язків повинна бути впорядкованою і, по можливості, симетричною. На його думку, це вказувало на її правильність і працездатність.

Прекрасне — це щось таке, що належить виключно смаку.

Іммануїл Кант

У своїй книзі «Лезо бритви» письменник Іван Єфремов висловлює думку, яку можна застосувати і до програмного забезпечення: «Краса — це найвищий ступінь доцільності,

ступінь гармонійної відповідності і поєднання суперечливих елементів у будь-якому пристрої, у будь-якій речі, будь-якому організмі. А сприйняття краси не можна ніяк інакше уявити, як інстинктивне.»Однак звідси випливає і суб'єктивізм краси. Взаємозв'язки, структуру можна усвідомити лише на основі певних знань і досвіду. Побачити красу в рішенні проблеми, відчути естетичні почуття, може тільки професіонал (або точніше, людина, яка має певний рівень знань). Щоб побачити красу в математичній формулі, потрібно бути математиком, інакше формула являє собою просто набір символів. Марно сперечатися з людиною щодо доцільності технічного рішення, якщо його підготовка і кваліфікація суттєво відрізняються від вашої. Йому буде надзвичайно складно змінити своє ставлення, засноване на попередньому досвіді. У деяких африканських племенах існують дивні з європейської точки зору звичаї у сфері краси, наприклад, у жінок племені Мурсі.

Дівчина з народу мурсі [6]

У сфері розробки програмного забезпечення ви можете зустріти не менш цікаві та різноманітні випадки “красивих” рішень.

Висновки

Краса, естетичні почуття щодо вирішуваної проблеми, є інтуїтивною мірою доцільності рішення. Інтуїція формується на основі попереднього досвіду і знань розробника. Дуже важливо в команді розробки досягти єдиного розуміння і ставлення до технічних рішень, щоб на інту-

їтивному рівні спростити розуміння системи, з якою ви працюєте. Читання літератури, навчання, спілкування з колегами дозволяють сформувати уявлення про розробку, які закладуть основу вашої інтуїції і поняття краси технічного рішення. Тим не менш, існують універсальні закони, пов'язані з групуванням, симетрією, порядком [2, с235], які викликають естетичні почуття і дозволяють спростити вирішення завдань.

Закінчити хочу цитатою Річарда Бакмінстера Фуллера: "Коли я працюю над проблемою, я не думаю про красу. Але коли завершую, і рішення виявляється некрасивим — я знаю: воно хибне."

---

[1] Dassault Celebrates 60 Years of Falcon Business Jets, As Advanced New Models Prepare to Take the Stage https://www.dassault-aviation.com/en/group/news/dassault-celebrates-60-years-of-falcon-business-jets-as-advanced-new-models-prepare-to-take-the-stage/
[2] Вілейанур Рамачандран, Мозок розповідає.
[3] Ред. Умберто Еко, Історія краси.
[4] М. В. Волькенштейн, Краса науки.
[5] Множина Мандельброта з неперервним кольоровим градієнтом https://en.wikipedia.org/wiki/Mandelbrot_set#/media/File:Mandel_zoom_00_mandelbrot_set.jpg
[6] Дівчина з народу мурсі https://uk.wikipedia.org/wiki/%D0%9C%D1%83%D1%80%D1%81%D1%96_(%D0%BD%D0%B0%D1%80%D0%BE%D0%B4)#/media/%D0%A4%D0%B0%D0%B9%D0%BB:Mursi_Tribe_(7936031138).jpg

Згадуємо математику

Математика запитали: Як приготувати чай?
Беремо чайник, наливаємо в нього воду, ставимо на вогонь, чекаємо, поки закипить, кладемо заварку в чашку, заливаємо її окропом, чекаємо кілька хвилин - чай готовий.
А якщо у нас вже є чайник з окропом? Елементарно. Виливаємо з нього окріп і зводимо завдання до попереднього.
Старий математичний жарт

Люди стикалися зі складними задачами задовго до появи комп'ютерів. Математики були серед перших, хто використовував методи зменшення складності у своїй роботі.

Якщо нова задача схожа на ту, що була вже вирішена, то доцільно спиратися на наявні результати, уникаючи повторення всього шляху з початку. Цей підхід настільки поширений при розв'язанні задач і доведенні теорем, що став предметом анекдотів.

Аналогічна стратегія застосовується і в розробці програмного забезпечення. Індустрія постійно розвивається, надаючи можливості для повторного використання бібліотек, модулів і пропонуючи готові рішення типових задач.

Попри очевидні переваги цього методу, багато розробників його ігнорують. Я часто чую думку, що через швид-

кий розвиток IT-сектора немає сенсу вивчати літературу старше 2-3 років, оскільки технології застарівають.

Щодо конкретних технологій це може бути вірно, але загальні поняття — такі як методи проєктування, архітектурні патерни, алгоритми — залишаються актуальними незалежно від мови програмування чи технології.

Щоб стати експертом у своїй галузі, необхідно спиратися на накопичений досвід і знання.

Таким чином, головна перевага використання готових рішень — це економія часу. Це особливо актуально при використанні бібліотек і компонентів. Однак готові архітектурні рішення можуть у короткостроковій перспективі ускладнити розробку, але в довгостроковій — покращити якість продукту. Тому перед вибором методики рішення необхідний глибокий аналіз поточних і майбутніх вимог проєкту. Те, що спочатку здається відповідним рішенням, з часом може стати його протилежністю.

Як приклад можу згадати історію, коли компанія вирішила швидко запустити новий проєкт, спираючись на свій існуючий продукт у тій же предметній галузі. Цей продукт успішно працював вже два роки і мав як обширну кодову базу, так і накопичений технічний борг.

Без особливого аналізу задачі було вирішено клонувати продукт (включаючи код і структуру бази даних) та модифікувати його під новий проєкт. Для роботи над новим проєктом була сформована нова команда розробників, до якої входив і я. Незабаром стало ясно, що проєкти насправді сильно відрізнялися. Більша частина коду і структури БД вимагали змін. Замість того щоб отримати готове рішення, новий проєкт успадкував усі проблеми старого. Кожен, хто мав справу з модифікацією застарілого коду, зрозуміє, в якій скрутній ситуації опинилися розробники. Більшу частину часу доводилося витрачати не на створення нових функцій, а на виправлення проблем, отриманих у спадок.

Висновки

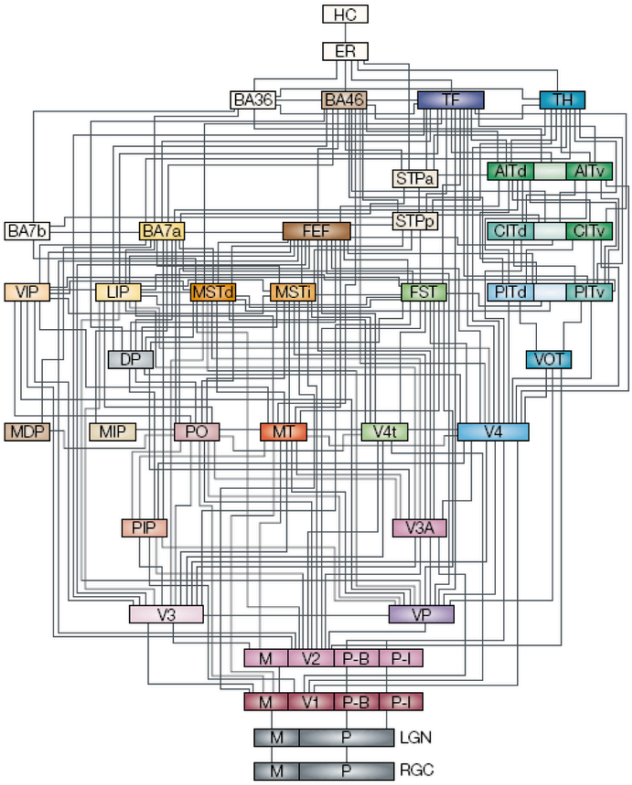
Повторне використання готових рішень дійсно може зменшити час, витрати і складність проєкту. Але такий підхід вимагає обдуманого застосування і глибокого аналізу переваг і недоліків. Особливо це важливо, коли зміни в рішенні можуть викликати складнощі, як в описаному прикладі.

Аналізуючи можливе рішення, слід враховувати його гнучкість у майбутньому. Наприклад, якщо ви вибрали певний фреймворк і адаптували під нього всю логіку ПЗ, ви фактично стаєте його заручником. Зміна фреймворка в майбутньому може стати трудомістким завданням. На відміну від цього, окремі компоненти, які інтегруються через інтерфейси, представляють менше ризиків у плані залежності.

Найважливіше почуття

Краще один раз побачити, ніж сто разів почути. Прислів'я

Для визначення того, в чому особливо сильний наш мозок, звернемося до історії того, як він формувався. Як згадувалося раніше, розвиток інтелекту – це відповідь на складне навколишнє середовище. Щоб знайти їжу, уберегтися від хижака і розмножитися, у людини повинні були розвинутися різні органи чуття: нюх, слух, зір. Найбільш «далекобійним» з них є зір, і, як наслідок, це орган чуття, який надає найбільший обсяг інформації. Вважається, що 80% інформації людина отримує саме через органи зору. Однак отримання інформації з зовнішнього світу – це лише перший крок. Не менш важливо її інтерпретувати. Процес «бачення» відбувається в мозку, і саме у людини розвиток зорових областей мозку досяг свого піку. Значна частина мозку відповідальна за зір, і в мозку існують десятки областей, пов'язаних з обробкою зорової інформації.

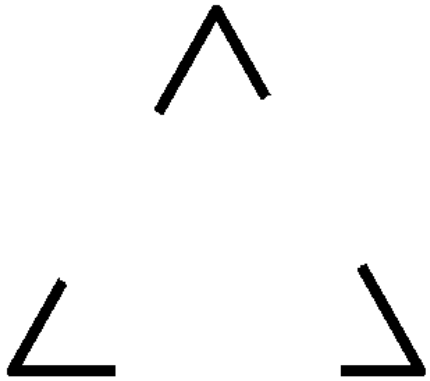


Діаграма Felleman і Van Essen (1991) анатомічної зв'язності шляхів зорової обробки. [1]

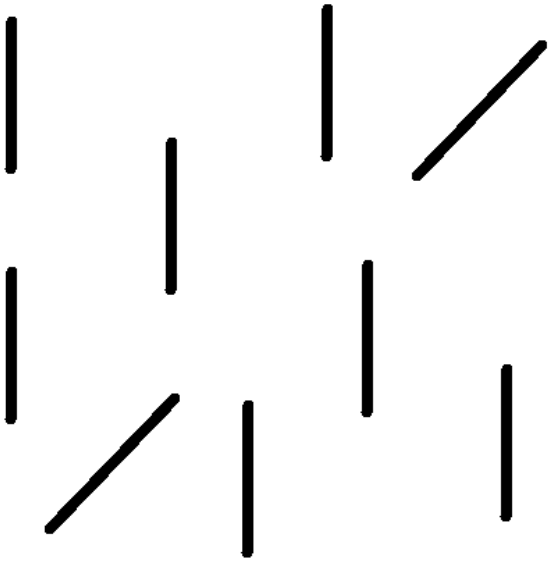
Обробка зорових сигналів відбувається на підсвідомому рівні. Зорові зони пов'язані з лімбічною системою, що відповідає за емоції. Ще до розпізнавання об'єкта і усвідомлення побаченого, мозок продукує емоційну реакцію на видимий предмет. Запам'ятайте цей факт, і пізніше побачимо, як його можна використовувати в ході вашої діяльності як розробника ПЗ. Наше сприйняття налаштоване на розпізнавання найбільш значущих для виживання об'єктів. У мозку існують окремі області, відповідальні за розпізнавання облич, рухів, форм, кольору тощо. При

цьому розпізнавання відбувається помітно швидше і з меншими зусиллями, ніж якби воно здійснювалося на свідомому рівні без використання цих спеціалізованих зон. Одним із ключових завдань для виживання було розпізнавання об'єктів навколишнього світу за частковою інформацією. Наприклад, було необхідно мати можливість побачити хижака, схованого в листі. Таким чином, у мозку сформувалися механізми, що дозволяють на основі часткової інформації сприймати об'єкт як цілісний, а також виділяти окремі частини об'єктів і об'єднувати їх.

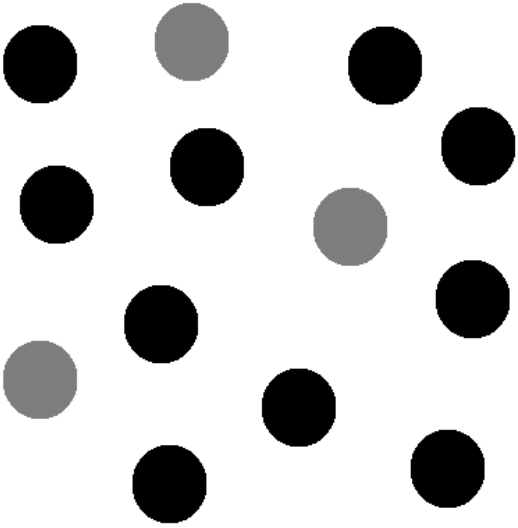
Розрізнені елементи, які припускають продовження контуру, будуть згруповані разом.



Нахилені лінії практично «кидаються» в очі, їх дуже легко виділити серед вертикальних ліній.



Кола іншого кольору чітко виділяються серед таких самих за формою чорних кіл.



Візуальні шаблони в програмному коді

Наш мозок ефективно навчається розпізнаванню різних візуальних шаблонів. Чим частіше ми стикаємося з певними типами зображень, тим швидше і легше ми їх впізнаємо. Це спрощує сприйняття візуальної інформації. Ці можливості мозку активно застосовуються в інструментах розробки. Наприклад, код програми може бути підсвічений різними кольорами, що робить його сприйняття більш інтуїтивним і зрозумілим.

```
class NodeGoto extends Base  {
    public const PRIORITY=1;
    public const NODE_PREFIX="goto_";

    protected $label;
    protected $one_time=false;
    protected $counter=false;

    public function __construct(array $parameters) {
        parent::__construct($parameters);

        $this->label=$parameters[INode::P_LABEL];
```

Форматування коду відповідно до загальноприйнятих стандартів і домовленостей дійсно значно полегшує його сприйняття. Я пам'ятаю, коли тільки починав кар'єру програміста, я надавав перевагу спочатку відформатувати чужий код згідно з моїм звичним стилем, перш ніж зануритися в вивчення його логіки. Багато розробників на інтуїтивному рівні усвідомлюють важливість стандартів форматування коду, навіть якщо не можуть чітко пояснити причину цієї важливості. Деякі мови програмування, такі як Python, у свою специфікацію включають вимоги до форматування коду. І якщо ви звикли програмувати, наприклад, на Java, і вирішите ознайомитися з кодом на Python, то, ймовірно, ви будете сприймати цей код повільніше. При цьому у вас можуть виникнути неприємні відчуття через незвичний стиль коду.

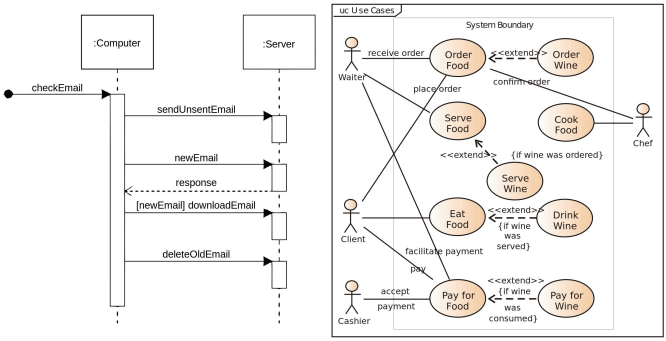
Не кодом єдиним

Розробка програмного забезпечення – це набагато більше, ніж просто написання коду. Під час розробки проєкту всі його учасники стикаються з такими речами,

як описи завдань, документація та інші матеріали. І тут ми можемо використовувати особливості нашої зорової системи для оптимізації сприйняття інформації, а саме:

- Розбивайте текст на абзаци по кілька речень, щоб виділити окремі логічні блоки.
- Створюйте виразні заголовки і робіть їх помітними, використовуючи різні кольори і розміри шрифту.
- Виділяйте ключову інформацію за допомогою різних стилів тексту, таких як жирний, курсив або підкреслення.

Більша частина роботи розробника пов'язана з текстом — чи то код, чи то документація або описи завдань. Але навіть найкраще написаний текст може бути неправильно інтерпретований читачем, і це може відрізнятися від початкового наміру автора. Не кажучи вже про те, що деякі люди можуть мати труднощі зі сприйняттям великих обсягів тексту. У таких випадках графічна візуалізація може стати справжнім порятунком. Презентації — гарний приклад цього. Якщо ви коли-небудь готувалися до публічного виступу, то, ймовірно, створювали слайди, які дублюють або доповнюють вашу промову, роблячи її більш наочною і зрозумілою для слухачів. Так само графічні елементи в матеріалах проєкту можуть покращити розуміння текстової інформації, активуючи при цьому різні частини мозку. На інтуїтивному рівні текст і графіка взаємодіють, посилюючи сприйняття і допомагаючи коригувати розуміння на ходу. Не випадково був створений UML (Unified Modeling Language) — мова, що пропонує обширний набір діаграм для опису різних аспектів проєкту.

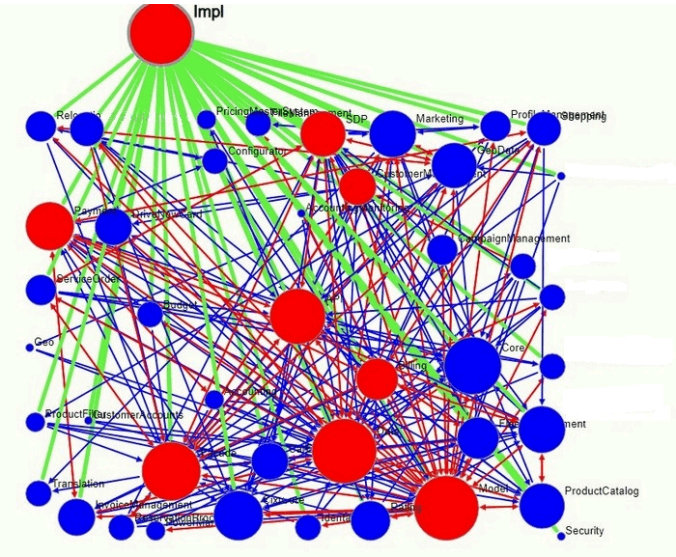


Приклади UML діаграм [2] [3]

Зворотний зв'язок

Графічне представлення можна використовувати не тільки для розробки, але й для контролю за процесом розробки і функціонування ПЗ. Представляючи дані у графічному вигляді, ви значно прискорюєте процес сприйняття і аналізу інформації. Наприклад, Grafana — програмна система візуалізації даних. На етапі розробки контроль здійснюється переважно на рівні коду, можливо, окремих метрик (наприклад, розмір методу або кількість рівнів вкладеності). Однак часто архітектору важливо мати загальний погляд на проєкт, і без візуалізації того, що вже реалізовано, зрозуміти проєкт буває вкрай складно. Деякий час тому я займався розробкою системи, що дозволяє візуалізувати програмну структуру проєкту для підвищення якості продукту і контролю за архітектурою проєкту. Ключовим інструментом аналізу було графічне представлення коду, модулів і зв'язків між ними. Інтерактивний інструмент дозволяв відобразити граф залежностей між модулями. Якщо модулі мали взаємну залежність, то такий зв'язок відображався червоним кольором, чим більше модулів посилалося на даний модуль,

тим більшим колом він відображався. Якщо модуль залежав від великої кількості інших модулів, то такий модуль підсвічувався червоним кольором. За бажання можна було виділити модуль і його залежності. Ось так виглядали зв'язки в проєкті, що розроблявся кількома десятками програмістів протягом п'яти років без належного архітектурного контролю.



Висновки

Досить велику частину мозку займають області, відповідальні за обробку і розпізнавання візуальної інформації. Незважаючи на те, що розробка ПЗ – це в першу чергу робота з текстами, не варто нехтувати можливостями значної частини мозку. Еволюційно сформувалися зони, що обробляють візуальну інформацію дуже ефективно. Зважаючи на особливості сприйняття, проєктну докумен-

тацію і код можна оформляти так, щоб вони краще сприймалися розробниками.

Доповнюючи текстову та аудіоінформацію візуальною, можна суттєво знизити ймовірність непорозуміння. А візуалізуючи наявні дані проєкту, можна значно підвищити швидкість і якість аналізу.

---

[1] Діаграма Felleman і Van Essen https://www.hms.harvard.edu/bss/neuro/bornlab/lab/figures/FVE_cnx.html
[2] The Sequence diagram of UML https://en.wikipedia.org/wiki/Sequence_diagram#/media/File:CheckEmail.svg
[3] A UML use case diagram for the interaction of a client (the actor) within a restaurant (the system) https://en.wikipedia.org/wiki/Use_case_diagram#/media/File:Use_case_restaurant_model.svg

Мова як робочий інструмент

Програмувати треба з використанням мови, а не на мові.
Стів Макконнелл, "Досконалий код"

Приблизно 75-150 тисяч років тому людина набула лінгвістичних навичок. Як це сталося, ніхто точно не знає. Деякі вчені припускають, що мова виникла еволюційно як засіб комунікації і з часом набула функції мислення. Є й протилежна думка: «Згідно з гіпотезою Стівена Джея Гулда, мова виникла з системи, яка дала нашим предкам більш складний спосіб мисленно уявляти світ і спосіб представляти самих себе в цьому уявленні» [1].

У нашому мозку формується уявлення про зовнішній світ, яке дозволяє моделювати і передбачати події у зовнішньому світі. «У нашому мозку зберігається модель світу, яка постійно порівнюється з реальністю» [2]. Частиною цієї моделі є інваріантні уявлення про об'єкти, тобто і звук, і картинка, і запах ведуть наш мозок до однієї і тієї ж асоціації.

Наприклад, побачивши птаха або почувши його спів, ми розуміємо, що цей об'єкт можна назвати словом «птах». Крім властивостей об'єкта зовнішнього світу, у нашому мозку зберігаються слова, що відповідають цьому об'єкту. Існує і зворотна залежність: чуючи слово «птах», ми можемо відтворити безліч асоціацій. Таким чином, слова дозволяють сформувати модель зовнішнього світу.

Розробляючи програмне забезпечення, ми також створюємо модель, яка відтворює певну частину нашої реальності, і нічого дивного, що для цього використовується мова. У мов програмування відсутня комунікаційна функція; програміст, використовуючи ту чи іншу мову програмування, фактично думає на цій мові. Мова є робочим інструментом розробника.

«Якщо з усіх інструментів у тебе є тільки молоток, то в кожній проблемі ти побачиш цвях» [3]. Відомий факт, що програмісти, довго використовуючи певну мову програмування, продовжують створювати конструкції у стилі цієї мови, перейшовши на іншу мову, яка має більш просунуті можливості.

Розробник концепції структурного програмування Едсгер Дейкстра писав: «Практично неможливо навчити хорошому стилю програмування студентів, які раніше мали справу з BASIC; як програмісти вони ментально покалічені без надії на відновлення». Широко відоме й інше висловлювання: «Справжній програміст на Fortran напише програму на Fortran, будь якою мовою».

Більшість програмістів, які використовують одну мову, вирішують задачі, використовуючи конструкції саме цієї мови, навіть якщо існує більш оптимальне рішення. Все це призводить до надмірної складності та розміру створюваної розробником програми. Як наслідок, витрачається більше часу на розробку, виникає більше помилок, і сама розробка стає значно дорожчою.

Не дивно, що з розвитком IT-галузі з'являється безліч мов програмування, орієнтованих на вирішення певних класів задач. Як і у випадку з програмістом, який вирішує всі завдання однією мовою, невдалий вибір мовної платформи може суттєво вплинути на успіх програмного проекту.

Колись я був досить наївний, сподіваючись, що мені досить буде добре вивчити одну мову.

Приписується Давиду Хейнемейеру Ханссону, розробнику Ruby on Rails.

Думаю, ви помічали: у кожній професійній області є своя термінологія і свій сленг, які дозволяють більш лаконічно і точно передавати думки. Професіонали спілкуються між собою не лише звичайною розмовною мовою, використання спеціальних слів робить це спілкування більш ефективним.

Можна піти далі: символи та позначення, з яких складаються формули математики та фізики, представляють собою лаконічну і зручну для використання мову цієї галузі.

$$\int_{-\infty}^{\infty} e^{-a(x+b)^2}\,dx = \sqrt{\frac{\pi}{a}}.$$

Важко уявити, що було б, якби математики використовували слова природної мови замість формул.

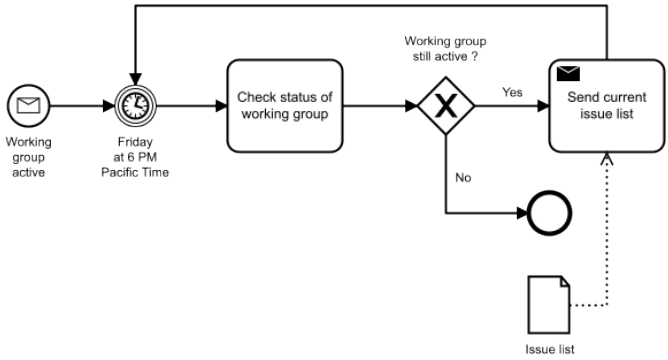
Створення такої нотації в цій предметній області значно підвищило ефективність роботи.

Той самий підхід можна використовувати і для написання коду. Вирішуючи завдання, ви спочатку придумуєте «мову», на якій рішення завдання буде простим і очевидним. Це може бути все, що завгодно: мова, нотація, графіка тощо (нижче я наведу кілька прикладів). На наступному етапі ви пишете код, який реалізує придумані вами конструкції.

Якщо придумане вами рішення охоплює цілий клас задач, то такий підхід може бути вкрай ефективним. Використовуючи ваше рішення, ви та ваші колеги будете

працювати значно швидше та якісніше. Фактично, ви створюєте інструмент, а потім вже за допомогою цього інструменту вирішуєте своє завдання.

— система умовних позначень для моделювання бізнес-процесів. Вона підтримується великою кількістю програмного забезпечення, дозволяє не лише документувати процеси, але й створювати додатки, які використовують BPMN-діаграми як вихідний код.



Приклад опису бізнес-процесу з використанням нотації BPMN [4]

Як описувалося вище, розробка ПЗ представляє собою процес перетворення інформації учасниками проекту. Чим менше змінюється вихідна інформація на шляху до використання її комп'ютером, тим менша ймовірність її спотворення, і тим менше часу будуть витрачати розробники. Тестування — це важлива частина розробки, і тести є частиною проекту. Нижче наведено приклад тесту, який дуже схожий на опис природною мовою.

```
Feature: ls
        In order to see the directory structure
        As a UNIX user
        I need to be able to list the current directory's contents
Scenario: List 2 files in a directory
        Given I am in a directory "test"
        And I have a file named "foo"
        And I have a file named "bar"
        When I run "ls"
        Then I should get:
        """
        bar
        foo
        """
features/ls.feature (END)
```

Приклад тесту фреймворку Behat

Деякий час тому мені довелося взяти участь у проекті автоматизації використання автопарку компанії. Передбачалося, що весь процес буде повністю автоматичним, автомобілі були оснащені спеціальним обладнанням для отримання доступу водієм, який зарезервував автомобіль.

Бронювання здійснювалося через спеціальний додаток, що розподіляє автомобілі автопарку. Одним із ключових компонентів системи мав бути модуль, що супроводжує сам процес поїздки – пошук вільного автомобіля, бронювання, повідомлення водія.

Алгоритм бізнес-процесу був досить складним, передбачав асинхронну взаємодію з різними системами, цикли очікування (водій міг не прийти, і тоді слід було звільнити автомобіль для іншої поїздки).

Розробник, який намагався реалізувати це завдання, став діяти в лоб, використовуючи знайому йому мову програмування загального призначення. Звичайно, він зіткнувся з серйозною проблемою, оскільки такі мови (Java, PHP, Python, C++ і т. п.) не розраховані на ефективну реалізацію тривалих періодів очікування та асинхронну взаємодію. Його код виглядав вкрай заплутаним і фактично не працював.

Насправді, основна частина логіки передбачала оркестрацію частин коду, які можна було б описати звичайною мовою програмування. Пошук відповідної для цього завдання бібліотеки, яка відповідала б усім обраним атрибутам якості, не дав успішного результату, і було прийнято рішення реалізувати власний опис процесу з використанням короткого набору операцій:

- дія;
- очікування;
- умовний перехід;
- безумовний перехід;
- обробка події.

Було створено невелику бібліотеку (кілька тисяч рядків), яка обробляла такий опис.

```
$process_nodes = [
    ["wait_next_day", "time" => "03:00"],
    ["if_end_condition",
        "then" => [
            ["goto_end"],
        ],
    ],
    ["send_on_time_message"],
    ["goto_wait_next_day"],
    ["send_event_message"],
    ["goto_wait_next_day"],
    ["end"],
];
```

Приклад опису бізнес-логіки

Завдання було успішно вирішено, функціональність була передана замовнику вчасно. Сама ж бібліотека виявилася досить зручною і була використана вже в іншому,

набагато більшому проекті. У цій системі одночасно працювало до півмільйона віртуальних процесів, реалізованих за допомогою цієї бібліотеки – обробка продажів, платежів, комунікація з користувачем та багато іншого. Таке відносно просте рішення дозволило заощадити сотні годин роботи програмістів.

В іншому проекті, було кілька десятків підключень до API сторонніх систем. Ті, хто стикався з подібними завданнями, знають, що кожна система має свої особливості, використовує різні формати запитів, протоколи та типи авторизації.

Було прийнято рішення описувати інтеграції за допомогою спеціальної конфігурації. Цей файл містив опис параметрів запиту, структуру запиту до зовнішньої системи, протокол, тип запиту тощо. Ця конфігурація читалася спеціальним додатком-сервером, який приймав стандартизовані запити від основної системи та транслював їх назовні. Таким чином, підключення до нової системи зводилося до створення конфігураційного файлу, а для виконання виклику використовувався стандартний клієнт.

Крім того, конфігурація містила приклади відповідей, які сервер міг надавати у тестовому середовищі, що значно спрощувало тестування та локальну розробку системи.

Однак найчастіше не потрібно винаходити щось нове, а достатньо ефективно використовувати набір існуючих мов і фреймворків. Наприклад, при розробці корпоративних систем мені часто вистачало мови загального призначення, фреймворку, який реалізує стандартний набір завдань, та мови SQL для роботи з даними.

Про SQL варто згадати окремо. Як згадувалося раніше, багато розробників схильні використовувати одну мову для всіх завдань, включаючи маніпуляції з даними. Дода-

ток витягує дані з сервера, і в циклах, застосовуючи різні умови та роблячи додаткові запити до БД, досягає бажаного результату. Застосування SQL для цих цілей може суттєво спростити та прискорити роботу застосунку.

Великий плюс у тому, що при написанні SQL коду вам потрібна лише консоль сервера БД, і не потрібно витрачати час на збірку та запуск додатка, що значно прискорює процес розробки. Також ви можете взаємодіяти безпосередньо з бізнес-аналітиком, який часто володіє SQL, що економить час на комунікації та тестуванні функціоналу.

Висновки

Вдале використання мови, правильний вибір нотації та чіткий опис завдання надають значні переваги у боротьбі зі складністю, що в кінцевому підсумку веде до покращення якості та підвищення продуктивності.

У своїй відомій статті «Срібної кулі немає» Фредерік Брукс стверджує: «Найбільший внесок у зростання продуктивності, надійності та простоти вносить все ширше використання мов високого рівня». Безліч мов програмування було створено з певними цілями. Наприклад, використовуючи Go, ви можете створювати високопродуктивні застосунки, які ефективно використовують системні ресурси. JavaScript, у свою чергу, природно реалізує подієву модель, а SQL дозволяє зручно маніпулювати даними.

Однак застосування мови не за її прямим призначенням може призвести до проблем. Створюваний код стає більш складним та заплутаним. Обсяг коду може збільшуватися в рази порівняно з ситуацією, коли використовується «правильна» мова для завдання. Саме з цієї причини на початковому етапі розробки вкрай важливий вибір інструментів, включаючи мови програмування.

Якщо в мові відсутні необхідні можливості, рекомендується використовувати відповідні бібліотеки та фреймворки. У разі їх відсутності можна розробити власну «мову» або нотацію.

Навчання програмістів використанню різних мов також відіграє важливу роль. Ґрунтуючись на особистому досвіді, можна стверджувати, що це часто складний процес, оскільки іноді потрібно змінити вкорінені підходи та звички, сформовані у розробників за роки роботи.

---

[1] Вілайанур Рамачандран, Мозок розповідає
[2] Джефф Хокінс, Сандра Блейксли, Про інтелект
[3] Quote Origin: If Your Only Tool Is a Hammer Then Every Problem Looks Like a Nail. https://quoteinvestigator.com/2014/05/08/hammer-nail/
[4] https://uk.wikipedia.org/wiki/BPMN

Вчимося спілкуватися

Язик до Києва доведе
Приказка

Як би дивно це не звучало, уміння спілкуватися та обмінюватися інформацією є одним із найважливіших способів боротьби зі складністю. Процес розробки базується на передачі інформації між учасниками проекту, і чим ефективніше спілкуються розробники, тим більша ймовірність успішного завершення проекту.

У попередніх розділах ми визначили складність як когнітивне навантаження, що виникає у конкретної людини при вирішенні проблеми. Ви можете намагатися вирішити задачу самостійно або звернутися до колеги за допомогою, знімаючи таким чином з себе розумове напруження. Можливо, колега вже стикався з подібною проблемою і знає відповідь, або підкаже, до кого звернутися чи що почитати. **Комунікація може значно зменшити навантаження та скоротити час на вирішення задачі.**

Перш ніж розпочати сеанс парного програмування, приберіть з кімнати всі гострі предмети.

Приписується Брендану Ейху

Однією з частин методології Екстремального програмування, запропонованої Кентом Беком [1], є парне програмування. Два програмісти, сидячи за одним робочим місцем, пишуть вихідний код. При цьому один пише, інший переглядає та дає підказки. Детально на описі під-

ходу зупинятися не будемо, але вкажемо на випадки, коли його застосування може бути найбільш ефективним.

Якщо програміст вирішує задачу в межах своєї компетенції та знань, то, зосередившись на роботі, він може швидко і якісно досягти результату. У такому випадку робота в парі не дасть особливої вигоди і, швидше за все, уповільнить процес.

Ситуація кардинально змінюється, коли вирішується завдання на стику компетенцій розробників. Наприклад, один має більше знань у предметній області, а інший — у галузі використовуваної технології. Кожен із них окремо зіткнувся б із проблемами через брак знань. Але, об'єднавши свої зусилля, розробники зможуть досягти результату набагато швидше.

Можу згадати випадок, коли потрібно було змінити звіт і прискорити його генерацію. Звіт був написаний мовою PHP і працював вкрай неефективно. Було вирішено переписати реалізацію, використовуючи SQL. На жаль, програміст не мав достатніх навичок роботи з цією мо-

вою, і завдання застопорилося. Проблема не вирішувалась до того моменту, як до її виконання підключився колега, який відмінно володів SQL і вмів оптимізувати запити. Разом вони швидко написали потрібний код.

Ніхто не може робити все самостійно. Іноді вам може знадобитися допомога, а іноді ви будете пропонувати її.

Чарльз Вудсон

Існують і більш прості ситуації, коли достатньо надіслати повідомлення у робочий чат і швидко отримати необхідну відповідь. Усім своїм колегам я намагаюся нагадувати про таку можливість і рекомендую не соромитися її використовувати. Така практика є проявом важливого командного феномену, який психологи називають «трансактивною пам'яттю» - колективною системою кодування, зберігання та пошуку інформації.

Трансактивна пам'ять команди розробників працює як розподілена база знань - кожен учасник є експертом у певній області, але при цьому знає, до кого звернутися за інформацією з інших питань. Наприклад, хтось краще розбирається в оптимізації SQL-запитів, інший - в архітектурі frontend-застосунків, а третій має глибокі знання бізнес-домену. З часом команда формує «метапам'ять» - розуміння того, хто що знає і де шукати потрібну експертизу. Саме тому згуртовані, давно працюючі разом команди зазвичай більш ефективні - вони вже сформували розвинену трансактивну пам'ять.

Учасники такої команди:

- знають сильні сторони та експертизу кожного колеги;
- довіряють компетенціям один одного;
- вміють швидко знаходити потрібну інформацію через правильних людей;
- ефективно розподіляють нові знання та досвід між собою;

- не витрачають час на повторне вирішення вже вирішених кимось проблем.

Проте важливо пам'ятати, що будь-яке звернення в офісі чи повідомлення у чаті може відволікти ваших колег від роботи. Тому важливо знаходити баланс між особистою роботою та комунікацією. Можу запропонувати просте правило: спочатку спробуйте розібратися з проблемою самостійно. Якщо протягом 15ти хвилин рішення не знайдено – звертайтесь за допомогою.

Якщо ви не можете пояснити щось простою мовою, ви не розумієте це достатньо добре.

Приписується Річарду Фейнману

З комунікацією тісно пов'язана проблема спотворення інформації під час її передачі. Ми розбирали це у розділі "Крах Вавилонської вежі". Головне, що слід пам'ятати: коли ви говорите або пишете комусь, варто врахувати ймовірність неправильного розуміння. Тому ставте контрольні питання і просіть переказати, що саме співрозмовник зрозумів. Уважно слухайте відповідь. Якщо відповідь довга, заплутана або терміни використовуються неправильно, це може вказувати на те, що людина вас не зрозуміла. Спробуйте пояснити те саме знову, але інакше.

Ті ж правила можна застосовувати і коли щось розповідають вам. Тільки переказувати почуте будете вже ви.

Люди іноді намагаються пояснити те, у чому самі не впевнені. У цьому випадку ставте уточнюючі питання. Якщо відповіді не задовольняють, зверніться до першоджерела, щоб уникнути спотворення інформації.

Крім проблем розуміння, на якість комунікації впливають її канали. Особисте спілкування — найефективніший спосіб. Воно надає безліч засобів виразності: можна говорити, жестикулювати, малювати, показувати екран. Живе спілкування дозволяє швидко вирішити питання, і

це одна з причин, чому учасникам проекту рекомендується періодично зустрічатися.

Однак багато проєктів сьогодні створюються розподіленими командами, що знаходяться на відстані тисяч кілометрів одна від одної. На щастя, існує безліч засобів для спілкування на відстані: телефони, месенджери, відеоконференції, електронна пошта. Ці інструменти покращують можливості комунікації, але головна проблема розподіленої роботи — це час відповіді та затримки в спілкуванні.

Швидкість має значення

Павло Дуров

Кожен стикався з ситуацією, коли написавши в чат ви деякий час, іноді досить довгий чекаєте на відповідь. Проте якщо ви на час очікування переключаєтеся на іншу

проблему, то втрачаєте контекст попередньої. Якщо ж вам відповідають скажімо наступного дня, то ви можете вже й забути, навіщо задавали питання. Проблема затримок у комунікації може бути досить серйозною.

Працюючи над проєктом в одній великій організації, я зіткнувся з тим, що моя робота безпосередньо залежала від підсистем, що знаходяться у відповідальності інших відділів. Спілкування відбувалося переважно у месенджерах, проводити особисті зустрічі не було можливості. Деякі працівники відділів з якими я спілкувався реагували повільно. В результаті проект значно затягнувся, витрати на комунікацію з'їли досить значну частину всього часу, витраченого на проєкт.

Якщо у вас є така можливість, намагайтеся відповідати якомога швидше. Якщо бачите, що переписка виходить за межі простої відповіді на питання, організовуйте зустріч і обговорюйте проблему. Якщо необхідно, підключайте інших учасників. Зустріч не повинна займати багато часу. По можливості не обмежуйтесь розмовами, демонструйте один одному код, тексти та дані, з якими ви працюєте.

Просте пояснення того, що ви робите, часто може суттєво допомогти спілкуванню, усуваючи початкове нерозуміння та закладаючи фундамент для покращення спільної роботи в майбутньому.

Ерін Мейєр, Карта культурних відмінностей

Не соромтеся показувати й розповідати іншим, чим ви займаєтесь. У майбутньому ця інформація може виявитися корисною, і до вас звернуться з питанням або за допомогою. З іншого боку, така розповідь часто призводить до того, що ви будете знаходити у себе помилки або логічні невідповідності.

Цьому факту є раціональне пояснення. Починаючи щось говорити, ви задіюєте додаткову зону мозку, відпо-

відальну за формування мови, і, як наслідок, підключення іншої частини мозку допомагає по-новому оцінити інформацію.

У крайньому випадку, якщо у вас немає друзів, ви можете використовувати кота або гумову качечку. Цей метод описаний у книзі «Програміст-прагматик» під назвою «Метод качечки» (Rubber duck debugging). Я сам чув про цю техніку ще від своїх батьків-інженерів, і вона називалася «мені потрібна шафа», тобто той, хто може мене вислухати.

Висновки

Взаємодія з колегами під час вирішення завдань може суттєво спростити їх вирішення. Це стосується не тільки дійсно складних завдань, але й невеликих проблем. Мої багаторічні спостереження показують, що дуже часто технічні фахівці тяжіють до індивідуальної роботи, і спілкування з оточуючими викликає у них труднощі. Слід звертати на це увагу і намагатися налагоджувати та покращувати взаємодію між членами команди.

У цьому розділі навмисно не розглядалися питання покращення навичок комунікації: розвиток емоційного інтелекту [2], розуміння поширених когнітивних викривлень [3], вирішення конфліктних ситуацій [4]. Все це окремі великі теми, що виходять за межі цієї книги.

Оскільки цей розділ присвячений технікам боротьби зі складністю, ця глава нагадує, що спілкування може суттєво допомогти в цій боротьбі.

---

[1] Кент Бек. Екстремальне програмування. Розробка через тестування,

[2] Гоулман Деніел. Емоційний інтелект. Чому він може значити більше, ніж IQ.
[3] Даніель Канеман. Думай повільно… вирішуй швидко.
[4] Гевін Кеннеді. Домовитися можна про все! Як досягати максимуму в будь-яких переговорах.

Зменшуємо кількість сценаріїв

Життя — це послідовність різних комбінацій, їх треба вивчити, стежити за ними, щоб завжди бути у вигідному становищі.
Оноре де Бальзак, Євгенія Гранде

У попередній главі «А що якщо?», ми розглянули можливі проблеми, які виникають через різноманіття даних і станів під час роботи вашої програми. У цьому розділі ми розберемо використання додавання, замість зміни, з метою спрощення розуміння того, як працює ваш код.

Ви, ймовірно, чули про «Принцип відкритості/закритості» в ООП, який є частиною SOLID. Цей принцип представляє собою окремий випадок підходу, при якому ви не змінюєте існуючий код або дані, а тільки додаєте. У контексті цього принципу ми говоримо про інтерфейс класу. У даному розділі ми розглянемо це питання більш широко.

Змінні. Я вже згадував про Erlang, де відсутня можливість модифікації змінних. Такий підхід ви можете використовувати і у своєму коді. Вибравши для змінної підходящу назву і один раз присвоївши їй значення, вам буде простіше розуміти код, особливо якщо в ньому є умовні оператори. Використання незмінних об'єктів також може запобігти помилкам, які виникають через те, що стан об'єкта може змінюватися неявно. Я не закликаю викори-

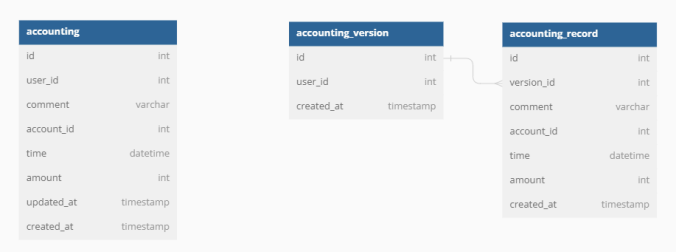
стовувати цей метод усюди, але його варто враховувати, особливо при багатопотоковій обробці.

База даних. Коли мова йде про БД, що містить структуровані дані, зміна їх структури може викликати збої в роботі. Тому при внесенні змін краще розширювати структуру, додаючи нові поля, а не змінювати або видаляти вже існуючі. Однак з даними все складніше, так як їх накопичення може привести до значного зростання їх обсягу. Але іноді заборона на зміну даних може спростити роботу.

Колись мені довелося стикатися з проектом, в якому в БД записувалося поточний стан розрахунків з клієнтами. Дані постійно доповнювалися, і все йшло добре до тих пір, поки не виникла необхідність у коригуванні даних у минулому. Розробники вирішили видаляти і змінювати старі записи після перерахунку. В результаті постійно виникали помилки, і за поточними даними було складно відновити хронологію подій та з'ясувати причини збою.

У таких випадках можна використовувати версіонування даних: створюється додаткова таблиця, завдяки якій здійснюється створення нових версій.

Структура таблиці зліва передбачає, що дані можуть змінюватися. Альтернативна структура з двох таблиць передбачає, що дані тільки додаються. Якщо потрібна корекція даних, у таблицю версій додається нова версія, а в основну таблицю додаються записи, відповідні поточному стану.

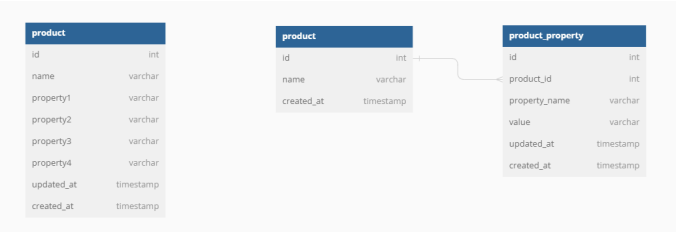


Приклад версіонування даних

При роботі з даними необхідно уважно ставитися до їх видалення. Намагайтесь уникати цієї операції. Замість видалення використовуйте додаткове поле «deleted_at», яке заповнюється, якщо дані позначаються як видалені.

Складнощі можуть виникати не тільки на етапі розробки, але і в процесі супроводу та налагодження програмного забезпечення. При відсутності операцій видалення і модифікації даних, процес діагностики і усунення проблем стає значно простішим.

Ще один ефективний метод, що дозволяє мінімізувати зміни в структурі даних, — це використання таблиці у форматі ключ-значення для тієї сутності, яка в значній мірі піддається змінам або розширенням.



Приклад структури, що дозволяє розширювати кількість властивостей об'єкта без зміни структури

У прикладі демонструється, як таблицю з властивостями продуктів можна перетворити в більш гнучку структуру. Такий підхід усуває необхідність у модифікації структури при додаванні нових властивостей продукту.

Програмні інтерфейси. При проектуванні програмних інтерфейсів версіонування вважається хорошою практикою. У контексті мікросервісної архітектури цей підхід стає особливо важливим, якщо ви хочете запобігти можливим помилкам при розгортанні нових версій продукту. Концепція версіонування API полягає у збереженні

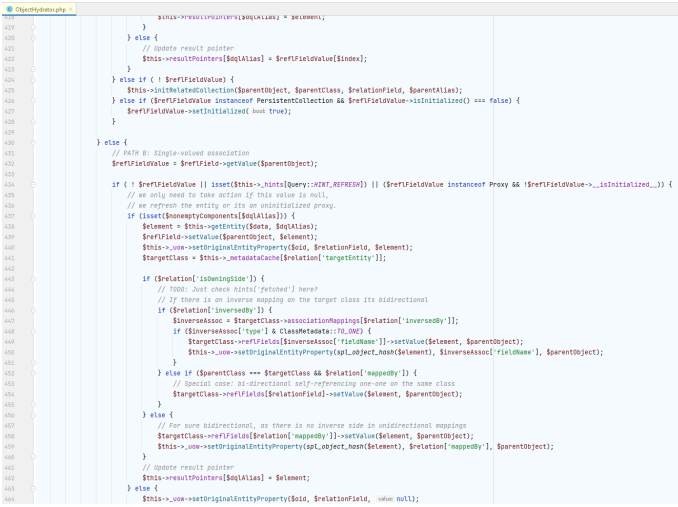
старої версії інтерфейсу і додаванні нової. Так, якщо у вас був ендпоінт «/api/v1/order» і відбулося зміна в структурі запиту, вам варто додати «/api/v2/order» з новою структурою, забезпечивши тим самим зворотну сумісність.

Програмний код. Мені неодноразово доводилося стикатися з кодом, який з часом обростав новими умовами і поступово з простого і очевидного перетворювався в монстра, змінювати якого взагалі не варто. Будь-яка, навіть на перший погляд нешкідлива зміна, призводить до несподіваних і часом дуже неприємних наслідків.

Зрозуміти і передбачити поведінку такого коду вкрай складно. Справа в тому, що, додаючи гілки виконання в код програми, програміст створює нові варіанти, за якими проходить логіка виконання.

Логіка змінюється, стаючи при цьому все більш складною. Особливо ускладнюється ситуація, коли розробники починають писати вкладені конструкції з умовними переходами, циклами і тому подібним. Варіантів, за якими може виконуватись програма, стає надзвичайно багато. Не дивно, що подібний код називають «спадщиною» (legacy), і розробники ніколи не зізнаються, що написали його самі.

Нижче представлений фрагмент подібного коду з популярної бібліотеки з відкритим кодом. Статичний аналіз коду показує, що логіка виконання може пройти по 535738 (NPath complexity) різним шляхам. І це далеко не межа. Мені доводилося бачити методи, для яких це число обчислювалося мільярдами (!!!).



Приклад складного коду

Як не дивно, але причина появи подібного коду полягає в першу чергу в підході до його написання. Слід дотримуватися наступних простих правил:

- Код повинен складатися з невеликих, незалежних, послідовних блоків.
- Модифікація коду передбачає додавання нових блоків, а не ускладнення існуючих.
- У разі збільшення обсягу коду, його частини переносяться в окремі функції, побудовані відповідно до принципів, викладених в двох перших пунктах.

Таким чином, ваш код в більшості випадків буде містити дві основні конструкції:

- Виконання операції або методу.
- Якщо умова виконується, то відбувається вихід з функції або блоку.

```
private function handle_event(Event $event)
{
    $event_type = $event->get_type();

    if (!isset($this->events_map[$event_type])) {
        return null;
    }

    if (!$this->is_waiting_for($event_type)) {
        return null;
    }

    $this->set_exec_time();
```

Приклад коду, побудованого за викладеними вище правилами

Цей підхід можна застосовувати в циклах, використовуючи оператори **continue** і **break**. Також використовуючи виключення в конструкціях try-catch, що існують у багатьох мовах.

У випадку використання ООП принцип додавання логіки може бути реалізований шляхом створення класів, що реалізують один інтерфейс, але містять різні варіанти логіки. Використовуючи шаблон проектування «Фабрика», можна створювати об'єкт необхідного класу, який і буде використовуватися в конкретному випадку. Такий підхід вимагає етапу попереднього аналізу, без якого неможливо створення групи однотипних абстракцій.

Як приклад, можу згадати реалізацію життєвого циклу угоди з контрагентом. Були виділені окремі етапи життєвого циклу, між якими міг здійснюватися перехід з виконанням необхідних дій. Операції переходу були реалізовані у вигляді окремих класів, а який саме клас застосовувався визначалося за ідентифікаторами станів, між якими відбувався перехід. Таким чином, при появі нових

бізнес-вимог і нових етапів угоди відбувалося додавання нового обробника, без зміни і впливу на вже працюючий код.

Висновки

Запропонований підхід заміни змін і видалень на додавання є перевіреним способом боротьби зі складністю за рахунок зменшення можливої кількості варіантів і станів, з якими вам доводиться мати справу в рамках окремого взятого компонента програми.

Стратегія мінімізації змін структур даних, коду та інтерфейсів дозволяє суттєво спростити не тільки розробку, але й супровід програмного продукту, а також зменшити ймовірність виникнення помилок.

Окремі випадки реалізації цього принципу ви, швидше за все, зустрічали і вже використовуєте у своїй роботі.

Фундамент для створення порядку

Люди використовують ментальні моделі або формати всередині себе (поняття і стереотипи), щоб керувати заплутаною, постійно змінюваною, часто хаотичною реальністю перед ними.
Люк Брабандер, Думай в інших форматах

Зіштовхуючись із реальним світом, людина отримує величезну кількість різноманітної інформації. Для того щоб сприйняти її як є, зрозуміти і використати, потрібні величезні обчислювальні потужності, яких, як ми з'ясували раніше, людина не має. Тим не менш, люди вміють існувати в цій непростій обстановці та навіть прогнозувати майбутні події. Основою такого успіху є побудова різноманітних ментальних моделей, у які укладаються факти, що надходять людині.

З моделями ми починаємо стикатися з самого раннього дитинства. Наші іграшки є моделями реальних об'єктів. Знайомство з цими моделями дає нам можливість краще розуміти світ.

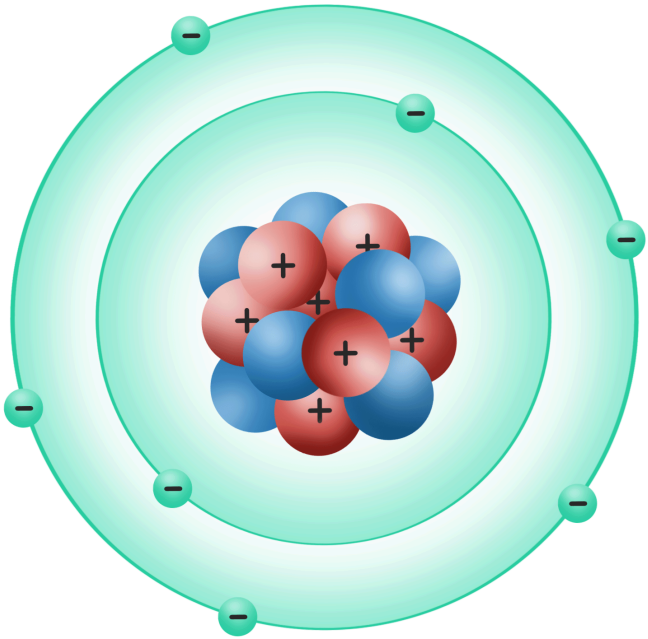
Граючи моделлю машинки, дитина розуміє, що у машини чотири колеса, є кабіна, де знаходяться водій і пасажири. Те, що в реальній машині є двигун, а в іграшковій немає, для дитини неважливо. Та й для багатьох дорослих машина насамперед це колеса, кермо, панель приладів і далеко не всі здогадуються, що в двигуні внутрішнього згоряння є мастило, яке потрібно регулярно міняти.

Людина виділяє саме ті властивості та поведінку об'єкта, які для неї важливі.

Для більшості поворот керма вправо означає, що й транспортний засіб поверне в тому ж напрямку. Ця ментальна модель дозволяє передбачити напрямок руху автомобіля та керувати ним. Однак, якщо її застосувати, наприклад, до мотоцикла, то виявиться, що вона недостатня — існує прийом, що називається "контрруління", що полягає в тому, щоб почати поворот праворуч, спочатку необхідно короткочасно повернути кермо ліворуч.

Основою ментальної моделі є абстракція, оскільки вона дозволяє спростити складні об'єкти та процеси. Використовуючи абстракцію, ми можемо виділити найважливіше і відкинути незначні деталі.

Моделі є основою наукової діяльності, оскільки дозволяють дослідникам зосередитися на певних аспектах вивченої ними області. Розробка моделей сприяє кращому розумінню закономірностей природи та полегшує процес створення нових технологій та відкриттів. На малюнку зображена модель атома, запропонована фізиками на початку двадцятого століття.



Але як це стосується розробки програм? Справа в тому, що сама програма є не лише інструментом вирішення завдань, а й моделлю тієї предметної області, в якій це завдання вирішується. І навіть якщо розробники не усвідомлюють цього, то під час написання програми вони створюють абстракції та будують моделі.

Моделі реального світу будувати простіше, ніж віртуального. Наприклад, виготовляючи ляльку, ви чітко розумієте, що вона має бути схожа на людину. У неї мають бути тулуб, дві руки, дві ноги та голова, тобто великі частини тіла, притаманні людині.

Під час побудови програми ми не створюємо модель, ми описуємо своє уявлення про неї. Комп'ютер же створює модель за даним описом. Однак опис може

бути суперечливим і під час роботи програми ці суперечності виявлятимуться у вигляді збоїв та помилок.

Моделі далеко не ідеальні. Різні мови програмування "ставляться" до побудови моделі по-різному. Є мови, парадигма яких не передбачає перевірки моделі на рівні опису.

Швидше за все, ви знаєте про типізацію - механізм, що дозволяє визначити та обмежити дані, які можуть бути присвоєні змінній. Типізація допомагає забезпечити безпеку та стабільність коду, запобігаючи помилкам, пов'язаним з неправильним використанням даних різних типів. Залежно від мови програмування, типізація може бути статичною (типи визначаються на етапі компіляції) або динамічною (типи визначаються під час виконання програми).

Мови зі слабкою динамічною типізацією (наприклад, PHP, JavaScript) дозволяють менше уваги приділяти побудові чітких абстракцій. Зазвичай розробка на них йде швидше, але як наслідок виникає більше помилок під час виконання програми. На противагу їм компілятор мови зі суворою статичною типізацією (наприклад, Haskell або Java) не дозволить вам запустити код, якщо текст вашої програми містить некоректне використання сутностей. Таким чином, деякі мови зобов'язують програміста заздалегідь продумувати деталі абстракцій, з якими вони мають справу.

Основою побудови абстракцій та всієї моделі, яка буде реалізована у вигляді ПЗ, є аналіз. В ідеальному світі, під час аналізу вимог, що надходять від усіх зацікавлених осіб, має бути сформована повна, несуперечлива модель предметної області, для якої створюється ПЗ. На початковій стадії аналізом займається бізнес-аналітик, далі естафету підхоплює архітектор.

У реальності ж часто буває так, що вимоги потрапляють безпосередньо до програмістів. І ті, не враховуючи всі вимоги, починають писати код. Таким чином, модель та абстракції в програмному коді починають формуватися самі по собі, випадковим чином, залежно від кваліфікації програміста, який їх створює.

Перевірка коректності роботи програмного забезпечення здійснюється QA інженерами або автоматизованими тестами, які перевіряють поведінку у окремих конкретних випадках.

У такій ситуації виникає потреба у повному покритті коду тестами, що досягається досить рідко. І навіть при 100% покритті немає гарантії, що ваша програма працюватиме правильно. Додаткове використання, наприклад, мутаційних тестів може показати наявність помилок та проблем у коді навіть при повному покритті модульними тестами.

Що ж буває потім? Виникають нові вимоги і програмний код змінюється. Бувають ситуації, коли доводиться переписувати весь існуючий код, оскільки немає можливості задовольнити нові вимоги - закладена модель в принципі не відповідає тому, що необхідно. Більш неприємна ситуація, коли розробникам вдається адаптуватися до нових вимог, ще сильніше ускладнивши програмний модуль. У коді виникають багато умовних переходів, з'являється код, який обробляє особливі випадки або вирішує задачу неприроднім способом (на програмістському сленгу - костилі). Сама програма розростається і стає все більш складною і менш контрольованою.

Існує емпіричний закон – чим більша команда розробки, тим більший обсяг буде мати розроблювана ними програма, незалежно від розміру вимог. Ця закономірність легко пояснюється тим, що при великій кількості

розробників суттєво зростає ймовірність створення альтернативних моделей та абстракцій у рамках одного програмного продукту.

Все перелічене означає, що основою створення якісного програмного забезпечення є аналіз. Ретельне вивчення вимог предметної області лежить в основі створення успішної моделі. Засоби, вкладені на етапі аналізу, окупляться у багато разів, а то й десятки разів на подальших етапах розробки.

Результатом аналізу має бути чіткий набір понять, якими користуються всі учасники проєкту, відсутність суперечностей та дублікатів. Якщо ви звернетеся до методології DDD, то однією з основних її частин є створення Єдиної мови (ubiquitous language) проєкту, яка також є результатом аналізу. У своїх проєктах я намагаюся контролювати набір термінів, що зустрічаються в коді програм. Такий підхід дозволяє знаходити альтернативні ментальні моделі, описані програмістом.

На жаль, люди, які мають сильне аналітичне мислення, зустрічаються відносно рідко. Фактично, це люди, які мають склад мислення вчених і можуть застосувати свої здібності у багатьох областях, а не тільки в розробці ПЗ.

Тим не менш, якщо ви керівник проєкту, то слід звернути особливу увагу на наявність сильного аналітика в команді. З точки зору витрат на проєкт, ця людина є найвигіднішим вкладенням. Фактично, вона створює фундамент проєкту та визначає його майбутній успіх.

Ось що пише про це Ф. Брукс: «Найважча частина розробки програмної системи полягає в тому, щоб точно вирішити, що розробляти. Жодна інша частина концептуальної роботи не є такою важкою, як встановлення детальних технічних вимог, включаючи всі інтерфейси для людей, машин та інших програмних систем. Жодна

інша частина роботи так не калічить результуючу систему, якщо вона зроблена неправильно. Жодна інша частина не є складнішою в виправленні пізніше.» [1]

Якщо ви хочете стати висококласним розробником, вам просто необхідно розвивати аналітичне та системне мислення, а також вміння будувати абстракції та моделі.

Висновки

Програмний код є описом ментальних моделей всіх розробників, які беруть участь у проєкті. Якщо моделі не узгоджені, то з високою ймовірністю ви отримаєте помилки та надмірний розмір проєкту. Тому вкрай важлива синхронізація цих ментальних моделей як на рівні розуміння (це досягається постійним спілкуванням розробників), так і на рівні опису (програмного коду). А саме:

- відсутність дублювання коду;
- контроль термінівщо використовуються;
- повторне використання об'єктів, функцій, структур даних.

Дії на етапі обробки вимог і проєктування мають найбільшу важливість для успіху проєкту. Вони закладають фундамент розробки, і від цих дій залежать подальші часові та матеріальні витрати.

Вони включають в себе:

- аналіз вимог;
- створення мови проєкту;
- побудову моделі предметної області.

Їм слід приділяти найбільше значення і залучати якнайбільш кваліфікованих фахівців. У цьому випадку кваліфікація визначається насамперед за досвідом успішної реалізації попередніх аналогічних проєктів.

Якось я спостерігав, як добре теоретично підготовлений фахівець, відомий своїми публікаціями, був прийнятий на посаду технічного керівника проєкту. Однак його практичні рішення були вкрай невдалими за широким колом питань, включаючи аналіз і реалізацію доменної моделі. За більш ніж рік його діяльності компанія зазнала серйозних фінансових збитків, так і не отримавши очікуваного результату.

---

[1] Фредерик Брукс. Срібної кулі немає — суттєві та часткові ознаки інженерії програмного забезпечення.

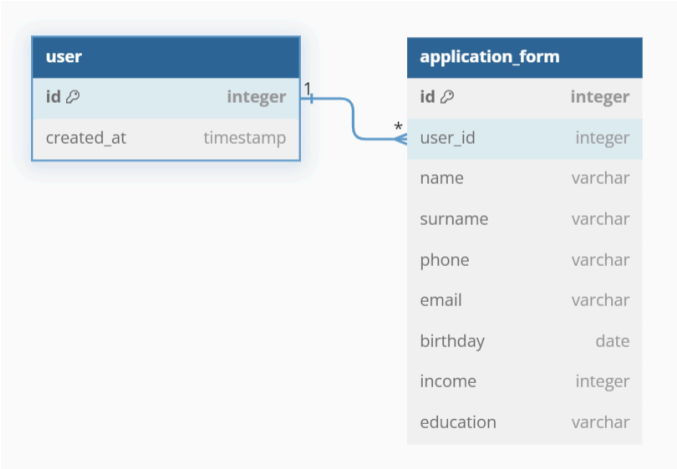
Як створити модель

Язиком плескати - не ціпом махати
Народна приказка

У попередній главі було сказано багато загальних слів з приводу важливості моделі при побудові програмного забезпечення, але майже нічого з приводу того, що саме треба робити. Само по собі слово «модель» може поплутати людину яка частіше пише код, ніж проектує програмне забезпечення. Популярні фреймворки представляють модель в першу чергу як відображення структур бази даних у коді і більшість програмістів саме так розуміє термін «модель».

Насправді модель предметної галузі включає в себе набагато більше. База даних потрібна для збереження стану об'єктів і її структура (звісно якщо вона є) далеко не завжди співпадає зі структурами даних якими програма маніпулює в оперативній пам'яті.

Співпадіння зазвичай буває у найпростіших випадках. Наприклад нам потрібно зберегти дані форми, яку заповнює клієнт. У цьому випадку очевидним рішенням буде створення таблиці, структура якої повторює структуру форми.



Програміст особливо не задумуючись створює таблицю в відповідними полями, екранну форму для вводу даних та бекенд який читає дані з запиту, перевіряє їх коректність та записує в таблицю. Завдання не потребує великих розумових зусиль і його може реалізувати програміст-початківець. Якщо створювати модель, то вона фактично буде співпадати з тим що можна побачити на екрані.

Але спробуємо додати вимог від замовника:

- форма, яку заповнює клієнт, містить багато полів, і її потрібно розбивати на декілька окремих частин (екранів);
- структура форми може змінюватись, деякі поля можуть зникнути, але можуть з'явитися і нові поля;
- необхідно зберігати всі дані які вводять клієнти, тобто мати архів заповнених клієнтами форм.

Якщо просто почати дописувати код орієнтуючись на вимоги, то скоріше за все ваш код почне обростати великою кількістю умов, будуть з'являтись таблиці не всі поля

яких будуть заповнені, і подальші зміни та вимоги реалізувати буде значно важче.

Тому перед тим як програмувати варто продумати модель, з якою ми будемо працювати і яка включає всі перераховані вимоги.

Спочатку виділимо **поле даних** і **екран** як окремі об'єкти. Так як існує вимога що структура екранних форм може змінюватись, зафіксуємо те, що поле належить до екрану вводу. Набір даних які ввів користувач представимо я набір даних ключ-значення (**актуальні дані користувача**). Таким чином дані не будуть залежати від того на якому екрані вони відображаються, але всі вони будуть належати до однієї **форми**.

Жирним виділені ті об'єкти які утворюють нашу модель. Запропоновані об'єкти можуть знайти своє відображення на рівні бази даних, але це не буде обов'язковим.

Якщо ви захочете, то зможете зберігати їх як у реляційній базі даних, так і у NoSQL сховищі або відправляти як повідомлення використовуючи брокер повідомлень. Бізнес логіка побудована на основі вимог не зміниться.

Отже при побудові моделі нам потрібно:

1. **Ідентифікувати компоненти**, які будуть використовуватись при побудові функціоналу.
2. **Придумати влучні назви**, які потім можна використовувати в коді і при спілкуванні. В нашому випадку це: поле даних, екран, форма.
3. **Визначити властивості**. Потрібно розуміти, що саме стоїть та тією чи іншою абстракцією. У нашому прикладі поле даних має назву, значення і прив'язане до екрану вводу.
4. **Визначити операції**, які може робити компонент. У нашому випадку значення поля даних можна зберегти та прочитати.

5. **Зв'язати компоненти між собою**. Для того щоб вирішити задачу до кінця, потрібно зібрати всі частини моделі до купи.

Загалом процес нагадує гру з дитячим конструктором. Єдина відмінність що конструктивні блоки треба створити самому.

Описаний приклад досить простий, логіка перетворення даних майже відсутня. Тому розглянемо більш складний приклад.

Від представника бізнесу прийшли вимоги — необхідно пропонувати різні умови клієнтам які давно працюють в системі і тим хто тільки зареєструвався. Очевидне рішення, розставити в коді умовні переходи там де визначались би умови згідно історії користування сервісом. Але ж глава не про це!

На момент отримання вимог програмний продукт реалізовував модель «клієнт — продукт». Клієнт обслуговується за умовами які є властивостями продукту. А до чого відноситься поставлена бізнесом вимога? Схоже, що у нас повинно з'явитись щось третє, яке зв'язує між собою існуючі сутності.

Розділення клієнтів на групи згідно характеристикам вік, дохід, місце проживання і т.і. називається сегментацією. Сегментація це один із ключових принципів маркетингу. Тобто представник бізнесу хоче сегментувати клієнтів. Проаналізувавши вектор розвитку продукту стало зрозуміло: якщо сьогодні виникла необхідність розділити клієнтів за ознакою новий/досвідчений то через деякий час виникне ще якась умова.

Таким чином модель «клієнт — продукт» була доповнена сутністю «сегмент». Сегмент, це приналежність клієнта до деякої групи, згідно вибраним умовам сегментування.

І замість умови в коді, модель предметної галузі була розширена до «клієнт — сегмент — продукт», що потягнуло за собою створення окремого модуля. На етапі формування пропозиції визначається до якого сегменту відноситься клієнт і за сегментом визначається той продукт, який пропонується клієнту.

Таким чином були зроблені дії описані вище:

- ідентифікація: щось повинно бути проміжною ланкою між клієнтом і умовами;
- визначення назви: «сегмент»;
- визначення властивостей: деяка умова, виконання якої відносить клієнта до сегменту;
- можливі операції: визначення приналежності клієнта до сегменту;
- зв'язки: клієнт відноситься до сегменту, до сегмента прив'язується продукт.

На розробку пішов додатковий час, дещо більший ніж очевидна модифікація коду. Кілька місяців функціонал не був затребуваний, але далі запити від замовника пішли один за одним і замість переписування програмного коду, розробники відбувались налаштуванням компонента. Таким чином, час витрачений на початковому етапі окупився економією на реалізації наступних задач. На відміну від очевидного рішення, яке би згодом привело до переписування та ускладнення коду.

Звісно буває так, що функціонал який створили розробники не знадобився, а час розробки був згаяний. Тому на етапі аналізу дуже важливо оцінити імовірність сценарію по якому піде розвиток продукту.

Якщо ви звернете увагу на слова які використовувались для опису моделі, ви не побачите технічних термінів. Це дуже важливий момент.

Найчастіше за все, заглядаючи в код програмного проекту побудованого на основі популярного фреймворка ви побачите файли і папки які будуть називатись Controller, Repository, Exception, Listener і т. д. Найближчим до того що може бути названо моделлю буде каталог Entity чи Models, але там будуть знаходитись компоненти які фактично відображають у коді структури бази даних. Але структури для збереженя даних і модель предметної галузі це не одне і теж саме. Не всі об'єкти мають стан, який треба зберігати.

Найчастіше розробники, не знаючи, куди віднести функціональність таких компонентів, створюють різноманітні «допоміжні» класи, у яких реалізуються ключові аспекти предметної області. Ще гірший варіант коли розробник намагається «всунути» об'єкти що можна віднести до моделі предметної галузі в існуючу структуру катало-

гів, таким чином заплутуючи тих хто буде працювати в майбутньому з їх кодом.

«Якщо дизайн або якась його центральна частина не відповідає доменній моделі, така модель має мало цінності, а правильність програмного забезпечення викликає сумніви. У той же час складні відображення між моделями та функціями дизайну важко зрозуміти і, на практиці, неможливо підтримувати в умовах змін дизайну. Виникає небезпечний розрив між аналізом і дизайном, через який ідеї, отримані в кожній із цих діяльностей, не впливають одна на одну.» [1]

Люк Скайуокер: «Темна сторона сильніша?»

Йода: «Ні, ні. Швидша, легша, більш спокуслива.»

Зоряні війни: Епізод V – Імперія завдає удар у відповідь

Але навіщо робити стільки додаткових дій, коли можна відбутись просто умовним переходом у коді? Справа у тому, що якщо не робити аналізу і не створювати моделі, зрозумілі всім учасникам проекту через деякий час розробники перестануть розуміти, що саме відбувається. Ментальна модель яка необхідна для чіткого розуміння продукту буде відсутня. В головах тих, хто створює продукт буде набір не прив'язаних ні до чого вимог, купа розрізнених шматків інформації. Загальна картина, що саме являє собою програмний продукт буде відсутня. Саме це створює нездолану перешкоду подальшому розвитку. Не дивлячись на збільшення витрат на розробку і збільшення кількості самих розробників розробка продукту буде сповільнюватись.

Висновки

При розробці програмного продукту важливо аналізувати всі вимоги на предмет їх відповідності існуючій

моделі, яку реалізує продукт. Наявність моделі дозволяє покращити розуміння того, що саме має робити продукт, з яких частин він складається. Для великих продуктів це необхідна умова сталого розвитку. Модель може розширюватись і перероблюватись у ході роботи над продуктом.

Коли ми говоримо «придумати влучні назви, які потім можна використовувати в коді та у спілкуванні», ми фактично описуємо процес створення Єдиної мови. Терміни «поле даних», «екран», «форма», а пізніше «сегмент» стають частиною цієї мови.

Терміни використовуються не лише в коді, але й у спілкуванні між розробниками та представниками бізнесу, що є ключовим аспектом Єдиної мови. Важливо відзначити, що вона не є статичною. Модель еволюціонує від «клієнт — продукт» до «клієнт — сегмент — продукт». Це демонструє, як мова проекту (і модель, яка за нею стоїть) розвивається разом з розумінням предметної області.

Коли ми говоримо про «ментальну модель, яка необхідна для чіткого розуміння продукту», то фактично описуємо одну з головних цілей Єдиної мови - створення спільного розуміння продукту серед усіх зацікавлених сторін.

Саме мова є основою для побудови абстракцій, а з абстракцій складається модель яка, як ми з'ясували вище необхідна для стабільності програмного продукту.

---

[1] Ерік Еванс, Предметно-орієнтоване проектування (DDD)

Від простого до складного

Складна система, яка працює, незмінно походить з простої системи, яка працювала. Складна система, розроблена з нуля, ніколи не працює і не може бути виправлена, щоб змусити її працювати. Вам доведеться почати заново з працюючої простої системи.
Джон Галл

Принцип, описаний в епіграфі, має конкретне практичне застосування. Його можна використовувати як при проектуванні архітектури, так і при щоденному написанні коду.

Основою використання принципу є наявність працюючої системи або модуля. Цей принцип покладено в основу розробки через тестування– TDD. TDD передбачає, що в першу чергу створюються тести, які перевіряють коректність написаного вами коду. Вони виконують ваш код, і якщо закралася помилка, то ви це побачите при запуску тесту. Можливість запустити написаний вами код і максимально швидко отримати результат дозволяє створити швидкий цикл зворотного зв'язку.

Якщо порівнювати такий підхід з окремим виділеним циклом тестування, ви отримуєте величезну перевагу в часі. Буває, що розробники відправляють свій код на тестування, не перевіряючи коректність його роботи, і роблять виправлення тільки після того, як тестувальник повертає їм завдання із зауваженнями. У цьому випадку

час, витрачений на виконання завдання, може драматично збільшитися.

Потрібно розуміти, що це крайній випадок, при виникненні якого необхідно вкрай серйозно спілкуватися з розробником, пояснюючи необхідність самому контролювати результати своєї роботи.

Для того, щоб використовувати підхід від «простого до складного» не обов'язково писати безліч тестів. Для початку потрібно мати можливість побачити результати роботи вашого коду.

Якщо для працездатності необхідна наявність багатьох компонентів, то може здатися, що необхідно всі їх реалізувати. Мої спостереження показують, що розробники вчиняють так досить часто.

Коли ж справа доходить до етапу перевірки, буває досить складно визначити, що саме працює неправильно. Адже на результат роботи впливає весь написаний код. У цьому випадку розробник найчастіше звертається до відладчика.

Замість того, щоб аналізувати код і шукати логічну невідповідність у написаному, процес розробки перетворюється на процес спостереження за тим, як працює програма. У цьому випадку код найчастіше ускладнюється, оскільки розробник намагається задовольнити випадки, що виникли при взаємодії всіх модулів.

Складні системи розвиватимуться з простих набагато швидше, якщо існують стабільні проміжні форми, ніж якщо їх немає. Герберт Саймон. Архітектура складності

Альтернативою написання всього коду відразу є використання заглушок. Створюючи код, ви в першу чергу створюєте його основну структуру, а деталі залишаєте на потім. При необхідності функції можуть повертати відповідні константи. Написавши основну логіку програми, ви

можете запустити код і таким чином на ранньому етапі перевірити його працездатність. Поступово замінюючи заглушки кодом, у вас з'являється можливість коригувати помилки не вдаючись до відладчика, оскільки вам точно відоме місце останніх змін у коді. Таким чином ви в першу чергу перевіряєте новий код, а не взаємодію компонентів.

```
public interface IMessageService {
    boolean sendMessage(String message, String recipient);
}

public class MessageService implements IMessageService {
    private boolean sendMessageResult;

    public MessageService(boolean sendMessageResult) {
        this.sendMessageResult = sendMessageResult;
    }

    public boolean sendMessage(String message, String recipient) {
        // TODO write implementation
        return sendMessageResult;
    }
}

public class MessageProcessor {
    private IMessageService messageService;

    public MessageProcessor(IMessageService messageService) {
        this.messageService = messageService;
    }

    public boolean processMessage(String message, String recipient) {
         return messageService.sendMessage(message, recipient);
    }
}

public class MessageProcessorTest {
    public static void main(String[] args) {
        IMessageService messageService = new MessageService(true);
        MessageProcessor messageProcessor = new MessageProcessor(messageService);
        boolean result = messageProcessor.processMessage("Hello", "test@example.com");
        System.out.println(result ? "SUCCESS" : "FAIL");
    }
}
```

Приклад коду з використанням заглушки

Заглушки можна використовувати не тільки в тестах і коді. Їх можна застосовувати і на рівні API, створюючи сервіси, які ще не містять логіки, але вже дозволять вам або вашим колегам писати код, який їх використовує.

Те ж саме стосується і вибірок із сховищ даних. Складний запит можна оформити у вигляді представлення (view) і працювати з його результатами як з таблицею. Якщо запит передбачає складну вибірку даних, то на початковому етапі ви також можете замінити його на набір даних, який очікуєте побачити, і реалізувати сам запит пізніше, або доручити реалізацію комусь іншому.

На мій погляд, основною проблемою у використанні підходу від «простого до складного» є аналітичні здібності розробника. Далеко не кожен може розбити завдання на зручні для використання частини, а після цього визначити послідовність, який код повинен бути написаний раніше, а який пізніше.

Як і в будь-якій іншій діяльності, такий підхід вимагає тренування. Якщо ви будете свідомо практикувати попереднє розбиття коду на зручні для використання частини, заміну деяких викликів на порожні методи або повернення даних-заглушок, то це стане вашою звичкою і ви максимально швидко навчитеся отримувати працюючу програму, нехай і з обмеженим функціоналом, який ви зможете розширювати по ходу розробки.

Проектуючи архітектуру додатка, ви можете виділити певні модулі і шари додатка для того, щоб їх можна було легко підміняти заглушками з мінімальним функціоналом. Особливо це зручно у випадку інтеграцій із зовнішніми системами або сервісами. Ізоляція коду, який взаємодіє із зовнішньою системою, від основної частини вашого додатка дозволяє створити працюючу систему не покладаючись на зовнішню залежність.

Важливим моментом при розробці від простого до складного є вміння абстрагуватися від деталей реалізації тих частин системи, які на даному етапі не є важливими. Намагаючись утримати в голові всі компоненти і зв'язки,

розробник відчуває більшу когнітивну напругу і як наслідок допускає більше помилок.

Ще один підхід, який дозволяє спростити написання коду - це використання так званих пісочниць. Замість того, щоб писати код в рамках великого програмного продукту, ви розробляєте код окремо. Наприклад, ви не впевнені, як саме працює функція чи компонент, який ви хочете використати. Створивши невеликий шматок коду, який працює незалежно від основного програмного продукту, ви можете потім легко його додати до основного коду і бути впевненим у його працездатності. Це може бути не тільки програмний код, а наприклад, регулярні вирази або складна структура даних у вигляді JSON. Нижче наведено список онлайн інструментів, які дозволяють швидко перевірити коректність вашого коду.

JavaScript Playground https://www.jsplayground.dev/

Online Python https://www.online-python.com/

Java online compiler https://www.w3schools.com/java/java_compiler.asp

PHP live regex https://www.phpliveregex.com/

JSONLint validator and formatter https://jsonlint.com/

Ви можете підібрати свій набір інструментів, який буде допомагати у вашій роботі. Звісно, не обов'язково пробувати створювати програмний код онлайн. Ви можете писати і запускати окремі частини коду незалежно від вашого проекту на вашому локальному комп'ютері.

Висновки

Намагайтеся отримати працездатну програму якомога раніше. Це дозволить зменшити складність при розробці за рахунок того, що ви будете чітко розуміти, які зміни в коді призвели до отриманого результату.

Невеликі послідовні зміни коду дозволять легко повернутися до працездатного стану, навіть якщо ви допустили помилку. Помилка проявиться відразу і усунути її буде легко. В іншому випадку, якщо ви створили велику кількість коду, ви будете мати справу не тільки з окремими помилками, але і з поведінкою, викликаною комбінаціями некоректно працюючого коду.

Основною проблемою застосування підходу від простого до складного є аналітичні здібності розробника. Досить часто для розробника складно декомпозувати завдання і виділити ті частини, які необхідно реалізувати в першу чергу.

Індикатором того, що в роботу взято занадто великий шматок логіки, є використання відладчика.

Не забувайте про можливість швидкого створення невеликих робочих прототипів коду за допомогою онлайн інструментів.

Задачі над якими потрібно поміркувати

У кожної проблеми є рішення. Вам просто потрібнобути достатньо креативним,щоб знайти його.
Тревіс Каланік

Розробники часто згадують про те, що хотіли б у ході роботи вирішувати складні технічні завдання. Насправді, більшість складнощів у своїй роботі вони самі і створюють.

Тим не менш, завдання, які вимагають серйозного напруження при пошуку рішення, все ж таки зустрічаються. І такі завдання теж потрібно вміти вирішувати. Найчастіше складність завдання полягає в тому, що необхідно задовольнити кілька суперечливих умов. Наприклад, досягти необхідної швидкодії при обмежених апаратних ресурсах. Або розробити простий і зрозумілий користувачеві інтерфейс при необхідності відобразити велику кількість даних. Або придумати алгоритм, що реалізує заплутані бізнес-вимоги.

Подібні завдання передбачають обдумування, пошук шляху вирішення. Спробуємо розібратися, як краще справлятися з подібними завданнями.

Іноді найкреативніше, що ви можете зробити — це заснути.

Деніел Пінк

Багато хто стикався з ситуацією, коли наполегливі зусилля з вирішення проблеми не давали бажаного ре-

зультату. Наполегливі спроби закінчувалися повним провалом, однак через деякий час, у найнесподіваніший момент рішення приходило само собою.

Усі знають історію давньогрецького вченого Архімеда, до якого прийшла ідея вирішення завдання, коли він приймав ванну. Його вигук «Еврика!» є синонімом раптового осяяння.

А ось як видатний хімік Дмитро Менделєєв описує відкриття періодичного закону елементів: «Запідозривши про існування взаємозв'язку між елементами ще в студентські роки, я не втомлювався обдумувати цю проблему з усіх боків, збирав матеріали, порівнював і зіставляв цифри. Нарешті настав час, коли проблема дозріла, коли рішення, здавалося, ось-ось готове було скластися в голові. Як це завжди бувало в моєму житті, передчуття близького розв'язання питання, що мучило мене, привело мене в збуджений стан. Протягом кількох тижнів я спав уривками, намагаючись знайти той магічний принцип, який відразу привів би в порядок усю купу накопиченого за 15 років матеріалу. І ось одного прекрасного ранку, провівши безсонну ніч і відчаявшись знайти рішення, я, не роздягаючись, приліг на диван у кабінеті і заснув. І уві сні мені зовсім виразно представилася таблиця. Я тут же прокинувся і накидав побачену уві сні таблицю на першому ж клаптику паперу, що підвернувся під руку.»

Наукові відкриття це приклади вирішення виключно складних завдань, однак це не означає, що схожі осяяння не відбуваються в звичайному житті.

Наш мозок сприймає і обробляє інформацію на підсвідомому рівні, більшість когнітивних процесів ми не контролюємо, тим не менш діяльність мозку йде в бажаному нами напрямку.

Обробка і запам'ятовування інформації не відбувається миттєво. Найкращий результат дає навчання з паузами між періодами навчання. Крім того, в процесі розумової діяльності формуються нові нейронні зв'язки. Цей процес не може бути миттєвим. Важливу роль у засвоєнні інформації відіграє відпочинок і особливо сон.

Таким чином, обдумуючи проблему, дізнаючись більше інформації про неї, ми готуємо мозок до її вирішення. Залежно від кількості інформації та заплутаності завдання така підготовка може тривати роки. На щастя, в нашому повсякденному житті зустрічаються завдання набагато простіші, і мозку для їх вирішення потрібно менше часу порівняно з фундаментальними науковими проблемами.

Отже, стикаючись із завданням, для якого ви не бачите очевидної відповіді, не варто намагатися вирішити його

впертим безперервним обдумуванням. З великою ймовірністю така діяльність виснажить вас і не дасть бажаного результату. Спробуйте переключитися на щось інше, зробіть паузу, поспілкуйтеся з оточуючими або займіться фізичною активністю.

Зміна діяльності може дати кращий результат, ніж напружена робота в спробі вирішити завдання. Дуже важливо знайти баланс між періодами напруження і відпочинку.

«Опинившись у глухому куті, потрібно зробити зворотне тому, що підказує інтуїція, - **не намагатися посилювати концентрацію** на проблемі протягом тривалого часу.» [1]

Якщо повернутися в сферу розробки ПЗ, то програміст, який п'є каву і спілкується з колегами на відсторонені теми, може бути не менш продуктивним, ніж інший співробітник, який весь робочий день проводить за клавіатурою і друкує код.

До вирішення завдання вас може підштовхнути випадкова асоціація або слово. Коли ми налаштовані на пошук проблеми, мозок підсвідомо намагається комбінувати різні, найнесподіваніші варіанти.

Особливе значення для засвоєння і переосмислення інформації має сон. Окрім банального очищення мозку від продуктів життєдіяльності, уві сні відбувається переосмислення інформації, отриманої під час неспання. [2]

Учасникам експерименту пропонували вирішити завдання. У завдання було кілька методів вирішення: очевидний, але трудомісткий і більш швидкий, але не настільки явний, як перший. Одна група піддослідних починала працювати над завданнями вранці і закінчувала ввечері. Інша група починала ввечері. Час на вирішення і періоди відпочинку були однакові, однак друга група мала можливість поспати. Після періоду сну значна група під-

дослідних знаходила короткий шлях вирішення завдання, на відміну від тих, хто не спав. Експеримент проводили в різний час доби і прийшли до висновку, що **ключовим фактором знаходження більш ефективного рішення був саме сон.** Сон відіграє важливу роль і при вивченні нової інформації. [3]

Таким чином, якщо ви стикаєтеся зі складним завданням або новою для себе галуззю, найбільш ефективним буде чергування періодів роботи і відпочинку, який включає повноцінний сон.

Ставте правильні запитання, якщо хочете знайти правильні відповіді.

Ванесса Редгрейв

Успішним підходом до вирішення складного завдання буває зміна формулювання завдання. Такий підхід може допомогти поглянути на проблему зовсім під іншим кутом зору і не тільки спростити вирішення, але іноді й усунути необхідність вирішувати завдання.

У 2007 році Nokia купила компанію Navteq для того, щоб вийти на ринок послуг навігації. Navteq мала розвинену інфраструктуру дорожніх сенсорів, що дозволяють відстежувати трафік на дорогах. Така інформація була необхідна для побудови маршруту в об'їзд заторів на дорозі.

Якщо формулювати завдання «Як дізнатися, де знаходиться дорожній затор?», то встановлення дорожнього сенсора є очевидним висновком, а придбання компанії, яка вже встановила сенсори, найбільш швидким, хоч і дорогим рішенням. Сума угоди склала 8,1 мільярда доларів.

Однак якщо сформулювати питання інакше: «Хто знає про дорожній затор?» то очевидною відповіддю на питання буде - водій, який в цьому заторі знаходиться. У 2008 році була заснована компанія Waze, яка, не маючи фінансових можливостей Nokia, вирішила те ж завдання,

витративши набагато, набагато менше. Її рішення базувалося на обміні інформацією між учасниками руху.

Змінити формулювання завдання можуть і додаткові запитання. Такими запитаннями можуть бути:

- Яка мета цього завдання?
- Що є найбільш складним у вирішенні завдання?Чи можливо прибрати цю умову з його формулювання?
- Наскільки критичним є дане завдання?Чому?
- Хто зацікавлений у його вирішенні?
- Скільки випадків покриває рішення даного завдання?

Одним із способів альтернативного формулювання завдання може бути його візуалізація. Спробуйте вирішити завдання. Летять качки, дві спереду, дві ззаду, одна посередині, три в ряд. Скільки всього качок?

Трохи подумайте, а потім спробуйте зобразити те, що описано в умові. Враховуйте те, що умови завдання повні і несуперечливі.

Творчість - це сила пов'язувати речі, які здаються несхожими.

Вільям Пломер

Вирішуючи проблему, ви фактично комбінуєте наявні у вас знання. Творчість багато в чому схожа на рекомбінацію генів в ДНК - нова комбінація може дати зовсім нову властивість. Дана книга також є комбінацією знань з різних областей - нейробіології і когнітивістики та розробки програмного забезпечення.

Вирішуючи завдання, не бійтеся фантазувати, не обмежуйте себе рамками тих знань, які у вас є, допускайте і намагайтеся обдумати варіанти, які можуть здаватися фантастичними. Цілком ймовірно, що такий підхід наштовхне вас вже на більш реалістичне рішення. Ця методика використовується в групових обговореннях, але ніхто не забороняє вам використовувати її і самостійно.

Спеціалізовані знання дуже корисні, але часто вони обмежують ваші творчі можливості. Обдумуючи проблему, генеруючи варіанти рішення, намагайтеся відкласти наявні у вас знання вбік, це дасть значно більшу кількість варіантів і поживи для роздумів.

Існує безліч творчих технік, що дозволяють вам збільшити кількість розглянутих варіантів рішень. Наприклад, спробуйте уявити, як би ви вирішували завдання, якби були кухарем.

Висновки

При розробці ПЗ іноді виникають завдання, які не вкладаються в типові рішення, вимагають обдумування і творчого підходу. У цьому випадку напружена, свідома робота над завданням може бути менш ефективною, ніж зміна діяльності або відпочинок. Дайте можливість вашій підсвідомості попрацювати для вас, підшукуючи підходяще рішення. Не обмежуйте свою фантазію і частину часу спробуйте забувати про свій досвід і наявні обмеження. Це сильно збільшить ваші творчі можливості.

Намагайтеся комбінувати наявні у вас рішення.

В ході одного проекту виникла необхідність візуалізувати граф бізнес-процесу як частини застосунка. Створення свого ПЗ, що вирішує дане завдання, було б невиправдано дорогим і зайняло б велику кількість часу. Як рішення, був знайдений існуючий, вільний у використанні редактор, що зберігає граф у текстовому форматі. Перетворити дані з файлу редактора на дані, що використовуються у застосунку, було елементарною задачею. Така комбінація дозволила дати швидке, якісне і в той же час дешеве рішення.

Існує багато літератури та методик, що стосуються творчих технік. Наприклад, Теорія розв'язання винахідницьких задач. Якщо у вас зустрічається велика кількість нестандартних завдань, вивчення подібних методик допоможе вирішувати їх ефективно. Так, відповідь на задачу про качок – три качки, що летять одна за одною: дві спереду, дві ззаду, одна посередині, три в ряд.

---

[1] Естаніслао Бахрах, Гнучкий розум.
[2] Wagner U, Gais S, Haider H, Verleger R, Born J. (2004) Сон надихає на осяяння. Nature. 427(6972): 352-5.
[3] Drosopoulos S, Wagner U, Born J. (2005) Сон покращує явне пригадування в пам'яті розпізнавання. Learn Mem. 12(1): 44-51.

Вирішуємо складні завдання в групі

Менеджмент — надто складний процес, щоб з ним могла впоратися одна людина.
Іцхак Адізес, «Ідеальний керівник. Чому ним не можна стати і що з цього випливає»

Сучасний світ з кожним днем стає все складнішим і динамічнішим. З одного боку, ми купаємося в океані інформації, з іншого — ця інформація не завжди правдива, повна та актуальна. Як колись давно людина придумала важіль для того, щоб підняти вантаж, який не могла підняти самостійно, сьогодні комп'ютери та програмне забезпечення дозволяють «підняти» обсяги даних, на обробку яких не здатний людський мозок. Однак це тільки погіршує ситуацію, оскільки самих даних стає все більше і більше, вони з'являються і змінюються з приголомшливою швидкістю.

Світ стає настільки складним, що людина перестає розуміти причинно-наслідкові зв'язки (саме цьому процесу зобов'язані своєю популярністю технології «великих даних» і машинне навчання). Те ж саме відбувається і з великими програмними системами — можливостей однієї людини часто не вистачає для вирішення завдань і розуміння того, як програмна система працює в цілому.

Для отримання найкращого результату надзвичайно важливо вміти приймати групові рішення, ефективно

взаємодіяти з іншими людьми, перекладати вирішення завдань з однієї людини на групу.

Мисливець дикого племені обережно пробирається лісом. Десь збоку він чує шелест листя і помічає злітаючого птаха. Постріл з лука. Птах, в якого потрапила стріла, падає на землю.

Групі піддослідних із кількох десятків людей показують банку з кульками і просять оцінити їх кількість. Відповіді досліджуваних усереднюють, і отримане число на частки відсотка відрізняється від реальної кількості кульок.

Що спільного в цих історіях? І перший, і другий випадок — це приклад прийняття рішення в умовах неповних даних, але такі, що призвели до точного результату. І подібне не є щасливим випадком. Мисливець регулярно

добуває собі їжу, а повторення експерименту дає схожий результат.

Мисливець не знає ваги стріли, відстані до птаха, швидкості його руху, але тим не менш влучає в ціль. Досліджувані не знають розміру кульок і об'єму банки, але їх груповий досвід дозволяє вирішити це завдання з дивовижною точністю.

У першому випадку результат дає спільна робота клітин мозку мисливця, а в другому — спільна робота членів групи. І в першому, і в другому випадку це обмін інформацією та використання досвіду тих, хто приймає рішення.

Кожну секунду ми приймаємо рішення — усвідомлені чи несвідомі. У результаті цього, трильйони клітин, що утворюють людський організм діють як єдине ціле. Якщо задуматися, як це відбувається, то й взагалі цей процес здається дивом. Наш мозок складається з приблизно 100 мільярдів нейронів. Це повністю децентралізована система: немає одного головного нейрона, який вирішує бігти чи битися, який маршрут краще вибрати, що замовити в ресторані.

Перелічені вище питання досить прості, приблизно ті ж завдання вирішують і тварини, та й варіантів вибору не так вже й багато. Але це на рівні всього організму, а як бути, якщо опуститися на рівень нервових клітин? Чи володіє окремий нейрон усією повнотою інформації? Чи може одна нервова клітина вашого мозку визначити, куди повернути на перехресті або оцінити швидкість наближення машини що прямує до вас?

А ось ще одна історія. 1986 рік, космічний шатл «Челленджер» вибухає при зльоті. Акції основних підрядників, які брали участь у запуску шатлу, падають через лічені хвилини після аварії. Відчутно більше за інших втратили у

ціні акції компанії, яка виготовила твердопаливний прискорювач, проблема з яким призвела до трагедії.

Якщо розглядати ціну акції на біржі як консенсус думок учасників ринку, то ми маємо дуже схожу ситуацію — тисячі людей на підставі своїх попередніх знань і досвіду приймають «рішення». У випадку з «Челленджером» біржа практично миттєво визначила винуватця катастрофи [1].

Це схоже на зображення на екрані, що складається з окремих точок. Кожна точка сама по собі нічого не означає, тільки всі разом вони представляють собою зображення.

Однак для того, щоб зображення було осмисленим, а рішення правильним, усі учасники повинні підпорядковуватися правилам, які дозволяють використовувати

інформацію, наявну в кожного, як елементи єдиної мозаїки.

Ми знаємо, що інформація в мозку передається у вигляді нервових імпульсів від однієї нервової клітини до іншої. Нервові клітини утворюють структури, в яких одна клітина узагальнює сигнали, отримані від інших клітин [2]. Результуючий сигнал передається далі по нейронній мережі. Фактично процес прийняття рішення в живому організмі являє собою безперервне «голосування» нервових клітин. Кожен нейрон, який приймає сигнал, є локальною «виборчою дільницею» для групи інших нейронів.

Мозок набагато гнучкіша структура, ніж усе інше тіло. Зв'язки в мозку і нервові клітини постійно змінюються під впливом зовнішнього середовища. Саме на цьому заснований процес навчання. Зіштовхуючись із чимось новим, ми рідко діємо найбільш ефективно. Перший раз сівши на велосипед, ви, швидше за все, впадете. Однак пробуючи знову і знову, навчаючи свій мозок і формуючи нові зв'язки, через деякий час ваше тіло робитиме правильні рухи (прийматиме правильні рішення).

Сучасна людина стикається з набагато складнішими завданнями. Це завдання зовсім іншого рівня складності (назвемо завдання рівня 2.0), коли окрема людина, подібно до одного нейрона, не в змозі з ними впоратися. У цьому випадку люди намагаються вирішувати проблеми колективно, використовувати знання та досвід групи.

Для вирішення завдань рівня складності 2.0 необхідна і складніша структура прийняття рішення, проте механізм взаємодії та правила ефективної взаємодії залишаються ті ж. Порівняємо роботу нейронів і учасників експерименту з кульками.

Кожен нейрон має свої обмежені дані (життєвий досвід окремої людини в експерименті), отримує нечітку

інформацію (банку з кульками) та генерує нервовий імпульс (припущення про кількість), який узагальнюється клітиною нервової мережі (середнє арифметичне в експерименті). Важливий момент для прийняття хорошого рішення групою людей: різнорідність групи. Чим більше відрізняються думки членів групи, тим краще буде результуюче рішення.

Як описувалось вище, рішення – це комбінація знань і досвіду, а також вхідних даних, навіть якщо вони нечіткі. Чим ширший досвід, тим краще вхідні дані можуть бути доповнені припущеннями тих, хто приймає рішення.

Повернімося до роботи мозку. Багато досліджень показують, що коли до людини приходить осяяння, в цей момент активізуються окремі, розрізнені ділянки мозку [3]. Творче рішення є нестандартною комбінацією наявних у людини різнорідних знань.

Досить популярна техніка мозкового штурму працює приблизно так само. На першому етапі збираються різні, навіть найшаленіші ідеї. На наступному етапі йде критика, комбінування, поліпшення цих ідей.

У цьому процесі кожен з етапів передбачає переробку інформації, отриманої на попередньому етапі. Точно так само як і обробка інформації в мозку передбачає проходження через кілька груп нейронів, кілька етапів.

Ми живемо у світі причинно-наслідкових зв'язків. Чим більше знань у людини з тієї області, в якій приймається рішення, тим більше ймовірність того, що рішення буде правильним. Цілком можливо, що експерт уже неодноразово вирішував подібну проблему.

У повсякденному житті ми часто користуємося інтуїцією або використовуємо це слово, якщо не можемо дати пояснення, чому ми прийняли те чи інше рішення. Однак інтуїція – це не магія. Інтуїція – це знання і досвід,

підсвідомо накопичений у мозку. Мозок у відповідь на зовнішню інформацію видає відповідь у вигляді емоцій або фізичних реакцій. Експерт високого рівня може не тільки дати правильну відповідь, але і інтерпретувати цю реакцію. Проте інтуїція стосовно питання у певній області може бути лише тоді, коли людина має досвід.

Повертаючись до колективного прийняття рішень, вага думки експерта при прийнятті рішення повинна бути більшою, ніж думка менш досвідченого учасника. При вирішенні специфічних проблем, які не мають аналогів у повсякденному житті, думка непрофесіонала і зовсім не повинна враховуватися через відсутність знань про причинно-наслідкові зв'язки в цій області. Побутові знання і логіка не можуть допомогти у вирішенні проблем, далеких від звичайного життя.

Колективні рішення можуть бути вкрай вдалими, але для того, щоб їх досягати, необхідно певним чином будувати спілкування у групі.

Голосування – популярне, але малоефективне групове рішення. Через свою простоту, воно застосовується дуже часто для вирішення широкого ряду завдань. Пряме голосування має такі недоліки: прямий причинно-наслідковий зв'язок і відсутність ваги думки того хто голосує.

Голосуючий найчастіше розуміє наслідки свого рішення. І якщо воно прямо пов'язане з особистою користю чи втратою, то рішення майже завжди очевидне.

Пофантазуємо і уявімо, що нейрон розуміє, до чого призведе переданий ним сигнал. Наприклад, до швидкого бігу, і, як наслідок, до нестачі кисню і харчування. Природним рішенням буде блокування такого сигналу, щоб продовжувати отримувати ресурси в необхідній кількості. І потрібно мати здатність мислити системно, щоб прийня-

ти правильне для всього організму рішення – не побіжиш зараз, через кілька секунд будеш убитий хижаком.

Більшість людей мислять простими логічними категоріями «якщо А, то Б» і це дає дуже широкі можливості для маніпуляцій колективним рішенням. Щоб уникнути цього, необхідно виключити суб'єктивне ставлення до прийнятого рішення або очевидний зв'язок між дією індивіда і наслідком.

У політичних виборах зробити це досить важко, хоча деякі виборчі системи, наприклад у США, мають більш складний механізм, ніж пряме голосування, саме для того, щоб підвищити "якість" голосуючих.

На щастя, у невеликих професійних групах ефективний процес прийняття рішення організувати набагато простіше, ніж вибори президента країни. Виглядати він може наступним чином.

Першим етапом є чітке формулювання проблеми, розуміння якого саме результату потрібно досягти під час обговорення. Наприклад, вибір сервера баз даних для нового проекту. Результатом обговорення буде назва і версія продукту, який буде використаний.

Наступний етап – критерії оцінки. Вибір тих атрибутів якості, які є найбільш важливими для вибору. Вже на цьому етапі можуть виникнути розбіжності, пов'язані з різним розумінням розв'язуваної задачі. Наприклад, хтось може вважати основним критерієм досвід команди розробників, а хтось – ефективність зберігання даних або швидкодію. Слід виключити неоднозначні або якісні критерії. Під якісними розуміється відсутність можливості їх об'єктивно оцінити або виміряти.

Визначення набору варіантів рішення. На цьому етапі всі охочі можуть внести свої пропозиції, один або кілька варіантів. На цьому етапі дуже важливо виключити елемент критики, оскільки це буде обмежувати можливий набір варіантів.

Останнім етапом буде порівняння отриманих варіантів згідно з початково вибраними критеріями.

Важливе значення має склад експертної групи. Як згадувалося вище, важливі знання та досвід учасників обговорення у вибраній проблемі. Мені доводилося спостерігати обговорення, на яких більша частина учасників не володіла областю, у якій проходило обговорення, і їх думки переважали над думкою експерта.

У групах із встановленою субординацією та ієрархією дуже важливо враховувати, щоб одні члени не тиснули своїм авторитетом та думкою. Я часто стикаюся з аргументацією: "у тебе більше авторитету", "він керівник, тому потрібно з ним погодитися" і тому подібне.

На обговорення впливають фактори, які не стосуються розв'язуваної проблеми (наприклад, посада учасника). Часто на результати обговорення впливають особисті стосунки.

Потрібно розуміти культуру і взаємини в організації для того, щоб мати можливість в принципі організовувати обговорення.

Важлива різнорідність учасників обговорення. Ваш неприємний у спілкуванні колега, який завжди має свою особливу думку, часто буває одним із найкорисніших у розв'язанні задач, пропонуючи несподівані ідеї або критикуючи думки авторитету.

Однак слід розрізняти ситуації, коли важливий фактор часу для прийняття рішення. Якщо час критичний, то найкраще мати або вибрати лідера, який буде приймати швидкі рішення, нехай і не найкращі.

Висновки

Робота людського мозку і процес прийняття ефективних рішень групою людей багато в чому схожі між собою. На сьогоднішній день мозок є найдосконалішим інструментом обробки інформації. Застосування механізмів роботи мозку до розв'язання задач вищого порядку дає дивовижні результати. Організація процесу прийняття рішення експертною групою на основі цих механізмів підвищить ефективність і точність рішень.

Слід враховувати такі фактори:

- різнорідність групи;
- наявність експертних знань у учасників групи;
- механізм агрегування думок групи;
- виключення особистої зацікавленості в рішенні.

Для агрегування думок можливо застосовувати статистичні методи, а також вводити додаткові обмеження, щоб думка одного учасника не могла переважити думки всіх інших. Для виключення особистої зацікавленості можна розробляти алгоритми, які дозволяють прибрати пряму причинно-наслідкову зв'язок між думкою і результатом.

Більше дізнатися про роботу мозку, розвиток мислення, нейропластичність та природу інсайту можна з науково-популярних книг:

- Девід Рок «Мозок, інструкція з експлуатації»
- Малкольм Гладуелл «Сила миттєвих рішень.Інтуїція як навичка»
- Джона Лерер «Як ми приймаємо рішення»
- Естаніслао Бахрах «Гнучкий розум»

---

[1] Джеймс Шуровьєскі. Мудрість натовпу.
[2] Джефф Хокінс. Про інтелект.
[3] Brain Activations and Functional Connectivity Patterns Associated with Insight-Based and Analytical Anagram Solving https://pmc.ncbi.nlm.nih.gov/articles/PMC7695184/

У пошуках колег

Молодий HR-менеджер приходить із купою резюме до директора:
— Дуже багато заявок, не знаю, як обрати найкращого…
Директор викидає більшу частину купи в смітник:
— Не люблю невдах!

Формування команди є одним із найважливіших компонентів стратегії управління складністю. Кожен член команди впливає на складність створюваного продукту. Створення артефактів проєкту — архітектура, написання коду, вибір інструментів, написання технічних завдань і тестів. Дії та рішення окремого члена команди можуть збільшувати або зменшувати складність проєкту.

Проте, окрім цього, важлива взаємодія між членами команди. Технічно підготовлений спеціаліст, який пише хороший код, але має погані навички комунікації, може створювати не менші проблеми, ніж спеціаліст з низькою кваліфікацією. Досить часто можна стикнутись з ситуаціями, коли замість реалізації того, що написано в технічному завданні, програміст писав код, що відображав його особисте розуміння проблеми. Члени команди повинні добре розуміти один одного і мінімально спотворювати інформацію при передачі.

Члени команди впливають на колективне прийняття рішень. У моїй практиці був випадок, який кардинально змінив ставлення до обговорення технічних завдань.

У групі зі мною працював доволі слабкий розробник. Найчастіше я пропонував йому готове технічне рішення в межах його компетенції. Але одного разу йому було запропоновано самому розібратися, як реалізувати необхідний функціонал. Коли він розповів своє бачення, я був сильно здивований, наскільки воно відрізнялося від того, що хотів запропонувати я. Його пропозиція була далеко не ідеальною, але вона була **іншою**. І в результаті гарячого обговорення було вироблено рішення, яке виявилося найкращим із усіх запропонованих. З цього був зроблений висновок: не можна нехтувати жодною думкою і, якщо дозволяє час, проводити обговорення технічних рішень із усіма членами команди, незалежно від їхньої компетенції.

Отже, кожен член команди впливає на результати роботи, тому до підбору колег потрібно підходити з усією серйозністю. Багаторічний практичний досвід показує: результати співбесід і роботи найчастіше відрізняються.

Ті, хто згодом виявилися найкращими працівниками, на співбесіді виглядали далеко не ідеально. Висновок — найкраща співбесіда, це випробувальний термін, а завдання співбесіди — відсіяти тих кандидатів, які точно не підходять.

Спробую розібрати деякі моменти при підборі кандидатів, заснованих на фактах із галузі роботи мозку.

Під впливом стресу людина не втрачає інтелект — вона втрачає доступ до нього.

Деніел Ґоулман

Співбесіда це стрес. Стрес тим більший, чим менше співбесід проходив кандидат. У цьому важливе протиріччя, хороший спеціаліст, який довго працює на одному місці, почуватиметься на співбесіді набагато менш ком-

фортно, ніж той, хто часто змінює місце роботи і, як наслідок, брав участь у багатьох співбесідах.

Різним людям потрібен різний час для адаптації. Був випадок, коли протягом усієї співбесіди кандидат виглядав дуже стиснутим, мало спілкувався. І тільки після закінчення співбесіди, коли ми залишилися вдвох, у неформальній обстановці, він перетворився на нормального співрозмовника. Людину наче підмінили — скутість зникла, і кандидат вів себе набагато вільніше.

Звідси напрошується висновок — необхідно зменшити рівень стресу кандидата, тоді ви зможете побачити найбільш наближену до природньої поведінку.

Зазвичай на співбесіді з боку роботодавця присутні кілька учасників, це також не додає впевненості в собі. Перед групою людей людина почувається менш упевненою.

Хорошою ідеєю може бути почати співбесіду з розповіді про компанію та проєкт, поступово залучаючи кандидата до спілкування. Таким чином, у того, хто проходить співбесіду, є час звикнути до навколишнього середовища і тих, хто з ним спілкується.

Одним із наслідків стресу є зменшення когнітивних здібностей. Пропонуючи на співбесіді вирішувати практичні завдання, ви фактично оцінюєте іншу людину.

Крім того, люди сильно відрізняються швидкістю мислення. Розробка програмного забезпечення далеко не та сфера діяльності, де швидше, автоматично означає краще.

Ставлячи кандидата в ситуацію з близьким терміном, коли відповідь потрібно дати за лічені хвилини, якщо не секунди, ви тим самим відсіюєте кандидатів, які готові ретельно, хоч і не швидко обдумувати рішення задачі.

До цього ж належать і всілякі завдання з онлайн-програмування, нестандартні питання або просто складні завдання, що потребують зосередження.

У дослідженні, що включало моделювання роботи мозку та метааналіз великої кількості досліджень, вчені визначили сильну негативну кореляцію між загальним інтелектом та швидкістю відповіді у складних тестах. Іншими словами, учасники з вищим інтелектом використовували більше часу, щоб дійти правильних рішень. [1]

Хорошою і перевіреною практикою є надсилання кандидату короткої анкети перед співбесідою. Це дозволяє структурувати розмову за підготовленими відповідями, а сам кандидат отримує час на обмірковування рішень і аргументацію своїх підходів. Крім того, заповнення анкети дає змогу заздалегідь оцінити рівень підготовки та відповідальність кандидата. У деяких випадках, якщо відповіді в анкеті свідчать про невідповідність вимогам,

це дозволяє зекономити час і відмовитися від подальшої співбесіди.

Важливою частиною тестування є перевірка логічного мислення кандидата. Дослідження підтвердили найбільшу кореляцію між навичками програмування та логічним мисленням [2]. Чітке розуміння причинно-наслідкових зв'язків не обмежується програмуванням і є надзвичайно важливим і під час опису задач на розробку, і під час тестування.

Оскільки розробка програмного забезпечення ґрунтується на спілкуванні, важливою якістю є вміння слухати та сприймати інформацію, а також чітко викладати свої думки. Наскільки у кандидата розвинені ці навички можна зрозуміти при безпосередньому спілкуванні.

Знання недостатньо; ми повинні застосовувати. Бажання недостатньо; ми повинні діяти.

Йоганн Вольфганг фон Ґете

Люди навчаються по-різному, деякі воліють брати практичне завдання, занурюватися у проблеми, що виникли, і в ході знаходження рішення набувати знань і досвіду. Значно менша частина, навпаки, вважає за краще читати книги, вивчати теорію. У цьому питанні дуже важливий баланс. Є багато спеціалістів із великим досвідом роботи, але із суттєвими пробілами у знаннях, що не дозволяє їм ефективно вирішувати робочі завдання.

Теоретики – це інша крайність, такі люди можуть багато розповісти і виглядати досить професійно, але коли справа доходить до практики, вони або з великими труднощами ці завдання виконують, або реалізація надмірно ускладнена. На етапі знайомства важливо зрозуміти, наскільки спеціаліст гармонійний з погляду теорії та практики. На жаль, повною мірою це можна зрозуміти лише

тоді, коли людина починає працювати та виконувати різноманітні практичні завдання.

Висновки

Кожен член команди має суттєвий вплив на її роботу. Підбір членів команди має надзвичайно важливе значення. Сама організація процесу підбору впливає на тип людей, які будуть обрані. Тестування навичок на самій співбесіді дасть вам більш стресостійких, але менш вдумливих працівників. Тому спочатку важливо визначити, який тип працівників вам потрібен.

Логічне мислення, навички комунікації, вміння вчитися – це ключові навички, які необхідні члену ефективної команди. Практичні знання та досвід є менш важливими і можуть бути досить однобокими.

З точки зору управління складністю, неможливо повноцінно реалізувати описані підходи, якщо інші члени команди не розуміють їх важливості та не дотримуються необхідних принципів. Навіть один учасник команди здатний створити достатньо проблем, щоб суттєво сповільнити загальний темп розробки та знизити якість програмного продукту. Саме тому ретельний підбір колег, які поділяють спільне бачення щодо управління складністю та здатні ефективно комунікувати, є критичним фактором для збереження простоти та підтримуваності програмного продукту.

---

[1] Learning how network structure shapes decision-making for bio-inspired computing
https://www.nature.com/articles/s41467-023-38626-y

[2] Relating Natural Language Aptitude to Individual Differences in Learning Programming Languages https://www.nature.com/articles/s41598-020-60661-8

Поглиблюємо розуміння, корисні принципи

Модель процесу розробки ПЗ

У кожній природничій науці стільки істини, скільки в ній є математики.
Іммануїл Кант

Як згадувалося в попередньому розділі, для розуміння та прогнозування люди будують моделі, що містять суттєві властивості модельованого об'єкта чи процесу. Побудуємо таку модель і ми, щоб краще зрозуміти, чому складність так суттєво впливає на створення програмного продукту.

Опис розробки ПЗ як процесу передачі та переробки інформації учасниками проекту було описано в самому першому розділі книги «Чим займаються розробники ПЗ».

Таким чином, наша модель складатиметься з первинного джерела інформації, зацікавлених осіб (стейкхолдерів), ланцюжка розробників, які перетворюватимуть цю інформацію на програмний продукт, і самого програмного продукту, який має відповідати цільовим характеристикам.

Виходячи з нашої моделі, ми подивимося, як різні параметри впливатимуть на час розробки програмного продукту та його якість.

Інформацію ми характеризуватимемо наступними параметрами: обсяг, достовірність і складність.

Достовірність (R) — діапазон значень від 0 до 1. Цей параметр менше 1, якщо інформація спотворена.

В даному контексті складність (C) відрізняється від того визначення, що було дано в попередніх главах. Тут ми її будемо розглядати не як когнітивне навантаження розробника, а як характеристику що дає нам уяву про заплутаність інформації. Це є величина комплексна, для реального продукту її можна розглядати як набір метрик програмного продукту. У нашій моделі ми визначимо складність як скалярне значення від 1 до нескінченності.

Вибір одиниці для початку інтервалу обумовлений тим, що складність за визначенням може мати тільки негативний вплив на успіх проекту. У подальших формулах ми використаємо це твердження.

Обсяг (V) – деяке позитивне число. З цим параметром все найпростіше, оскільки з одиницями вимірювання кількості інформації знайомий кожен.

Визначимо ті характеристики розробника, які нам важливі для побудови моделі.

p – продуктивність (productivity). Ця характеристика визначає, наскільки швидко розробник сприймає та генерує інформацію. Наприклад, бізнес-аналітик буде тим продуктивнішим, чим швидше він може поспілкуватись з зацікавленими особами та написати завдання програмісту.

Чим більш продуктивний програміст, тим швидше він напише код. $p > 0$

Для нашої моделі важливо те, наскільки «дбайливо» розробник поводиться з інформацією, зберігаючи її достовірність:

i – інтелект (intelligence). Перероблюючи інформацію, розробник може спотворити її.

Ця характеристика залежить від того, наскільки розробник добре сприймає інформацію, від його розумових здібностей і знань. Чим вищий інтелект, тим вищим буде поріг складності, при якому інформація є для розробника очевидною і буде опрацьована без спотворень.

Якщо складність інформації перевищує цю характеристику розробника, то розробник починає її спотворювати. Дуже грубо цю характеристику можна прирівняти до IQ.

a – заумність (abstruseness). Розробник може вносити додаткову складність в інформацію, з якою він має справу. Збільшення будь-яких метрик складності в ході розробки може характеризувати цю властивість розробника. Одну й ту саму інформацію можливо подати по-різному, програмний код можна написати різними способами.

Мої особисті спостереження говорять, що розробники дуже сильно відрізняються в плані внесення надлишкової складності в проект.

Це і якість коду, і архітектура, і використання різних технологій та компонентів. Якщо для розробника a = 1, то він передає інформацію як є, без привнесення додаткової складності. Для більшості розробників a > 1: у реальному проекті це дублювання коду, створення хаотичних зв'язків між компонентами, використання нестандартних рішень.

Так з'являється технічний борг, легасі код і системи, які в певний момент починають у прямому сенсі розвалюватися.

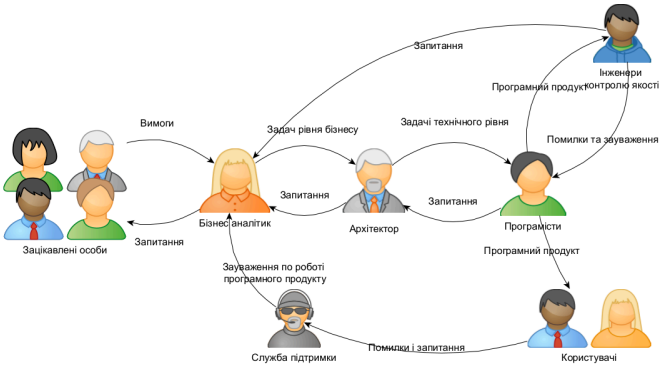
Але може бути і інша ситуація, проаналізувавши інформацію, побудувавши вдалу модель, розробник може спростити задачу. У цьому випадку a < 1.

f – показник критичного мислення або зворотного зв'язку (feedback). Цей параметр відображає здатність спеціаліста перевіряти отриману інформацію та виявля-

ти в ній помилки чи невідповідності. Параметр приймає значення від 0 до 1, де 1 означає ідеальне критичне мислення. Наприклад, тестер з показником близьким до 1 знайде практично всі помилки в програмі, а розробник одразу помітить суперечності у вимогах і повідомить про це автора технічного завдання. В нашій моделі, якщо достовірність інформації (R) менша за рівень критичного мислення виконавця (f), тобто R < f, то виконавець помічає проблему і повертає завдання на доопрацювання. Однак якщо достовірність R перевищує рівень критичного мислення f, то неочевидні помилки можуть залишитися невиявленими, і програмний продукт переходить або на наступний етап розробки, або відразу на робоче оточення.

Ці характеристики є функціями від складності: чим складніша інформація, тим легше її спотворити, тим важче знайти логічне протиріччя або помилку.

Наша модель представлятиме собою ланцюжок розробників, які приймають інформацію від зацікавлених осіб, обробляють її і передають далі. Розробники можуть давати зворотний зв'язок, тим самим уточнюючи інформацію, що потрапляє до них, виправляючи спотворення, що виникли в ході розробки. При передачі по ланцюжку найчастіше достовірність інформації (R) зменшується, складність (C) збільшується, оскільки кожен розробник розуміє її по-своєму і вносить свої зміни.



Процес комунікації при розробці програмного проекту

Складність сама по собі не становить інтересу, важливо як вона впливає на успішність проекту. Проект можна вважати успішним, якщо всі завдання по проекту були реалізовані з заданою якістю в запланований термін. Під якістю будемо розуміти відповідність продукту початковим вимогам: на кінцевому етапі достовірність (R) повинна бути не меншою ніж встановлений коефіцієнт якості.

Для різних продуктів вимоги до якості можуть суттєво відрізнятися. Персональний сайт і програма управління космічним апаратом мають суттєві відмінності у вимогах до якості.

На складність задачі для розробника буде впливати складність проекту. Реалізуючи кожну наступну вимогу, розробник працює не на порожньому місці. Нові вимоги повинні інтегруватися в існуючий проект і ми також повинні цю складність враховувати. Це та сама властивість, яка робить роботу з проектами з історією вкрай неприємною.

Невелика зміна у існуючому модулі з підвищеною складністю може зайняти дуже багато часу і бути зовсім нетривіальною — це перевірено на практиці багато разів.

Отже складність задачі для розробника будемо рахувати як

$$\frac{C_{task} + C_{project}}{2}$$

Складність проекту буде збільшуватись з кожною реалізованою задачею, але вплив кожної наступної задачі на складність проекту буде зменшуватись:

$$C_{n+1} = C_n + \frac{C_{task}}{N} * K$$

де N кількість реалізованих на поточний момент задач, а K деякий постійний коефіцієнт, який дозволить нам відобразити величини з якими ми працюємо на одному графіку.

Великий початковий внесок першої задачі можна пояснити необхідністю встановлення фреймворку та інших необхідних компонент, налаштування оточення розробки, опанування багатьох загальних речей стосовно проєкту.

Тепер спробуємо написати залежності, які допоможуть нам оцінити час на реалізацію проєкту.

T – час на опрацювання інформації виконавцем.

Чим більший обсяг інформації, чим вона складніша, тим довший час необхідний для її сприйняття та створе-

ння своєї версії (задачі, коду, тест-кейсу). Однак чим більш продуктивний і розумний розробник, тим швидше він зможе впоратись з задачею. Більш детально про вплив метрик складності на час можна прочитати у дослідженні [1].

$$T = \frac{V * c}{p}$$

c – це саме та складність яка створює когнітивне навантаження. Якщо складність перевищує інтелектуальні здібності, то вона починає суттєво впливати на розробку.

Якщо C < i (intelligence), то c буде 1, тобто не впливатиме на час розробки. Якщо C > i, то будемо рахувати її як c = 1 + (C - i) . Тобто для складної задачі (C > i) маємо:

$$T = \frac{V * (1 + C - i)}{p}$$

Перетворюючи інформацію, розробник змінює її характеристики: достовірність і складність. Як і у попередньому випадку, якщо C > i маємо:

$$R_{n+1} = \frac{R_n}{1 + C - i} \qquad C_{n+1} = a * C_n$$

Останньою складовою нашої моделі комунікацій буде зворотний зв'язок, який дають одні розробники іншим і який дозволяє виправляти помилки, впливати на достовірність R.

Якщо R < f/(1 + C - i) , то вважаємо що розробник знайшов помилку в реалізації. У цьому випадку він повертає задачу попередньому розробнику.

У нашій моделі значення F буде покращувати значення коефіцієнта R. У формулі це значення пропорційне до критичного мислення розробника. Чим менш достовірна інформація, тим більше можна побачити невідповідностей:

$$F = f * (1 - R) / (1 + C — i)$$

Крім того, будемо вважати, якщо тестер знайшов помилку і повернув задачу розробнику, він тим самим прояснив деякі невраховані розробником сценарії і розробник краще зрозумів задачу. Таким чином незважаючи на високу складність задачі розробник зможе її реалізувати. Проте час розробки значно збільшиться, так як буде включати в себе додаткові цикли тестування і виправлення помилок.

Зробимо припущення для спрощення формул: вихідні вимоги, за якими створюється програмний проект, розділені на рівні частини. У цьому випадку з формул можна прибрати V (обсяг завдання що виконується), а розмір проекту вимірювати кількістю завдань, необхідних для реалізації проекту.

Зберемо команду: аналітик (1), програміст (2), тестер (3). Порахуємо час на реалізацію одного завдання:

$$T_1 = \frac{1 + C_0 - i_1}{p_1}$$

Складність завдання після опрацювання:

$$C_1 = C_0 * a_1$$

Достовірність, розробник міг щось не так зрозуміти або переплутати:

$$R_1 = \frac{R_0}{1 + C_0 - i_1}$$

Далі інформація у якості завдання на розробку потрапляє до програміста. Розрахуємо ті сами параметри для етапу програмування:

$$T_2 = \frac{1 + C_1 - i_2}{p_2} \qquad C_2 = C_1 * a_2 \qquad R_2 = \frac{R_1}{1 + C_1 - i_2}$$

Тестер перевіряє реалізацію виходячи з вимог завдання і витрачаючи на це деякий час, без зміни складності та достовірності програмного продукту.

$$T_3 = \frac{1 + C_1 - i_3}{p_3}$$

Якщо достовірність більша за рівень критичності тестера, то завдання потрапляє на робоче оточення (f3 < R2) . Якщо навпаки, задача повертається на доопрацювання програмістом.

Загальний час на завдання складає:

$$T = T_1 + T_2 + T_3$$

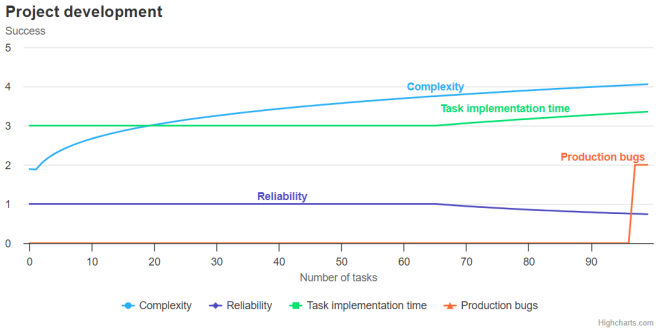
$$T = \frac{1 + C_0 - i_1}{p_1} + \frac{1 + C_1 - i_2}{p_2} + \frac{1 + C_1 - i_3}{p_3}$$

$$T = \frac{1 + C_0 - i_1}{p_1} + \frac{1 + C_0 * a_1 - i_2}{p_2} + \frac{1 + C_0 * a_1 - i_3}{p_3}$$

Як ми бачимо, найбільший вплив на час реалізації має складність C0 та параметр a1. Саме параметр a1 визначає, наскільки аналітик якісно впорався з аналізом вимог. Що підтверджує емпіричне спостереження, описане в главі 7 (Модель – фундамент для створення порядку): сильний аналітик створює фундамент проекту і має найбільший вплив на його майбутній успіх.

А тепер запрограмуємо цю модель і подивимось як вона поведе себе за різних обставин.

Почнемо з проекту невеликого розміру, з командою достатньої кваліфікації, але без розробника який розуміється на керуванні складністю.

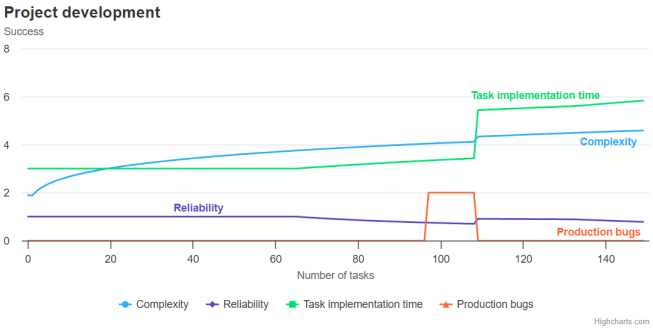


Як видно з графіку, на початковому етапі проєкту все йде за графіком, програмне забезпечення працює, але складність поступово зростає.

Через деякий час починають з'являтися неточності, і навіть виникає помилка на робочому оточенні. Проте проєкт можна вважати успішним. Умовний час на розробку становить 311 одиниць і 61 на комунікацію між розробниками.

У нашій моделі 3 розробники однакової продуктивності, тому на реалізацію однієї задачі за звичайних обставин іде 3 одиниці часу. Відповідно на 100 задач потрібно 300 умовних одиниць часу, але складність починає впливати на розробку і час роботи над задачами збільшується.

Наш проєкт продовжує розвиватися, від зацікавлених осіб надходять нові задачі.

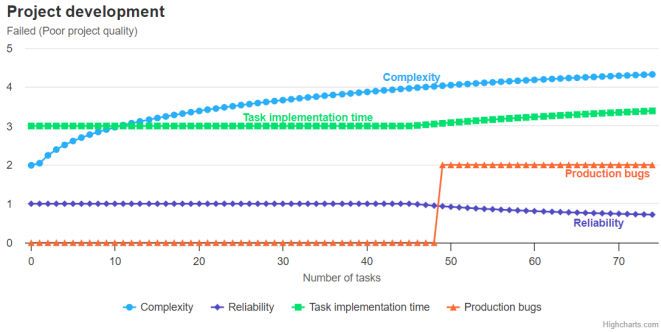


Бачимо різке зростання часу, який потрібний на реалізацію поточних задач. Це відбувається в момент, коли складність проекту досягає рівня кваліфікації та інтелектуальних здібностей розробників. У параметрах моделі цей рівень становить 4 умовних одиниці. Часові витрати на розробку і комунікацію становлять 585 і 110 одиниць. Проєкт починає буксувати. За рахунок проблем, що були знайдені на робочому оточенні і виправлені, якість проекту трошки збільшується, але продовжує далі погіршуватись у процесі розробки. Якщо ми продовжимо роботу далі, то приблизно через 660 задач розробники втратять можливість вносити зміни. Проєкт не зможе далі розвиватися.

Подивимось, що буде якщо погіршити якість нашої команди, а саме на 10% збільшити «заумність» програміста. Тобто наш віртуальний розробник буде писати більш складний код. Наш проект зазнає краху набагато швидше, а саме стане некерованим після 335 задач.

Проте цільова якість нашого проекту була закладена достатньо низька, а саме 0.75. Але і тестувальник який працює на проекті не дуже кваліфікований, його рівень в нашій моделі складає 0.7, тому він досить довго «не помічав» невеликі помилки розробника.

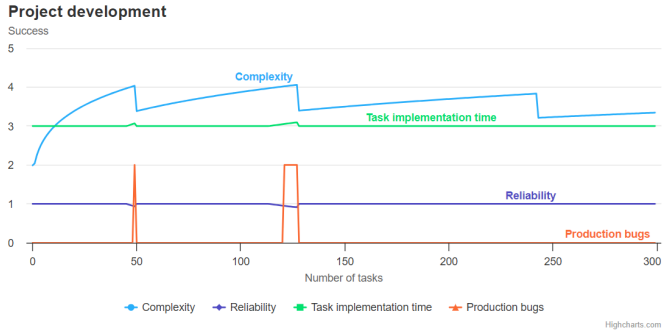
Подивимось що буде якщо значно підвищити вимоги до програмного забезпечення, умовно нашу команду яка розробляла новинний сайт залучити до розробки банківської системи з суттєво вищими вимогами до якості, а саме 0.95.



Наша модель показує, що дуже скоро команда досягне критичного рівня помилок на робочому оточенні і не зможе закінчити проект.

У випадку, коли починають виникати проблеми з розробкою і функціонуванням програмної системи, розробники починають частіше згадувати слово «рефакторинг». Це процес зміни внутрішньої структури програми без зміни її зовнішньої поведінки. Основна мета рефакторингу - поліпшення якості коду, його читабельності та підтримуваності. З точки зору нашої моделі, рефакторинг - це переробка програмного продукту для зменшення його складності. Але якщо ви будете розуміти, що таке керування складністю, ви не будете допускати збільшення цієї характеристики програмного продукту. Так виглядає процес розробки, який включає у себе рефакторинг. У момент, коли виникають проблеми, розробникам критично необхідно переробити програмний продукт. Головна проблема в тому, що це вимагає додаткового часу і високого рівня кваліфікації. На жаль, дуже часто розробники,

розуміючи необхідність переробки, не можуть її втілити в життя через недостатню кваліфікацію — впорядкувати хаос набагато важче, ніж його не допустити.



Сходинки на графіку складності відображають рефакторинг програмного продукту. Час на переробку не відображено на графіку, оскільки це виходить за межі нашої моделі.

Висновки

У даній главі описана модель, яка дозволяє формалізувати бачення процесу розробки як конвеєра з перетворення вимог замовників у програмний продукт. Модель відтворює розповсюджені сценарії розвитку програмних продуктів та дозволяє сформувати бачення впливу учасників проекту на його успіх. Знання мов програмування, алгоритмів, фреймворків і патернів програмування впливає значно менше на успіх програмного продукту, ніж вміння уважно слухати і писати зрозумілий код. Модель показує критичну важливість контролю складності в ході розробки, а також вибору членів команди.

[1] Evaluating code complexity triggers, use of complexity measures and the influence of code complexity on maintenance time. https://link.springer.com/article/10.1007/s10664-017-9508-2

Навіщо потрібна архітектура?

Архітектор — це той, хто знає різницю між тим, що можна зробити, і тим, що слід зробити.
Ларрі Маквой

Розробляючи програмні продукти, постійно доводиться чути слово «архітектура». У цьому розділі розберемо, чому архітектура необхідна для великих і складних систем.

Архітектура програмного забезпечення — це набір принципів, правил та обмежень, що визначають структуру та поведінку програмного продукту і його компонентів.

Чим масштабніший програмний продукт, тим більше значення має архітектура. Розглянемо аналогії з інших областей. Для прикладу візьмемо організацію груп людей залежно від їх кількості.

Ця аналогія доречна, оскільки часто змінюються умови, в яких функціонує програмний продукт (наприклад, збільшується кількість користувачів), а також може зростати обсяг даних, з якими доводиться мати справу.

На початку свого існування люди представляли собою окремі племена. Сьогодні такі племена можна зустріти в джунглях Амазонії, африканських пустелях або на островах Океанії — у всіх тих місцях, куди ще не дісталася сучасна цивілізація. Правила поведінки людей у таких групах прості, проблемні питання вирішуються колегіально. При

розмірі в кілька десятків людей група стабільна і більше залежить від наявності ресурсів, насамперед їжі, а не від внутрішніх правил поведінки. Максимальний розмір — кілька сотень людей.

Зі збільшенням продуктивності праці збільшилися і групи людей, які проживають спільно; суспільство розшарувалося. З'явилася керівна верхівка, і відбулося перерозподілення ресурсів. Стабільність такого суспільства підтримувалася насамперед силовими методами. Така політична структура називається вождівством. Ця архітектура суспільства дозволяла стабільно існувати значно більшій кількості людей, ніж у первісному племені — десяткам тисяч. Однак при подальшому збільшенні кількості груп, що мають силу, виникали конфлікти, що порушували звичний порядок речей.

Подальше збільшення соціуму було неможливе без прийняття законів — наборів правил, які регламентують поведінку членів суспільства. Законодавство є основою стабільності сучасних суспільств, саме воно дозволяє спільно існувати мільйонам людей і визначає структуру суспільства та поведінку сучасної людини.

Якщо порівнювати з розробкою ПЗ, невелика утиліта може бути написана як завгодно, головне, щоб вона виконувала свої прості функції. Для таких примітивних програм архітектура не має значення. При збільшенні розмірів і функціональності додаток розбивається на шари і модулі з певною відповідальністю. Додаток ще може контролюватися «вождем», технічним керівником проєкту.

При подальшому збільшенні масштабу виникає необхідність у правилах, що регламентують взаємодію підсистем. Розвиток системи визначається насамперед правилами та протоколами, а не директивами з боку керівника проєкту.

Отже, архітектура необхідна насамперед для того, щоб при збільшенні розміру система зберігала стабільність. При розробці ПЗ нестабільність проявляється у вигляді зростаючої кількості помилок, критичному зниженні продуктивності. Зі збільшенням розміру і складності застосунку розробники втрачають контроль над ним — збільшуються вартість і час, необхідні для модифікації системи. Для збереження контролю критично необхідне дотримання прийнятих протоколів і правил. З іншого боку, подібні обмеження зменшують швидкість розробки.

Порівняння розробки великого програмного продукту з організацією держуправління не настільки далекі одне від одного, як могло би здаватись. Культура команди розробки і наявність архітектури мають таке саме значення як мораль і закони в суспільстві. В цьому контексті архітектор виступає у ролі законодавця.

Проблема з програмістами в тому, що ви ніколи не можете сказати, що робить програміст, доки не стане надто пізно.

Сеймур Крей

Але одних законів замало, у людей повинно бути розуміння і бажання цих законів дотримуватись. Тому друга дуже важлива функція архітектора - навчання та виховання команди. Культура написання коду має прямий вплив на його складність. Культура спілкування в команді безпосередньо впливає на якість передачі інформації. При директивному управлінні зникає той самий вкрай важливий зворотній зв'язок, який корегує помилки, що виникають при розробці.

Неодноразово мені доводилось отримувати слушні зауваження від колег з приводу архітектурних правил які я сам і запроваджував. Людина зі своїми обмеженими розумовими можливостями не може пам'ятати про все. І

коли ти зосереджений на реалізації конкретної задачі, то легко можеш забути про додаткові вимоги. Саме дружня і відкрита атмосфера спілкування в команді дозволяє такі помилки виправляти.

Але часто буває і навпаки. Людина, що відповідає за архітектуру, в директивному порядку спускає своє рішення виконавцям. І відсутність бажання слухати зауваження, та корегувати рішення призводить до значних часових витрат з боку команди розробки, а крім того погіршує її моральний стан. Програміст стає заручником чужих недолугих рішень і це є одним із факторів що призводить до так званого «вигоряння».

Але повернемось до архітектури. Якщо брати аналогію з області біології, то архітектура грає роль скелету, який визначає зовнішній вигляд істоти. Скелет мають всі великі організми. Якби скелет був відсутній, організм не міг би підтримувати свою форму. Як наслідок, він би не міг нормально функціонувати. Скелет, а в розробці програм архітектура задають можливі напрямки розвитку, однак і обмежують межі зростання.

Уявімо, що людина буде необмежено рости; таке зустрічається при порушеннях у роботі гормональної системи організму і називається гігантизм. Органи не справляються з функціями підтримки життєдіяльності при збільшенні масштабу і це неминуче призводить до смерті.

Зростання програмного продукту призводить до проблем у функціонуванні. На щастя, програмні системи змінювати легше, ніж живі. У міру зростання архітектура системи може змінюватися, таким чином адаптуючись до нових умов. Найкращим варіантом розвитку подій є такий, коли можливість зростання закладена в систему з самого початку. Однак є ризик, що на початковому етапі

буде закладена така архітектура, яка не дозволить проєкту вкластися у бюджет і терміни.

Якщо вам не соромно за першу версію вашого продукту, ви запустилися занадто пізно

Рід Хоффман

Досвід показує, що найбільш успішним є підхід із накопиченням технічного боргу (складності), однак із можливістю максимально швидкого розвитку. Головне зробити продукт яким почнуть користуватись. У разі успіху з'являться наступні версії, з кращою архітектурою, в іншому ж випадку архітектура не буде мати значення, бо не буде користувача, який зможе її оцінити.

Висновки

Архітектура програмного застосунку впливає практично на всі його аспекти. Однак потреба в архітектурі виникає із зростанням програмного продукту. Вона дозволяє задати необхідні для стабільної роботи і розвитку обмеження, але в цілому зменшує швидкість розробки програмного продукту. Уміння зберігати баланс між стабільністю продукту та швидкістю розробки є одним із основних завдань архітектора програмного продукту.

Але архітектор повинен не тільки бути «законодавцем», але і вихователем. Культурний код команди впливає на успіх проекту не менше, ніж формальні архітектурні правила. Культура команди є гарантією того, що заявлена архітектура буде реалізована в програмному продукті.

Рекомендована література

1. Джаред Даймонд. Зброя, мікроби і сталь. Витоки нерівності між народами. Від егалітаризму до клептократії.
2. Джеффри Уэст. Масштаб.

3. Len Bass, Paul Clements, Rick Kazman. Software architecture in practice.
4. Деніел Койл. Культурний код. Секрети успішної взаємодії в команді.

Три стовпи складних систем

Природа проста і не розкошує зайвими причинами речей.
Ісаак Ньютон

Коли займаєшся розробкою програмних застосунків на різних рівнях, від проєктування архітектури до кодування, виникають думки про універсальні принципи, які допомагають приймати рішення, про які не шкодуватимеш у майбутньому.

Спостерігаючи за складними системами, створеними природою, не перестаєш захоплюватися, з одного боку, видимою складністю цих систем, а з іншого боку — їхньою лаконічністю. Наприклад, простий розрахунок обсягу інформації, якою закодовано будову людини, показує всього 1,5 Гб, а відмінність однієї людини від іншої становить частки відсотка!

Системи, створювані людьми, простіші: міста, соціальні структури, інженерні споруди. Проте вони все одно достатньо складні, щоб розглядати їх як приклади. До них також належать і програмні системи. Вважається, що програми є одними з найскладніших творінь людини.

Порівнюючи властивості систем, створених як природою, так і людиною, можна виділити три універсальні властивості: модульність, ієрархічність і типізація складових частин.

Нижче ми коротко розглянемо їх і наведемо конкретні приклади. Мова йтиме не просто про складні системи, а

саме про ті, що володіють високою стабільністю та надійністю.

Модульність

Складні системи частіше за все складаються з менших, але відносно самодостатніх частин-модулів. Наприклад, живі організми складаються з окремих клітин. Якщо розглядати людський організм, то можна виділити окремі органи. Місто складається з будівель. Армія (складна соціальна структура) — з окремих підрозділів. Для програмної системи такими частинами можуть бути функції, пакети, об'єкти тощо.

Частини, з яких складається складна система, приховують у собі особливості своєї реалізації та функціонування, а також взаємодіють із навколишнім середовищем через визначений інтерфейс. У програмуванні цей принцип отримав максимальний розвиток в об'єктно-орієнтованій парадигмі. Можливо, саме тому ООП найчастіше використовується при створенні великих систем.

Створюючи модуль, проєктувальник має можливість відволіктися від деталей його внутрішньої реалізації. З точки зору когнітивних здібностей людини, такий підхід дозволяє спростити процес проєктування, дозволяючи зосередитися на суттєвих функціях об'єкта та його інтерфейсі. Однак, розробляючи наповнення модуля, можна відволіктися від оточення, оскільки модуль має мати чітку специфікацію інтерфейсу.

Принцип модульності корисний як з точки зору створення об'єкта, так і з точки зору його функціонування. Об'єкт більш стабільний, якщо навколишнє середовище не має доступу всередину об'єкта. Будинок буде швидше руйнуватися, якщо в стіні чи даху буде отвір. Якщо генерал

буде напряму віддавати наказ солдату, минаючи його командира, то підрозділ стане небоєздатним. Ну і що відбувається з організмом, коли пошкоджується його оболонка, говорити не треба.

У програмному забезпеченні наслідками порушення принципу модульності стають помилки та нестабільність роботи. Саме тому глобальні змінні вважаються поганим тоном у розробці. В ООП для обмеження доступу до даних об'єкта використовуються спеціальні засоби. Хорошою перевіркою, чи є ваш код модульним, слугує можливість написання модульних (UNIT) тестів. Якщо вам легко вдається написати такий тест, то скоріше за все ваш код достатньо добре захищений від зовнішніх впливів.

Ієрархічність

У більшості складних об'єктів можна виділити рівні, на яких знаходяться сутності одного класу. Дуже простий приклад — книга. Рівні ієрархії об'єктів — літери, слова, речення, абзаци, розділи тощо. У даному випадку об'єкти більш високого рівня включають низькорівневі об'єкти. Проте можливі й інші варіанти. Наприклад, ієрархія управління. Інформація або керуючий вплив передається послідовно від рівня до рівня. Прикладом ієрархії управління може бути армія. Наказ головнокомандувача реалізується у вигляді послідовності наказів командирів підрозділів.

Навіть якщо розглядати таку на перший погляд невпорядковану структуру як кора мозку людини, можна виділити 6 шарів, що відповідають за певні функції [4]. Кровоносна система людини складається з невеликої кількості магістральних судин, які через менші судини та капіляри перерозподіляють кров по всьому організму.

Відображенням цього принципу при проєктуванні ПЗ є дуже поширена рівнева (шарова) архітектура. При написанні коду принцип ієрархічності може виражатися в тому, що на одному рівні не повинні зустрічатися артефакти різних масштабів. Наприклад, робота з окремими властивостями об'єкта і водночас масивами об'єктів.

Типізація

Розглядаючи об'єкти на одному рівні ієрархії, можна помітити, що в складних системах ці об'єкти однотипні: конструктивні блоки будівель, клітини в організмі, підрозділи в силових структурах тощо. Типізація передбачає, що об'єкти мають схожі властивості, однаковий інтерфейс. Саме це дає можливість і однорідно взаємодіяти з цими елементами системи, і керувати ними з більш високих рівнів.

Контейнерні перевезення - це яскравий приклад, де використання типових розмірів контейнерів змінило цілу індустрію. Уявіть порт, де кожен контейнер має власний унікальний розмір та систему кріплень. Це зробило б неможливим:

- використання стандартних вантажних кранів у портах;
- ефективне розміщення на кораблях-контейнеровозах;
- швидке перевантаження між кораблем, залізницею та вантажівками. Саме стандартизація морських контейнерів (ISO-контейнери з фіксованими розмірами 20 і 40 футів) зробила можливим сучасний рівень світової торгівлі.

Труби для водопроводу випускаються в типових розмірах. Для кожного розміру визначені точні зовнішній та

внутрішній діаметри, а також крок різьби, що забезпечує надійне з'єднання труб різних виробників. До кожного розміру труб виготовляються всі необхідні з'єднувальні елементи - від простих муфт до складних переходів, що дозволяє створювати водопровідні системи будь-якої складності.

Приклади використання принципу типізації при розробці ПЗ

- Мікроядерна архітектура, в якій розширення функціоналу реалізується за допомогою плагінів з типовим інтерфейсом.
- Шаблони проєктування «Стратегія», «Ланцюжок обов'язків», «Абстрактна фабрика» тощо.
- Структури даних у БД — таблиці, поля, колекції. Уявіть, наскільки незручно було б працювати з даними, якби для звернення до кожної окремої структури вимагалася б особлива команда.

Вирішуючи повсякденні задачі, не слід забувати про цей принцип. Якщо ви у кількох розрізнених задачах можете побачити типове рішення і звести їх у єдиний клас чи використати єдиний інтерфейс, ви можете значно пришвидшити розробку та зробити її більш якісною.

Існує декілька характерних ознак, які вказують на можливість застосування типізації для покращення коду:

Схожі структури даних. Якщо ви помічаєте, що в різних частинах системи використовуються схожі набори полів або властивостей - це перший кандидат для типізації. Наприклад, якщо ви працюєте з різними типами повідомлень (email, SMS, push-notifications), які мають спільні характеристики (отримувач, текст, статус доставки), варто створити загальний інтерфейс для всіх типів повідомлень.

Повторювані алгоритми обробки. Коли один і той самий алгоритм застосовується до різних, але схожих за структурою даних - це привід для створення спільного типу. Наприклад, якщо ви валідуєте дані користувачів з різних форм за схожими правилами, можна створити загальний механізм валідації на основі типізованих правил.

Дублювання коду з невеликими відмінностями. Часто дублювання коду виникає через те, що розробник не помітив можливість абстрагувати спільну функціональність за допомогою типізації.

Які ж практичні наслідки з описаних принципів?

В першу чергу про них потрібно пам'ятати на етапі аналізу та проєктування. На жаль, далеко не завжди розділення на модулі та шари є очевидним. Побудова моделі предметної області є складним аналітичним завданням. Переробляйте модель предметної області доти, поки вона не буде відповідати описаним вище принципам.

Модулі повинні бути закриті від впливу ззовні, рівні ієрархії не повинні змішуватися, об'єкти, що виконують однакові функції, повинні бути типізовані.

Добре допомагають при розробці шаблони проєктування — перевірені часом рішення, які можна використовувати при реалізації компонентів вашої системи.

Висновки

У цій главі розглянуті принципи, які присутні в будь-якій складній системі, що стабільно функціонує. Розробник під час роботи над системою повинен постійно аналізувати свої дії з точки зору дотримання зазначених принципів. Інакше складність системи буде неконтрольовано зростати, а її стабільність зменшуватися.

Правила, описані в цій главі, можна застосовувати для отримання відповідей на питання про розбиття системи на компоненти та побудові моделі предметної області

Що таке добре і що таке погано

Один чоловік зустрів у лісі дикукобилуі взяв її собі.
— Ого-го! — сказали сусіди, — от просто взяв і завів кобилу, пощастило тобі!
— Не знаю, пощастило мені чи ні… — відповів він.
Його син став об'їжджати кобилу, але вона була норовливою і скинула його. Синзламав ногу.
— Ах! Яке нещастя! — вигукували сусіди, — як погано!
— Я не знаю, добре це чи погано, — відповідав чоловік.
Незабаром почалася війна, і всіх придатних юнаків забрали в армію. Сусідські сини теж пішли на війну і загинули.
— Добре тобі, — говорили залишені без дітей люди, — твій син залишився живий.
— Я не знаю, добре це чи погано, — все одно відповідав чоловік…
Буддійська притча

Людина схильна до мінімізації розумових зусиль, необхідних для виконання завдання. Це може включати спрощення процесів прийняття рішень, уникнення складних завдань або використання очевидних шляхів вирішення проблем. Ключовим аспектом є оптимізація процесів мислення таким чином, щоб досягти бажаних результатів з меншими витратами. Одним з таких прийомів є використання спрощених, крайніх характеристик.

Правильний – неправильний, хороший – поганий, швидкий - повільний. Такий поділ дозволяє менше дума-

ти і спростити прийняття рішень. Однак навколишня дійсність набагато складніша і різноманітніша, і часто вибір з кількох альтернатив є складним завданням.

У IT-спільнотах досить часто виникає явище, яке називається холівар (від англійського holy war) - опис гарячої, часто емоційної та малопродуктивної суперечки чи дискусії. Наприклад, яка ОС краща, Linux або Windows. Такі суперечки виникають саме через багатогранність обговорюваних предметів. І в таких суперечках ми як аргументи можемо почути описані вище якісні характеристики, що відображають емоційне ставлення людини до предмета або посилання на попередній досвід. Використання таких аргументів майже ніколи не призводить до конструктивних рішень.

Отже, для того щоб використовувати оцінки «хороший»/«поганий», «гірший»/«кращий» насамперед необхідно визначити характеристику предмета, за якою йде оцінка чи робиться порівняння.

Розглянемо, що необхідно робити для зваженої оцінки. Цей підхід можна застосовувати для вирішення будь-яких завдань, у тому числі й для вибору технічних рішень, архітектури, інструментів.

Спершу складемо список властивостей, які мають бути враховані. Це може бути що завгодно: швидкодія, час або витрати на розробку, точність розрахунків тощо. Таких властивостей не повинно бути багато. У розробці програмного забезпечення список таких властивостей вже існує і називається атрибутами якості. Атрибути якості — це ключові характеристики, які визначають рівень якості програмного забезпечення. Оцінюючи програмне забезпечення за цими характеристиками, можна визначити, наскільки ефективно і надійно воно виконує свої функції.

Упорядкуємо вибрані властивості за важливістю. Без розстановки пріоритетів у разі суперечностей неможливо буде зробити вибір на користь того чи іншого рішення.

Припустимо, ви вибираєте автомобіль. Залежно від можливостей та потреб набір властивостей може сильно відрізнятися. Якщо ви збираєтеся подорожувати, ви шукатимете надійний та економічний автомобіль. Якщо ж ви зібралися в експедицію, то на першому місці у списку з'явиться прохідність.

Якщо ви не один або вам багато чого знадобиться взяти з собою, то до списку потрібно буде додати місткість. І для остаточного вибору доведеться вирішувати, яка з властивостей буде важливішою.

Папуга навчився говорити «залежить від контексту» і був прийнятий на посаду системного архітектора.

Автор невідомий

Досить часто люди, які працюють з проєктами з історією (legacy) не найкраще відгукуються про технічні рішення, що використовуються в проєкті. Однак для об'єктивної оцінки необхідно розуміти, в який момент часу і чому таке рішення приймалося. Цілком ймовірно, що з часом змінилися умови, вимоги та пріоритети. І те, що в якийсь момент вважалося «хорошим», з часом перетворилося на «погане».

Або навпаки, рішення, яке вважалося б вдалим для повноцінного, працюючого на повну силу проєкту, може виявитися шкідливим на початковому етапі. До такого можна віднести використання складної архітектури, особливо якщо відсутній досвід створення аналогічних проєктів.

Насамперед слова «правильно»/«неправильно», «добре»/«погано» доводиться чути від фахівців середньої кваліфікації. Вони вже досить упевнені у своїх силах, щоб висловлювати свою думку. Однак нестача знань і досвіду не дає їм можливості оцінити всі можливі плюси та мінуси.

Висновки

При обговоренні питань, ухваленні рішень, використання якісних оцінок з дуже малою ймовірністю приведе до необхідного результату. Найчастіше їх використання свідчить про погане розуміння проблеми або недостатню кваліфікацію того, хто використовує такі оцінки.

Якісні оцінки є маркером того, що обговорення може зайти в глухий кут. У цьому випадку необхідно просити уточнити, який саме критерій використовується для такої оцінки.

Як виміряти програміста

Якщо ти знаєш своїх ворогів і знаєш себе, ти можеш перемогти у сотні битв без жодної поразки.
Сунь-Цзи

Працюючи у великій корпорації і спілкуючись з десятками спеціалістів, пов'язаних з розробкою програмного забезпечення, у мене виникали різні ідеї щодо оцінки персоналу: кваліфікації, якості роботи, продуктивності.

Справа в тому, що в невеликих колективах всі працівники на видноті. Якщо один з членів команди не справляється з покладеними на нього завданнями, це одразу відчувають його колеги.

У великій компанії ситуація складніша. Велика структура не може існувати без управління, правил, робочих процесів. Вплив окремого працівника на результати не такий суттєвий і помітний, як у маленьких колективах. В

середньому це дозволяє працювати менш продуктивно, без серьозного впливу на кінцевий результат.

Зменшення середнього індивідуального внеску в групову роботу зі збільшенням кількості членів групи називається ефектом Рінгельмана і був описаний ще у далекому 1913 році. Тому у великих компаніях періодично з'являються працівники, які паразитують на компанії або просто малоефективні.

У великих організаціях багато залежить від думки безпосереднього керівника, який, у свою чергу, далеко не завжди може адекватно оцінити працівника або для оцінки використовує рівень особистої лояльності, а не робочі якості співробітника. І навіть якщо в організації введено методики оцінки типу 360 або KPI, вони також не дозволяють системно і якісно оцінити роботу програміста. Область розробки програмного забезпечення особлива тим, що результат і виконана робота прямо не пов'язані: можна працювати менше, а результат отримувати кращий. Причому те, що зроблено, може не дати видимого результату сьогодні, проте матиме суттєвий ефект у майбутньому.

Мені доводилося мати справу з розробниками, які реалізовували «одноденні» рішення — з великою ймовірністю код, написаний ними, доводилося переписувати в майбутньому. Але більшість людей що їх оточували, як замовники, так і колеги, були цілком задоволені їхньою роботою. Проте для проекту який постійно розвивається та розрахований на довготривале використання, такі розробники є малокорисними, а іноді навіть шкідливими. Вони сприяють виникненню помилок у роботі програмного продукту, збільшенню часу реалізації нових завдань.

Оцінити знання працівника можна шляхом проходження тестів і проходженням сертифікацій. А ось як бути з продуктивністю і якістю роботи?

Карма має конкретний вплив:неправедні вчинкизавжди викликають страждання, а добродійні завжди приносять щастя. Якщо ви сієте добро, то пожнете щастя; якщо ви сієте зло, то пожнете страждання.

Далай-лама XIV

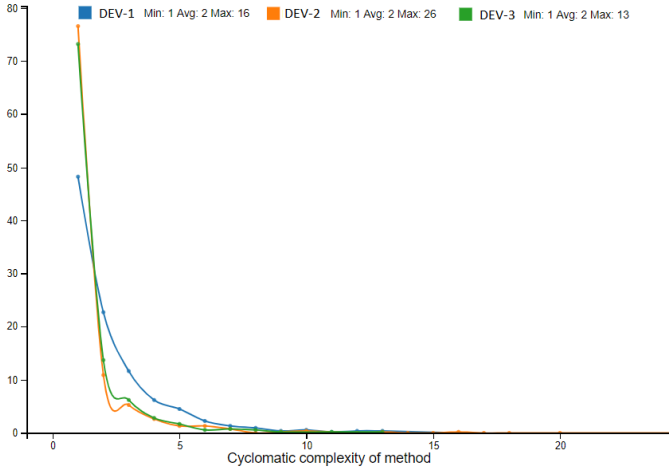
У деяких релігіях існує поняття « карма «: наслідки твоїх дій впливають на твоє майбутнє. Та ж ідея може бути застосована і до програміста, який працює над проектом. В даному випадку кармою є та складність, яку програміст вносить у проект. За неї він і його колеги будуть розплачуватися у наступних ітераціях втраченим часом і помилками.

Один і той самий функціонал може бути реалізований багатьма різними способами. Як це буде зроблено, залежить саме від того, хто пише код. При аналізі великого обсягу коду виявляється, що одні програмісти пишуть код складніше, ніж інші. А саме — для коду, написаного одним програмістом, усереднені метрики складності будуть більші, ніж для коду, написаного іншим.

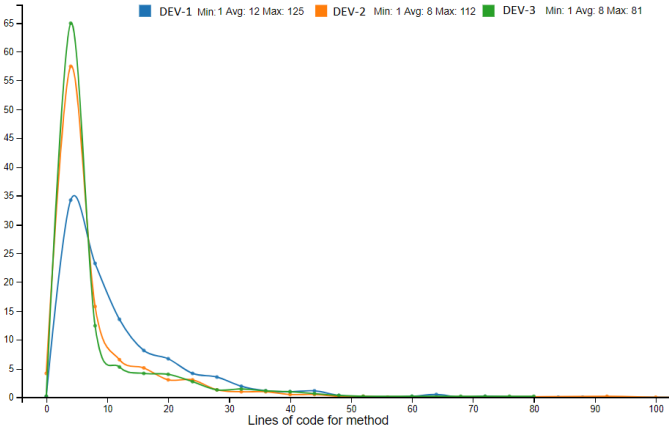
В якості ілюстрації візьмемо дані аналізу кількох програмістів, які працюють над одним і тим самим проектом. Для різних артефактів програмного проекту були розраховані метрики складності, потім в залежності від того, хто писав код був проведений перерахунок значень метрик у розрізі програміста. По горизонталі відкладаються значення метрик, по вертикалі кількість артефактів, що мають цю метрику. Кожна крива на графіку представляє дані по конкретному програмісту. Для можливості порівняння дані були пронормовані залежно від кількості коду, написаного програмістом.

Нижче представлені графіки, побудовані на основі різних метрик по трьох програмістах.

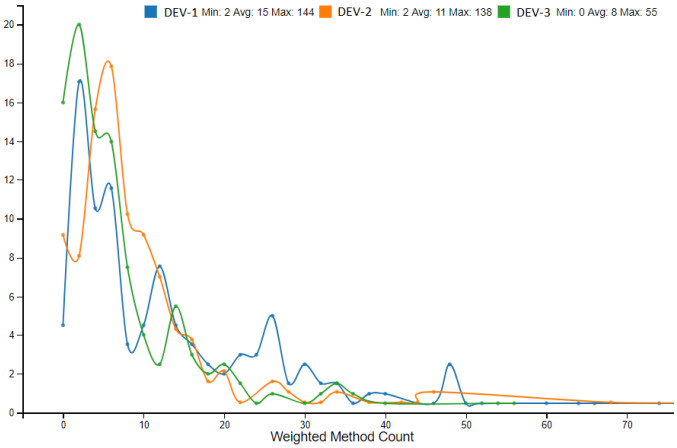
Цикломатична складність методу



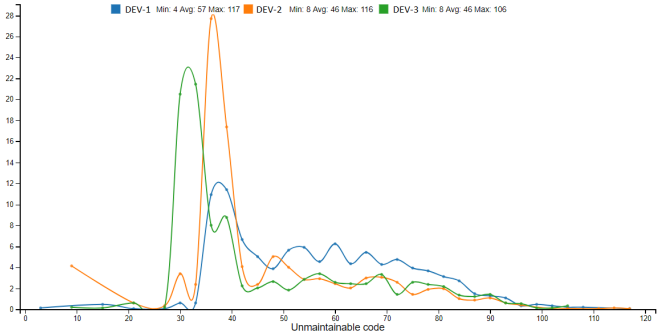
Кількість рядків коду в методі



Зважений підрахунок методів



Складність супроводу коду. Розраховувалась як обернена до оригінальної метрики Індекс супроводжуваності коду для зручності побудови графіка.



Як видно з графіків, синя крива знаходиться вище, що означає більш складний код, написаний програмістом DEV-1. Програмісти DEV-2 та DEV-3 пишуть код приблизно однакової складності. Формально всі вони мають одну і ту ж кваліфікацію, але аналіз показує, що між їхнім кодом є відчутна різниця. Особливо цікаві ці дані, коли тобі відомий досвід людей, ставлення до них у колективі та їх самооцінка.

Ці графіки дозволяють порівнювати членів команди і звертати увагу на тих, з ким варто попрацювати щодо навичок написання коду.

Приходить секретар влаштовуватися на роботу.

Рекрутер: З якою швидкістю ви можете друкувати на клавіатурі?

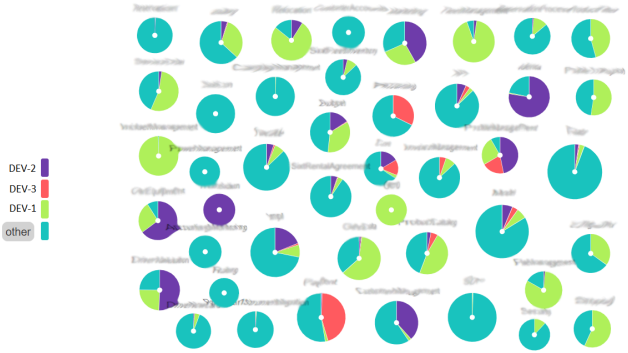
Секретар: Ну… 1000 знаків за хвилину.

Рекрутер: Хіба можна з такою швидкістю друкувати?

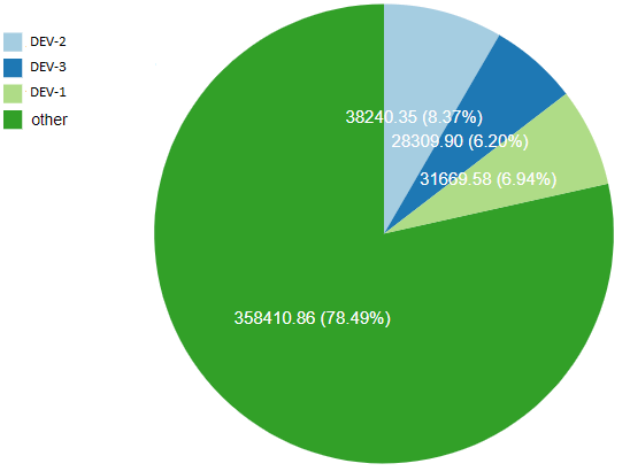
Секретар: Друкувати-то можна, але така нісенітниця виходить!

Те, що продуктивність програмістів може дуже сильно відрізнятися, відомо дуже давно: «Програміст з річною зарплатою 20 000 доларів може бути в 10 разів продуктивнішим, ніж програміст з річною зарплатою 10 000 доларів. І навпаки. Дані не показали абсолютно ніякої кореляції між досвідом та продуктивністю» [1]. Можна точно стверджувати, що одні розробники працюють швидше за інших. Але якщо поставити питання, як формалізувати вимірювання продуктивності, то відповідь буде далеко не очевидною. У виробництві, де продукція вимірюється в одиницях продукції, можна поділити кількість продукції на час і отримати потрібний результат. Але що можна назвати продукцією в контексті розробки ПЗ? Рядки коду, файли, вирішені задачі? Всі ці речі відрізняються за обсягом і складністю. І навіть якщо дати одну і ту ж задачу двом програмістам і заміряти час на її вирішення, ми все одно не отримаємо об'єктивної картини, хто з них «кращий». Як вже писалося вище, рішення можуть суттєво відрізнятися: код, написаний з більшою швидкістю, може виявитися непридатним через деякий час і зникнути з проекту. Існує думка, що виміряти продуктивність у масштабах усього бізнесу неможливо [2], і з цим я сперечатися не буду. Однак, якщо оперувати артефактами

всередині програмного проекту, то дещо виміряти можна. Кореляція між обсягом коду (рядків, класів, методів тощо), написаного програмістом, і функціональністю програми існує. І можна спробувати виміряти внесок, який кожен розробник зробив у проект, і порівняти їх між собою. Розглянемо два випадки. У першому програміст Василь для вирішення задачі просто скопіював вже наявний код. У другому випадку програміст Микола створив компонент, який використав як для першої, так і для другої задачі. При цьому, вирішуючи аналогічну задачу, він знову зможе скористатися цим компонентом і заощадить час. Якщо ж Микола поділиться цим напрацюванням з колегами, і ті почнуть застосовувати його розробку, то цей код стає ще більш цінним — і, як наслідок, внесок Миколи в проект буде збільшуватися з кожним новим застосуванням створеного ним компонента. Як визначити формально, який код має більшу вагу? Свого часу схожу задачу вирішували творці Google щодо документів в інтернеті. І якщо ми можемо використовувати Page Rank для визначення важливості документа, то чому б не застосувати його до коду? Отже, внеском програміста в проект я буду називати обсяг написаного коду, помножений на його значущість (Page Rank). Слід розуміти, що рахувати його також можна по-різному, і справа не в абсолютній точності, а у можливості порівняння. Розглянемо все тих же трьох реальних розробників нашого проекту. На рисунку нижче показані пакети, в які вносили зміни розробники. Розмір сектора діаграми відповідає відсотку рядків коду в модулі, написаних програмістом. Як бачимо, розробники досить часто працювали в рамках одних і тих же модулів.



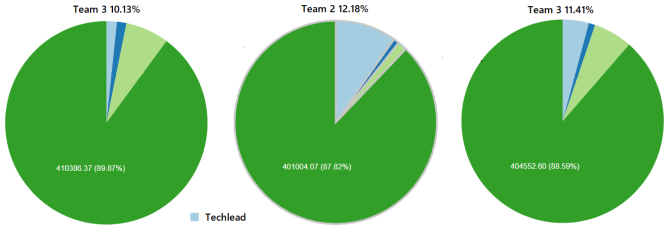
А на цій діаграмі зображений загальний внесок кожного з розробників у проект: кількість рядків коду, помножених на їх значущість.



Ці ж дані представлені у вигляді таблиці:

Розробник	Рядків коду, тисячі рядків	Рядків коду, %	Внесок у проект, %	Середня значущість коду
DEV-1	50	9	6,94	0,139
DEV-2	36	7	8,37	0,233
DEV-3	23	4	6,2	0,269

Середня значущість коду отримана діленням внеску в проект на кількість рядків написаних розробником. Наприклад: 6.94 / 50 ~ 0.139Найбільше пише DEV-1 (50 тисяч рядків, або 9% коду проекту), але його середня значущість майже в два (!) рази менша, ніж у DEV-3. Якщо врахувати, що цей код ще й більш складний, то можна зрозуміти, звідки береться легасі — чим складніший код і чим його більше, тим важче його супроводжувати. Усі розрахунки показують, що розробнику DEV-1 необхідно приділити пильну увагу. Подібна методика дозволяє робити висновки не тільки про роботу окремих програмістів, але і цілих команд. Діаграма нижче показує внесок у той же самий проект в розрізі команд.



Як бачимо, команди більш-менш збалансовані, внесок у проект у них приблизно однаковий. Однак, схоже, принципПарето можна застосувати і до написання коду. Видно, що практично всюди є свій «локомотив», програміст який реалізує більшу частину функціоналу, що розробляється командою. А також те, що одні розробники в рази відрізняються від інших з точки зору внеску в проект. Ну, і цікавий висновок для команди 2. Техлід пише в кілька разів більше ніж інші члени команди. Дуже схоже, що він діє за принципом «Хочеш зробити добре — зроби це сам»…

Висновки

Хороший програміст (для довгострокового проекту) — той, хто «інвестує» у майбутнє — збільшує кількість коду, який можна використовувати повторно, і мінімізує складність, що гальмує розвиток. Я не просто так зробив уточнення про тривалість проекту. Слід розуміти: якісна оцінка залежить від обраного критерію. Якщо задачу потрібно вирішити якомога оперативніше, то кращим буде той програміст, який працює швидше, а не пише зрозумілий код. Для аутсорс компаній, які беруть з клієнта погодинну оплату і фактично перепродують час своїх працівників, можливо, кращим програмістом буде «середнячок», той хто не видає клієнту ідеальний результат, але дозволяє при цьому заробити своїй компанії. Запропонована методика дозволяє формально оцінити команду розробників і, як наслідок, підвищити її ефективність або своєчасно прийняти управлінські рішення.

---

[1] Фредерік Брукс, Міфічний людино-місяць
[2] Мартін Фаулер, Не можна виміряти продуктивність https://martinfowler.com/bliki/CannotMeasureProductivity.html

Повітряний замок, який став реальністю

Добре, що нікому не потрібно чекати ні хвилини, щоб почати робити світ кращим.
Анна Франк

Деякий час тому я потрапив на проєкт як пожежник. Цілком успішний бізнес зазнавав збитків через проблеми з програмним забезпеченням. Це були як помилки в роботі, так і проблеми, пов'язані з продуктивністю, що не дозволяли обслуговувати необхідну кількість клієнтів. Одна з помилок у коді призвела до прямої втрати великої суми грошей. Історія про неї стала місцевою легендою, її мені переповідали різні люди, що давно працювали на проєкті. Які втрати були у бізнесу з інших причин, можна тільки здогадуватися.

Проблеми стали досить відчутними приблизно через рік від самого початку проєкту. І розробники вирішили переписати проєкт, зробивши все «правильно». Від моноліту було вирішено перейти до мікросервісів, а також замінити один фреймворк на інший. Те, що заміна технології допоможе вирішити помилки в розробці – дуже поширена хибна думка, описана в розділі **«Лакмусові папірці»**. Проєкт став ще складнішим, продуктивність не збільшилася, а кількість помилок зросла. Те, що мало стати окремими сервісами, перетворилося на розподілений моноліт — всі вони використовували одну й ту ж базу даних.

З'явилися нові помилки, пов'язані з новою реалізацією. Наприклад, один з «сервісів» записував дані в БД і надсилав повідомлення через брокер повідомлень іншому «сервісу». Однак логіка була побудована так, що перший «сервіс» міг не встигнути записати дані, а другому ці дані були необхідні для роботи. Якщо повідомлення починало оброблятися до того, як у сховищі з'являлися дані, виникала помилка. Або не виникала, якщо сервер бази даних виявлявся швидшим за брокера повідомлень.

Супроводжувати інфраструктуру і шукати причину проблем стало складніше. Для того, щоб виправити становище, керівництво компанії вирішило замінити IT менеджмент.

Відтоді минуло кілька років. Не можна сказати, що помилки на робочому середовищі зникли зовсім, проте їх кількість і критичність значно зменшилися. Часто вони пов'язані не з проблемами в коді, а з особливостями роботи зовнішніх систем, з якими відбувається взаємодія. Бізнес звик до того, що новий функціонал з'являється в строк, водночас відділ розробки не робить героїчних зусиль для того, щоб вкластися в терміни. Кількість розробників принципово не змінилася, кількість сервісів зменшилася – система розпалася на кілька компонентів, що взаємодіють через API, працюють у своїй зоні відповідальності, маючи свої окремі сховища.

Проєкт не переписувався з нуля, багато частин коду залишилися колишніми. Фреймворки, мови програмування та інші технології також не змінилися.

Що ж сталося? У першу чергу з проєкту зникли компоненти з надмірною складністю. Вони були перепроєктовані та реалізовані заново, це було зроблено передусім через відсутність зрозумілих моделей предметної області (глава **«Фундамент для створення порядку»**). Частина компонентів була типізована, таким чином багато нових завдань реалізуються за стандартною схемою. Те, що командою першої хвилі реалізовувалося програмуванням, у поточній версії вирішується зміною конфігурації і займає значно менше часу на розробку. Більшість принципів, описаних у цій книзі, були використані на практиці при внесенні змін.

Проєкт далеко не ідеальний, постійний розвиток призводить до того, що іноді доводиться йти на компроміси і залишати вирішення проблем на потім. Однак через постійний контроль і управління складністю залишається час на усунення технічного боргу.

І звісно великий внесок у стабільність проекту вносять суміжні до групи розробки відділи: команди тестування і підтримки інфраструктури. Без їх чіткої роботи було б набагато важче досягти успіху.

Ідеї втілюються у реальне програмне забезпечення, яке працює незважаючи на наявність «історії», повітряний замок програмного продукту не розвалюється під тиском надмірної складності.

Якісний програмний продукт може і повинен бути реальністю, і сподіваюся, що ця книга допоможе вам самим переконатися в цьому на практиці.

Від чого розвалюються повітряні замки? Введення у керування складністю при створенні програмних систем

Руслан Дмитракович

Мова видання: українська

Ілюстрації: Юлія Дмитракович. Частина ілюстрацій згенерована за допомогою AI (Midjourney).

Електронна версія книги доступна на сайті
https://complexity-book.com

PDF згенеровано 2026-04-18