Метапрограмування

Мультипарадигмовий підхід в інженерії програмного забезпечення

Київ, 2015 — 2025

Аннотація

Усі програми - це дані. Одні дані інтерпретуються, як значення, а інші - як типи цих значень, треті - як інструкції з обробки перших двох. Будь- які парадигми та техніки програмування - це лише спосіб формувати метадані, які дають правила і послідовність потоку обробки інших даних. Мультипарадигменне програмування бере краще з всіх парадигм і будує з них синтаксичні конструкції, що дозволяють більш зрозуміло та зручно описати предметну область. Ми зв'язуємо високорівневі DSL (доменні мови) із низькорівневими машинними інструкціями через безліч шарів абстракцій. Тут важливо не фанатично слідувати одній парадигмі, а найефективніше відображати завдання для виконання на машинному рівні. Найефективніше - це з меньшою кількістю шарів та залежностей, найбільш зручно для розуміння людиною, для підтримки та модифікації, забезпечення надійності та тестованості коду, розширюваності, перевикористання, зрозумілості та гнучкості конструкцій метаданих на кожному рівні. Ми вважаємо, що такий підхід дозволить отримувати як швидкі перші результати в розробці кожного завдання, так і не втрачати темпів при більшому потоку змін на етапах, коли проект вже достиг високої зрілості та складності. Ми постараємося розглянути прийоми та принципи програмування з різних парадигм через призму метапрограмування та не стільки змінити цим програмну інженерію, як її осмислення новими поколіннями інженерів.

Зміст

1. Вступ

Постійне переосмислення своєї діяльності, навіть найпростішої, має супроводжувати інженера все його життя. Звичка записувати свої думки словами та відточувати формулювання дуже допомагає у цьому. Текст цей з'явився як мої уривчасті нотатки, написані в різні роки, які я накопичував і критично вичитував десятки разів. Часто, я не погоджувався з собою, перечитуючи уривок після того, як він полежить деякий час. Тому я доводив текст до того, поки сам не погоджувався з написаним після тривалих періодів витримки матеріалу. Своїм завданням я прийняв писати якомога коротше і неодноразово переписував великі фрагменти, знаходячи, що їх можна виразити коротше. Це дозволило брати участь у формуванні книги всім охочим із спільноти Метархія, швидко знаходити помилки та неточності завдяки читачам, а також багатьом просто зручніше сприймати у вигляді книги. Актуальну версію завжди можна знайти на https:// github.com/ HowProgrammingWorks/ Book, вона доповнюватиметься постійно. Прошу надсилати запити на виправлення та доповнення до [issues] (https:// github.com/ HowProgrammingWorks/ Book/ issues) англійською мовою, нові ідеї у [discussions] (https:// github.com/ HowProgrammingWorks/ Book/ discussions) будь- якою мовою, а свої доповнення та виправлення оформляти у вигляді pull-request у репозиторій книги.

Програмування – це мистецтво та інженерія розв'язання задач за допомогою обчислювальної техніки. Інженерія, тому що воно покликане отримувати користь із знань, а мистецтво, тому що знаннями програмування на сучасному етапі розвитку, на жаль, не обмежується і змушене вдаватися до інтуїції та слабко осмисленого особистого досвіду. Завдання програміста не у знаходженні математично вірного рішення, а у відшуканні узагальненого механізму рішення, здатного нас приводити до знаходження прийнятного рішення за обмежений час у якомога більшому класі завдань. Іншими словами, у знаходженні абстрактного класу рішень. Не всі парадигми програмування припускають рішення покрокове, але фізична реалізація обчислювальної техніки та природа людського мислення передбачають покроковість. Складність у цьому, що ці дії далеко не завжди зводяться до машинних операцій і залучають зовнішню взаємодію з пристроями введення/ виводу і датчиками, а через них, із зовнішнім світом і людиною. Ця обставина створює велику невизначеність, яка не дозволяє математично суворо довести правильність способу

вирішення всіх завдань і, тим більше, суворо вивести таке рішення з аксіом, як це характерно для точних наук. Однак, окремі алгоритми можуть бути виведені аналітично, якщо вони зводяться до чистих функцій. Тобто, до функцій, які у будь-який момент для певного набору вхідних даних однозначно дають однаковий результат. Чиста функція не має історії (пам'яті або стану) і не звертається до зовнішніх пристроїв (які можуть мати такий стан), може звертатися тільки до інших чистих функцій. Від математики програмування успадкувало можливість шукати точні рішення аналітично, та й сама обчислювальна машина функціонує строго в рамках формального математичного апарату. Але процес написання програмного коду не завжди може бути зведений до формальних процедур, ми змушені приймати рішення в умовах великої невизначеності та конструювати програми інженерно. Програміст обмежений і часом розробки програми, тому ми зменшуємо невизначеність завдяки введенню конструктивних обмежень, які не є суворо виведеними із завдання і засновані на інтуїції та досвіді конкретного спеціаліста. Простіше кажучи, через відсутність оптимального алгоритму, програміст може вирішувати завдання будь-яким способом, який дає прийнятні результати за розумний час, і який може бути реалізований за такий час, поки завдання ще залишається актуальним. У таких умовах ми повинні брати до уваги не тільки міру наближення рішення до оптимального, а й знання програміста, володіння інструментарієм та інші наявні ресурси. Адже навіть доступ до знань вже готових програмних рішень обмежений авторським правом, правами володіння вихідним кодом та документацією, що відповідають ліцензійними обмеженнями, не тільки на програмні продукти, а й на книги, відео, статті, навчальні матеріали тощо. Все це суттєво ускладнює та уповільнює розвиток галузі, але згодом доступність знань незворотно зростає, вони просочуються у вільне ходіння в мережі через популяризаторів, ентузіастів та рух вільного програмного забезпечення.

1.1. Підходи до вивчення програмування

Багато хто думає, що головне уміння програміста — це писати код. Насправді програмісти частіше читають код і виправляють його. А основні критерії якості, коду - зрозумілість, читабельність та простота.

Головне уміння програміста - це читання та виправлення

коду.

Кожна тема містить приклади хорошого коду та поганого коду. Ці приклади зібрані з практики програмування та ревью проектів. Спеціально заготовлені приклади поганого коду будуть працездатні, але сповнені антипаттернів та проблем, які потрібно виявити та виправити. Навіть сама перша практична робота в курсі буде пов'язана з виправленням коду, підвищення його читабельності. Якщо давати традиційні завдання (написати функцію по сигнатурі, алгоритм, клас), то початківець, очевидно, реалізує його не найкращим чином, але захищатиме свій код, бо це перше, що він написав. А якщо завдання буде "взяти приклад чужого поганого коду, знайти проблеми та виправити", не переписати з нуля, а покращити за кілька кроків, фіксуючи та усвідомлюючи ці кроки, то включається критичний підхід.

Виправлення поганого коду - один із найефективніших способів навчання.

Початківець одержує приклади рев'ю коду і за аналогією прагне виправити своє завдання. Такі ітерації повторюються неодноразово, не втрачаючи критичного настрою. Дуже добре, якщо буде наставник, який спостерігає за покращеннями, і може коригувати та підказувати. Але наставник у жодному разі не повинен виконувати роботу за новачка, а скоріше наштовхувати його на те, як треба думати про програмування і де шукати рішення.

Наставник – незамінний на будь-якому етапі професійного зростання.

Далі будуть завдання по написанню свого коду. Дуже рекомендуємо початківцям обмінюватися між собою цими рішеннями для перехресного ревью. Звичайно, перед цим потрібно застосувати лінтери та форматери коду, які проаналізують синтаксис, знаходячи в ньому помилки, і виявлять проблемні місця за великою кількістю шаблонів коду. Потрібно домогтися того, щоб колега розумів виражену тобою думку, а не витрачав час на синтаксис і форматування.

Застосовуйте дружнє ревью коду, перехресне ревью, лінтери та форматери.

Переходимо до вправ на зниження зв'язуваності між кількома абстракціями, потім між модулями, тобто зробити так, щоб потрібно було якнайменше знати про структури даних однієї частини програми з іншої її частини. Зниження мовного фанатизму досягається паралельним вивченням з самого початку кількох мов програмування та перекладами з однієї мови на іншу. Між **JavaScript** та **Python** перекладати дуже просто, а **C** буде складніше, але ці три мови, які б вони не були, не можна не включити в курс.

З перших кроків не допускайте жодного фанатизму: мовного, фреймворкового, парадигменного.

Зниження фреймворкового фанатизму — заборона для початківців використовувати бібліотеки та фреймворки, і зосередитись на максимально нативному коді без залежностей. Зниження парадигмального фанатизму - намагатися комбінувати процедурне, функціональне, ООП, реактивне та автоматне програмування. Ми спробуємо показати, як ці комбінації дозволяють спростити патерни та принципи з GoF та SOLID.

Наступна важлива частина курсу – вивчення антипаттернів та рефакторингу. Спочатку ми дамо огляд, а потім практикуватимемося на реальних прикладах коду з живих проектів.

1.2. Приклади на мовах JavaScript, Python та C

Приклади коду ми писатимемо різними мовами, але перевага віддаватиметься не найкращим, красивим і швидким, а тим, без яких не можна обійтися. Ми візьмемо **JavaScript** , як найпоширеніший, **Python**, тому що є області, де без нього не можна і **C**, як мова досить близька до асемблера, все ще актуальна та має найбільший вплив на сучасні мови по синтаксису та по закладеним у нього ідеям. Всі три дуже далекі від мови моєї мрії, але це те, що ми маємо. На перший погляд **Python** дуже відрізняється від **JavaScript** та інших C-подібних мов, хоча це тільки на перший погляд, ми покажемо, що він дуже схожий на **JavaScript** через те, що система типів, структури даних, а особливо вбудовані колекції в них

дуже схожі. Хоча синтаксично, відмінність в організації блоків коду за допомогою відступів і фігурних дужок **{}** впадає у вічі, але насправді, така відмінність не дуже вже важлива, а між **JavaScript** і **Python** набагато більше спільного, ніж у обох з мовою **C**.

Почнемо ми не з вивчення синтаксису, а одразу з читання поганого коду та пошуку у ньому помилок. Давайте подивимося наступні фрагменти, перший буде на JavaScript:

```
let first_num = 2;
let secord_num = 3;
let sum = firstNum + secondNum;
console.log({ sum });
```

Спробуйте зрозуміти, що тут написано і в чому можуть бути помилки. А потім порівняйте цей код із його перекладом на **C**.

```
#include <stdio.h>

int main() {
  int first_num = 2;
  int secord_num = 3;
  int sum = firstNum + secondNum;
  printf("%d\n", sum);
}
```

Помилки тут ті самі, їх легко може виявити людина,коли розглядатиме код, яка навіть не знає основ програмування. А наступний фрагмент коду буде на Python, він робить абсолютно те саме і містить однакові помилки.

```
first_num = 2;
secord_num = 3;
sum = firstNum + secondNum;
print({ 'sum': sum  });
```

Далі ми часто порівнюватимемо приклади коду різними мовами, шукатимемо і виправлятимемо помилки, оптимізуватимемо код, покращуючи насамперед його читальність і зрозумілість.

1.3. Моделювання: абстракції та повторне використання

В основі будь- якого програмування лежить моделювання, тобто створення моделі розв'язання задачі або моделі об'єктів і процесів у пам'яті машини. Мови програмування надають синтаксиси для конструювання обмежень під час створення моделей. Будь- яка конструкція та структура, покликана розширити функціональність та введена в модель, призводить до додаткових обмежень. Підвищення рівня абстракції, навпаки, може знімати частину обмежень і зменшувати складність моделі і коду програми, що виражає цю модель. Ми постійно балансуємо між розширенням функцій і згортанням їх у більш узагальнену модель. Цей процес може та повинен бути багаторазово ітеративним.

Дивно, але людина здатна успішно вирішувати завдання, складність яких перевищує можливості її пам'яті та мислення, за допомогою побудови моделей та абстракцій. Точність цих моделей визначає їх користь для прийняття рішень та вироблення керуючих впливів. Модель завжди не точна і відображає лише малу частину реальності, одну чи кілька її сторін чи аспектів. Однак, в обмежених умовах використання, модель може бути невідмінною від реального об'єкта предметної області. Є фізичні, математичні, імітаційні та інші моделі, але нас цікавитимуть насамперед інформаційні та алгоритмічні моделі.

Абстракція - це спосіб узагальнення, що зводить безліч різних, але схожих між собою випадків до однієї моделі. Нас цікавлять абстракції даних та абстрактні алгоритми. Найпростіші приклади абстракції в алгоритмах - це цикли (ітераційне узагальнення) та функції (процедури та підпрограми). За допомогою циклу ми можемо описати багато ітерацій одним блоком команд, припускаючи його повторюваність кілька разів, з різними значеннями змінних. Функції також повторюються багато разів з різними аргументами. Приклади абстракції даних - це масиви, асоціативні масиви, списки, множини, тощо. У застосунках абстракції потрібно поєднувати в рівні – шари абстракцій. Низькорівневі абстракції вбудовані у мову програмування (змінні, функції, масиви, події). Абстракції більш високого рівня містяться у програмних платформах, рантаймах, стандартних бібліотеках та зовнішніх бібліотеках або їх можна побудувати самостійно із простих абстракцій. Абстракції так називаються тому, що вирішують

абстрактні узагальнені завдання загального призначення, не пов'язані з предметною областю.

Побудова шарів абстракцій - це чи не найважливіша задача програмування від успішного вирішення якої залежать такі характеристики програмного рішення, як гнучкість налаштування, простота модифікації, здатність до інтеграції з іншими системами та період життя рішення. Всі шари, які не прив'язані до предметної області та конкретних прикладних завдань, ми називатимемо системними. Над системними шарами програміст надбудовує прикладні шари, абстракція яких навпаки знижується, універсальність зменшується та конкретизується застосування, прив'язуючись до конкретних завдань.

Абстракції різних рівнів можуть бути як в одному адресному просторі (одному процесі або одному застосунку), так і в різних. Відокремити їх один від одного і здійснити взаємодію між ними можна за допомогою програмних інтерфейсів, модульності, компонентного підходу і просто зусиллям волі, уникаючи прямих викликів із середини одного програмного компонента в середину іншого, якщо мова програмування або платформа не дбають про запобігання такій можливості. Так слід чинити навіть усередині одного процесу, де можна було б звертатися до будь-яких функцій, компонентів та модулів з будь-яких інших, навіть якщо вони логічно відносяться до різних шарів. Причина цього в необхідності знизити зв'язність шарів та програмних компонентів, забезпечивши їх взаємозамінністю, повторним використанням та роблячи можливою їх роздільну розробку. Одночасно потрібно підвищувати зв'язність усередині шарів, компонентів та модулів, що забезпечує зростання продуктивності коду, простоту його читання, розуміння та модифікації. Якщо ж нам вдасться уникнути зв'язаності між різними рівнями абстракцій, і за допомогою декомпозиції домогтися того, щоб один модуль завжди міг бути повністю охоплений увагою одного інженера, то процес розробки стане масштабованим, керованим і більш передбачуваним. Подібна ідея покладена в основу архітектури мікросервісів, але більш загальний принцип застосовується для будь-яких систем, і не важливо, чи це будуть незалежно запущені мікросервіси або модулі, запущені в одному процесі.

Слід зазначити, що чим краще система розподілена, тим краще вона централізована. Тому, як вирішення завдань у таких системах знаходяться на адекватних рівнях, де вже достатньо інформації для

прийняття рішень, обробки та отримання результату, відсутня жорстка зв'язаність моделей різного рівня абстракції. При такому підході немає зайвих ескалацій завдання на верхні рівні, уникаються “перегріви” вузлів прийняття рішень, мінімізована передача даних і підвищена оперативна швидкодія.

1.4. Алгоритм, программа, синтаксис, мова

Є багато термінів, пов'язаних із програмуванням. Для визначеності нам слід з'ясувати різницю між ними. Краще зосередитися на неформальному розумінні, ніж формальному визначенні. Найстарішим поняттям тут є алгоритм, який ми пам'ятаємо зі шкільного курсу математики. Наприклад, алгоритм Евкліда знаходження найбільшого спільного дільника двох цілих чисел.

Алгоритм (Algorithm)

Алгоритм - це ще не програма, а ідея вирішення задач, описана формально. Так, щоб вона була зрозуміла іншим, вона перевіряється та реалізується. Алгоритм не можна запустити, його можна перетворити на код якоюсь мовою програмування. Алгоритм містить опис операцій і може бути записаний різними способами: формулою, блок-схемою, списком дій, людською мовою. Він завжди обмежений конкретним класом завдань, що він вирішує за кінцевий час. Часто ми можемо спростити та оптимізувати алгоритм, звузивши клас задач. Наприклад, переходячи від обчислення суми цілих і дробових чисел до обчислення суми лише цілих, ми можемо зробити більш ефективну реалізацію. Можна і розширити клас задач для цього прикладу, дозволивши подавати на вхід ще й рядкові подання чисел. Це зробить алгоритм більш універсальним, але менш ефективним. Ми маємо вибирати, що саме ми оптимізуємо. В даному випадку краще розділити алгоритм на два, один перетворюватиме всі числа до одного типу даних, а другий складатиме.

Приклад реалізації алгоритму НСД (GCD)

На JavaScript алгоритм Евкліда, для знаходження НСД (загальної міри чи найбільшого спільного дільника) можна написати так:

```
const gcd = (a, b) => {
  if (b === 0) return a;
  return gcd(b, a % b);
};
```

Або навіть коротше, але стане не менш зрозумілим, якщо ви порівняєте два ці варіанти і простежте які конструкції замінені:

```
const gcd = (a, b) => (b === 0 ? a : gcd(b, a % b));
```

Цей простий алгоритм є рекурсивним, тобто звертається до себе для обчислення наступного кроку і передбачає вихід, коли b доходить до 0. Для алгоритмів ми можемо визначати обчислювальну складність, класифікувати їх за ресурсами процесорного часу та пам'яті, необхідним для вирішення задачі.

Програма (Program)

У попередньому прикладі ми мали справу з функцією, це частина програми, але щоб вона запрацювала, її потрібно викликати та передати їй дані. Програмний код та дані, об'єднані в одне ціле – це і є програма. Ніклаус Вірт, автор багатьох мов, у тому числі і Pascal, має книгу «Алгоритми + Структури даних = Програми». Її назва схопила дуже важливу істину, яка глибоко відобразилася не тільки у світогляді читачів, а й у назвах курсів у провідних ВНЗ, і навіть на співбесідах, коли випробуваного просять сконцентруватися саме на цих двох речах. За перші 50 років існування індустрії програмного забезпечення виявилося, що структури даних не менш важливі, ніж алгоритми. Більше того, багато відомих програмістів роблять на них основну ставку, наприклад, відома цитата Лінуса Торвальдса: «Погані програмісти турбуються про код. Хороші програмісти турбуються про структури даних та зв'язки між ними». Справа в тому, що вибір структур даних багато в чому визначає те, яким буде алгоритм, обмежує його в рамках обчислювальної складності та семантики завдання, яке програміст розуміє через дані, розкладені в пам'яті, набагато краще, ніж через послідовність операцій.

Ерік Реймонд висловив це так: «Розумні структури даних і тупий код працюють набагато краще, ніж навпаки».

Код дозволяє порозумітися

Однак той самий Лінус Торвальдс сказав нам «Балаканина нічого не варта. Покажіть мені код». Це зовсім не суперечить сказаному вище. Я думаю, що він мав на увазі те, що програмний код не допускає двозначності. Це універсальна мова, яка дозволяє програмістам знаходити спільну мову навіть тоді, коли природні мови через свою багатозначність не дозволяють точно зрозуміти один одного, в такому випадку це можна зробити просто поглянувши на код.

Інженерія (Engineering)

Отримання практичної користі з наявних ресурсів за допомогою науки, техніки, різних методик, організаційної структури, прийомів та знань – це наступний рівень — інженерія.

Я пам'ятаю, що в перші роки вивчення програмування для мене вже було важливо, щоб код використовувався людьми, покращував їхнє життя і жив довго. Олімпіадні завдання здавались мені нецікавими, навчальні завдання надто надуманими, хотілося сконцентруватися на тому, що люди запускатимуть на своїх комп'ютерах щодня: програми баз даних, форми та таблиці, мережеві та комунікаційні програми, програми, що керують апаратурою, працюють з датчиками, та безліч інструментів для програмістів.

Так само, як і в інших інженерних галузях, у програмуванні дуже важлива користь для людини, а не правильність або стрункість концепції. Інженерія покликана використовувати наукові досягнення, а в тих місцях, де наукових знань, що є на сьогодні, недостатньо, інженерія застосовує інтуїцію, інженерну культуру, метод спроб і помилок, застосування неусвідомленого досвіду та досвіду, що має недостатнє наукове осмислення.

У цьому й перевага інженерії та її недолік. Ми маємо безліч різних та суперечливих рішень одного завдання, ми не завжди знаємо чому щось не працює, але це ще добре, ми іноді дивуємось, чому щось працює. Такий підхід призводить до накопичення поганих практик у проектах і такого сплетіння хороших і поганих практик, що розділити їх дуже складно, тому часто зусилля витрачаються повторно для вирішення завдання. Ніклаус Вірт сказав «Програми стають повільнішими швидше, ніж "залізо" стає швидше» і ми часто

стикаємося з тим, що написати програму наново простіше, ніж виправляти в ній помилки.

Інженерія програмного забезпечення (Software engineering)

Залучення інженерії до індустрії програмного забезпечення включає архітектуру, дослідження, розробку, тестування, розгортання та підтримку програмного забезпечення.

Індустрія програмного забезпечення перетворилася на потужну галузь промисловості, обросла допоміжними технологічними практиками, які дозволяють зменшити вплив її недоліків, вже наведених вище та зробити кінцевий продукт достатньо надійним, щоб він приносив прибуток, але недостатньо якісним, щоб можна було випускати нові і нові його версії .

«Більшість програм на сьогоднішній день подібні до єгипетських пірамід з мільйона цеглинок одна на одній і без конструктивної цілісності — вони просто побудовані грубою силою і тисячами рабів» // Алан Кей

Програмування (Programming)

Отже, програмування - це мистецтво та інженерія вирішення завдань за допомогою обчислювальної техніки. Тут важливо відзначити, що обчислювальна техніка дуже сильно впливає на те, як ми програмуємо, диктує те, які парадигми та підходи працюватимуть ефективніше і даватимуть результат, доступний нам за ресурсами, витраченими на програмування та обчислювальними ресурсами, необхідними для виконання створених програм.

Кодування (Coding)

Якщо виділити з програмування лише написання вихідного коду програми за допомогою певного синтаксису (мови), стилю та парадигми за готовим ТЗ (технічним завданням), то ми називаємо це кодуванням, хоч слово можна вважати застарілим.

Розробка може бути поділена на проектування та кодування, і це дає більш ефективне застосування сил на довгій дистанції, але

часто доводиться починати програмувати без ТЗ і без попереднього проектування. Розроблені таким чином системи називають прототипами, MVP (minimum viable product), пілотними системами або стендами. Їх користь полягає у перевірці гіпотез про корисність для споживача або економічну ефективність їх використання.

Програміст не завжди усвідомлює, що він робить прототип або продукт, іноді ми отримуємо прототип, зроблений так добротно, наче готовий продукт, або готовий продукт, зроблений як тимчасове рішення. Проте є ентузіасти, які люблять свою роботу, і саме на них тримається ця галузь, суперечлива та повна проблем.

«Більшість хороших програмістів роблять свою роботу не тому, що чекають оплати або визнання, а тому, що отримують задоволення від програмування» // Лінус Торвальдс

Розробник vs програміст

Є прихильники кожної з назв, часто самоназва розробник або програміст пов'язана з особливою професійною гордістю і навіть пихатим ставленням до прихильників іншої назви. Було б добре розділити ці професії приблизно так само, як розділилися професії водія та автомеханіка. Звичайно, автомеханік може говорити, що водії нічого не тямлять у автомобілях, але масово людей возять саме водії. Так само і в IT, програміст повинен концентруватися на абстракціях і створенні програмних компонентів, а розробник - на застосуванні готових компонентів до завдання, що вимагає зовсім інших знань і навичок, окрім програмування.

Різницю між програмістом і розробником можна показати на прикладі створення інформаційних систем (ІС). У світі є потреба масового виробництва ІС, забезпечення потреб промисловості, транспорту, сфери обслуговування, логістики, торгівлі, медицини тощо. Але зараз інформаційні системи (ІС) – це дороге задоволення і для їхньої розробки потрібні висококваліфіковані кадри. ІС, як клас систем, передбачає бази даних, інтерфейси користувача і бізнес-логіку. Майже завжди ІС потребує інтеграції з іншими ІС. Таким чином, для розробки ІС потрібні знання з СУБД (SQL або noSQL), фронтенду (форми, UI/UX), бекенду (сервер додатків) та API. Тому ІС мають велику вартість володіння, а експлуатація пов'язана з високими ризиками. Адже ІС створюють універсальні інженери-програмісти, які пишуть для кожної ІС багато системного коду з

нуля. Якщо розділити прикладне та системне програмування на дві різні спеціальності та використовувати високорівневу платформу, яку зробили системні програмісти, то ми зможемо перевикористовувати до 80% коду в різних системах. Прикладні програмісти (тобто розробники) зможуть тоді зосередитися лише на завданнях, пов'язаних із специфікою предметної галузі. Це істотно знижує вимоги до прикладних програмістів, а використання принципів вільного програмного забезпечення дозволяє об'єднати зусилля зі створення платформи та виключити ризики, пов'язані з володінням платформою. Open source ліцензії не дають вендорам довільно змінювати свою політику стосовно споживачів та системних інтеграторів, тому що вони не мають блокуючого контролю над платформою і можуть бути замінені конкурентами.

Такий підхід вже застосовувався неодноразово і дозволив зробити доступнішим бухгалтерське та офісне ПЗ, розробку веб-сайтів, навіть комп'ютерні ігри зараз не пишуть з нуля, а використовують платформи, на яких навіть початківці можуть швидко показати вражаючі результати.

Щоб запровадити такий підхід для ІС, нам потрібні нові професії, нова система підготовки кадрів, новий підхід до постановки завдань і навіть замовник таких ІС має думати про них інакше. Існуючий попит має суттєво змінитись, вирости завдяки тому, що тепер спеціалізовані ІС будуть доступні набагато ширшому колу споживачів і перестануть бути розкішшю.

Складність та простота

Давайте ж прагнути до того, щоб наші програми були простими і для споживача і для нас самих, як людей, які їх багато разів модифікуватимуть і постійно стикатимуться з тими рішеннями, які ми заклали в них під час первинної розробки. А якщо ми обмежені в часі і змушені писати неефективний або малозрозумілий код, то слід планувати його переробку, рефакторинг та оптимізацію до того, як ми забудемо його структуру і у нас вивітряться всі ідеї щодо покращення. Накопичення проблем у коді називається "технічний борг" і він призводить не тільки до того, що програми стають менш гнучкими та зрозумілими, а й до того, що наші молодші колеги, підключаючись до проектів, читають та вбирають не найкращі практики та переймають наш оверинжиніринг. Простота вирішення складних завдань є метою хорошого програміста, приховування

складності за програмними абстракціями — це метод досвідченого інженера.

«Я завжди мріяв про те, щоб моїм комп'ютером можна було користуватися так само легко, як телефоном; моя мрія збулася: я вже не можу розібратися, як користуватись моїм телефоном» // Бйорн Страуструп

1.5. Декомпозиція та поділ відповідальності

No translation

1.6. Огляд спеціальності інженер-програміст

Навколо програмування, як і навколо будь- якої сфери своєї діяльності, людина встигла побудувати величезну кількість забобонів і омани. Найперше джерело проблеми, це термінологія, адже різні парадигми, мови та екосистеми насаджують свою термінологію, яка не тільки суперечить між собою, але і нелогічна навіть всередині окремої спільноти. Більше того, багато програмістів самоучки і одинаки, видумують самобутню, ні на що не схожу термінологію і концепції, дублюючи один одного. Ошаленілі маркетологи теж деструктивно впливають на IT галузь у цілому, на формування світогляду і термінології. Викручуючи очевидні речі, заплутуючи та ускладнюючи концепції, вони забезпечують невичерпну лавину проблем, на якій тільки і підтримується весь софтверний бізнес. Переманюючи користувачі і програмістів на свої технології, гіганти промисловості часто створюють дуже привабливі і правдоподібні концепції, що ведуть в підсумку до несумісності, війни стандартів і до явної залежності від постачальника програмної платформи. Угруповання, що захоплюють увагу людей, десятиліттями паразитують на їх увазі та бюджетах. Ведені гординею та марнославством, деякі розробники й самі розповсюджують сумнівні, а іноді й свідомо тупикові ідеї. Адже робити програмне забезпечення добре – зовсім не вигідно для виробника. Ситуація значно краща у сфері вільного ПЗ та відкритого коду, але децентралізовані ентузіасти надто роз'єднані, щоб ефективно протидіяти потужній пропаганді гігантів галузі.

Як тільки якась технологія або екосистема розвивається достатньо, щоб на ній можна було створювати хороші рішення, вона обов'язково застаріває або виробник припиняє її підтримку або стає

занадто складною. На моїй пам'яті вже змінилося понад п'ять таких технологічних екосистем.

1.7. Огляд парадигм програмування

Математика розглядає програму, як функцію, яку можна декомпозувати (розділити) на більш прості функції так, щоб програма- функція була їх суперпозицією. Тобто, грубо кажучи, програма є складною формулою, перетворювачем даних, коли на вхід подаються умови завдання, а на виході ми отримуємо рішення. Не всякий програміст знайомий із цією точкою зору, хоч вона й не ідеальна, але корисна для переосмислення своєї діяльності. Більш поширена протилежна точка зору, яку легко одержати з практики програмування. Полягає вона в написанні програм виходячи з уявлення користувача, з малюнків екранів інтерфейсу користувача, і з інструментарію, мови, платформи і бібліотек. В результаті ми отримуємо не програму- функцію, а велику систему станів, в якій відбувається комбінаторний вибух переходів і поведінка якої непередбачувана навіть для автора, не те що для користувача. Але не можна відразу відкидати цей, здавалося б, жахливий підхід. У ньому є конструктивне зерно, і полягає воно в тому, що не всі програми можна за короткий термін реалізувати у функціональній парадигмі, як перетворювачі даних. Тим більше, що людська діяльність вся складається з кроків і зміни станів навколишніх предметів за принципом покрокових маніпуляцій ними, а уявити її у вигляді функцій було б досить неприродним для нашого мислення.

Парадигма задає набір ідей та понять, припущень та обмежень, концепцій, принципів, постулатів, прийомів та технік програмування для вирішення завдань на обчислювальній машині.

У цьому розділі ми розглянемо деякі з них поверхово, а далі в книзі буде спеціальний розділ з більш детальним обговоренням кожної парадигми. Існують мови, які підтримують одну парадигму, а є мультипарадигмові мови. Ми звернемо увагу на різні мови та відмінності реалізації парадигм у них.

Для людини природно уявлення про будь- яку дію, як про набір кроків або алгоритм, це імперативний підхід. Ці кроки можуть бути або лінійними або прийняттям рішення про перехід до іншого кроку плану, замість того, щоб виконувати дії послідовно. Прийняття рішення для машини це операція, порівняння, що веде до розгалуження алгоритму, що дає варіанти (зазвичай два). Дії можна

умовно поділити на внутрішні та зовнішні. У внутрішніх беруть участь лише процесор та пам'ять, дія виконується відразу, без очікування, і має певний результат, який доступний безпосередньо на наступному етапі алгоритму. Зовнішні дії - це звернення до зовнішніх пристроїв вводу-виводу (мережа, диски, інші пристрої), і вони вимагають очікування реакції від пристрою, яка прийде за, зазвичай, невідомий заздалегідь, час. Ми надсилаємо керуючий сигнал периферійним пристроям про те, що їм потрібно щось зробити і передаємо їм потрібні для цього дані. Далі у нас знову є два варіанти, або чекати результату, і це буде називатися блокуючим режимом вводу-виводу або перейти до наступного кроку алгоритму, не чекаючи результату, і це буде неблокуючий ввід-вивід. Такий поділ спричинено значною різницею у тривалості внутрішніх та зовнішніх дій. Більшість зовнішніх дій пов'язані з фізичними операціями над зовнішнім середовищем. Наприклад, передача даних через бездротову або провідну мережу, запис або читання з фізичного носія, взаємодія з датчиком, реле або приводом. Такі операції часто мають не цифрову, а аналогову природу, тому потрібно додаткове перетворення даних, очікування перехідного процесу, очікування потрібного показання датчика або сигналу від пристрою і т.д. Пристрої вводу-виводу часто мають свій контролер, в якому виконується окремий потік операцій, а взаємодія між центральним процесором та пристроями вводу- виводу також вимагає узгодження, що займає час і може закінчитися невдало.

Парадигма пропонує узагальнену модель розв'язання задач, певний стиль, шаблони, приклади хороших та поганих рішень, які застосовуються для написання програмного коду.

2. Базові концепти

No translation

```
// Single-line comment
```

```
/*
  Multi-line
  comments
*/
```

```
# Single-line comment
```

```
"""
  Multi-line
  comments
"""
```

```
select name from PERSON -- comments in sql
```

```
<!-- commented block in xml and html -->
```

```
; Single-line comment in Assembler and LISP
```

2.1. Значення, ідентифікатор, змінна та константа, літерал, присвоєння

Найважливіше у програмуванні — це давати зрозумілі імена ідентифікаторам (змінним, константам, параметрам, функціям, класам і т.д.) і розташовувати їх в програмі так, щоб вони були видні у потрібних місцях. При цьому моємо намагатися скоротити їх область видимості. Ідентифікатори можуть бути глобальними, оголошеними у файлі або імпортованими з інших модулів програми.

Оголошення та присвоєння (assignment)

```
let migrationYear = 622;
```

Цей запис на JavaScript і тут **migrationYear** — ім'я ідентифікатора, **=** це оператор присвоєння, **622** — значення, а **let** означає, що значення може бути переприсвоєно, тобто, змінено.

На C і C++ аналогічний код виглядатиме так:

```
int migrationYear = 622; // тут тип int представляет целое
число
```

У Python, як і JavaScript, оператор присвоєння не вимагає вказувати тип, він визначається з типу літералу. А при присвоєнні ми не можемо заборонити переприсвоєння ідентифікатору іншого значення.

```
migrationYear = 622
```

Операція присвоєння у багатьох мовах повертає значення (має поведінку виразу). Таким чином, можна написати:

```
let year;
const migration = (year = 622); // теперь оба идентификатора
console.log({ migration, year }); // имеют значение 622
```

Але такий синтаксис погано читається і може ввести в оману, як і завдання кількох ідентифікаторів через кому:

```
let year = 622, migration = 'Hijrah', i, counter = 0;
```

Економія в коді це добре, але не заощаджуйте літери та рядки за рахунок читабельності коду. Натомість краще написати:

```
let year = 622;
let migration = 'Hijrah';
let i;
let counter = 0;
```

А ще краще відразу вирішити, які ідентифікатори не змінюватимуть свого значення і задати їх через **const**:

```
const year = 622;
const migration = 'Hijrah';
let i;
let counter = 0;
```

Змінні та константи

У повсякденному спілкуванні навіть професіонали часто плутають

поняття, називаючи константу змінною, адже слово ідентифікатор занадто довге, а слово константа занадто багатозначне. Це трапляється з локальними константами, які ми називаємо у стилі **lowerCamel** , наприклад, усередині функцій. Їх значення може бути різним за різних викликів функції, а є глобальні значення, яких однакове у будь-якому місці програми або файлу, і ми називаємо їх у стилі **UPPER_SNAKE**.

```
const WATCH_TIMEOUT = 5000; // Глобальная константа
```

```
const cityName = 'Beijing'; // Локальная константа
```

```
let distance = 0; // Переменная
```

Ми можемо змінювати значення змінної на відміну від константи, а для деяких мов, як JavaScript і Python, можемо змінювати і тип, але це сильно погіршуватиме читаність коду, наприклад:

```
let migration = 'Hijrah'; // используем переменную первый раз
console.log(migration); // вывод строки
migration = 622; // используем повторно для других целей
console.log(migration); // вывод числа
```

Літерал (Literal) — запис значення коду програми

Ідентифікатори зв'язуються зі своїми значеннями в пам'яті під час роботи програми, але під час написання коду ми іноді хочемо задавати значення синтаксично (тобто записувати їх символами в правилах конкретної мови програмування). Наприклад: літерал числа та рядки, літерал об'єкта та масиву, навіть літерал функції та класу (але їх ми побачимо в наступних розділах). У різних мовах різні правила запису літералів, але вони частково збігаються і зазвичай їх легко прочитати. Літерали можуть задавати не тільки значення, а тип та систему числення за допомогою префіксів та суфіксів. Давайте подивимося приклади:

- Рядок **it's**: в одинарних лапках '**it\'s**', тут символ бекслеш **** дозволяє використовувати лапку всередині рядка; у подвійних

"it's", тут \ не потрібен, але може використовуватися для подвійної лапки всередині рядка та інших символів;
- Ціле число 255 можна записати, як 0xFF у шістнадцятковій системі; 0o377 у вісімковій; а ось запис 0377 це стара форма, вона не підтримується в Python починаючи з 3 версії, а у JavaScript працює, але погано читається; 0b11111111 у бінарному записі; літерал 255n із суфіксом n у JavaScript створює значення типу BigInt; а 255u; 255l, 255ul у C++ та низці інших мов, де суфікс u (unsigned) і задає беззнаковий тип, а суфікс l (long) задає довге число;
- Дробове число 1.618 записується: 1.618; в C++ і Java 1.618f число з плаваючою точкою; і 1.618d число з подвійною точністю, а 1618e-3 в експоненційній формі;
- Число -12000, крім звичайного запису, може бути записано: -12000n; -1.2e4; -12e3; -12e3f; -1.2e4d; тут символи n, f та d позначають тип, а e експоненційну форму; у різних мовах підтримуються різні форми запису;
- Булеві літерали: true та false;
- Літерал масиву: [1, 2, 3] у JavaScript та Python; в C і C++, Java, C# { 1, 2, 3, 4 };
- Літерал словника: { 'uno': 1, 'due': 2, 'tre': 3 } у Python; і літерал об'єкту { uno: 1, due: 2, tre: 3 } у JavaScript;

Ця книга не є посібником з якоїсь мови, а особливості конкретної мови можна дізнатися з її специфікації, підручника або прикладів коду, яких тут достатньо, щоб освоїти запис усіх необхідних літералів JavaScript і Python.

Приклад програми

Тут багато конструкцій знайомі, але є й нові (функція, цикл, умова), вони будуть докладніше розглянуті в наступних розділах, хоча їхнє просте використання не складно прочитати і зрозуміти вже зараз.

```
'use strict';

const MAX_PURCHASE = 2000; // Глобальна константа

// Функція обчислення суми покупок із масиву цін
// prices - параметр (масив цін)
const calculateTotal = (prices) => {
```

```
  // Задаємо змінну з початковим значенням
  let amount = 0; // Перебираємо елементи масиву у циклі
  for (const price of prices) {
    // локальна константа price не змінюється у циклі
    // на кожному проході додаємо до суми
    amount += price;
  }
  return amount; // Повертаємо суму з функції
};

// Задаємо масив цін товарів
const purchase = [1500, 100, 10, 50];
// Викликаємо функцію обчислення суми та передаємо їй аргумент
// З функції значення аргументу буде доступним через параметр
const total = calculateTotal(purchase);
// Результат записуємо в локальну константу
// Якщо сума не перевищила максимальної, то виводимо її
if (total <= MAX_PURCHASE) {
  console.log({ total }); // Виводить: {total: 1660}
}
```

2.2. Типи даних, скалярні, посилання та структурні типи

Тип

Тип - множина значень та операцій, які можуть бути виконані над цими значеннями. Наприклад, в **JavaScript** тип **Boolean** передбачає два значення **true** , **false** та логічні операції над ними. Тип **Null** передбачає одне значення **null**, а тип **Number** безліч раціональних чисел з додатковими обмеженнями на мінімальне та максимальне значення, а також обмеження на точність та математичні операції **+ - * ** / % ++ -- > < >= <= & | ~ ^ << >>**.

Типи даних (Data Types)

```
const values = [5, 'Kiev', true, { size: 10 }, (a) => ++a];
```

```
const types = values.map((x) => typeof x);
console.log({ types });
```

Посилання (Reference)

Значення типу посилання вказує на об'єкт. Для **JavaScript** це все похідні типу **Object**, якими є всі об'єкти, функції, масиви, типізовані масиви та об'єкти інших вбудованих класів (або вбудованих прототипів).

Структурні типи (Composed types)

Композитні типи чи структури складаються з кількох скалярних значень (для JavaScript усі вони є об'єктами). Скалярні значення поєднуються в одне таким чином, щоб над цим об'єднаним значенням можна було виконувати набір операцій. Наприклад: об'єкт, масив, множина, кортеж.

Перерахований тип (Enumerated type)

Прапорець (Flag)

Прапорець це значення (часто бульового або перелічуваного типу), що визначає стан чогось, іншої програмної абстракції, екземпляра, процесу, пристрою. Наприклад, ознака закриття з'єднання, ознака завершення пошуку структури даних і т.д. Наприклад:

```
let flagName = false;
```

Рядок (String)

У більшості мов програмування до кожного символу в рядку можна звернутися через синтаксис доступу до елементу масиву, наприклад, квадратні дужки **[]**. Але в деяких мовах окремі символи можна також перезаписувати, змінювати рядок. Є мови програмування, де рядки незмінні, як JavaScript та Python.

2.3. Оператор та вираз, блок коду, функція, цикл, умова

Інструкція (Instruction) - один крок алгоритму обчислень, наприклад, інструкція процесора виконується CPU.

Оператор (Statement) - найменша синтаксична частина мови програмування, що виконується інтерпретатором, середовищем або компілюється в машинний код.

Команда (Command) – атомарне завдання для командного процесора.

Вираз (Expression) — синтаксична конструкція мови програмування, призначена для виконання обчислень.

Вираз складається з ідентифікаторів, значень, операторів та виклику функцій. Приклад:

```
(len - 1) * f(x, INTERVAL);
```

Блок коду (Code block) – логічно пов'язана група інструкцій чи операторів.

Блоки створюють область видимості. Блоки можуть бути вкладені. Приклади різних мов: {}, (+ a b), begin end, в Python блоки виділяються відступами.

Цикл (Loop) – багаторазове виконання блоку операторів.

```
const MAX_VALUE = 10;

console.log('Begin');
```

```
for (let i = 0; i < MAX_VALUE; i++) {
  console.dir({ i, date: new Date() });
}
console.log('The end');
```

Умова (Conditional statements) - синтаксична конструкція, що дозволяє виконати різні дії або повертає різні значення (тернарний оператор) залежно від логічного виразу (що повертає true або false).

Рекурсія (Recursion) — завдання алгоритму обчислення функції через виклик її самої (прямий чи непрямий) або визначення функції через неї саму.

Непряма рекурсія - коли функція визначена або викликає себе не безпосередньо, а через інший або ланцюжок функцій.

Хвостова рекурсія — окремий випадок, коли рекурсивний виклик є останньою операцією перед поверненням значення, що завжди може бути перетворено на цикл, навіть автоматичним способом. Не хвостова також може бути перетворена в цикл та оптимізована, але більш складним способом, зазвичай вручну.

2.4. Контекст та лексичне оточення

Область видимості (Scope)

Область видимості - частина коду, з якої "видно" ідентифікатор. Розглянемо приклад:

```
const level = 1;

const f = () => {
  const level = 2;
  {
    const level = 3;
    console.log(level); // 3
```

```
  }
  console.log(level); // 2
};
```

У сучасному стандарті JavaScript область видимості породжується функцією або будь-яким блоком операторів, що мають фігурні дужки {}. У Python scope породжується лише функціями. Порівняйте цей код із попереднім прикладом:

```
level = 1

def f():
  level = 2
  if level == 2:
    level = 3
    print(level) // 3
  print(level) // 3

f()
```

Лексичний контекст (Lexical environment)

Набір ідентифікаторів доступний у локальному блоці або функції. Якщо ідентифікатор не знайдено в лексичному контексті, пошук продовжиться в батьківському контексті, адже контексти мають вкладену структуру. Якщо дійшовши до кореня ідентифікатора не знайдено, буде проведено пошук у глобальному контексті. Для JavaScript лексичні контексти обмежуються блоками '{}' і функціями, а Python — лише функціями.

Лексичний контекст чи лексичне оточення мають вкладеність, тобто крім локальних змінних у блоці, що породжує контекст, є й вищий блок зі своїм контекстом. Якщо ідентифікатор визначений у контексті, то його видно у всіх вкладеннях, якщо тільки не відбувається перекриття імен. Перекриття – це випадок, коли у вкладеному контексті теж оголошено ідентифікатор, що вже є у зовнішньому, тоді значення із зовнішнього контексту стає недоступним і ми маємо доступ лише до значення ідентифікатора з внутрішнього контексту.

Об'єктний контекст

Об'єкт, доступний з методів і функцій через спеціальний ідентифікатор **this** теж використовується як контекст. У більшості мов програмування метод пов'язується з **this** при створенні об'єкта класу. Але JavaScript функції можуть бути прикріплені до **this** за допомогою **bind** або одноразово викликані в контексті об'єкта через **call** і **apply**. Усі функції, крім стрілочних, можуть бути пов'язані з об'єктами.

Глобальний контекст (Global context)

Якщо ідентифікатор не знаходиться у жодному з вкладених лексичних контекстів, то буде виконано його пошук у глобальному об'єкті- довіднику, який є глобальним контекстом (JavaScript **global** або **window**).

2.5. Процедурна парадигма, виклик, стек та куча

Процедура або підпрограма (Procedure, Subroutine) — це логічно пов'язана група інструкцій або операторів, яка має ім'я.

Процедура сприяє повторному використанню коду і може бути викликана з різних частин програми багато разів і з різними аргументами. Процедура не повертає значень, на відміну від функцій, а в деяких мовах (але не в JavaScript) може модифікувати свої аргументи. У багатьох мовах процедура описується синтаксисом функцій (наприклад, типу void).

Функція (Function) – абстракція перетворення значень. Функція однозначно відображає одну множину значень в іншу множину значень.

Функція може бути задана блоком операторів чи виразом. Функція має набір аргументів. Функція може бути викликана через ім'я або вказівник. Функція сприяє повторному використанню коду і може бути викликана з різних частин програми багато разів і з різними

аргументами. У JavaScript функція описується за допомогою **function** або синтаксису стрілок (лямбда-функцій).

Сигнатура функції (Function signature) включає: ім'я (ідентифікатор), кількість аргументів та їх типи (а іноді і імена аргументів), тип результату.

Метод – функція або процедура, пов'язана з об'єктом чи класом.

```
{
  a: 10,
  b: 10,
  sum() {
    return this.a + this.b;
  }
}
```

```
const colorer = (s, color) => `\x1b[3${color}m${s}\x1b[0m`;

const colorize = (name) => {
  let res = '';
  const letters = name.split('');
  let color = 0;
  for (const letter of letters) {
    res += colorer(letter, color++);
    if (color > COLORS.length) color = 0;
  }
  return res;
};

const greetings = (name) =>
  name.includes('Augustus')
    ? `${SALUTATION}, ${colorize(name)}!`
    : `Hello, ${name}!`;
```

Usage

```
const fullName = 'Marcus Aurelius Antoninus Augustus';
console.log(greetings(fullName));

const shortName = 'Marcus Aurelius';
console.log(greetings(shortName));
```

2.6. Функція вищого порядку, чиста функція, побічні ефекти

No translation

2.7. Замикання, функції зворотного виклику, обгортки та події

No translation

2.8. Винятки та обробка помилок

No translation

2.9. Завдання до розділу

Завдання 1. Візьмемо приклад, який ми вже розглядали, але до нього додано кілька помилок. Скопіюйте цей код в окремий файл і виправте його, щоб він не лише працював, але був гарним та зрозумілим. За зразок можна брати код із книги та лекцій.

```
const Items = [
  { CENA: 40 } ,{ CENA : 120 },{
  CENA: '505',
  }, { CENA: 350 }];

For (const ITEM of items){
console.log(`Price: ${item.price}`);
}
```

Завдання 2. Тепер давайте зробимо функцію, яка обчислить суму

всієї покупки. Дайте функції зрозумілу назву і додайте такі правила: потрібно перевіряти, чи є ціна числом (за допомогою **typeof**), підсумовуємо тільки позитивні ціни, а якщо знаходимо не число або негативне число, то помилка видається за допомогою **throw**.

У ході виконання завдання, пошукайте в інтернеті документацію по **for..of**, **throw**, **console.log**, функціям та масивам. Найкраще шукати у MDN (mozilla developers network).

Потрібно, щоб код запускався в командному рядку через node.js або в браузері.

Завдання 3. Візьміть цю структуру даних та доповніть її товарами та групами товарів за прикладом тих, які вже є:

```
const purchase = {
  Electronics: [
    { name: 'Laptop', price: 1500 },
    { name: 'Keyboard', price: 100 },
  ],
  Textile: [{ name: 'Bag', price: 50 }],
};
```

Помістіть код у файл і виведіть всю структуру на екран, запустивши код у node.js або браузері.

Завдання 4. Напишіть функцію **find**, яка буде проходити по структурі з попереднього завдання та знаходити товар за його ім'ям (перевіряючи всі групи товарів). Імена можуть повторюватися, але цього разу нас цікавить лише перший товар, у якого ім'я збіглося.

Приклад використання функції **find**:

```
const result = find(purchase, 'Laptop');
console.log(result);
```

Повинно вивести: **{ name: 'Laptop', price: 1500 }**

Завдання 5. Тепер розширимо попереднє завдання: потрібно так змінити функцію **find** , щоб вона повертала масив, що містить усі товари із зазначеним ім'ям. Якщо жодного не знайшли, то пустий масив.

3. Стан застосунку, структури даних та колекції

No translation

3.1. Підходи до роботи зі станом: stateful and stateless

No translation

3.2. Структури та записи

```
#include <stdio.h>

struct date {
  int day;
  int month;
  int year;
};

struct person {
  char *name;
  char *city;
  struct date born;
};

int main() {
  struct person p1;
  p1.name = "Marcus";
  p1.city = "Roma";
  p1.born.day = 26;
  p1.born.month = 4;
  p1.born.year = 121;

  printf(
    "Name: %s\nCity: %s\nBorn: %d-%d-%d\n",
    p1.name, p1.city,
    p1.born.year, p1.born.month, p1.born.day
  );
```

```
  return 0;
}
```

Pascal

```
program Example;

type TDate = record
  Year: integer;
  Month: 1..12;
  Day: 1..31;
end;

type TPerson = record
  Name: string[10];
  City: string[10];
  Born: TDate;
end;

var
  P1: TPerson;
  FPerson: File of TPerson;

begin
  P1.Name := 'Marcus';
  P1.City := 'Roma';
  P1.Born.Day := 26;
  P1.Born.Month := 4;
  P1.Born.Year := 121;
  WriteLn('Name: ', P1.Name);
  WriteLn('City: ', P1.City);
  WriteLn(
    'Born: ',
    P1.Born.Year, '-',
    P1.Born.Month, '-',
    P1.Born.Day
  );
  Assign(FPerson, './record.dat');
```

```
  Rewrite(FPerson);
  Write(FPerson, P1);
  Close(FPerson);
end.
```

Rust

```
struct Date {
    year: u32,
    month: u32,
    day: u32,
}

struct Person {
    name: String,
    city: String,
    born: Date,
}

fn main() {
    let p1 = Person {
        name: String::from("Marcus"),
        city: String::from("Roma"),
        born: Date {
            day: 26,
            month: 4,
            year: 121,
        },
    };

    println!(
        "Name: {}\nCity: {}\nBorn: {}-{}-{}\n",
        p1.name, p1.city,
        p1.born.year, p1.born.month, p1.born.day
    );
}
```

TypeScript: Interfaces

```
interface IDate {
  day: number;
  month: number;
  year: number;
}

interface IPerson {
  name: string;
  city: string;
  born: IDate;
}
```

```
const personToString = (person: IPerson): string => {
  const { name, city, born } = person;
  const { year, month, day } = born;
  const fields = [
    `Name: ${name}`,
    `City: ${city}`,
    `Born: ${year}-${month}-${day}`,
  ];
  return fields.join('\n');
};
```

```
const person: IPerson = {
  name: 'Marcus',
  city: 'Roma',
  born: {
    day: 26,
    month: 4,
    year: 121,
  },
};

console.log(personToString(person));
```

TypeScript: Classes

```
class DateStruct {
  day: number;
  month: number;
  year: number;
}

class Person {
  name: string;
  city: string;
  born: DateStruct;
}
```

JavaScript: Classes

```
class DateStruct {
  constructor(year, month, day) {
    this.day = day;
    this.month = month;
    this.year = year;
  }
}

class Person {
  constructor(name, city, born) {
    this.name = name;
    this.city = city;
    this.born = born;
  }
}
```

```
const personToString = (person) => {
  const { name, city, born } = person;
  const { year, month, day } = born;
  const fields = [
    `Name: ${name}`,
    `City: ${city}`,
    `Born: ${year}-${month}-${day}`,
  ];
```

```
  return fields.join('\n');
};
```

```
const date = new DateStruct(121, 4, 26);
const person = new Person('Marcus', 'Roma', date);
console.log(personToString(person));
```

JavaScrip: Objects

```
const person = {
  name: 'Marcus',
  city: 'Roma',
  born: {
    day: 26,
    month: 4,
    year: 121,
  },
};

console.log(personToString(person));
```

JavaScrit: strucute serialization

```
const v8 = require('v8');
const fs = require('fs');
```

Take from previous example:

- class DateStruct
- class Person

```
const date = new DateStruct(121, 4, 26);
const person = new Person('Marcus', 'Roma', date);

const v8Data = v8.serialize(person);
const v8File = './file.dat';
```

```
fs.writeFile(v8File, v8Data, () => {
  console.log('Saved ' + v8File);
});
```

File: file.dat

```
FF 0D 6F 22 04 6E 61 6D 65 22 06 4D 61 72 63 75
73 22 04 63 69 74 79 22 04 52 6F 6D 61 22 04 62
6F 72 6E 6F 22 03 64 61 79 49 34 22 05 6D 6F 6E
74 68 49 08 22 04 79 65 61 72 49 F2 01 7B 03 7B
03
```

Nested structures

```
#include <stdio.h>
#include <map>
#include <string>
#include <vector>

struct Product {
  std::string name;
  int price;
};

void printProduct(Product item) {
  printf("%s: %d\n", item.name.c_str(), item.price);
}

void printProducts(std::vector<Product> items) {
  for (int i = 0; i < items.size(); i++) {
    printProduct(items[i]);
  }
}

int main() {
  std::map<std::string, std::vector<Product>> purchase {
    { "Electronics", {
```

```
      { "Laptop", 1500 },
      { "Keyboard", 100 },
      { "HDMI cable", 10 },
    } },
    { "Textile", {
      { "Bag", 50 },
    } },
  };

  std::vector electronics = purchase["Electronics"];
  printf("Electronics:\n");
  printProducts(electronics);

  std::vector textile = purchase["Textile"];
  printf("\nTextile:\n");
  printProducts(textile);

  Product bag = textile[0];
  printf("\nSingle element:\n");
  printProduct(bag);

  int price = purchase["Electronics"][2].price;
  printf("\nHDMI cable price is %d\n", price);
}
```

Python

```
purchase = {
  'Electronics': [
    { 'name': 'Laptop', 'price': 1500 },
    { 'name': 'Keyboard', 'price': 100 },
    { 'name': 'HDMI cable', 'price': 10 },
  ],
  'Textile': [
    { 'name': 'Bag', 'price': 50 },
  ],
}

electronics = purchase['Electronics']
```

```
print({ 'electronics': electronics })

textile = purchase['Textile']
print({ 'textile': textile })

bag = textile[0]
print({ 'bag': bag })

price = purchase['Electronics'][2]['price']
print({ 'price': price })
```

JavaScript

```
const purchase = {
  Electronics: [
    { name: 'Laptop', price: 1500 },
    { name: 'Keyboard', price: 100 },
    { name: 'HDMI cable', price: 10 },
  ],
  Textile: [{ name: 'Bag', price: 50 }],
};

const electronics = purchase.Electronics;
console.log(electronics);

const textile = purchase['Textile'];
console.log(textile);

const bag = textile[0];
console.log(bag);

const price = purchase['Electronics'][2].price;
console.log(price);

const json = JSON.stringify(purchase);
console.log(json);
const obj = JSON.parse(json);
console.log(obj);
```

3.3. Масив, список, множина, кортеж

No translation

3.4. Словник, хеш-таблиця та асоціативний масив

No translation

3.5. Стек, черга, дек

No translation

3.6. Дерева та графи

No translation

3.7. Проекції та відображення наборів даних

No translation

3.8. Оцінка обчислювальної складності

No translation

4. Розширені концепції

No translation

4.1. Що таке технологічний стек

No translation

4.2. Середовище розробки та налагодження коду

No translation

4.3. Ітерування: рекурсія, ітератори та генератори

No translation

4.4. Структура додатку: файли, модулі, компоненти

No translation

4.5. Об'єкт, прототип та клас

No translation

4.6. Часткове застосування та каррування, композиція функцій

No translation

4.7. Чейнінг для методів та функцій

No translation

4.8. Домішки (mixins)

No translation

4.9. Залежності та бібліотеки

No translation