O'REILLY®

Nauka programowania opartego na testach

Jak pisać przejrzysty kod w kilku językach programowania

Saleem Siddiqui

Helion

Tytuł oryginału: Learning Test-Driven Development: A Polyglot Guide to Writing Uncluttered Code

Tłumaczenie: Lech Lachowski

ISBN: 978-83-283-9040-9

Authorized Polish translation of the English *Learning Test-Driven Development* ISBN 9781098106478

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

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

Spis treści

ROZDZIAŁ 1.

Problem z pieniędzmi

> Za nic mam prostotę po tej stronie złożoności, ale oddałbym życie za prostotę po drugiej stronie złożoności.
>
> — Oliver Wendell Holmes Jr.

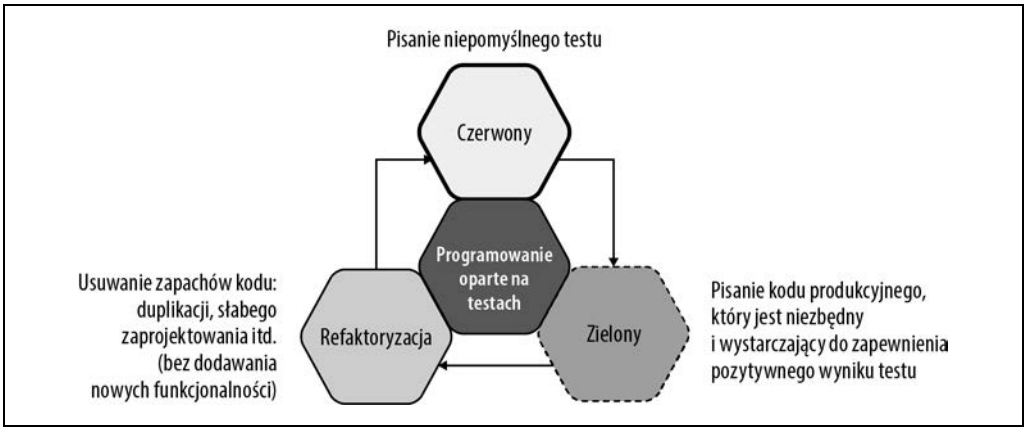
Nasze środowisko programistyczne jest gotowe. W tym rozdziale poznasz trzy fazy związane z programowaniem opartym na testach, a potem przy użyciu TDD napiszesz kod swojej pierwszej funkcjonalności.

„Czerwony, zielony, refaktoryzacja" — elementy konstrukcyjne TDD

W programowaniu opartym na testach stosuje się proces (cykl), na który składają się trzy fazy:

1. **Czerwona** (ang. *red*). Piszemy test kończący się niepowodzeniem (obejmujący możliwe niepowodzenia kompilacji). Uruchamiamy zestaw testowy, aby zweryfikować, czy testy dają negatywny wynik.
2. **Zielona** (ang. *green*). Piszemy tylko tyle kodu produkcyjnego, aby test był zielony, czyli kończył się powodzeniem. Uruchamiamy zestaw testowy w celu przeprowadzenia weryfikacji.
3. **Refaktoryzacja** (ang. *refactor*). Usuwamy wszelkie zapachy kodu. Mogą one być związane z duplikacją, zakodowanymi na stałe wartościami lub niewłaściwym użyciem idiomów językowych (np. zastosowanie rozwlekłej pętli zamiast wbudowanego iteratora). Jeśli podczas refaktoryzacji zepsujemy jakiekolwiek testy, przed zakończeniem tej fazy priorytetowo traktujemy przywrócenie ich stanu zielonego.

Jest to cykl **„czerwony, zielony, refaktoryzacja"** (ang. *Red-Green-Refactor* — RGR), który pokazałem na rysunku 1.1. Te trzy fazy są podstawowymi elementami konstrukcyjnymi programowania opartego na testach. Cykl będziesz stosował do całego kodu, który napiszesz podczas pracy z tą książką.



Rysunek 1.1. Cykl „czerwony, zielony, refaktoryzacja" jest fundamentem, na którym opiera się TDD

Trzy fazy cyklu „czerwony, zielony, refaktoryzacja" są podstawowymi elementami konstrukcyjnego TDD.

RGR w akcji

Trzech faz cyklu RGR będziemy używać w całej książce. Ściśle zastosujemy się do tego reżimu, dlatego ważne jest, by zacząć powoli, a potem przyspieszyć. W tym rozdziale w odpowiednich fragmentach te trzy fazy będą wyraźnie zaznaczone. W kolejnych rozdziałach od fazy czerwonej do zielonej będziemy przechodzić dość dziarsko i często płynnie. Potem będziemy kierować naszą uwagę na określanie tego, co należy poddać refaktoryzacji. Przejścia między fazami staną się coraz płynniejsze w miarę wzrastania tempa naszego programowania. Jednak te trzy fazy będą zawsze obecne i przeprowadzone dokładnie w tej samej kolejności.

Na czym polega problem?

Mamy problem z pieniędzmi. Nie chodzi jednak o to, z czym boryka się prawie *każdy* z nas, czyli z niewystarczającą ilością pieniędzy! Ten problem polega raczej na tym, że chcemy „śledzić nasze pieniądze".

Załóżmy, że musimy zbudować arkusz kalkulacyjny do zarządzania pieniędzmi w więcej niż jednej walucie, np. do zarządzania portfelem akcji.

Spółka	Indeks	Akcje	Kurs	Wartość całkowita
IBM	NASDAQ	100	124 USD	12 400 USD
BMW	DAX	400	75 EUR	30 000 EUR
Samsung	KSE	300	68 000 KRW	20 400 000 KRW

Aby zbudować ten arkusz kalkulacyjny, musielibyśmy wykonywać proste operacje arytmetyczne na liczbach w dowolnej walucie:

5 USD · 2 = 10 USD

10 EUR · 2 = 20 EUR

4002 KRW : 4 = 1000,5 KRW

Chcielibyśmy również przeliczać waluty, np. jeśli 1 euro daje nam 1,2 dolara, a 1 dolar wymienia się na 1100 wonów południowokoreańskich, możemy wykonać takie operacje:

5 USD + 10 EUR = 17 USD

1 USD + 1100 KRW = 2200 KRW

Każda z powyższych linii będzie jedną (maleńką) funkcjonalnością, którą zaimplementujemy przy użyciu TDD. Mamy już kilka funkcjonalności do zaimplementowania. Aby móc skoncentrować się tylko na jednej rzeczy, funkcjonalność, nad którą będziemy pracować, zaznaczymy **pogrubioną czcionką**. Kiedy skończymy opracowywać daną funkcjonalność, zasygnalizujemy naszą satysfakcję, ~~wykreślając ją~~.

Od czego powinniśmy zacząć? Na wypadek gdyby tytuł tej książki nie był wystarczająco oczywistą wskazówką, podpowiem, że zaczniemy od napisania testu.

Nasz pierwszy test z negatywnym wynikiem

Zacznijmy od wdrożenia pierwszej funkcjonalności z naszej listy:

5 USD · 2 = 10 USD

10 EUR · 2 = 20 EUR

4002 KRW : 4 = 1000,5 KRW

5 USD + 10 EUR = 17 USD

1 USD + 1100 KRW = 2200 KRW

Najpierw napiszemy niepomyślny test, odpowiadający fazie *czerwonej* cyklu RGR.

Go

W nowym pliku o nazwie *money_test.go* w folderze *go* napiszmy nasz pierwszy test:

```
package main ❶

import (
    "testing" ❷
)

func TestMultiplication(t *testing.T) { ❸
    fiver := Dollar{ ❹
```

```
            amount: 5,
        }
        tenner := fiver.Times(2) ❺
        if tenner.amount != 10 { ❻
            t.Errorf("Oczekiwano 10, otrzymano: [%d]", tenner.amount) ❼
        }
}
```

❶ Deklaracja pakietu.

❷ Importujemy pakiet `"testing"`, którego używamy później w `t.Errorf`.

❸ Nasza metoda testowa, która musi zaczynać się od `Test` i mieć jeden argument `*testing.T`.

❹ Struktura reprezentująca 5 dolarów. `Dollar` jeszcze nie istnieje.

❺ Testowana metoda `Times`, która również jeszcze nie istnieje.

❻ Porównanie wartości rzeczywistej z oczekiwaną.

❼ Upewniamy się, że test zakończy się wynikiem negatywnym, jeżeli wartość oczekiwana nie będzie równa wartości rzeczywistej.

Ta funkcja testowa zawiera nieco boilerplate'owego kodu.

Instrukcja `package main` deklaruje, że cały powstały kod jest częścią pakietu `main`. Jest to wymagane dla samodzielnych programów wykonywalnych Go. W języku Go zarządzanie pakietami (*https://oreil.ly/yvh3S*) to zaawansowana funkcjonalność. Omówiłem ją szerzej w rozdziale 5.

Następnie importujemy pakiet `testing` za pomocą instrukcji `import`. Ten pakiet zostanie wykorzystany w teście jednostkowym.

Funkcja testu jednostkowego stanowi większość kodu. Deklarujemy encję reprezentującą 5 dolarów. Jest to zmienna o nazwie `fiver`, którą inicjujemy ze strukturą przechowującą wartość 5 w polu `amount`. Następnie mnożymy `fiver` przez 2 i oczekujemy, że wynik wyniesie 10 dolarów, czyli pole `amount` zmiennej `tenner` musi mieć wartość 10. Jeśli tak nie jest, wypisujemy ładnie sformatowany komunikat o błędzie z rzeczywistą wartością (niezależnie od tego, jaka ona będzie).

Kiedy uruchomimy ten test za pomocą polecenia `go test -v .` z poziomu folderu *go* w głównym folderze projektu TDD, powinniśmy otrzymać błąd:

```
... undefined: Dollar
FAIL    tdd [build failed]
FAIL
```

Otrzymujemy głośno i wyraźnie komunikat, że to nasz pierwszy nieudany test!

Polecenie `go test -v .` powoduje uruchomienie testów w bieżącym folderze, a `go test -v ./...`[1] uruchamia testy w bieżącym folderze i wszystkich podfolderach. Przełącznik `-v` generuje szczegółowe dane wyjściowe.

[1] Trzy kropki w poleceniach `go test -v ./...` i `go fmt ./...` należy wpisywać dosłownie; to jedyne wystąpienia w tej książce, kiedy *nie oznaczają* one pominiętego kodu!

JavaScript

Napiszmy nasz pierwszy test w nowym pliku o nazwie *test_money.js* utworzonym w folderze *js*:

```
const assert = require('assert'); ❶

let fiver = new Dollar(5); ❷
let tenner = fiver.times(2); ❸
assert.strictEqual(tenner.amount, 10); ❹
```

❶ Importujemy pakiet `assert`, który będzie potrzebny później do asercji.

❷ Obiekt reprezentujący 5 dolarów. `Dollar` jeszcze nie istnieje.

❸ Testowana metoda `times`, która również jeszcze nie istnieje.

❹ Porównywanie wartości rzeczywistej z wartością oczekiwaną w instrukcji asercji `strictEqual`.

JavaScript ma minimalną ilość boilerplate'owego kodu — jedyną linią poza kodem testowym jest instrukcja `require`. Daje nam ona dostęp do pakietu `assert` menedżera pakietów NPM.

Po tej linii następują trzy linie kodu, które tworzą nasz test. Tworzymy obiekt reprezentujący 5 dolarów, mnożymy go przez 2 i oczekujemy, że wynik będzie równy 10.

W specyfikacji ES2015 wprowadzone zostało słowo kluczowe `let` (*https://oreil.ly/jBMPk*) do deklarowania zmiennych oraz słowo kluczowe `const` (*https://oreil.ly/GfYQ5*) do deklarowania stałych.

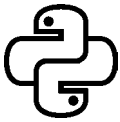
Kiedy uruchomimy ten kod z poziomu głównego folderu projektu TDD za pomocą polecenia `js/test_money.js`, powinniśmy otrzymać błąd, który zaczyna się tak:

```
ReferenceError: Dollar is not defined
```

To nasz pierwszy test z negatywnym wynikiem. Hurra!

Polecenie `node` *`nazwa_pliku`*`.js` uruchamia kod JavaScriptu z pliku *nazwa_pliku.js* i generuje dane wyjściowe. Używamy tego polecenia do uruchamiania testów.

Python



Napiszmy nasz pierwszy test w nowym pliku o nazwie *test_money.py* utworzonym w folderze *py*:

```
import unittest ❶

class TestMoney(unittest.TestCase): ❷
  def testMultiplication(self): ❸
    fiver = Dollar(5) ❹
    tenner = fiver.times(2) ❺
    self.assertEqual(10, tenner.amount) ❻

if __name__ == '__main__': ❼
    unittest.main()
```

❶ Importujemy pakiet `unittest` niezbędny dla klasy bazowej `TestCase`.

❷ Nasza klasa testowa, która musi być podklasą klasy `unittest.TestCase`.

❸ Aby nasza metoda była kwalifikowana jako metoda testowa, jej nazwa musi zaczynać się od `test`.

❹ Obiekt reprezentujący 5 dolarów. `Dollar` jeszcze nie istnieje.

❺ Testowana metoda `times`, która również jeszcze nie istnieje.

❻ Porównywanie wartości rzeczywistej z wartością oczekiwaną w instrukcji `assertEqual`.

❼ Idiom `main` zapewnia, że ta klasa może być uruchamiana jako skrypt.

Python wymaga zaimportowania pakietu `unittest`, utworzenia podklasy klasy bazowej `TestCase` oraz zdefiniowania funkcji, której nazwa zaczyna się od `test`. Aby móc uruchomić tę klasę jako samodzielny program, potrzebujemy powszechnie stosowanego idiomu Pythona (*https://docs.python.org/3/library/__main__.html*), który uruchomi funkcję `unittest.main()`, gdy wykonamy bezpośrednio polecenie `test_money.py`.

Ta funkcja testowa opisuje oczekiwany sposób działania kodu. Definiujemy zmienną o nazwie `fiver` i inicjujemy ją w pożądanej (ale jeszcze nieutworzonej) klasie `Dollar` z argumentem konstruktora o wartości `5`. Następnie mnożymy `fiver` przez `2` i zapisujemy wynik w zmiennej `tenner`. Na koniec oczekujemy, że pole `amount` zmiennej `tenner` będzie miało wartość `10`.

Kiedy uruchomimy ten kod za pomocą polecenia `python3 py/test_money.py -v` z poziomu folderu `TDD_PROJECT_ROOT`, otrzymamy błąd:

```
NameError: name 'Dollar' is not defined
```

To nasz pierwszy test z negatywnym wynikiem. Hurra!

Polecenie `python3 nazwa_pliku.py -v` powoduje uruchomienie kodu Pythona znajdującego się w pliku *nazwa_pliku.py* i wyświetlenie szczegółowych danych wyjściowych. Używamy tego polecenia do uruchamiania naszych testów.

Gramy w zielone

Napisaliśmy nasze testy w sposób zgodny z oczekiwanym działaniem, na razie beztrosko ignorując wszystkie błędy składniowe. Czy to rozsądne?

Na *samym* początku — czyli tu, gdzie jesteśmy — dobrze jest zacząć od najmniejszego fragmentu kodu, który skieruje nas na ścieżkę progresu. Oczywiście nasze testy kończą się niepowodzeniem, ponieważ nie zdefiniowaliśmy, czym jest `Dollar`. *Może* się wydawać, że to idealny moment, by powiedzieć: „no raczej!”. Jednak odrobina cierpliwości jest uzasadniona z dwóch powodów:

1. Właśnie zakończyliśmy *pierwszy* etap naszego *pierwszego* testu — fazę *czerwoną*. To nawet nie początek, a sam początek początku.
2. W miarę postępów będziemy mogli zwiększyć tempo (i to zrobimy). Należy jednak wiedzieć, że kiedy potrzeba, możemy zwolnić.

Kolejną fazą cyklu RGR jest faza *zielona*.

Oczywiste jest, że musimy wprowadzić abstrakcję `Dollar`. Ta sekcja definiuje, jak wprowadzić tę i inne abstrakcje, aby nasz test przeszedł.

Go

Na końcu kodu w pliku *money_test.go* dodajemy pustą strukturę `Dollar`.

```
type Dollar struct {
}
```

Jeśli uruchomimy teraz nasz test, otrzymamy nowy błąd:

```
... unknown field 'amount' in struct literal of type Dollar
```

Postęp!

Ten komunikat o błędzie kieruje nas do wprowadzenia w strukturze `Dollar` pola o nazwie `amount`. Zróbmy to więc, używając na razie typu danych `int` (który jest wystarczający do naszych celów):

```
type Dollar struct {
    amount int
}
```

Rozszerzenie definicji struktury `Dollar` raczej przewidywalnie prowadzi nas do następnego błędu:

```
... fiver.Times undefined (type Dollar has no field or method Times)
```

Wyłania się tu pewien wzorzec: gdy coś (pole lub metoda) jest niezdefiniowane, otrzymujemy ze środowiska uruchomieniowego Go błąd `undefined`. Wykorzystamy te informacje, aby w przyszłości przyspieszyć nasze cykle TDD. Na razie dodajmy funkcję o nazwie `times`. Ze sposobu, w jaki napisaliśmy nasz test, wiemy, że funkcja ta musi przyjmować liczbę (mnożnik) i zwracać kolejną liczbę (wynik).

Ale jak obliczyć wynik? Znamy podstawy arytmetyki i wiemy, jak pomnożyć dwie liczby. Gdybyśmy jednak mieli napisać najprostszy działający kod, bylibyśmy usprawiedliwieni, jeżeli *zawsze* zwracalibyśmy wynik oczekiwany przez nasz test, czyli strukturę reprezentującą 10 dolarów:

```
func (d Dollar) Times(multiplier int) Dollar {
    return Dollar{10}
}
```

Jeśli uruchomimy teraz nasz kod, powinniśmy otrzymać w terminalu krótką i przyjemną odpowiedź:

```
=== RUN   TestMultiplication
--- PASS: TestMultiplication (0.00s)
PASS
```

Oto magiczne słowo `PASS`, które oznacza, że test zakończył się pomyślnie!

JavaScript

W pliku *test_money.js* zaraz po linii `const assert = require('assert');` zdefiniujmy pustą klasę o nazwie `Dollar`:

```
class Dollar {
}
```

Jeśli uruchomimy teraz plik *test_money.js*, otrzymamy błąd:

```
TypeError: fiver.times is not a function
```

Postęp! Komunikat o błędzie jasno informuje, że dla obiektu o nazwie `fiver` nie ma zdefiniowanej funkcji o nazwie `times`. Wprowadźmy ją więc w klasie `Dollar`:

```
class Dollar {
    times(multiplier) {
    }
}
```

Uruchomienie testu powoduje teraz pojawienie się nowego błędu:

```
TypeError: Cannot read properties of undefined (reading 'amount') ❶
```

❶ Ten komunikat pochodzi z Node.js v16; wersja v14 wyświetla nieco inny komunikat o błędzie.

Nasz test oczekuje obiektu z właściwością `amount`. Ponieważ z metody `times` niczego nie zwracamy, wartość zwracana jest `undefined`, co oznacza, że nie ma właściwości `amount` (ani tak naprawdę żadnej innej właściwości).

W języku JavaScript w funkcjach i metodach nie deklaruje się bezpośrednio żadnych typów zwracanych. Jeśli zbadamy wynik funkcji, która niczego nie zwraca, przekonamy się, że wartość zwracana jest `undefined`.

Jak więc sprawić, by nasz test stał się zielony? Jaka jest najprostsza rzecz, która może zadziałać? A co powiesz na to, żebyśmy zawsze tworzyli i zwracali obiekt reprezentujący 10 dolarów?

Spróbujmy! Dodajmy `constructor` inicjujący obiekty z określoną kwotą oraz metodę `times`, która uparcie tworzy i zwraca obiekty „10 dolarów”:

```
class Dollar {
    constructor(amount) { ❶
        this.amount = amount; ❷
    }

    times(multiplier) { ❸
        return new Dollar(10); ❹
    }
}
```

❶ Funkcja `constructor` jest wywoływana za każdym razem, gdy tworzony jest obiekt `Dollar`.

❷ Inicjujemy zmienną `this.amount` z podanym parametrem.

❸ Metoda `times` przyjmuje parametr.

❹ Prosta implementacja: zawsze zwracamy 10 dolarów.

Gdy uruchomimy teraz nasz kod, nie powinniśmy otrzymać żadnych błędów. To nasz pierwszy zielony test!

Ponieważ `strictEqual` i inne metody z pakietu `assert` generują dane wyjściowe tylko wtedy, gdy asercje się *nie powiodą*, pomyślne uruchomienie testu będzie całkiem ciche bez danych wyjściowych. Poprawimy to zachowanie w rozdziale 6.

Python

Ponieważ `'Dollar' is not defined`, zdefiniujmy go w kodzie w pliku *test_money.py* przed klasą `TestMoney`:

```
class Dollar:
  pass
```

Jeśli uruchomimy teraz nasz kod, otrzymamy błąd:

```
TypeError: Dollar() takes no arguments
```

Postęp! Ten komunikat o błędzie mówi nam wyraźnie, że aktualnie nie ma możliwości inicjowania obiektów `Dollar` z jakimikolwiek argumentami, takimi jak 5 i 10, które mamy w kodzie. Naprawmy to więc, dostarczając najkrótszy możliwy inicjator:

```
class Dollar:
  def __init__(self, amount):
    pass
```

Teraz komunikat o błędzie z naszego testu zmienia się:

```
AttributeError: 'Dollar' object has no attribute 'times'
```

Widać tu pewien wzorzec: nasz test nadal kończy się niepowodzeniem, ale za każdym razem z *nieco innych* powodów. Gdy definiujemy abstrakcje — najpierw `Dollar`, a następnie pole `amount` — komunikaty o błędach „poprawiają się”, przechodząc do kolejnego etapu. To znak rozpoznawczy TDD: stały postęp w kontrolowanym tempie.

Przyspieszmy trochę. Zdefiniujmy funkcję `times` *oraz* określmy jej minimalne zachowanie, aby przejść do etapu zielonego. Jakie jest niezbędne minimalne zachowanie? Oczywiście takie, by zawsze zwracać obiekt „10 dolarów”, który jest wymagany przez nasz test!

```
class Dollar:
  def __init__(self, amount): ❶
    self.amount = amount ❷

  def times(self, multiplier): ❸
    return Dollar(10) ❹
```

❶ Funkcja `__init__` jest wywoływana za każdym razem, gdy tworzony jest obiekt `Dollar`.

❷ Inicjujemy zmienną `self.amount` z podanym parametrem.

❸ Metoda `times` przyjmuje parametr.

❹ Prosta implementacja: zawsze zwracamy 10 dolarów.

Jeśli uruchomimy teraz nasz test, otrzymamy krótką i przyjemną odpowiedź:

```
Ran 1 test in 0.000s

OK
```

Możliwe, że test nie zostanie wykonany w `0.000s`, ale nie traćmy z oczu magicznego słowa `OK`. To nasz pierwszy zielony test!

Czyszczenie

Czy czujesz się zdezorientowany, że do fazy zielonej doszliśmy przez zakodowanie w testach wartości 10 dolarów na sztywno? Nie przejmuj się: etap refaktoryzacji pozwala nam pozbyć się tego dyskomfortu i dowiedzieć się, *w jaki sposób* możemy usunąć zakodowaną na sztywno i zduplikowaną wartość 10 dolarów.

Refaktoryzacja to trzeci i ostatni etap cyklu RGR. W tym momencie możemy nie mieć wielu linii kodu; ważne jest jednak, aby wszystko było uporządkowane i zwarte. Jeśli mamy jakiś bałagan formatowania lub wykomentowane wiersze kodu, nadszedł czas, by to uporządkować.

Najistotniejsze jest usunięcie duplikatów i uczynienie kodu czytelnym. Na pierwszy rzut oka może się wydawać, że w około 20 linijkach kodu, które napisaliśmy, nie może być żadnej duplikacji. A jednak wdarła się pewna subtelna, ale znacząca duplikacja.

Tę duplikację możemy znaleźć, jeśli zauważymy w naszym kodzie kilka dziwactw:

1. Napisaliśmy wystarczająco dużo kodu, aby zweryfikować, czy „podwojenie 5 dolarów da nam 10 dolarów". Jeżeli zdecydujemy się zmienić dotychczasowy test i przyjąć założenie, że „podwojenie 10 dolarów powinno dać nam 20 dolarów" — co jest równie rozsądnym stwierdzeniem — będziemy musieli zmienić *zarówno* test, jak i kod obiektu `Dollar`. Pomiędzy tymi dwoma segmentami kodu istnieje pewna zależność, czyli **logiczne powiązanie**. Generalnie tego rodzaju powiązania należy unikać.
2. Zarówno w teście, jak i w kodzie mamy magiczną liczbę 10. Jak na to wpadliśmy? Oczywiście policzyliśmy to w pamięci. Wiemy, że podwojenie 5 dolarów powinno dać nam 10 dolarów. Dlatego wpisaliśmy wartość 10 zarówno w teście, jak i w kodzie dolara. Powinniśmy zdać sobie sprawę, że 10 w encji `Dollar` to tak naprawdę `5 * 2`. Gdy to zrozumiemy, będziemy mogli usunąć tę duplikację.

Zduplikowany kod jest często symptomem jakiegoś głębszego problemu: brakującej warstwy abstrakcji kodu lub niewłaściwych powiązań pomiędzy różnymi częściami kodu[2].

Usuńmy tę duplikację, a tym samym pozbądźmy się również powiązania.

[2] Warto przytoczyć tutaj opinię Kenta Becka: „Jeśli zależność jest problemem, duplikacja jest symptomem".

Go

Zastąpmy wartość 10 w funkcji `Times` jej odpowiednikiem w postaci operacji `5 * 2`:

```
func (d Dollar) Times(multiplier int) Dollar {
    return Dollar{5 * 2}
}
```

Test powinien pozostać zielony.

Gdy napiszemy to w ten sposób, uświadomimy sobie, że brakuje nam warstwy abstrakcji. Zakodowana na sztywno wartość `5` to tak naprawdę `d.amount`, a `2` to `multiplier`. Zastąpienie tych zakodowanych na sztywno liczb odpowiednimi zmiennymi daje nam nietrywialną implementację:

```
func (d Dollar) Times(multiplier int) Dollar {
    return Dollar{d.amount * multiplier}
}
```

Super! Test nadal jest wykonywany pomyślnie, a jednocześnie usunęliśmy duplikację i powiązanie.

Został jeszcze jeden, ostatni element czyszczenia.

Podczas inicjowania w teście struktury `Dollar` użyliśmy bezpośrednio nazwy pola `amount`. Przy inicjowaniu struktury możliwe jest pominięcie także nazw pól, tak jak zrobiliśmy to w metodzie `Times`[3]. Możliwe są oba style: używanie bezpośrednich nazw albo ich nieużywanie. Należy jednak być konsekwentnym. Zmieńmy funkcję `Times`, aby określić nazwę pola:

```
func (d Dollar) Times(multiplier int) Dollar {
    return Dollar{amount: d.amount * multiplier}
}
```

Pamiętaj o cyklicznym uruchamianiu polecenia `go fmt ./...`, aby naprawić wszelkie problemy z formatowaniem w kodzie.

JavaScript

Zamieńmy wartość 10 w metodzie `times` na jej odpowiednik w postaci operacji `5 * 2`:

```
times(multiplier) {
    return new Dollar(5 * 2);
}
```

Test powinien pozostać zielony.

Oczywista staje się teraz brakująca abstrakcja. Zamiast wartości `5` możemy użyć `this.amount`, a w miejsce `2` wstawić `multiplier`:

```
times(multiplier) {
    return new Dollar(this.amount * multiplier);
}
```

[3] Jeżeli w strukturze jest wiele pól (co nas na razie nie dotyczy), kolejność pól musi być taka sama zarówno w definicji struktury, jak i podczas inicjowania, *albo* nazwy pól muszą być określone podczas inicjowania struktury. Zobacz *https://gobyexample.com/structs*.

Super! Test jest nadal zielony, a jednocześnie wyeliminowaliśmy zarówno zduplikowaną wartość 10, jak i powiązanie.

Python

Zamieńmy wartość 10 w metodzie `times` na jej odpowiednik w postaci operacji `5 * 2`:

```
def times(self, multiplier):
  return Dollar(5 * 2)
```

Zgodnie z oczekiwaniami test pozostaje zielony.

Ujawnia to leżącą u podstaw abstrakcję. Wartość `5` to tak naprawdę `self.amount`, a `2` to `multiplier`:

```
def times(self, multiplier):
  return Dollar(self.amount * multiplier)
```

Hurra! Test jest nadal zielony, a jednocześnie zniknęły duplikacja i powiązanie.

Zatwierdzanie zmian

Ukończyliśmy naszą pierwszą funkcjonalność, używając TDD. Nie zapominajmy, że ważne jest częste zatwierdzanie kodu w naszym repozytorium kontroli wersji.

Zielony test to doskonała okazja do zatwierdzenia kodu.

W oknie powłoki wpiszmy te dwa polecenia:

```
git add . ❶
git commit -m "dokonanie: pierwszy zielony test" ❷
```

❶ Dodajemy do indeksu Gita wszystkie pliki, w tym wszystkie wprowadzone w nich zmiany.

❷ Zatwierdzamy indeks Gita w repozytorium z podanym komunikatem.

Zakładając, że kod dla wszystkich trzech języków znajduje się we właściwych folderach, powinniśmy otrzymać taki komunikat:

```
[main (root-commit) bb31b94] dokonanie: pierwszy zielony test ❶
4 files changed, 56 insertions(+)
create mode 100644 go/go.mod
create mode 100644 go/money_test.go
create mode 100644 js/test_money.js
create mode 100644 py/test_money.py
```

❶ Liczba szesnastkowa `bb31b94` reprezentuje kilka pierwszych cyfr unikatowego skrótu SHA, związanego z zatwierdzeniem kodu. Ta liczba będzie inna dla każdego użytkownika i zatwierdzenia.

Oznacza to, że wszystkie nasze pliki znalazły się bezpiecznie w repozytorium kontroli wersji Gita. Możemy to zweryfikować, wykonując w powłoce polecenie `git log`, które powinno dać mniej więcej taki wynik:

```
commit bb31b94e90029ddeeee89f3ca0fe099ea7556603 (HEAD -> main) ❶
Author: Saleem Siddiqui ...
Date: Sun Mar 7 12:26:06 2021 -0600

    dokonanie: pierwszy zielony test ❷
```

❶ To jest pierwsze zatwierdzenie z pełnym skrótem SHA.

❷ To jest komunikat, który wpisaliśmy dla naszego pierwszego zatwierdzenia.

Należy zdać sobie sprawę, że repozytorium Gita, do którego przesłaliśmy nasz kod, również znajduje się w naszym lokalnym systemie plików (w folderze *.git* w katalogu głównym `TDD_PROJECT_ROOT`). Chociaż nie chroni nas to przed przypadkowym rozlaniem kawy na klawiaturę (zawsze używaj kubków z przykrywką), daje pewność, że jeśli gdzieś się zaplączemy, będziemy mogli wrócić do poprzedniej znanej działającej wersji. W rozdziale 13. prześlemy cały nasz kod do repozytorium GitHuba.

Tej strategii zatwierdzania kodu w naszym lokalnym repozytorium Gita będziemy używać w każdym rozdziale, korzystając z tego samego zestawu poleceń.

W każdym rozdziale będziemy stosować te dwa polecenia, `git add .` i `git commit -m` *`_komunikat zatwierdzenia_`*, do częstego zatwierdzania naszego kodu.

Jedynym zmieniającym się elementem będzie komunikat zatwierdzenia, który będzie zgodny z semantycznym stylem zatwierdzania i będzie zawierał krótki, jednoliniowy opis zmian.

Komunikaty `git commit` używane w tej książce są zgodne z semantycznym stylem zatwierdzania (*https://oreil.ly/MhE1b*).

Gdzie jesteśmy?

W tym rozdziale wprowadziłem programowanie oparte na testach (TDD), pokazując *pierwszy* cykl „czerwony, zielony, refaktoryzacja". Po pomyślnym wdrożeniu pierwszej małej funkcjonalności możemy ją wykreślić. Na naszej liście funkcjonalności znajdujemy się w tym miejscu:

~~5 USD · 2 = 10 USD~~

10 EUR · 2 = 20 EUR

4002 KRW : 4 = 1000,5 KRW

5 USD + 10 EUR = 17 USD

1 USD + 1100 KRW = 2200 KRW

wyzwania, poświęćmy chwilę na przejrzenie naszego kodu i podelektowanie się nim. Poniżej przedstawiłem kod źródłowy dla wszystkich trzech języków. Znajdziesz go również w repozytorium kodów dołączonym do książki. Aby zachować zwięzłość, w kolejnych rozdziałach będę podawał tylko nazwę odpowiedniej gałęzi.

Go

Tak wygląda teraz plik *money_test.go*:

```
package main
import (
    "testing"
)

func TestMultiplication(t *testing.T) {
    fiver := Dollar{amount: 5}
    tenner := fiver.Times(2)
    if tenner.amount != 10 {
        t.Errorf("Oczekiwano 10, otrzymano: [%d]", tenner.amount)
    }
}

type Dollar struct {
    amount int
}

func (d Dollar) Times(multiplier int) Dollar {
    return Dollar{amount: d.amount * multiplier}
}
```

JavaScript

Tak wygląda w tym momencie plik *test_money.js*:

```
const assert = require('assert');

class Dollar {
    constructor(amount) {
      this.amount = amount;
    }

    times(multiplier) {
        return new Dollar(this.amount * multiplier);
    }
}

let fiver = new Dollar(5);
let tenner = fiver.times(2);
assert.strictEqual(tenner.amount, 10);
```

Python

Tak wygląda teraz plik *test_money.py*:

```
import unittest

class Dollar:
  def __init__(self, amount):
    self.amount = amount

  def times(self, multiplier):
    return Dollar(self.amount * multiplier)
```

```
class TestMoney(unittest.TestCase):
  def testMultiplication(self):
    fiver = Dollar(5)
    tenner = fiver.times(2)
    self.assertEqual(10, tenner.amount)

if __name__ == '__main__':
    unittest.main()
```

Kod dla tego rozdziału znajduje się w folderze o nazwie *r01* w repozytorium kodów, które możesz pobrać pod adresem *https://ftp.helion.pl/przyklady/naprop.zip*. Odpowiedni folder istnieje dla każdego rozdziału, w którym będziemy pisać kod.

W rozdziale 2. przyspieszymy pracę i zbudujemy kilka kolejnych funkcjonalności.

TDD: napisz kod, któremu można zaufać!

Od oprogramowania wymaga się solidności i poprawności, a równocześnie oczekuje wydajnego działania i skalowalności. Jako programista odpowiadasz za zapewnienie aplikacjom tych właśnie cech. Niezależnie od tego, jakiego języka programowania używasz, praca zgodnie z paradygmatem TDD umożliwi Ci otrzymanie testowalnego kodu o wysokiej jakości. Choć te korzyści przeważają nad niedogodnościami podejścia TDD, wielu programistów narzeka na czasochłonność, pracochłonność i sprawiającą problemy implementację programowania opartego na testach.

To przewodnik dla programistów, którzy chcą stosować podejście TDD w swojej codziennej praktyce. Pokazuje, jak korzystać z programowania sterowanego testami podczas pracy w trzech różnych językach: Go, JavaScripcie i Pythonie. Dzięki tej książce zrozumiesz, w jaki sposób zastosować klasyczny paradygmat „dziel i zwyciężaj" do budowania testów jednostkowych i w efekcie radzić sobie nawet z bardzo rozbudowaną architekturą oprogramowania. Liczne przykłady o stopniowo rosnącym poziomie zaawansowania pozwolą Ci płynnie nabierać wprawy i pewności w tworzeniu testów jednostkowych, a także ich używaniu. Szybko się przekonasz, że wprowadzenie TDD do codziennej praktyki kodowania jest bardzo opłacalną decyzją: kod będzie czysty, zrozumiały, elegancki i prosty w utrzymaniu!

W książce między innymi:

- działanie TDD w różnych językach, frameworkach testowych i koncepcjach domenowych
- TDD a ciągła integracja
- konfiguracja środowiska ciągłej integracji
- refaktoryzacja i przeprojektowywanie przy użyciu TDD
- testy jednostkowe w JavaScripcie
- jak TDD ułatwia pisanie czystego kodu w Go, JavaScripcie i Pythonie

Saleem Siddiqui jest programistą, autorem książek i znakomitym dydaktykiem programowania. Zdobył doświadczenie w tworzeniu oprogramowania dla służby zdrowia, handlu detalicznego, a także sektora rządowego i farmaceutycznego. Przekonuje, że dzięki uważnemu wdrażaniu metodyki TDD można uniknąć wielu kosztownych błędów.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

ISBN 978-83-283-9040-9
9 788328 390409

Cena: 59,00 zł