Глава 4

Прагматична параноя

Порада 36 Написати ідеальну програму неможливо

Це вас зачіпає? А не мало б. Прийміть цю пораду як аксіому, оцінивши та привітавши її, оскільки ідеального програмного забезпечення не існує. Нікому за коротку історію обчислювальної техніки ще не вдавалося написати ідеальну програму, і ви навряд чи станете першим. Якщо ж ви не приймете це як даність, то врешті-решт витратите дарма час та сили, переслідуючи нездійсненну мрію. То як же програмісту-прагматику дістати вигоду з цієї гнітючої реальності? Саме даній темі і присвячено цю главу.

Кожний вважає, що він є найкращим водієм у світі, а всі решта проскакують знаки заборони, лавірують між дорожніми смугами, не даючи сигналів повороту, набирають текст повідомлень на мобільному телефоні та взагалі живуть не за правилами. А ми ж їздимо обережно, остерігаючись біди ще до того, як вона станеться, передбачаючи неочікуване та ніколи не ставлячи себе у таке становище, з якого не можна виплутатися.

Аналогія з програмуванням цілком очевидна. Ми постійно маємо справу з чужим кодом, який може й не відповідати високим стандартам, а також — із вхідними даними, які можуть і не бути достовірними. Тому ми привчені програмувати, будучи насторожі. І якщо у нас виникає якийсь сумнів, то ми перевіряємо всю надану нам інформацію. Для виявлення неякісних даних ми користуємося перевірками, не довіряючи даним, які отримуємо від потенційних атакуючих зловмисників чи тролів. Ми виконуємо перевірку на узгодженість, накладаємо обмеження на стовпчики у таблиці бази даних і, загалом, почуваємося чудово.

Але програміст-прагматик іде далі, *не довіряючи навіть собі*. Прекрасно знаючи, що нікому не дано написати ідеальний код, включно з нами самими, програмісти-прагматики вибудовують бастіони захисту проти власних помилок.

Перший захисний захід описується у розділі "Тема 23. Проєктування за контрактом", коли клієнти та постачальники повинні узгодити права та обов'язки. Далі у розділі "Тема 24. Мертві програми не брешуть" пояснюється, що обробка програмних помилок не завдає жодної шкоди. Тому слід намагатися здійснювати перевірки якомога частіше та переривати програму, якщо справи стануть кепські. А у розділі "Тема 25. Стверджувальне програмування" описується простий метод оперативної перевірки, який полягає у написанні коду, що активно перевіряє припущення програміста.

По мірі того, як ваша програма стає все більш динамічною, ви бачите, що маніпулюєте системними ресурсами (оперативною пам'яттю, файлами, пристроями тощо), ніби жонглюючи кулями. Тому у розділі "Тема 26. Як збалансувати ресурси" пропонуються способи не втратити жодної такої кулі. А найголовніше — як просуватися до мети дрібними кроками, щоб не звалитися у прірву з краю скелі, як пояснюється у розділі "Тема 27. Не випереджуйте світло фар свого автомобілю".

У світі недосконалих систем, безглуздих часових шкал, забавних інструментів та нездійсненних вимог доводиться діяти обережно. Як одного разу сказав відомий кінорежисер Вуді Аллен: "Коли всі фактично будують до вас підступи, єдиною хорошою думкою стає параноя".

Тема 23 Проєктування за контрактом

Ніщо так не вражає людей, як здоровий глузд та відвертість.

Ралф Волдо Емерсон, *Есеї*[1] (Essays)

Мати справу з обчислювальними системами нелегко, а мати справу з людьми ще складніше. Але як біологічний вид ми з давніх давен з'ясовували питання людських взаємовідносин. Деякі рішення, до яких ми прийшли за останні кілька тисячоліть, можуть згодитися і для написання програм. І одним з найкращих рішень, які гарантують прямоту відносин, є *контракт*. Контракт визначає права та обов'язки обох сторін, що його уклали. Крім того, у ньому передбачена угода сторін, що стосується наслідків, якщо одна з них не виконає контракту.

Скажімо, ви могли укласти контракт про наймання на роботу, у якому вказано, скільки годин ви повинні працювати, а також правила, яких

[1] Ralph Waldo Emerson — американський есеїст, поет, філософ, пастор, суспільний діяч XIX століття. — *Приміт. перекл.*

ви зобов'язані дотримуватися. У свою чергу, роботодавець платить вам зарплатню та решту надбавок. Якщо кожна зі сторін дотримується своїх обов'язків, то виграють від цього усі.

Ідея контракту використовується у всьому світі (формально чи неформально) з метою допомогти людям налагодити взаємовідносини. А чи може той самий принцип допомогти у налагоджуванні взаємодій між програмними модулями? Так, може.

Принцип проєктування за контрактом

Бертран Меєр (Bertrand Meyer; *Object-Oriented Software Construction* [Mey97]) розробив *принцип проєктування за контрактом* для мови Eiffel[2]. Це проста, але ефективна методика, спрямована на документування (та узгодження) прав і обов'язків програмних модулів, щоб забезпечити правильність програми. А що таке правильна програма? Це така програма, яка виконує лише те, що від неї потрібно, — не більше і не менше. Документування та верифікація такої вимоги і складають саму суть *проєктування за контрактом* (Design by Contract, DBC).

Кожна функція та метод у програмній системі *виконує* якусь *дію*. Перш ніж почати цю *дію*, функція може передбачати якийсь стан оточуючого світу, а після завершення вона може зробити заяву про стан оточуючого світу. Ці припущення та вимоги Меєр описує так.

- **Попередні умови.** Це вимоги підпрограми, які визначають, що повинно бути істинним для її виклику. Підпрограма взагалі не повинна викликатися, якщо її попередні умови буде порушено. На код виклику покладається відповідальність за передавання якісних даних (див. врізання "Хто несе відповідальність?" далі у цій главі).
- **Умови після.** Це стан оточуючого світу після завершення підпрограми, тобто, те, що нею гарантується. Наявність умови після у підпрограми передбачає, що програма неодмінно *завершиться*, а отже, нескінченні цикли не припускаються.
- **Інваріанти класу.** Клас гарантує, що дана умова для коду виклику завжди істинна. У процесі внутрішньої обробки у підпрограмі інваріант може і не дотримуватися, але до моменту виходу з підпрограми та передавання управління коду виклику інваріант повинен бути

[2] Частково ґрунтуючись на колишніх роботах Дейкстри (Dijkstra), Флойда (Floyd), Гоара (Hoare), Вірта (Wirth) та ін.

неодмінно істинним. (Слід, однак, мати на увазі, що клас не може надати необмежений доступ для запису будь-якому члену даних, який бере участь у інваріанті.)

Таким чином, контракт між підпрограмою та будь-яким потенційним кодом виклику може бути складено так.

- Якщо всі попередні умови підпрограми задовольняються кодом виклику, то підпрограма гарантує істинність всіх умов після та інваріантів при своєму завершенні.

Якщо ж одна зі сторін контракту не виконує його умови, то викликається (попередньо узгоджений) засіб захисту порушених прав, наприклад: виникає виключення чи переривання програми. Що б не сталося, не сумнівайтесь, що недотримання умов контракту вважається програмною помилкою. Немає жодної гарантії, що цього взагалі не станеться, саме тому попередні умови і не можна використовувати для виконання таких операцій, як перевірка даних, що вводяться користувачем, на достовірність.

У одних мовах ці принципи дотримуються краще, ніж в інших. Наприклад, у мові Clojure підтримуються попередні умови та умови після, а також більш широкий інструментарій, який надається за *специфікацією*. Нижче наведено приклад функції для відкриття внеску у банку за допомогою попередніх умов та умов після.

```
(defn accept-deposit [account-id amount]
  { :pre [ (> amount 0.00)
           (account-open? account-id) ]
    :post [ (contains? (account-transactions account-id) %) ] }
  "Accept a deposit and return the new transaction id"
  ;; Тут виконується решта обробки...
  ;; Повернути новостворену транзакцію:
  (create-transaction account-id :deposit amount))
```

Для функції `accept-deposit()` тут задано дві попередні умови. Перша полягає у тому, що сума внеску має бути більше нуля, а друга — у тому, що рахунок повинен бути відкритим та дійсним. Останнє визначається шляхом виклику функції `account-open?()`. Існує також така умова після: функція гарантує, що нова транзакція (повернення з даної функції значення, представленого у відсотках “%”) може перебувати серед інших транзакцій для даного рахунку.

Якщо викликати функцію `accept-deposit()` з позитивною сумою для відкриття внеску та дійсним рахунком у банку, то вона створить далі транзакцію підходящого типу та виконає всю решту необхідної обробки.

Але якщо у програмі є помилка і даній функції якимось чином передано від'ємну суму для відкриття внеску, то під час виконання виникне такий виняток:

```
Exception in thread "main"...
Caused by: java.lang.AssertionError:
          Assert failed: (> amount 0.0)
```

Крім того, дана функція вимагає, щоб вказаний рахунок був відкритий та дійсний. Якщо це не так, з'явиться виняток:

```
Exception in thread "main"...
Caused by: java.lang.AssertionError:
          Assert failed: (account-open? account-id)
```

В інших мовах є засоби, що дають непоганий результат, попри те, що вони не стосуються безпосередньо проєктування за контрактом. Наприклад, у мові Elixir використовуються *оператори захисту* для диспетчеризації викликів функції за кількома її тілами, що наявні:

```
defmodule Deposits do
  def accept_deposit(account_id, amount) when (amount > 100000) do
    # Викликати диспетчер!
  end
  def accept_deposit(account_id, amount) when (amount > 10000) do
    # Додаткові федеральні вимоги для
    # Тут виконується деяка обробка...
  end
  def accept_deposit(account_id, amount) when (amount > 0) do
    # Тут виконується деяка обробка...
  end
end
```

У даному випадку виклик функції `accept_deposit()` з великою сумою може вимагати додаткових кроків та обробки. Але якщо спробувати викликати дану функцію з сумою, що менша чи дорівнює нулю, то виникне наведений нижче виняток, який сповіщає, що зробити такий внесок не можна.

```
** (FunctionClauseError) no function clause
    matching in Deposits.accept_deposit/2
```

Це простіше, ніж просто перевіряти вхідні дані. У цьому випадку функцію `accept_deposit()` просто не можна викликати, якщо її аргументи вказані поза заданими межами.

Порада 37 Проєктуйте за контрактом

Проєктування за контрактом та розробка на основі тестування

Чи є потреба у проєктуванні за контрактом там, де розробники практикують модульне тестування, розробку на основі тестування (Test-Driven Development — TDD), тестування на основі властивостей чи захисне програмування? Звісно, є.

Проєктування за контрактом та тестування — це різні підходи до більш широкої теми правильності програм. Обидва підходи цінні та придатні у різних ситуаціях. Проєктування за контрактом дає низку переваг порівняно з іншими підходами до тестування.

- Воно не потребує жодної підготовки чи імітації.
- Визначає параметри вдалого чи невдалого результату у *всіх* випадках, тоді як тестування може бути одночасно націлене лише на один конкретний випадок.
- Розробка на основі тестування та інші різновиди тестування відбуваються у циклі збирання лише під час тестування, а проєктування за контрактом та затвердження — постійно: під час проєктування, розробки, розгортання та супроводу.
- Розробка на основі тестування не зосереджена на перевірці внутрішніх інваріантів у коді, що тестується, а діє більшою мірою за принципом "чорного ящика" для перевірки відкритого інтерфейсу.
- Проєктування за контрактом більш ефективне та дотримується принципу DRY точніше, ніж захисне програмування, де *всі* повинні перевірити дані на достовірність, якщо цього більше ніхто не робить.

Розробка на основі тестування є чудовою методикою, але, як і багато інших методик розробки, вона може призвести до зосередження на вдалому шляху, а не на реальному світі, де повно неякісних даних, поганих виконавців, невдалих версій та кепських специфікацій.

У розділі "Тема 10. Ортогональність" глави 2 "Прагматичний підхід" ми рекомендували писати "скромний" код. А тут акцент робиться на "ледачому" коді, який передбачає, що ви суворо ставитесь до того, що приймається, перш ніж починати щось робити, а також обіцяєте якомога менше у відповідь. Не слід, однак, забувати, що якщо у контракті ви погодилися приймати все що завгодно та наобіцяли на виході цілий світ, то для збереження такого контракту вам доведеться написати чимало коду! При програмуванні будь-якою мовою, хай вона буде функціональною, об'єк-

тно-орієнтованою чи процедурною, проєктування за контрактом змушує вас *думати*.

Інваріанти класу та функціональні мови

Вся справа у назвах. Мова Eiffel є об'єктно-орієнтованою, і тому Меєєр назвав дане поняття "інваріантом класу". Але насправді це більш загальне поняття, що означає *стан*. У об'єктно-орієнтованій мові стан пов'язаний з екземплярами класів, але ж стан існує й у інших мовах. Скажімо, у функціональній мові стан, як правило, передається функції та приймається оновленим у результаті її виконання. Поняття інваріантів виявляються настільки ж корисними і за цих обставин.

Реалізація проєктування за контрактом

Просте перерахування меж сфери вхідних значень, граничних умов та того, що підпрограма обіцяє (а ще важливіше те, чого вона *не* обіцяє) надати, перед тим, як писати код, — це величезний крок вперед у розробці більш якісного програмного забезпечення. Не сформулювавши всі ці умови, ви, по суті, повертаєтесь до *програмування за збігом*, яке розглядається у розділі "Тема 38. Програмування за збігом" глави 7 "По ходу кодування". Саме з цього починаються, цим закінчуються та зазнають краху багато проєктів.

У тих мовах, де проєктування за контрактом у початковому тексті не підтримується, це може бути все, чого вдасться досягти, але і це не так вже погано. Адже проєктування за контрактом — це всього лише методика *проєктування*. Навіть не вдаючись до автоматичної перевірки, можна впровадити контракт до початкового тексту у вигляді коментарів чи як модульні тести і, тим не менше, отримати від цього вельми реальну вигоду.

Твердження

Попри те, що документування подібних припущень слугує хорошим початком, можна отримати ще більшу вигоду, змусивши компілятор автоматично перевіряти контракт. Таку перевірку можна частково симулювати у деяких мовах за допомогою *тверджень* (assertions) — перевірки логічних умов під час виконання (див. далі розділ "Тема 25. Стверджувальне програмування"). А чому лише частково? Чи можна за допомогою тверджень домогтися всього, на що лише здатне проєктування за контрактом?

На жаль, не можна. Перш за все, у об'єктно-орієнтованих мовах може не підтримуватися розповсюдження тверджень за ієрархією наслідування. Це означає, що якщо перевизначити у базовому класі метод, що має контр-

акт, то твердження, які реалізують цей контракт, не будуть викликані коректним чином, якщо тільки не продублювати їх вручну у новому коді. І треба не забути викликати інваріант класу (і всі інваріанти базового класу) вручну перед виходом з кожного методу. Головна складність полягає у тому, що контракт не виконується автоматично. А у інших середовищах винятки, що генеруються з тверджень, складених у стилі проєктування за контрактом, можуть бути відключені глобально чи повністю проігноровані початковим текстом.

Крім того, не передбачене таке поняття, як "старі" значення, тобто, значення, які існували на момент входу до методу. Якщо ви користуєтесь твердженнями для перевірки контрактів, то повинні ввести код до попередньої умови, щоб зберегти будь-яку інформацію, яка знадобиться в умові після, якщо мова взагалі допускає таке. Скажімо, у мові Eiffel, де, власне, зародилося проєктування за контрактом, для цієї мети достатньо скористатися *виразом* `old`.

Нарешті, звичайні системи часу виконання та бібліотеки не призначені для підтримки контрактів, тому подібні виклики не перевіряються. Це велика втрата, оскільки на межі між прикладним кодом та бібліотеками, що у ньому застосовуються, виявляється більшість складнощів (докладніше про це — у розділі "Тема 24. Мертві програми не брешуть").

Проєктування за контрактом та аварійне завершення

Проєктування за контрактом витончено вписується до поняття дострокового аварійного завершення (див. далі у розділі "Тема 24. Мертві програми не брешуть"). Використовуючи механізм тверджень чи проєктування за контрактом для перевірки достовірності попередніх умов, умов після та інваріантів, можна домогтися дострокового аварійного завершення програми та повідомити більш точні дані про неполадку, що виникла.

Припустимо, що є метод, який обчислює квадратні корені. Цьому методу потрібна попередня умова проєктування за контрактом, яка обмежує сферу його дії додатними числами. Якщо передати даному методу від'ємне значення параметру `sqrt` у тих мовах, де підтримується проєктування за контрактом, то врешті-решт буде отримане повідомлення про помилку на кшталт `sqrt_arg_must_be_positive` (аргумент для витягання квадратного кореня має бути додатним) разом з результатами трасування стеку. Це, звісно, краще, ніж альтернатива, що є в інших мовах на кшталт Java, C та C++, де в результаті передавання методу від'ємного значення

параметру `sqrt` повертається спеціальне значення `NaN` (не число). І якщо деякий час потому спробувати виконати у програмі які-небудь математичні операції над значенням `NaN`, то врешті-решт буде отримано геть неочікувані результати.

Набагато простіше знайти та діагностувати неполадку, достроково досягнувши аварійного завершення у місці виникнення цієї неполадки.

Хто несе відповідальність?

Хто ж відповідає за перевірку попередньої умови: код виклику чи підпрограма, яка викликається? Якщо проєктування за контрактом реалізується як частина мови, то за таку перевірку не відповідає ні те, ні інше. Адже попередня умова перевіряється підспудно після того, як підпрограма буде викликана у коді виклику, але до входу до самої підпрограми. Скажімо, якщо треба виконати явну перевірку параметрів, це повинно бути зроблено у *коді виклику*, оскільки самій підпрограмі взагалі недоступні параметри, які порушують попередню умову. (У тих мовах, де відсутня вбудована підтримка проєктування за контрактом, щоб перевірити подібні твердження, програма, *що викликається*, повинна мати для цієї мети преамбулу та/або заключну частину.)

Розглянемо у якості прикладу програму, що вводить число з консолі, витягає з нього квадратний корінь за допомогою функції `sqrt()` та виводить результат. У функції `sqrt()` є така попередня умова: її аргумент не повинен бути від'ємним. Якщо користувач введе до консолі від'ємне число, то код виклику зобов'язаний вжити заходів, щоб воно не було передане до функції `sqrt()`. У цього коду виклику є багато варіантів дій: перервати виконання, видати попередження та вимагати ввести інше число в консолі, зробити число додатним чи додати до результату, що повертається функцією `sqrt()`, символ уявної одиниці (*i*). Але який би вибір не було зроблено, це абсолютно не стосується самої функції `sqrt()`.

Висловлюючи сферу дії функції `sqrt()`, яка витягає квадратний корінь, у її попередній умові, ви фактично переносите тягар відповідальності за правильність на код виклику `sqrt()`, де йому і місце. І тоді ви можете благополучно проєктувати функцію `sqrt()`, твердо знаючи, що її вхідні дані перебуватимуть у межах, що задано контрактом.

Семантичні інваріанти

Семантичні інваріанти можуть бути використані для висловлення незмінних вимог, чогось на кшталт "філософського контракту". Нам якось довелося розробляти програмний комутатор транзакцій за дебетовими

банківськими картками. Головна вимога полягала у тому, щоб користувач дебетової картки не міг двічі виконати одну й ту саму транзакцію на своєму рахунку. Іншими словами, якого б роду режим відмови не виник, помилка повинна викликати заборону обробки транзакції, але тільки не обробку дубльованої транзакції. Це просте правило, яке випливає безпосередньо з вимог, виявляється дуже корисним, коли доводиться розбиратися зі складними випадками усунення помилок. І воно спрямовує на шлях докладного проєктування та реалізації у багатьох галузях.

В жодному разі не плутайте вимоги, що є фіксованими та незмінними правилами, з вимогами, які являють собою лише стратегії, що можуть змінитися разом з новим режимом керування. Саме тому ми користуємося тут терміном *семантичні інваріанти*, оскільки йому належить центральне місце у визначенні самої *суті* предмету, і він не повинен підкорюватися примхам стратегії, якій більшою мірою відповідають більш динамічні бізнес-правила.

Коли ви виявляєте вимогу, яка може уточнюватися, вживіть заходів, щоб вона стала добре відомою частиною будь-якої документації, що складається, хай це буде маркований список у документі вимог, який підписується у трьох примірниках, або ж велика замітка на спільній білій дошці, яку бачать усі. Спробуйте сформулювати цю вимогу чітко та однозначно. Скажімо, якщо повернутися до прикладу з дебетовими картками, то така вимога може бути сформульована таким чином.

Помилка — на користь споживача.

Це чітке, коротке, однозначне формулювання, яке можна застосувати на найрізноманітніших ділянках системи. Це наш контракт з усіма користувачами системи та наша гарантія її поведінки.

Динамічні контракти та агенти

Досі йшлося про контракти як фіксовані, незмінні специфікації. Але у сфері автономних агентів це зовсім не обов'язково повинно бути саме так. За визначенням, *автономні агенти* вільні відкидати ті запити, які вони не бажають брати до уваги. Вони вільні узгоджувати контракт заново, заявляючи: "Я не можу цього надати, але якщо ви дасте мені те і це, тоді я міг би надати вам щось інше".

Безумовно, будь-яка система, що спирається на технологію агентів, перебуває у *вирішальній* залежності від договірних стосунків, навіть якщо вони генеруються динамічно.

Уявіть: за достатньої кількості компонентів та агентів, які здатні узгоджувати між собою контракти, щоб досягти поставленої мети, ми могли б просто дати програмному забезпеченню можливість розв'язати кризу його продуктивності замість нас.

Але якщо ми не можемо користуватися контрактами вручну, то не зможемо скористатися ними і автоматично. Тому наступного разу, коли проєктуватимете якесь програмне забезпечення, спроєктуйте для нього контракт.

Інші розділи, пов'язані з цією темою

- **Тема 24.** Мертві програми не брешуть.
- **Тема 25.** Стверджувальне програмування.
- **Тема 38.** Програмування за збігом, **глава 7** "По ходу кодування".
- **Тема 42.** Тестування на основі властивостей, **глава 7** "По ходу кодування".
- **Тема 43.** Будьте обережні, **глава 7** "По ходу кодування".
- **Тема 45.** Пастка вимог, **глава 8** "До початку проєкту".

Завдання

- Розгляньте такі питання для роздумів: якщо проєктування за контрактом є настільки ефективним, то чому воно не знаходить більш широкого застосування? Чи важко сформулювати контракт? Чи змушує це вас замислитися над тими питаннями, які ви інакше проігнорували б? Чи змушує це вас взагалі ДУМАТИ?! Очевидно, що це небезпечний інструментальний засіб!

Вправи

14. Розробіть інтерфейс для кухонного блендера. Врешті-решт це повинен бути орієнтований на веб та інтернет речей блендер, але поки що потрібен лише інтерфейс для керування ним. В ньому повинні бути засоби для встановлення десяти швидкостей, причому 0 означає вимкнення блендеру. Працювати з блендером вхолосту не можна, а його швидкість можна змінювати лише по черзі та на одиницю, тобто, від 0 до 1 та від 1 до 2, але не від 0 до 2. Нижче наведено відповідні методи. Додайте відповідні попередні умови та умови після та інваріант.

```
int getSpeed()
void setSpeed(int x)
boolean isFull()
void fill()
void empty()
```

15. Скільки всього чисел у ряду 0, 5, 10, 15, …, 100?

Тема 24 Мертві програми не брешуть

Чи доводилося вам помічати, що іноді інші люди можуть раніше за вас побачити, що з вами щось не так? Те саме відбувається і з кодом. Скажімо, якщо з однією з ваших програм станеться щось погане, то неполадка іноді виявляється у сторонній бібліотеці чи каркасі. Причина може бути, наприклад, у тому, що до бібліотечної функції було передано порожнє значення `nil` або ж порожній список. Можливо, у хеші відсутній ключ, чи значення, яке, як нам здавалося, містить хеш, насправді містить список. А можливо, у мережі чи у файловій системі сталася помилка, яка не була перехоплена, через що були отримані порожні чи зіпсовані дані. Логічна помилка, яку було зроблено пару мільйонів операцій тому, означає, що у селекторі оператору `case` більше не очікується значення `1`, `2` чи `3`, і тому несподівано відбувається перехід до гілки `default`. І це ще одна причина, з якої у кожній інструкції `switch` неодмінно повинна бути присутня гілка `default` на той випадок, якщо станеться щось "неможливе".

Дуже легко впасти у помилку "цього не може статися". Більшості з нас доводилося писати код, у якому не перевірялося успішне закриття файлу та належне написання оператору трасування. І за решти рівних умов нам це, найімовірніше, було не потрібно, оскільки ми вважали, що наш код не зазнає краху за будь-яких нормальних умов. Але ми використовуємо захисне програмування. Зокрема, перевіряємо дані на достовірність, робочий код — на працездатність, а версії завантажених залежностей — на правильність.

Всі помилки дають якусь інформацію для роздумів. Можна, звісно, переконати себе, що помилки не станеться, і просто проігнорувати її. Замість цього програмісти-прагматики говорять собі: якщо виникла помилка, значить, відбулося щось дуже і дуже кепське. Тому не забувайте читати всі ці огидні повідомлення про помилки (див. розділ "Програміст у чужій країні" глави 3).

Принцип "піймав — відпустив" — лише для ловлі риби

Деякі розробники вважають хорошим стилем програмування перехоплювати всі винятки чи відновлювати нормальну роботу програми після їхнього виникнення, а також повторно генерувати їх після виводу якогось повідомлення. В їхньому коді можна знайти чимало таких місць, як у наведеному нижче фрагменті, де порожній оператор `raise` повторно генерує поточний виняток.

```
try do
    add_score_to_board(score);
rescue InvalidScore
    Logger.error("Can't add invalid score. Exiting");
    raise
rescue BoardServerDown
    Logger.error("Can't add score: board is down. Exiting");
    raise
rescue StaleTransaction
    Logger.error("Can't add score: stale transaction. Exiting");
    raise
end
```

А ось що написав би програміст-прагматик:

```
add_score_to_board(score);
```

Ми віддаємо перевагу саме такому стилю з двох причин. По-перше, прикладний код не затьмарюється обробкою помилок. По-друге, що, мабуть, ще важливіше, прикладний код виявляється менш зв'язаним. У наведеному вище багатослівному прикладі доводиться перераховувати кожен виняток, який може бути згенерований у методі `add_score_to_board()`. Якщо автор цього методу введе ще один виняток, то його код ледве помітно застаріє. А у більш прагматичному другому прикладі новий виняток розповсюджується автоматично.

Порада 38 Користуйтесь достроковим аварійним завершенням програми

Аварійне завершення замість відправки на смітник

Одна з переваг якомога більш раннього виявлення неполадок полягає у можливості дострокового аварійного завершення. І часто-густо це найкраще, що можна зробити. У якості альтернативи можна продовжити,

записавши зіпсовані дані до якоїсь актуальної бази даних чи давши пральній машині команду на двадцятому за рахунком циклі обертання її барабану з білизною. Саме такий принцип підтримується у мовах Erlang та Elixir. Джо Армстронг, винахідник мови Erlang та автор книги *Programming Erlang: Software for a Concurrent World* [Arm07] (Програмування на Erlang: програмне забезпечення для паралельного світу), часто говорив таке: "Захисне програмування — це марне витрачання часу. Припускайте аварійне завершення!" У таких середовищах передбачається, що програми можуть відмовляти, але відмова перебуває під управлінням *супервізора*, який відповідає за виконання коду та знає, що саме слід робити, якщо код відмовить. До подібних заходів стосується очищення після збою, перезапуск коду тощо. А що, як відмовить сам супервізор? Цією подією керує його власний супервізор, що призводить до архітектури, яка складається з *дерев супервізорів*. Така методика є доволі ефективною та сприяє застосуванню згаданих вище мов у високонадійних, стійких до відмов системах.

В інших середовищах простий вихід з працюючої програми може виявитися неприйнятним. Адже у ній могли бути затребувані ресурси, які можуть бути і не звільнені. А можливо, доведеться написати протокольні повідомлення, очистити відкриті транзакції чи організувати взаємодію з іншими процесами.

Тим не менше основний принцип залишається тим самим: якщо у прикладному коді виявляється, що відбулося щось, що вважалося неможливим, такий код більше не вважається життєздатним. І все, що у такому коді робиться далі, стає підозрілим, а отже, його виконання слід перервати якомога скоріше. Мертва програма, як правило приносить набагато менше шкоди, ніж зіпсована.

Інші розділи, пов'язані з цією темою

- **Тема 20.** Налагодження, **глава 3** "Основні інструментальні засоби".
- **Тема 23.** Проєктування за контрактом.
- **Тема 25.** Стверджувальне програмування.
- **Тема 26.** Як збалансувати ресурси.
- **Тема 43.** Будьте обережні, **глава 7** "По ходу кодування".

Тема 25 Стверджувальне програмування

Самоосуд нам дарує незрівнянну насолоду: коли ми ганимо самі себе, то відчуваємо, що ніхто інший не має права ганити нас.

Оскар Вайлд, *Портрет Доріана Ґрея*,
переклад Ростислава Доценка

Мабуть, є така мантра, яку кожен програміст повинен запам'ятати, ледве почавши свою кар'єру. Це фундаментальний постулат, переконання, яке ми вчимося застосовувати до вимог, проєктів, початкового коду, коментарів та практично до всього, що ми робимо. І він звучить так.

Це ніколи не може статися…

Прикладами цього слугують такі твердження: "Цей застосунок ніколи не буде використовуватися за кордоном, то навіщо його інтернаціоналізувати?", "Змінна `count` не може приймати від'ємне значення" чи "Протоколювання не може дати збій". Намагайтесь уникати у своїй практиці подібного самообману, особливо під час програмування.

Порада 39 Користуйтесь твердженнями, щоб запобігти неможливому

Кожного разу, коли ви хапаєте себе на думці "але цього, звісно, ніколи не може статися", вводьте код для її перевірки. І зробити це найпростіше за допомогою тверджень. У реалізаціях багатьох мов можна виявити деяку форму тверджень у вигляді оператора `assert`, де перевіряється логічна умова[3]. Такі перевірки можуть виявитися неоціненними. Скажімо, якщо параметр чи результат взагалі не повинен бути порожнім `(null)`, його слід перевірити явним чином, як показано нижче.

```
assert (result != null);
```

В реалізації Java можна (і треба) ввести до твердження описовий рядок таким чином:

```
assert result != null && result.size() > 0 : "Empty result from XYZ";
```

[3] У мовах C та C++ твердження зазвичай реалізуються у вигляді макрокоманд, а у мові Java вони заборонені за замовчуванням. Щоб дозволити їх, слід викликати віртуальну машину Java з параметром — `enableassertions` з командного рядку та залишити їх дозволеними.

Твердження корисні також для перевірки функціонування алгоритмів. Скажімо, якщо написати алгоритм витонченого сортування під назвою `my_sort`, перевірити його працездатність можна, наприклад, таким чином:

```
books = my_sort(find("scifi"))
assert(is_sorted?(books))
```

В жодному разі не користуйтесь твердженнями замість реальної обробки помилок. Твердження дозволяють перевірити те, що ніколи не повинно статися. Навряд чи вам захочеться написати код, аналогічний до наведеного нижче.

```
puts("Enter 'Y' or 'N': ")
ans = gets[0]                          # Взяти перший символ з відповіді
assert((ch == 'Y') || (ch == 'N'))  # Явно невдала ідея!
```

Те, що більшість реалізацій оператора `assert` просто завершує процес, якщо твердження не виконується, не є причиною для того, щоб це відбувалося і у написаних вами версіях програмного забезпечення. Скажімо, якщо вам треба звільнити ресурси, що використовуються, перехопіть виняток, який виник у твердженні, або перервіть вихід з програми, а далі виконайте свій обробник помилок. Треба лише зробити так, щоб код, який виконується протягом цих мілісекунд, що спливають, перш за все не спирався на інформацію, яка послужила причиною того, що твердження не справдилося.

Твердження та побічні ефекти

Якщо код, що вводиться для виявлення помилок, фактично призводить до нових помилок, то становище лише погіршується. Таке може статися і з твердженнями, якщо перевірка умови має побічні ефекти. Наприклад, написання наступного фрагменту коду навряд чи можна вважати вдалою ідеєю:

```
while (iter.hasMoreElements()) {
    assert(iter.nextElement() != null);
    Object obj = iter.nextElement();
    // ....
}
```

Виклик `.nextElement()` у твердженні має побічний ефект, який проявляється у тому, що ітератор пропускає елемент, що витягується, і тому у циклі `while` буде оброблено лише половину елементів, які є у колекції.

У такому випадку було б краще написати даний цикл таким чином:

```
while (iter.hasMoreElements()) {
  Object obj = iter.nextElement();
  assert(obj != null);
  // ....
}
```

Це різновид невизначеної, ледве вловимої програмної помилки під назвою *гайзенбаг*[4], яка змінює поведінку системи, що налагоджується. Ми також вважаємо, що тепер, коли у більшості мов є пристойна підтримка функцій, які циклічно перебирають колекції, такого роду явний цикл не потрібний та є невдалим різновидом їхньої обробки.

Залишайте увімкненим режим тверджень

Стосовно тверджень існує вельми розповсюджена помилка:

> Твердження вносять деякі видатки до коду. Вони перевіряють те, що ніколи не повинно статися, і тому приводяться у дію лише програмною помилкою у коді. І щойно код буде перевірено та доставлено користувачу, вони більше не знадобляться, а отже, режим тверджень можна відключити, щоб код виконувався швидше. Твердження є засобом налагодження початкового коду.

У наведеному вище міркуванні є два заздалегідь хибних припущення. По-перше, у ньому припускається, що під час тестування виявляються всі програмні помилки. Насправді у будь-якій складній програмі навряд чи вдасться перевірити навіть найменшу частку всіх перестановок, що зроблено у початковому коді. І по-друге, оптимісти забувають, що їхні програми виконуються у небезпечному середовищі. Під час тестування щури навряд чи перегризуть кабель зв'язку, гравець вичерпає оперативну пам'ять по ходу комп'ютерної гри, а журнальні файли заполонять весь розділ запам'ятовуючого пристрою. Подібні події скоріше можуть відбутися за виконання програми у реальному експлуатаційному середовищі. Тому вашою першою лінією оборони повинна стати перевірка будь-яких можливих помилок, а другою лінією — застосування тверджень, щоб спробувати виявити пропущені помилки.

Вимикання режиму тверджень під час запуску програми до експлуатації можна порівняти з ходженням по натягнутому під куполом цирку дроті

[4]Назва гайзенбаг (Heisenbug) походить від принципу невизначеності Гайзенберга з квантової механіки та, власне, програмної помилки (bug). Докладніше про це див. за адресою: http://www.eps.mcgill.ca/jargon/jargon.html#heisenbug.

без страхової сітки лише тому, що це вже одного разу вдалося зробити. У цьому є драматизм та видовищність, але не страховка від летального наслідку.

Навіть якщо вам доводиться вирішувати питання продуктивності, вимикайте лише ті твердження, які дійсно негативно впливають на продуктивність. Скажімо, наведений вище приклад сортування може бути вкрай важливою частиною застосунку, яка повинна виконуватися якомога швидше. Додавання до неї перевірки означає ще один прохід даних, що може виявитися неприйнятним. Тому зробіть цю конкретну перевірку необов'язковою, але решту перевірок залиште без змін.

Застосування тверджень в умовах реальної експлуатації приносить чималий прибуток

Колишній сусід Енді керував невеликою компанією-стартапом, що виробляла мережеві пристрої. Одним з секретів їхнього успіху було рішення залишити твердження на місці у робочих версіях. Ці твердження були достатньо грамотно складені, щоб повідомляти всю інформацію, яка мала стосунок до збоїв, а також представлені кінцевому користувачу через витонченого виду інтерфейс. Такий рівень організації відгуків реальних користувачів у конкретних умовах експлуатації дозволяв розробникам затикати дірки у захисті та усувати ледь помітні, важко відтворювані програмні помилки, а отже, виробляти неймовірно стійке, надійне програмне забезпечення. Ця невелика маловідома компанія виробляла настільки добротну продукцію, що невдовзі її було придбано за сотні мільйонів доларів. Але це так, до слова.

Інші розділи, пов'язані з цією темою

- **Тема 23.** Проєктування за контрактом.
- **Тема 24.** Мертві програми не брешуть.
- **Тема 42.** Тестування на основі властивостей, **глава 7** "По ходу кодування".
- **Тема 43.** Будьте обережні, **глава 7** "По ходу кодування".

Вправи

16. Швидка перевірка на почуття реальності. Які з перерахованих нижче "неможливих" подій можуть все ж статися?

– Місяць, у якому менше 28 днів.
– Код помилки, що повертається з системного виклику та означає неможливість доступу до поточного каталогу.
– У мові C++: `a = 2`, `b = 3`, але `(a + b)` не дорівнює `5`.
– Трикутник, сума кутів якого не дорівнює 180°.
– Хвилина, що не налічує 60 секунд.
– `(a + 1) <= a`

Тема 26 Як збалансувати ресурси

…запалена свічка неодмінно створює тінь...

Урсула Ле Гуїн, *Чарівник Земномор'я*,
переклад Анатолія Сагана

Коли ми програмуємо, то так чи інакше маніпулюємо ресурсами: оперативною пам'яттю, транзакціями, потоками виконання, мережевими з'єднаннями, файлами, таймерами, тобто, всіма засобами з обмеженою доступністю. І найчастіше ресурси використовуються за цілком передбачуваним шаблоном: спочатку ресурс виділяється, потім використовується, і, нарешті, звільняється.

Тим не менше у багатьох розробників немає постійного плану з виділення та звільнення ресурсів. Тому дамо у зв'язку з цим таку пораду.

Порада 40 Завершуйте те, що почали

Цю пораду легко застосувати у більшості випадків. Вона просто означає, що функція чи об'єкт, які виділяють ресурс, повинні відповідати за його звільнення. Розглянемо застосування цієї поради на конкретному прикладі невдало написаного на Ruby фрагменту коду, що відкриває файл, читає з нього інформацію про клієнта, оновлює окреме поле даних та записує результат назад до файлу. Заради простоти та більшої ясності даного прикладу з нього виключено перевірку помилок, як показано нижче.

```
def read_customer
    @customer_file = File.open(@name + ".rec", "r+")
    @balance = BigDecimal(@customer_file.gets)
end
```

```
def write_customer
    @customer_file.rewind
    @customer_file.puts @balance.to_s
    @customer_file.close
end

def update_customer(transaction_amount)
    read_customer
    @balance = @balance.add(transaction_amount,2)
    write_customer
end
```

На перший погляд, підпрограма `update_customer()` з наведеного вище фрагменту коду виглядає цілком обґрунтовано. Мабуть, для реалізації її логіки знадобиться прочитати запис, оновити баланс на поточному рахунку та записати оновлений запис назад. Тим не менше за такою охайністю приховується головна складність. Підпрограми `read_customer()` та `write_customer()` тісно пов'язані, оскільки разом користуються спільною змінною екземпляру `customer_file`. Спочатку підпрограма `write_customer()` відкриває файл та зберігає посилання на нього до змінної `customer_file`, а далі підпрограма `read_customer()` виконує збережене посилання, щоб після його завершення закрити файл. Але ця спільна змінна навіть не згадується у підпрограмі `update_customer()`.

Чому це погано? Розглянемо приклад програміста-невдахи, який отримав завдання внести таку зміну до специфікації застосунку, що він супроводжує: баланс на рахунку повинен оновлюватися лише у тому випадку, якщо нове значення не є від'ємним. З цією метою він звертається до початковому коду та вносить такі зміни до підпрограми `update_customer()`:

```
def update_customer(transaction_amount)
    read_customer
    if (transaction_amount >= 0.00)
        @balance = @balance.add(transaction_amount,2)
        write_customer
    end
end
```

Під час тестування цей код начебто поводить себе як слід. Але коли цей код запускається до експлуатації, він зазнає краху за кілька годин, завершуючись попередженням про те, що відкрито *надто багато* файлів. І виявляється, що за певних обставин підпрограма `write_customer()` не викликається. І коли це відбувається, файл не закривається.

Вельми *невдалим* рішенням даної задачі була б спроба реалізувати особливий випадок у підпрограмі `update_customer()`:

```
def update_customer(transaction_amount)
    read_customer
    if (transaction_amount >= 0.00)
        @balance += BigDecimal(transaction_amount, 2)
        write_customer
    else
        @customer_file.close # Погане рішення!
    end
end
```

І хоча таке рішення усуває ускладнення, оскільки файл тепер буде закрито незалежно від нового балансу на рахунку, тим не менше, тепер дане виправлення означає, що *всі* три підпрограми зв'язані спільною змінною `customer_file`, а слідкування за тим, чи відкритий файл чи закритий, починає перетворюватися на повний безлад. В результаті наш програміст потрапляє до пастки, і вся справа починає швидко йти шкереберть, якщо продовжити її у тому ж дусі. Це і є ознака незбалансованості!

Порада "Завершуйте те, що почали", наказує, що в ідеальному випадку підпрограма, яка виділяє ресурс, повинна його звільнити. Щоб скористатися цією порадою, реорганізуємо трохи код, що тут розглядається, як показано нижче.

```
def read_customer(file)
    @balance=BigDecimal(file.gets)
end

def write_customer(file)
    file.rewind
    file.puts @balance.to_s
end

def update_customer(transaction_amount)
    file=File.open(@name + ".rec", "r+")              # >--
    read_customer(file)                               #    |
    @balance = @balance.add(transaction_amount,2)     #    |
    file.close                                        # <--
end
```

Замість того, щоб зберігати посилання на файл, ми змінили у даному випадку початковий код таким чином, щоб передавати посилання у якості параметру[5]. І тепер вся відповідальність за маніпулювання файлом покладається на підпрограму `update_customer()`, яка відкриває файл та (після завершення того, з чого вона починається) закриває його перед самим поверненням. Таким чином, у даній підпрограмі балансується користуван-

[5] Див. пораду 50 у главі 5.

ня файлом як зовнішнім ресурсом, оскільки операції його відкриття та закриття відбуваються у одному й тому самому місці, і цілком очевидно, що на кожну операцію відкриття файлу припадає відповідна операція його закриття. Крім того, завдяки реорганізації коду виводиться з обігу кепська спільна змінна.

До коду, що тут розглядається, можна внести ще одне важливе вдосконалення. У багатьох сучасних мовах строк дії ресурсу можна укласти у межі деякого замкненого блоку. Скажімо, у мові Ruby є різновид оператору `open`, що передає посилання на відкритий файл такому блоку, як показано нижче у проміжку між операторами `do` та `end`.

```
def update_customer(transaction_amount)
    File.open(@name + ".rec", "r+") do |file|          # >--
        read_customer(file)                            #    |
        @balance = @balance.add(transaction_amount,2)  #    |
        write_customer(file)                           #    |
    end                                                # <--
end
```

У даному випадку змінна `file` виходить у кінці блоку зі сфери своєї дії, і на цьому зовнішній файл закривається. А програміст позбувається необхідності пам'ятати, що слід неодмінно закрити файл і тим самим звільнити ресурс, оскільки це гарантовано буде зроблене автоматично.

Якщо ж вас охоплюють сумніви, то завжди варто скоротити сферу дії.

Порада 41 Дійте локально

Вкладене виділення ресурсів

Основний шаблон виділення ресурсів може бути розширено для тих підпрограм, яким водночас необхідний не один ресурс. І для цього є лише наступні дві пропозиції.

- Звільнити ресурси у порядку, протилежному до того, у якому їх було виділено. Таким чином ресурси не залишаться завислими, якщо один з них містить посилання на інші.
- Якщо одна і та сама низка ресурсів виділяється у різних місцях початкового коду, вони повинні виділятися у тому самому порядку. Завдяки цьому знижується імовірність взаємного блокування. Скажімо, якщо процес A зажадає ресурс 1 та готовий зажадати ре-

сурс 2, тоді як процес B уже зажадав ресурс 2 та намагається отримати ресурс 1, то обидва процеси перейдуть до стану безкінечного очікування.

Незалежно від того, якого роду ресурси використовуються, хай це будуть транзакції, мережеві з'єднання, оперативна пам'ять, файли, потоки виконання чи вікна, застосовується такий основний шаблон: той, хто виділяє ресурс, повинен відповідати за його звільнення. Втім, в деяких мовах цей принцип може бути розвинуто далі.

Об'єкти та винятки

Баланс між виділенням та звільненням ресурсів нагадує конструктор та деструктор класу у об'єктно-орієнтованому програмуванні. Зокрема, клас являє собою ресурс, конструктор являє собою виділення конкретного об'єкту даного ресурсу, а деструктор видаляє його зі сфери видимості.

Якщо ви програмуєте об'єктно-орієнтованою мовою, то можете виявити, що для вас є зручною інкапсуляція ресурсів у класах. Щоразу, коли вам знадобиться ресурс конкретного типу, ви отримуєте екземпляр об'єкту даного класу. А коли об'єкт виходить зі сфери своєї дії чи утилізується збірником "сміття", деструктор цього об'єкту звільняє ув'язнений у ньому ресурс. Такий підхід дає особливі переваги при програмуванні тими мовами, де винятки можуть заважати звільненню ресурсів.

Баланс у часі

У даній темі розглядаються загалом ефемерні ресурси, що використовуються для виконання певного процесу. Але вам, можливо, доведеться розглянути й інші складні питання, які могли бути залишені поза увагою.

Як, наприклад, обробляти журнальні файли? Скажімо, якщо ви, формуючи дані, вичерпаєте все вільне місце на запам'ятовуючому пристрої, то чи слід замість цього організувати ротацію та очищення журнальних файлів? А як щодо видалення неофіційних файлів налагодження? Якщо ви вводите протокольні записи до бази даних, то чи є аналогічний процес, що визначає закінчення строку їхньої дії? Щодо всього, що ви створюєте і що потребує скінченного ресурсу, думайте, як це збалансувати.

І нарешті, чи не залишилось ще щось поза вашою увагою?

Баланс винятків

У мовах, що підтримують винятки, звільнення ресурсів може бути ускладнене. Скажімо, якщо генерується виняток, то як гарантувати очищення всього, що було виділено до цього винятку? Відповідь на це питання залежить від підтримки цієї можливості мовою. Як правило, для цього є дві можливості.

1. Скористатися сферою видимості змінної (наприклад, стековими змінними у C++ чи Rust).
2. Скористатися оператором `finally` у блоці операторів `try/catch`.

За звичайними правилами дотримання сфери видимості у таких мовах, як C++ чи Rust, пам'ять, що виділяється для змінної, буде утилізовано, щойно змінна вийде зі сфери своєї видимості шляхом повернення з функції, завершення блоку коду чи винятку. Але для очищення будь-яких зовнішніх ресурсів можна також здійснити прив'язку до деструктору змінної. У наведеному нижче прикладі коду мовою Rust файл автоматично закривається, щойно змінна `accounts` вийде зі сфери своєї видимості.

```
{
    let mut accounts = File::open("mydata.txt")?;      // >--
    // Користуємося змінною 'accounts'                 //    |
    ..                                                 //    |
}                                                      // <--
// Тепер змінна 'accounts' перебуває поза сферою
// своєї видимості, а файл автоматично закривається
```

Інша можливість, якщо тільки вона підтримується у конкретній мові, полягає у застосуванні оператору `finally`, як показано нижче. Цей оператор гарантує, що вказаний код буде виконано незалежно від того, чи буде згенеровано виняток у блоці операторів `try/catch`.

```
try
    // щось що не викликає довіри
catch
    // згенеровано виняток
finally
    // очистити у будь-якому випадку
```

Але тут є одна хитрість.

Антишаблон винятків

Нам часто доводилося спостерігати, як програмісти пишуть такий код:

```
begin
    thing = allocate_resource()
```

```
    process(thing)
finally
    deallocate(thing)
end
```

Чи можете ви побачити помилку у цьому коді? Що, як виділити ресурс не вдасться та виникне виняток? Він буде перехоплений у операторі `finally`, який спробує звільнити ресурс `thing`, який не було виділено.

Правильний шаблон для звільнення ресурсів у середовищі з винятками виглядає так:

```
thing = allocate_resource()
begin
  process(thing)
finally
  deallocate(thing)
end
```

Коли не можна збалансувати ресурси

Іноді основний шаблон для виділення ресурсів виявляється просто непридатним. І зазвичай це буває у тих програмах, у яких застосовуються динамічні структури даних. Скажімо, одна підпрограма може виділити сферу пам'яті та зв'язати її з якою-небудь більш великою структурою, де вона може залишатися деякий час.

Вихід з цього становища полягає у тому, щоб встановити семантичний інваріант для виділення пам'яті. При цьому необхідно вирішити, хто повинен відповідати за дані у агрегованій структурі даних. Що відбудеться, якщо звільнити структуру верхнього рівня? Для цього є три основні можливості.

- Структура верхнього рівня відповідає за звільнення будь-яких підструктур, які вона містить. З цих структур далі видаляються дані, що у них перебувають, тощо.
- Структура верхнього рівня просто звільняється. А будь-які структури, на які вона вказує та на які ніде більше не робляться посилання, "зависають".
- Структура верхнього рівня відмовляється звільнятися, якщо вона містить будь-які підструктури.

Вибір конкретної можливості залежить від обставин кожної структури даних окремо. Тим не менше її необхідно висловити явно для кожної структури даних та узгоджено реалізувати своє рішення. Реалізація будь-

якої з цих можливостей у таких процедурних мовах, як C, може виявитися складною, оскільки самі структури даних неактивні. У таких випадках ми воліємо написати для кожної з основних структур даних модуль, що забезпечує стандартне виділення та звільнення ресурсів, яких потребує дана структура. (Такий модуль може також надати такі ресурси, як виведення на друк за налагодження, серіалізація та десеріалізація, а також перехоплювачі обходу.)

Перевірка балансу

Програмісти-прагматики нікому не довіряють, в тому числі і самим собі, і тому ми вважаємо, що завжди варто писати такий код, у якому перевіряється, чи дійсно ресурси звільнено належним чином. У більшості застосунків це, як правило, означає створення оболонок для кожного типу ресурсу, а також їхнє застосування для відслідковування всіх операцій виділення та звільнення ресурсів. У деяких місцях логіка програми буде диктувати перебування ресурсів у певному стані, тому для його перевірки слід скористатися оболонками. Наприклад, у верхній частині головного циклу обробки даних у програмі, що довго виконується та обслуговує запити, імовірно, є єдина точка, де очікується надходження чергового запиту. Це зручне місце, де можна гарантувати, що використання ресурсу не зросло з моменту останнього виконання даного циклу. На більш низькому, хоча й менш корисному рівні можна вкласти кошти у інструментальні засоби, які, між іншим, перевіряють програми, що виконуються, на наявність витоку пам'яті.

Інші розділи, пов'язані з цією темою

- **Тема 24.** Мертві програми не брешуть.
- **Тема 30.** Перетворювальне програмування, **глава 5** "Гнучкість чи ламкість".
- **Тема 33.** Розривання часового зв'язування, **глава 6** "Паралельність".

Завдання

- За відсутності способів, які завжди гарантують звільнення ресурсів, може допомогти послідовне застосування деяких методик. Раніше у цьому розділі пояснювалося, яким чином встановлення семантичного інваріанту для основних структур даних може призвести

до правильних рішень з приводу звільнення оперативної пам'яті. З'ясуйте, як це робиться. Для цього вам, можливо, доведеться повернутися до матеріалу розділу "Тема 23. Проєктування за контрактом".

Вправи

17. Дехто з тих, хто програмує на C та C++, як правило, встановлюють значення вказівника таким, що дорівнює `NULL`, після звільнення сфери пам'яті, на яку він посилається. Чим ця ідея хороша?
18. Дехто з тих, хто програмує на Java, встановлюють значення `NULL` для об'єктної змінної після завершення використання об'єкту. Чим ця ідея хороша?

Тема 27 Не випереджайте світло фар свого автомобілю

Важко робити передбачення,
особливо — про майбутнє.

Лоуренс "Йогі" Берра[6], з датського прислів'я

Уявіть темну дощову ніч. Двомісний автомобіль різко розвертається на крутих поворотах вузької звивистої гірської дороги, ледве вписуючись у них. І раптом під колесо автомобіля потрапляє шпилька для волосся, в результаті чого він вдаряється у слабке дорожнє загородження, стрімко падає униз та розбивається вщент об землю у долині. Працівники місцевої поліції прибувають на місце пригоди, оглядають машину, і старший офіцер, сумно хитаючи головою, говорить: "Видно, випередили світло фар свого автомобілю".

Невже двомісний автомобіль міг їхати швидше за світло? Ні, звісно, оскільки перевищити цю межу швидкості не дозволяють закони фізики. Офіцер мав на увазі здатність водія керувати машиною та вчасно зупинитися, реагуючи на дорожню обстановку, яка освітлюється світлом фар.

Світло фар розповсюджується у обмежених рамках, які називаються *проєкційною відстанню*, а далі воно стає надто розсіяним, щоб ефективно освітлювати дорогу. Крім того, фари проєктують світло по прямій осьовій лінії, нічого не освітлюючи за її межами, в тому числі повороти, підйоми

[6] Lawrence "Yogi" Berra — відомий американський бейсболіст. — *Приміт. перекл.*

та западини на дорозі. Згідно даних Національного управління безпекою руху на трасах (National Highway Traffic Safety Administration — NHTSA), середня відстань, яку освітлюють фари ближнього світла, складає близько 50 м. Але, на жаль, безпечний гальмівний шлях на швидкості 85 км/год складає 57 м, тоді як на швидкості 110 км/год — вже близько 140 м[7]. Отже, випередити світло фар свого автомобіля дуже легко.

У розробці програмного забезпечення дальність світла наших "передніх фар" настільки ж обмежена. Ми не можемо зазирнути надто далеко у майбутнє, і щодалі ми відхиляємо свій погляд від основної осі, то темніше стає. Тому програміст-прагматик повинен дотримуватися такого твердого правила:

Порада 42 Завжди робіть невеликі кроки

Намагайтесь завжди робити невеликі, помірковані кроки, перевіряючи реакцію у відповідь та коректуючи свої дії, перш ніж продовжити далі. І в жодному разі не робіть надто великий крок та не беріться за надто велику задачу.

А що конкретно мається на увазі під реакцією у відповідь? Все, що незалежним чином підтверджує чи спростовує вашу дію. Наприклад:

- результати у циклі "читання — обчислення — висновок" (REPL) забезпечують реакцію у відповідь на ваше розуміння API та алгоритмів;
- модульні тести забезпечують реакцію у відповідь на останню зміну коду;
- демонстрація користувачам та її обговорення забезпечують реакцію у відповідь на функціональні засоби, зручність та простоту користування.

А яку задачу слід вважати надто великою? Будь-яку, що потребує "передбачення майбутнього". Подібно до обмеженої дальності світла передніх фар автомобіля, ми можемо також зазирнути у майбутнє, можливо, на один чи два кроки, а, можливо, найбільше — на кілька годин чи днів. Вийшовши за ці межі, можна дуже швидко перейти від *обґрунтованого*

[7] Відповідно до такої формули NHTSA: безпечний гальмівний шлях — це відстань, яку проходить автомобіль за час реакції водія + шлях гальмування, за умови, що середній час реакції складає 1,5 с, а прискорення — 5,19 м/с^2.

припущення до *пустих домислів*. Легко впасти у гріх передбачення майбутнього, коли доводиться:

- оцінювати дати завершення робіт на місяці вперед;
- планувати розширюваність супроводу у перспективі;
- передбачати майбутні потреби користувачів;
- прогнозувати технічну придатність у майбутньому.

Але ми вже чуємо ваші голосні заперечення: "Хіба ми не проєктуємо з урахуванням майбутнього супроводу?" Так, звісно, але лише до певного моменту, тобто, настільки, наскільки ми в змозі дивитися вперед. Щобільше вам доводиться передбачати майбутнє, то більший ризик, що ви помилитесь. І замість того, щоб витрачати зусилля на проєктування для невизначеного майбутнього, можна завжди вдатися до розробки замінного коду. Намагайтесь полегшити відмову від непридатного коду та його заміну на більш придатний код. Можливість замінити код допомагає також його зціпленню, зв'язуванню та дотриманню принципу DRY, що призводить до більш якісного проєктування. І хоча ви можете бути впевнені у майбутньому, завжди існує імовірність, що не за горами з'явиться щось зовсім негадане, ніби чорний лебідь.

Чорні лебеді

У своїй книзі "Чорний лебідь: про (не)ймовірне у реальному житті" [Tal10] Нассім Ніколас Талеб стверджує, що всі значущі події в історії походили від резонансних, важко передбачуваних та рідкісних подій, що виходять за межі звичайних очікувань. Ці неймовірні події, попри їхню статистичну рідкісність, мають неспівставний з нею вплив. Крім того, наші когнітивні викривлення призводять до того, що ми не помічаємо, як зміни потихеньку проникають до нашої роботи, доходячи до країв (див. розділ "Тема 4. Суп із каменів та варені жаби" глави 1 "Філософія прагматизму").

На час виходу у світ першого видання цієї книги у комп'ютерних журналах та доступних онлайн форумах розгорілася полеміка з такого питання: "Хто переможе у війнах настільних GUI між стилями Motif та OpenLook[8]?" Це питання було поставлене невірно. Цілком можливо, що ви взагалі не чули про ці технології, оскільки жодна з них не перемогла, і у цій галузі швидко переважила вебтехнологія, орієнтована на браузери.

[8] Motif та OpenLook були стандартними графічними інтерфейсами для системи X-Windows на робочих станціях Unix.

Порада 43 Уникайте передбачення майбутнього

Завтра часто-густо виглядає дуже схожим на сьогодні. Але на це не слід особливо покладатися.

Інші розділи, пов'язані з цією темою

- **Тема 12.** Трасуючі кулі, **глава 2** "Прагматичний підхід".
- **Тема 13.** Прототипи та пам'ятні нотатки, **глава 2** "Прагматичний підхід".
- **Тема 40.** Рефакторинг, **глава 7** "По ходу кодування".
- **Тема 41.** Тестувати, щоб кодувати, **глава 7** "По ходу кодування".
- **Тема 48.** Сутність гнучкості, **глава 8** "До початку проєкту".
- **Тема 50.** Кокосами не обійтися, **глава 9** "Прагматичні проєкти".