O'REILLY®

Reguły programowania

JAK PISAĆ LEPSZY KOD

CHRIS ZIMMERMAN

Helion

Tytuł oryginału: The Rules of Programming: How to Write Better Code

Tłumaczenie: Piotr Rajca

ISBN: 978-83-289-0130-8

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/regpro
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

Tak proste, jak to możliwe, lecz nie prostsze

Programowanie jest trudne.

Przypuszczam jednak, że tego sam się już dowiedziałeś. Każdy, kto kupił i czyta książkę *Reguły programowania*, najprawdopodobniej spełnia dwa warunki:

- urodził się, by programować, przynajmniej w pewnym stopniu;
- jest sfrustrowany tym, że programowanie nie jest prostsze, niż jest w rzeczywistości.

Istnieje wiele powodów tego, że programowanie jest trudne, oraz wiele strategii, które można zastosować, aby je ułatwić. W tej książce przyjrzymy się starannie wybranym sposobom, w jakie programiści pakują się w kłopoty, i zasadom unikania tych błędów, a wszystkie te informacje bazują na moich wieloletnich doświadczeniach w popełnianiu własnych błędów i kopiowaniu błędów innych.

Istnieje pewien ogólny wzorzec „reguł", wspólny temat, który większość z nich dzieli. Najlepiej podsumowuje go cytat z Alberta Einsteina opisujący cele teoretyka fizyki: „Tak proste, jak to możliwe, lecz nie prostsze"[1]. Einstein miał przez to na myśli, że najlepsza teoria fizyczna to ta, która jest najprostsza i całkowicie opisuje wszystkie obserwowalne zjawiska.

Przenosząc tę ideę na grunt programowania, najlepszym sposobem wdrożenia rozwiązania dowolnego problemu jest ten najprostszy, który spełnia wszystkie wymagania tego problemu. Najlepszy kod to najprostszy kod.

[1] Einstein niemal na pewno nie użył dokładnie tych słów — potomność wyświadczyła Einsteinowi przysługę, poprawiając jego aforyzmy. Najbliższy zapisany odpowiednik to: „Trudno zaprzeczyć, że najwyższym celem wszelkiej teorii jest uczynienie nieredukowalnych elementów podstawowych tak prostymi i tak nielicznymi, jak to tylko możliwe, jednak bez konieczności rezygnacji z adekwatnego odwzorowania choćby jednego elementu doświadczenia". Czyli w zasadzie to samo, tylko nie aż tak zwięźle. Co więcej, faktyczny cytat Einsteina jest trochę za długi jak na tytuł reguły i tego rozdziału.

Wyobraź sobie, że piszesz kod, aby policzyć liczbę ustawionych bitów w liczbie całkowitej. Istnieje wiele sposobów, by to zrobić. Możesz użyć sztuczek bitowych[2], aby zerować kolejno poszczególne bity, i liczyć, ile bitów zostało wyzerowanych:

```
int countSetBits(int value)
{
    int count = 0;

    while (value)
    {
        ++count;
        value = value & (value - 1);
    }

    return count;
}
```

Ewentualnie możesz zdecydować się na zastosowanie implementacji bez pętli, która zlicza ustawione bity, wykorzystując ich jednoczesne przesuwanie i maskowanie:

```
int countSetBits(int value)
{
    value = ((value & 0xaaaaaaaa) >> 1) + (value & 0x55555555);
    value = ((value & 0xcccccccc) >> 2) + (value & 0x33333333);
    value = ((value & 0xf0f0f0f0) >> 4) + (value & 0x0f0f0f0f);
    value = ((value & 0xff00ff00) >> 8) + (value & 0x00ff00ff);
    value = ((value & 0xffff0000) >> 16) + (value & 0x0000ffff);

    return value;
}
```

W końcu możesz po prostu napisać najbardziej oczywisty z możliwych kodów:

```
int countSetBits(int value)
{
    int count = 0;

    for (int bit = 0; bit < 32; ++bit)
    {
        if (value & (1 << bit))
            ++count;
    }

    return count;
}
```

Pierwsze dwie odpowiedzi są sprytne… i bynajmniej nie jest to komplement[3]. Szybki rzut oka nie wystarczy, aby zorientować się, jak te przykłady działają — każdy z nich ma mały fragment kodu, który u większości wywoła reakcję „czekaj… co?”. Przy odrobinie wysiłku intelektualnego

[2] Przepraszam wszystkich Czytelników, którzy nie używają języka C++, za to, że w kolejnych trzech przykładach będziemy się bawić bitami. Obiecuję, że w dalszej części książki operacji na bitach już nie będzie.

[3] W wiarygodnym alternatywnym wszechświecie ta reguła nosi nazwę „spryt nie jest cnotą”.

można się zorientować, co się dzieje, a zastosowanie sztuczki będzie całkiem zabawne. Ale zrozumienie tego wszystkiego wymaga trochę wysiłku.

Mocny początek! Powiedziałem Ci, co robią funkcje przed pokazaniem kodu, a ich nazwy dodatkowo to potwierdzają i określają. Gdybyś nie wiedział, że kod liczy ustawione bity, rozgryzienie któregokolwiek z dwóch pierwszych przykładów byłoby jeszcze bardziej pracochłonne.

W przypadku trzeciej, ostatniej odpowiedzi sprawa wygląda zupełnie inaczej. Oczywistym jest, że funkcja liczy ustawione bity. Jest tak prosta, jak to możliwe, lecz nie prostsza, i to czyni ją lepszą od dwóch wcześniejszych[4].

Pomiar prostoty

Istnieje wiele opinii na temat tego, co czyni kod prostym.

Możesz się zdecydować, by mierzyć prostotę w oparciu o to, jak łatwo inna osoba może zrozumieć kod. Jeśli losowo wybrany kolega może przeczytać kawałek kodu i zrozumieć go bez wysiłku, to kod jest odpowiednio prosty.

Możesz się też zdecydować na zmierzenie prostoty w oparciu o łatwość tworzenia kodu — nie tylko czasu potrzebnego do jego wpisania, ale także czasu potrzebnego na uzyskanie kodu w pełni funkcjonalnego i wolnego od błędów[5]. Przygotowanie skomplikowanego kodu wymaga czasu; tworzenie prostego kodu jest znacznie łatwiejsze.

Oczywiście te dwie miary w dużym stopniu się pokrywają. Kod, który jest łatwy do napisania, zwykle jest również łatwy do odczytania. Istnieją też inne ważne miary złożoności kodu, których możesz użyć:

Ilość napisanego kodu

Prostszy kod jest zazwyczaj krótszy, choć możliwe jest zmieszczenie dużej złożoności w jednym wierszu kodu.

Liczba wprowadzanych pomysłów

Prosty kod zazwyczaj bazuje na koncepcjach, które wszyscy w zespole znają; nie wprowadza nowych sposobów myślenia o problemach ani żadnej nowej terminologii.

[4] Nowoczesne procesory udostępniają specjalną instrukcję przeznaczoną do liczenia bitów ustawionych w wartości — na przykład `popcnt` w procesorach x86, której wykonanie zajmuje jeden cykl procesora. Można również skorzystać z instrukcji SIMD, aby policzyć wiele bitów jeszcze szybciej niż przy użyciu instrukcji `popcnt`. Ale wszystkie te podejścia są trudne do zrozumienia, a dostępne instrukcje zależą od używanego procesora. Ja wolałbym zobaczyć najprostszą wersję funkcji `countSetBits`, chyba że byłby naprawdę, ale to naprawdę dobry powód, aby użyć jakiegoś bardziej skomplikowanego rozwiązania.

[5] Oczywiście wolnego od błędów w granicach błędu eksperymentalnego. Zawsze są błędy, których jeszcze nie znalazłeś.

Ile czasu zajmuje wyjaśnienie kodu

> Prosty kod jest łatwy do wytłumaczenia — w trakcie przeglądu kodu jest on na tyle oczywisty, że recenzent może jedynie rzucić na niego okiem. Skomplikowany kod wymaga wyjaśnień.

Kod, który wydaje się prosty według jednej miary, będzie prosty również według innych. Musisz tylko wybrać, która z tych miar w największym stopniu koncentruje się na Twojej pracy — osobiście sugeruję jednak, by zaczynać od łatwości tworzenia kodu i łatwości jego zrozumienia. Jeśli skupisz się na szybkim przygotowaniu kodu łatwego do odczytania, to będziesz tworzył prosty kod.

…ale nie prostszy

Lepiej, aby kod był prostszy, ale wciąż musi on rozwiązać problem, który zamierza rozwiązać.

Wyobraź sobie, że próbujesz policzyć, na ile sposobów można wejść na drabinę mającą pewną liczbę szczebli, przy założeniu, że podczas jednego kroku wspinasz się o jeden, dwa lub trzy szczeble. Jeśli drabina ma dwa szczeble, to można na nią wejść na dwa sposoby — albo wejdziesz na pierwszy szczebel, albo nie. Podobnie na drabinę mającą trzy szczeble można wejść na cztery sposoby — wejść na pierwszy szczebel, wejść na drugi szczebel, wejść na pierwszy i drugi szczebel lub wejść bezpośrednio na najwyższy szczebel. Na drabinę o czterech szczeblach można wejść na siedem sposobów, na taką z pięcioma szczeblami — na trzynaście sposobów itd.

Można napisać prosty kod, który będzie to obliczał rekurencyjnie:

```
int countStepPatterns(int stepCount)
{
    if (stepCount < 0)
        return 0;

    if (stepCount == 0)
        return 1;

    return countStepPatterns(stepCount - 3) +
           countStepPatterns(stepCount - 2) +
           countStepPatterns(stepCount - 1);
}
```

Podstawowa idea działania tej funkcji polega na tym, że każda wspinaczka po drabinie musi prowadzić na najwyższy szczebel z jednego z trzech szczebli umieszczonych poniżej. Dodanie liczby sposobów wejścia na każdy z tych szczebli daje liczbę sposobów wejścia na najwyższy szczebel. Potem pozostaje już tylko kwestia określenia przypadków bazowych. Poprzedni kod dopuszcza ujemne liczby kroków jako przypadek bazowy, aby uprościć rekurencję.

Niestety to rozwiązanie nie działa. To znaczy — w zasadzie działa, przynajmniej dla małych wartości parametru `stepCount`, ale wykonanie wywołania `countStepPatterns(20)` trwa niemal dwa razy dłużej niż wykonanie wywołania `countStepPatterns(19)`. Komputery są naprawdę szybkie,

lecz wzrost wykładniczy, taki jak ten, dogoni tę prędkość. W moim teście przykładowy kod zaczął się robić dość powolny, gdy wartość `stepCount` przekroczyła 20.

Jeśli będziesz musiał policzyć liczbę sposobów wejścia na dłuższą drabinę, ten kod okaże się zbyt prosty. Podstawowym problemem jest to, że wszystkie pośrednie wyniki wywołania `countStepPatterns` są wielokrotnie przeliczane, a to prowadzi do wykładniczych czasów wykonania. Standardowym rozwiązaniem tego problemu jest zastosowanie memoizacji — zapamiętywanie obliczonych wartości pośrednich i ponowne ich stosowanie, jak w poniższym przykładzie:

```
int countStepPatterns(unordered_map<int, int> * memo, int rungCount)
{
    if (rungCount < 0)
        return 0;

    if (rungCount == 0)
        return 1;

    auto iter = memo->find(rungCount);
    if (iter != memo->end())
        return iter->second;

    int stepPatternCount = countStepPatterns(memo, rungCount - 3) +
                           countStepPatterns(memo, rungCount - 2) +
                           countStepPatterns(memo, rungCount - 1);

    memo->insert({ rungCount, stepPatternCount });
    return stepPatternCount;
}

int countStepPatterns(int rungCount)
{
    unordered_map<int, int> memo;
    return countStepPatterns(&memo, rungCount);
}
```

Po zaimplementowaniu memoizacji każda wartość jest obliczana tylko raz i wstawiana do mapy — kontenera `unordered_map`. Kolejne wywołania funkcji `countStepPatterns` znajdują obliczoną wartość w mapie w mniej więcej stałym czasie, dzięki czemu wykładniczy wzrost złożoności zostaje wyeliminowany. Kod korzystający z techniki memoizacji jest nieco bardziej skomplikowany, lecz wydajność nie stanowi dla niego problemu.

Możesz również zdecydować się na wykorzystanie techniki programowania dynamicznego, by kosztem większej złożoności koncepcyjnej uzyskać prostszy kod:

```
int countStepPatterns(int rungCount)
{
    vector<int> stepPatternCounts = { 0, 0, 1 };

    for (int rungIndex = 0; rungIndex < rungCount; ++rungIndex)
    {
        stepPatternCounts.push_back(
            stepPatternCounts[rungIndex + 0] +
```

```
            stepPatternCounts[rungIndex + 1] +
            stepPatternCounts[rungIndex + 2]);
    }

    return stepPatternCounts.back();
}
```

To rozwiązanie także działa dostatecznie szybko, a jest jeszcze prostsze od wcześniejszej rekurencyjnej wersji funkcji korzystającej z memoizacji.

Czasami lepiej jest uprościć problem niż rozwiązanie

Problemy w początkowej rekurencyjnej wersji funkcji `countStepPatterns` pojawiały się dla dłuższych drabin. Najprostszy kod działał doskonale dla niewielkiej liczby szczebli, ale zderzał się z wykładniczą ścianą wydajności, gdy szczebli było dużo. Kolejne wersje funkcji omijały problem wydajności kosztem nieco większej złożoności… ale wkrótce stanęły w obliczu innego problemu.

Jeśli uruchomię poprzedni kod, by obliczyć wywołanie `countStepPatterns(36)`, otrzymam prawidłową odpowiedź, 2 082 876 103. Jednak wywołanie `countStepPatterns(37)` zwraca wartość –463 960 867, co jest oczywistym błędem!

Dzieje się tak dlatego, że wersja C++, której używam, przechowuje liczby całkowite jako wartości 32-bitowe ze znakiem, a podczas obliczania `countStepPatterns(37)` nastąpiło przepełnienie. Istnieje 3 831 006 429 sposobów wejścia na drabinę mającą 37 szczebli, a ta wartość jest zbyt duża, aby zapisać ją w 32-bitowej liczbie całkowitej ze znakiem.

Może zatem nasz kod wciąż jest zbyt prosty? Wydaje się, że całkiem rozsądne jest założenie, że funkcja `countStepPatterns` będzie działać prawidłowo dla drabin o dowolnej długości, prawda? Język C++ nie ma standardowego rozwiązania operowania na naprawdę dużych liczbach całkowitych, ale istnieje (i to wiele) bibliotek *open source*, które implementują różne rodzaje liczb całkowitych o dowolnej precyzji. Ewentualnie, kosztem kilkuset wierszy kodu, mógłbyś zaimplementować własne rozwiązanie:

```
struct Ordinal
{
public:

    Ordinal() :
        m_words()
        { ; }
    Ordinal(unsigned int value) :
        m_words({ value })
        { ; }

    typedef unsigned int Word;

    Ordinal operator + (const Ordinal & value) const
    {
        int wordCount = max(m_words.size(), value.m_words.size());
```

```
        Ordinal result;
        long long carry = 0;

        for (int wordIndex = 0; wordIndex < wordCount; ++wordIndex)
        {
            long long sum = carry +
                            getWord(wordIndex) +
                            value.getWord(wordIndex);

            result.m_words.push_back(Word(sum));
            carry = sum >> 32;
        }

        if (carry > 0)
            result.m_words.push_back(Word(carry));

        return result;
    }

protected:
    Word getWord(int wordIndex) const
    {
        return (wordIndex < m_words.size()) ? m_words[wordIndex] : 0;
    }

    vector<Word> m_words;
};
```

Zastosowanie struktury `Ordinal` w poprzednim przykładzie w miejsce zmiennej typu `int` pozwoli uzyskiwać prawidłowe wyniki dla drabin o dowolnej długości:

```
Ordinal countStepPatterns(int rungCount)
{
    vector<Ordinal> stepPatternCounts = { 0, 0, 1 };

    for (int rungIndex = 0; rungIndex < rungCount; ++rungIndex)
    {
        stepPatternCounts.push_back(
            stepPatternCounts[rungIndex + 0] +
            stepPatternCounts[rungIndex + 1] +
            stepPatternCounts[rungIndex + 2]);
    }

    return stepPatternCounts.back();
}
```

Czyli co… problem rozwiązany? Dzięki wprowadzeniu typu `Ordinal` możemy obliczać prawidłowe odpowiedzi dla znacznie dłuższych drabin. Jasne, dodanie kilkuset wierszy kodu w celu zaimplementowania struktury `Ordinal` nie jest optymalne, zwłaszcza biorąc pod uwagę fakt, że rzeczywista funkcja `countStepPatterns` liczy tylko 14 wierszy, ale czy nie jest to cena, którą trzeba zapłacić za poprawne rozwiązanie problemu?

Prawdopodobnie nie. Jeśli nie ma prostego rozwiązania problemu, warto go dokładnie przeanalizować, zanim zaakceptujemy skomplikowane rozwiązanie. Czy problem, który próbujesz rozwiązać, jest rzeczywiście tym, który wymaga rozwiązania? A może przyjmujesz niepotrzebne założenia, które komplikują Twoje rozwiązanie?

W tym przypadku, jeśli faktycznie liczysz możliwe sposoby wejścia na prawdziwą drabinę, prawdopodobnie możesz przyjąć jakąś jej maksymalną długość. Jeśli maksymalna długość drabiny wynosi, powiedzmy, 15 szczebli, to każde z rozwiązań przedstawionych w tym rozdziale będzie bardzo dobre, nawet naiwny przykład rekurencyjny przedstawiony jako pierwszy. Wystarczy, że dodasz do jednego z nich wywołanie `assert`, określając w ten sposób wbudowane ograniczenie funkcji, i możesz ogłosić zwycięstwo:

```
int countStepPatterns(int rungCount)
{
    // UWAGA (chris) przy użyciu wartości typu int nie można reprezentować wzorca wejść
    // na drabinę, jeśli będzie ona miała więcej niż 36 szczebli…

    assert(rungCount <= 36);

    vector<int> stepPatternCounts = { 0, 0, 1 };

    for (int rungIndex = 0; rungIndex < rungCount; ++rungIndex)
    {
        stepPatternCounts.push_back(
            stepPatternCounts[rungIndex + 0] +
            stepPatternCounts[rungIndex + 1] +
            stepPatternCounts[rungIndex + 2]);
    }

    return stepPatternCounts.back();
}
```

Ewentualnie, jeśli wymagana jest obsługa naprawdę długich drabin — powiedzmy takich jak drabiny do inspekcji turbin wiatrowych — to czy wystarczyłoby uwzględnienie przybliżonej liczby kroków? Prawdopodobnie, a jeśli tak, to łatwo można by zastąpić liczby całkowite wartościami zmiennoprzecinkowymi. Tak łatwo, że nawet nie zamierzam pokazywać kodu takiego rozwiązania.

Popatrz, zawsze można doprowadzić do przepełnienia. Rozwiązywanie skrajnych przypadków brzegowych dla problemu zawsze doprowadzi do zbyt skomplikowanego rozwiązania. Nie daj się złapać w pułapkę rozwiązywania problemu zdefiniowanego w najbardziej ścisły sposób. O wiele lepiej jest mieć proste rozwiązanie dla części problemu, która faktycznie wymaga rozwiązania, zamiast skomplikowanego rozwiązania dla szerszej definicji problemu. Jeśli nie możesz uprościć rozwiązania, spróbuj uprościć problem.

Proste algorytmy

Czasami to zły wybór algorytmu zwiększa złożoność Twojego kodu. W końcu każdy problem można rozwiązać na wiele sposobów, a niektóre z nich są bardziej skomplikowane niż inne. Proste algorytmy prowadzą do prostego kodu. Problem w tym, że prosty algorytm nie zawsze jest oczywisty!

Załóżmy, że piszesz kod mający posortować talię kart. Oczywistym podejściem będzie zasymulowanie metody, której prawdopodobnie nauczyłeś się jako dziecko — podzielenia talii na dwa stosy, a następnie lekkiego ich wygięcia, nasunięcia nieznacznie obu stosów na siebie i jednoczesnego puszczania kart z obu stosów, tak by miały one mniej więcej równe szanse, aby w połączonej talii znaleźć się na następnym miejscu; te czynności trzeba kilkukrotnie powtórzyć[6].

Implementacja takiego tasowania mogłaby wyglądać następująco:

```
vector<Card> shuffleOnce(const vector<Card> & cards)
{
    vector<Card> shuffledCards;

    int splitIndex = cards.size() / 2;
    int leftIndex = 0;
    int rightIndex = splitIndex;

    while (true)
    {
        if (leftIndex >= splitIndex)
        {
            for (; rightIndex < cards.size(); ++rightIndex)
                shuffledCards.push_back(cards[rightIndex]);

            break;
        }
        else if (rightIndex >= cards.size())
        {
            for (; leftIndex < splitIndex; ++leftIndex)
                shuffledCards.push_back(cards[leftIndex]);

            break;
        }
        else if (rand() & 1)
        {
            shuffledCards.push_back(cards[rightIndex]);
            ++rightIndex;
        }
        else
        {
```

[6] Talia zostanie w miarę dobrze losowo potasowana po 7 powtórzeniach takiej procedury. Po czterech czy pięciu powtórzeniach karty w ogóle nie będą jeszcze ułożone losowo. Owszem, moja rodzina się denerwuje, gdy czeka, aż wielokrotnie potasuję karty przed kolejnym rozdaniem. „Chcemy grać w karty, Chris, a nie patrzeć, jak tasujesz". Wiedza to niebezpieczna rzecz.

```
            shuffledCards.push_back(cards[leftIndex]);
            ++leftIndex;
        }
    }

    return shuffledCards;
}

vector<Card> shuffle(const vector<Card> & cards)
{
    vector<Card> shuffledCards = cards;

    for (int i = 0; i < 7; ++i)
    {
        shuffledCards = shuffleOnce(shuffledCards);
    }

    return shuffledCards;
}
```

Ten algorytm symulowanego tasowania kart działa, a kod, który tu napisałem, jest jego dość prostą implementacją. Będziesz musiał włożyć nieco wysiłku, by upewnić się, że wszystkie sprawdzenia indeksów są prawidłowe, ale nie jest to zbyt dużym problemem.

Jednak istnieją prostsze algorytmy tasowania talii kart. Na przykład możesz przygotować potasowaną talię po jednej karcie na raz. W tym celu w każdej iteracji należy wybrać jedną nową kartę i zamienić ją z losową kartą z talii. Można to nawet zrobić, używając jednego wektora reprezentującego talię kart:

```
vector<Card> shuffle(const vector<Card> & cards)
{
    vector<Card> shuffledCards = cards;

    for (int cardIndex = shuffledCards.size(); --cardIndex >= 0; )
    {
        int swapIndex = rand() % (cardIndex + 1);
        swap(shuffledCards[swapIndex], shuffledCards[cardIndex]);
    }

    return shuffledCards;
}
```

W oparciu o wprowadzone wcześniej miary prostoty ta wersja tasowania kart jest lepsza. Napisanie jej zajęło mniej czasu[7]. Jest łatwiejsza do przeczytania i zrozumienia. Ma mniej kodu. Łatwiej ją wytłumaczyć. Jest prostsza i lepsza — nie z powodu kodu, ale z powodu lepszego wyboru algorytmu.

[7] Zmierzono to eksperymentalnie; w Twoim przypadku może to wyglądać inaczej. Ja miałem trochę zabawy z indeksami i warunkami podczas pisania pierwszej wersji sortowania, a zapewnienie jej właściwego działania wymagało wykonania kilku prób. Kod korzystający z losowego wyboru zadziałał za pierwszym razem.

Nie trać z oczu celu

Prosty kod jest łatwy do przeczytania — a najprostszy kod można przeczytać od razu, od góry do dołu, tak jak czyta się książkę. Jednak programy to nie książki. Łatwo uzyskać kod, który jest trudny do analizy i zrozumienia, jeśli przepływ sterowania w tym kodzie także nie jest prosty. Kiedy kod jest zagmatwany, kiedy zmusza cię do przeskakiwania z miejsca na miejsce, aby śledzić przepływ sterowania, znacznie trudniej jest go czytać.

Zagmatwany kod może być wynikiem zbyt usilnych prób wyrażenia każdej idei w dokładnie jednym miejscu. W ramach przykładu przyjrzyjmy się kodowi pierwszej wersji sortowania kart. Fragmenty kodu obsługujące prawy i lewy stos kart wyglądają dość podobnie. Logika przenoszenia jednej karty lub serii kart na potasowany stos mogłaby zostać podzielona na osobne funkcje, a następnie wywołana w funkcji `shuffleOnce`:

```
void copyCard(
    vector<Card> * destinationCards,
    const vector<Card> & sourceCards,
    int * sourceIndex)
{
    destinationCards->push_back(sourceCards[*sourceIndex]);
    ++(*sourceIndex);
}

void copyCards(
    vector<Card> * destinationCards,
    const vector<Card> & sourceCards,
    int * sourceIndex,
    int endIndex)
{
    while (*sourceIndex < endIndex)
    {
        copyCard(destinationCards, sourceCards, sourceIndex);
    }
}

vector<Card> shuffleOnce(const vector<Card> & cards)
{
    vector<Card> shuffledCards;

    int splitIndex = cards.size() / 2;
    int leftIndex = 0;
    int rightIndex = splitIndex;

    while (true)
    {
        if (leftIndex >= splitIndex)
        {
            copyCards(&shuffledCards, cards, &rightIndex, cards.size());
            break;
        }
        else if (rightIndex >= cards.size())
```

```
            {
                copyCards(&shuffledCards, cards, &leftIndex, splitIndex);
                break;
            }
            else if (rand() & 1)
            {
                copyCard(&shuffledCards, cards, &rightIndex);
            }
            else
            {
                copyCard(&shuffledCards, cards, &leftIndex);
            }
        }

        return shuffledCards;
    }
```

Poprzednia wersja funkcji `shuffleOnce` była czytelna od samego początku aż do końca; ta nie jest. To czyni ją trudniejszą do odczytania. Podczas czytania kodu `shuffleOnce` znajdziesz funkcję `copyCard` lub `copyCards`. Następnie musisz się dowiedzieć, co robią, wrócić do początkowej funkcji i dopasować argumenty przekazane z `shuffleOnce` do parametrów funkcji `copyCard` lub `copyCards`. Jest to o wiele trudniejsze niż czytanie pętli użytej w początkowym kodzie funkcji `shuffleOnce`.

A zatem ta nowa wersja funkcji, napisana zgodnie z zasadą „nie powtarzaj się”, nie tylko wymaga więcej czasu na zaimplementowanie[8], lecz także jest trudniejsza do odczytania. Co więcej, jej kod jest dłuższy! Próba usunięcia powielania sprawiła, że kod stał się bardziej skomplikowany, a nie prostszy.

Oczywiście eliminacja powtórzeń w kodzie zdecydowanie jest warta rozważenia! Jednak ważne jest, by zdawać sobie sprawę, że ma ona swoją cenę — w przypadku niewielkich ilości kodu i prostych koncepcji lepiej jest po prostu zostawić powtarzające się fragmenty kodu. Dzięki temu będzie on łatwiejszy zarówno do napisania, jak i do czytania.

Jedna reguła, by rządzić innymi

Wiele z pozostałych reguł w tej książce będzie powracać do przedstawionego tu tematu prostoty, utrzymywania kodu tak prostym, jak to tylko możliwe, ale nie prostszym.

W swej istocie programowanie jest walką ze złożonością. Dodawanie nowych funkcji często oznacza komplikowanie kodu — a gdy kod staje się bardziej skomplikowany, coraz trudniej się z nim pracuje, zaś postęp jest coraz wolniejszy. W końcu można dotrzeć do horyzontu zdarzeń, za którym każda próba pójścia naprzód — naprawienia jakiegoś błędu lub dodania funkcji — powoduje tyle samo problemów, ile rozwiązuje. Dalszy postęp jest praktycznie niemożliwy.

[8] To także zostało ustalone eksperymentalnie. W rzeczywistości doprowadzenie kodu do postaci, w której dało się go skompilować, wymagało kilku prób, ponieważ wahałem się między używaniem wskaźników i referencji.

W końcu to złożoność zabije Twój projekt.

Oznacza to, że skuteczne programowanie polega na opóźnianiu tego, co nieuniknione. Tworząc nowe możliwości i poprawiając błędy, staraj się wprowadzać do kodu tak mało złożoności, jak to tylko możliwe. Szukaj możliwości eliminacji złożoności lub zaprojektuj wszystko tak, aby nowe możliwości nie dodawały wiele do ogólnej złożoności systemu. Staraj się także zadbać o to, by wzajemna współpraca w ramach Twojego zespołu była możliwie jak najprostsza.

Jeśli jesteś sumienny, będziesz w stanie w nieskończoność odwlekać to, co nieuniknione. Pierwsze wiersze kodu Sucker Punch napisałem 25 lat temu, a od tego czasu baza kodu stale ewoluowała. Nie widać końca — nasz kod jest o wiele bardziej skomplikowany niż 25 lat temu, ale udało nam się zachować kontrolę nad tą złożonością i wciąż jesteśmy w stanie robić efektywne postępy.

Skoro my potrafiliśmy zarządzać złożonością, to Ty też możesz. Zachowaj czujność, pamiętaj, że złożoność jest ostatecznym wrogiem, a dobrze sobie poradzisz.

Młody programista szybko sobie uzmysławia, że opanowanie języka programowania nie oznacza umiejętności pisania dobrego kodu. Zanim się ją opanuje, trzeba spędzić wiele bezsennych nocy na próbach usunięcia błędów czy rozwiązania innych problemów. Programowanie jest po prostu trudną sztuką. Czy istnieje sposób, aby choć trochę ją ułatwić?

W książce znajdziesz inspirujące spostrzeżenia zarówno dla początkujących, jak i doświadczonych programistów!

Paul Daugherty, Group Chief Executive of Technology i CTO, Accenture

Właśnie w tym celu powstał ten przewodnik po filozofii oprogramowania. Znajdziesz w nim 21 pragmatycznych reguł, którymi kierują się najlepsi programiści. Dzięki spostrzeżeniom zawartym w książce zmienisz podejście do programowania i szybko się przekonasz, że pozwoli Ci to na pisanie lepszego, czytelniejszego i niezawodnego kodu. Poszczególne reguły zostały zilustrowane jego rzeczywistymi przykładami, ułatwiającymi zrozumienie prezentowanych treści. Ten zajmująco i zabawnie napisany przewodnik nie tylko zainspiruje Cię do programistycznego rozwoju, ale również będzie nieocenioną pomocą przy szkoleniu nowych członków zespołu.

Poznaj reguły, którymi kierują się najlepsi:

- Tak prosty, jak to możliwe, ale nie prostszy
- Pierwsza lekcja optymalizacji: nie optymalizuj
- Błędy są zaraźliwe
- Kod, który nie jest wykonywany, nie działa
- I wiele innych!

Chris Zimmerman dawniej pracował w Microsofcie. W 1997 roku założył studio gier wideo Sucker Punch Productions i przez 25 lat kierował w nim zespołem programistów, z którym zrealizował wiele udanych projektów, w tym wydaną w 2020 roku grę *Ghost of Tsushima*. Jakiś czas temu zdecydował się na częściową emeryturę, aby napisać tę książkę.

Oto świetne wskazówki dla początkujących i subtelne lekcje dla ekspertów!

Mark Cerny, Lead System Architect, PlayStation 4 i 5

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-289-0130-8
9 788328 901308
Cena: 79,00 zł
O'REILLY®