Testowanie na podstawie właściwości

inż. Paulina Brzęcka 184701 inż. Marek Borzyszkowski 184266

14 grudnia 2024

Spis treści

1 Wstęp

Istnieje wiele koncepcji testowania oprogramowania, jedną z nich jest testowanie na podstawie właściwości. Ten dokument ma na celu ukazanie:

1. na czym polega testowanie na podstawie właściwości.
2. Czym różni się ono od klasycznego podejścia do testowania.
3. Jakie są strategie testowania na podstawie właściwości.
4. Jak przy wykorzystaniu konceptu QuickCheck znaleźć wartości początkowe nieprzechodzące testy.

2 Geneza

Testowanie na przykładach (Example-Based Testing)

W testowaniu na przykładach sprawdzamy funkcję na podstawie konkretnych wartości wejściowych i oczekiwanych rezultatów.

Załóżmy, że chcemy przetestować funkcję dodawania.

Testowanie na przykładach:

Listing 1: Testowanie na przykładach

```
[Test]
let ``Add two numbers, expect their sum``() =
    let testData = [ (1,2,3); (2,2,4); (3,5,8); (27,15,42) ]
    for (x,y,expected) in testData do
        let actual = add x y
        Assert.AreEqual(expected,actual)
```

Deweloper potrafi napisać coś takiego:

Listing 2: Testowany kod

```
let add x y =
    match x,y with
    | 1,2 -> 3
    | 2,2 -> 4
    | 3,5 -> 8
    | 27,15 -> 42
    | _ -> 0
```

Dodatkowo odgrażając się, że podąża za regułą TDD, czyli poprawia swój kod minimalnie, dopóki test nie będzie zielony :)

Ograniczenia:

- Musimy ręcznie wymyślić testowe dane wejściowe.
- Testy obejmują tylko przypadki, które sami zdefiniujemy.
- Trudno przewidzieć wszystkie możliwe scenariusze (np. wartości graniczne, liczby ujemne, liczby bardzo duże itp.).

Można wymusić prawidłowy kod na podstawie generatora liczb losowych:

Listing 3: Przykładowe rozwiązanie

```
[Test]
let ``Add two random numbers 100 times, expect their sum``() =
    for _ in [1..100] do
        let x = randInt()
        let y = randInt()
        let expected = x + y
        let actual = add x y
        Assert.AreEqual(expected,actual)
```

Natomiast w tym przypadku musimy zaimplementować w teście `add`, aby przetestować funkcję `add`.

Testowanie na podstawie właściwości (Property-Based Testing)

Zamiast testować pojedyncze przypadki, definiujemy ogólne właściwości, które funkcja powinna zawsze spełniać. Są różne narzędzia, np. `FsCheck`, które automatycznie generują dane wejściowe i sprawdzają, czy właściwości są spełniane.

Przykład właściwości dla funkcji dodawania:

- Przemienność.
- Dodanie liczby 1 dwukrotnie jest tym samym co dodanie liczby 2 raz.
- Dodanie liczby 0 nie zmienia wartości liczby.

Listing 4: Testowanie na właściwościach

```
let add x y = x + y

let commutativeProperty x y =
    let result1 = add x y
    let result2 = add y x
    result1 = result2

[Test]
let addDoesNotDependOnParameterOrder() =
    Check.Quick commutativeProperty

let add1TwiceIsAdd2Property x _ =
    let result1 = x |> add 1 |> add 1
    let result2 = x |> add 2
    result1 = result2

```

```
17 [Test]
18 let addOneTwiceIsSameAsAddTwo() =
19     Check.Quick add1TwiceIsAdd2Property
20
21 let identityProperty x _ =
22     let result1 = x |> add 0
23     result1 = x
24
25 [Test]
26 let addZeroIsSameAsDoingNothing() =
27     Check.Quick identityProperty
```

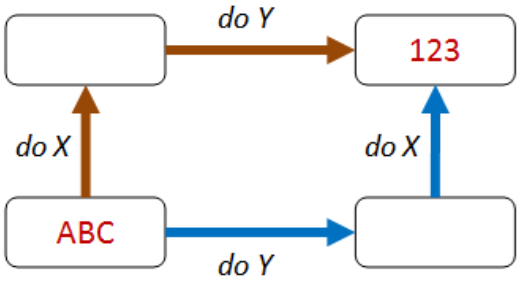
Dzięki podejściu *property-based* mamy większą pewność, że implementacja jest właściwa. Dodatkowo to podejście pozwala lepiej zrozumieć wymagania i istotę całego problemu.
Zdefiniowanie właściwości w bardziej złożonych przypadkach potrafi być bardzo problematyczne. Natomiast istnieją pewne techniki, które pomagają w ich definiowaniu.

3 Strategie

Testowanie na podstawie właściwości nie jest proste w zastosowaniu, przynajmniej przy pierwszej próbie. Istnieją jednak pewne schematy wskazujące drogę, jak stworzyć takie testy.

3.1 Różne ścieżki, ten sam wynik

Jedną z podstawowych strategii skorzystanie z komutatywności niektórych operacji. Można to zrobić poprzez wykonanie operacji w różnej kolejności, jak zaprezentowano w Rys. 1.

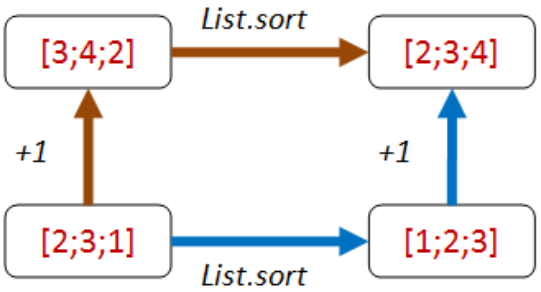


Rysunek 1: Strategia - komutatywność

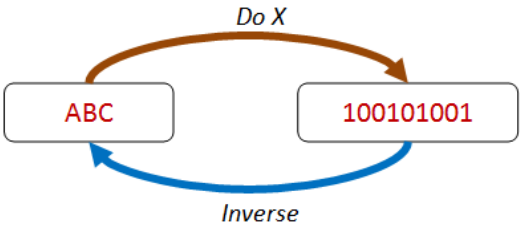
Przykładem takiej strategii może być komutatywność dodawania z List. 4, gdzie wykorzystano `add x y`, jak i odwrotność tej operacji `add y x`. Innym przykładem jest test metody `sort` danej listy. Wykonanie sortowania, a następnie dodanie do każdego elementu listy `1` powinno dać taki sam efekt jak dodanie `1` do każdego z elementów listy, a następnie jej posortowanie Rys. 2 List. 5.

Listing 5: Test sortowania listy z wykorzystaniem strategii komutatywnej

```
1     let addThenSort_eq_sortThenAdd sortFn aList =
2         let add1 x = x + 1
3
4         let result1 = aList |> sortFn |> List.map add1
5         let result2 = aList |> List.map add1 |> sortFn
6         result1 = result2
```



Rysunek 2: Strategia - komutatywność - sortowanie listy



Rysunek 3: Strategia - inwersja

3.2 Tam i z powrotem

Test z wykorzystaniem inwersji Rys. 3, polega na sprawdzeniu, czy po zaaplikowaniu testowanej funkcji, następnie wykorzystaniu funkcji odwrotnej do otrzymania początkowych wartości. Przykładem takiego testu mogą być przeciwne operacje matematyczne jak:

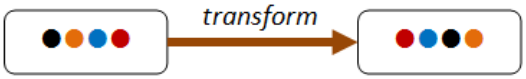
- `dodawanie/odejmowanie`
- `mnożenie/dzielenie`
- `potęga/logarytm`.

Innymi przykładami są operacje niekoniecznie matematyczne:

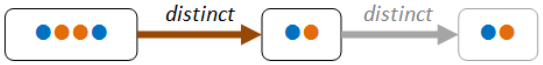
- `serializacja/deserializacja`
- `zapis/odczyt z pliku`
- `wstaw/sprawdź czy zawiera`.
- `odwrócenie listy/odwrócenie listy`

3.3 Są rzeczy niezmienne

Czasami testowana funkcja przetwarzając dane zachowuje część ich właściwości Rys. 4. Chociażby funkcje `sort` lub `map` wykonane na liście `n` elementów, zwracaja odpowiednio zmodyfikowaną listę `n` elementową.



Rysunek 4: Strategia - niezmienność



Rysunek 5: Strategia - idempotentność

3.4 Z czasem rzeczy przestają się zmieniać

Inną właściwością funkcji może być niezmienność wyniku funkcji po ponownym jej zaaplikowaniu Rys. 5. Innymi słowy, wykonanie funkcji 2 razy daje taki sam efekt, jak jednokrotne jej zaaplikowanie.

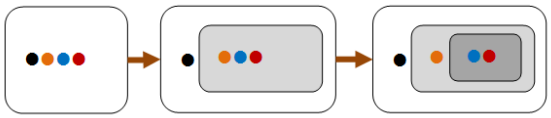
Przykładami takich operacji, dla których taki typ testu miałby zastosowanie to metoda `distinct` wykonana na danej liście, lub wykonanie `update` na danej bazie danych.

3.5 Dziel i rządź

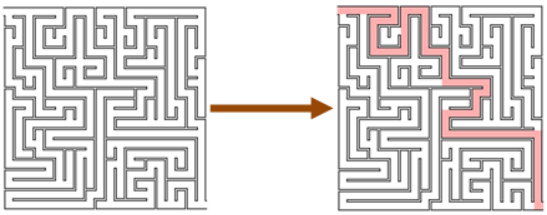
Istnieją sposoby na testowanie na podstawie właściwości jest wykorzystanie rekursywności struktur przekazywanych do funkcji, takich jak `listy`, `drzewa` Rys. 6. Przykładem może być sprawdzenie za pomocą tej metody funkcji sort List. 6.

Listing 6: Test sortowania listy z wykorzystaniem strategii rekursywnej

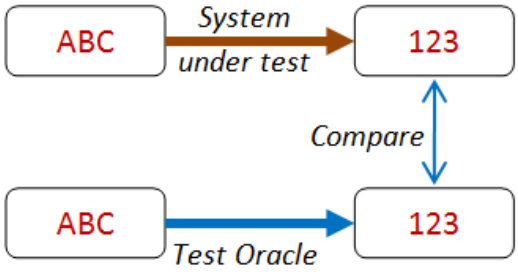
```
let rec firstLessThanSecond_andTailIsSorted sortFn (aList:int list) =
  let sortedList = aList |> sortFn
  match sortedList with
  | [] -> true
  | [first] -> true
  | [first;second] -> first <= second
  | first::second::rest->
    first <= second &&
    let tail = second::rest
    // check that tail is sorted
    firstLessThanSecond_andTailIsSorted sortFn tail
```



Rysunek 6: Strategia - rekursywność



Rysunek 7: Strategia - łatwe sprawdzenie



Rysunek 8: Strategia - wyrocznia

3.6 Łatwiej zweryfikować niż zaimplementować

Niekiedy testowana funkcja jest skomplikowana, ale jej rezultat da się łatwo sprawdzić. Przykładem może być funkcja wyszukująca wyjście z labiryntu Rys. 7, gdzie sam algorytm wyszukiwania odpowiedniej ścieżki jest skomplikowany, natomiast samo sprawdzenie, czy ścieżka dobrze prowadzi do wyjścia można w łatwy sposób zweryfikować.

3.7 Testowanie z wyrocznią

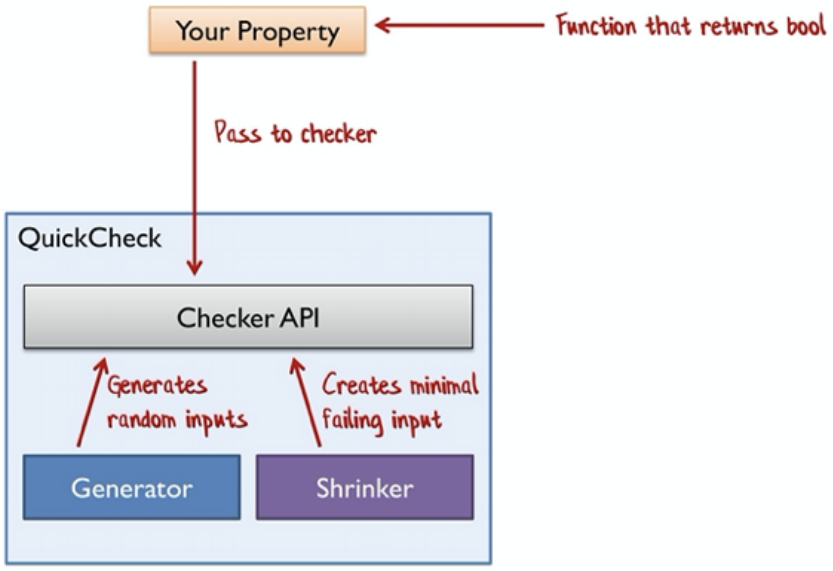
Zdaża się, że funkcjonalność została już napisana i trzeba ją zrefactorować, przepisać, napisać od nowa Rys. 8. Warto wtedy wykorzystać wartości zwracane przez oryginalnie zaimplementowany algorytm jako pewną wartość wyniku, pewnego rodzaju wyrocznię, uznając go jako prawdę. W taki sposób można sprawdzić, czy nowa funkcja w pewnym stopniu pokrywa się ze starą funkcją. Czasami też istnieje wiele algorytmów doprowadzających do tego samego wyniku, mające różne złożoności, czy też działające równolegle. Można wykożystać wtedy najprostrzy algorytm jako wyrocznię, ze względu na najmniejsze prawdopodobieństwo napisania takiego algorytmu z błędem. Następnie, przy wykorzystaniu wyroczni, stworzyć bardziej skomplikowany (często efektywniejszy) algorytm.

4 QuickCheck

QuickCheck został stworzony w języku Haskell jako pierwsze narzędzie wspierające testowanie oparte na właściwościach. Jest on inspiracją dla bibliotek w innych językach, takich jak FsCheck dla .NET.

Cechy QuickCheck

- Automatyczne generowanie danych testowych.
- Sprawdzanie, czy zdefiniowane właściwości funkcji są spełniane dla wielu losowych przypadków.
- W przypadku wykrycia błędu narzędzie redukuje (*shrinking*) dane wejściowe, aby znaleźć minimalny przykład prowadzący do błędu.



Rysunek 9: Schemat działania QuickCheck

Działanie

1. **Checker API** wykrywa typ wejścia funkcji.
2. Wywoływany jest generator odpowiedniego typu.
3. Następuje generowanie przypadków testowych.
4. Przypadki testowe są przekazywane do testowanej właściwości.

Funkcje

- **Check.Quick** – uruchamia szybki test, sprawdzając właściwość dla domyślnej liczby losowych przypadków (np. 100).

- **Check.Verbose** – działa jak Check.Quick, ale wyświetla więcej szczegółowych informacji o danych testowych.
- **Generowanie danych wejściowych** – FsCheck wspiera generowanie danych dla typów prostych (np. `int`, `float`, `string`) i bardziej złożonych struktur, takich jak listy czy rekordy.

Pisanie właściwości w FsCheck

FsCheck umożliwia definiowanie właściwości w formie funkcji logicznych. Właściwości opisują, jakie warunki zawsze muszą być spełnione przez funkcję.

Przykład: Odwracanie listy

Dla funkcji `reverse`, która odwraca listę, możemy zdefiniować właściwość:

- Odwrócenie listy dwukrotnie powinno zwrócić pierwotną listę.

Listing 7: Definicja właściwości dla odwracania listy

```
let reverseProperty (xs: int list) =
    List.rev (List.rev xs) = xs

Check.Quick reverseProperty
```

Generacja danych testowych

FsCheck pozwala na tworzenie własnych generatorów danych wybranego typu. Przydatne, gdy chcemy testować funkcję na specyficznych danych.
Na początku tworzymy generator, następnie generujemy dane testowe podając maksymalny rozmiar przypadku testowego (w przypadku stuktur danych) oraz liczbę przypadków testowych.
Zarówno przy generowaniu danych testowych, jak i przy późniejszej redukcji danych (shrinking) wykorzystywaniy jest arbitraż (Arb).
Jest to mechanizm definiowania sposobu generowania losowych danych dla konkretnego typu. Dzięki arbitrażowi możesz dostosować sposób generowania danych, wprowadzić dodatkowe ograniczenia, a nawet zdefiniować własne typy i ich generatory.

Listing 8: Przykład generatora dla liczb dodatnich

```
let intGenerator = Arb.generate<int>
Gen.sample 100 3 intGenerator  // [-37; 24; -62]

// Przyklad generowania liczb dodatnich:
type PositiveInt = PositiveInt of int

let positiveIntGen =
    Gen.suchThat ((<) 0) Arb.generate<int>
Arb.register<PositiveInt>(positiveIntGen)

type Generators =
    static member PositiveInt() =
        Arb.fromGen positiveIntGen
Arb.register<Generators>() // rejestracja generatora by Quick.Check mogl zostac uzyty

```

```
let intListGenerator = Arb.generate<int list>
Gen.sample 5 10 intListGenerator
// [ []; []; [-4]; [0; 3; -1; 2]; [1]; [1]; []; [0; 1; -2]; []; [-1; -2]]

let stringGenerator = Arb.generate<string>
Gen.sample 10 3 stringGenerator // ["";  "eiX$a^"; "U%OIkaGr"]

type Point = {x:int; y:int; color: Color}
let pointGenerator = Arb.generate<Point>
Gen.sample 50 10 pointGenerator
(*
{x = -8; y = 12; color = Green -4;};
    {x = 28; y = -31; color = Green -6;};
    {x = 11; y = 27; color = Red;};
    {x = -2; y = -13; color = Red;};
    {x = 6; y = 12; color = Red;};
    // itd
*)
```

Shrinking

Po fazie generowania danych losowych dane wejściowe są ustawione od najmniejszej do największej wartości. Jeśli jakiekolwiek dane wejściowe spowodują, że właściwość przestanie być spełniona, narzędzie zaczyna "zmniejszać"pierwszy parametr, aby znaleźć mniejszą wartość. Dokładny proces zmniejszania zależy od typu danych (można go również nadpisać) *(W przypadku liczb prowadząc do coraz mniejszych wartości.)*

Listing 9: Przykład shrinking na liczbach

```
let isSmallerThan80 x = x < 80
isSmallerThan80 100 // false, so start shrinking

Arb.shrink 100 |> Seq.toList//  [0; 50; 75; 83; 94; 97; 99]
isSmallerThan80 0 // true
isSmallerThan80 50 // true
isSmallerThan80 75 // true
isSmallerThan80 83 // false, so shrink again

Arb.shrink 83 |> Seq.toList//  [0; 44; 66; 77; 80; 81; 83]
isSmallerThan80 0 // true
isSmallerThan80 44 // true
isSmallerThan80 66 // true
isSmallerThan80 77 // true
isSmallerThan80 80 // false <- najmniejsza porazka
// wynik: Falsifiable, after 10 tests (2 shrinks)
```

Narzędzie jest bardzo przydatne do określenia, gdzie znajdują się granice błędów w testowaniu Shrink działa na customowych typach złożonych, dodatkowo można też generować własne sekwencje oraz zasady w jaki sposób przeprowadzać customowe shrinkowanie.

Listing 10: Shrinkowanie ciągu znaków

```
Arb.shrink "abcd" |> Seq.toList
// ["bcd"; "acd"; "abd"; "abc"; "abca"; "abcb"; "abcc"; "abad"]
```

Konfiguracja

Czasem może zaistnieć potrzeba własnego dostosowania liczby testów itp. W tym celu można odpowiednio skonfigurować narzędzie:

Listing 11: Dostosowanie konfiguracji testów

```
let config = {
    Config.Quick with
        MaxTest = 1000
    }
    Check.One(config,isSmallerThan80 )
    // result: Ok, passed 1000 tests. (a nie powinno :)

    let config = {
    Config.Quick with
        MaxTest = 10000
    }
    Check.One(config,isSmallerThan80 )
    // result: Falsifiable, after 8660 tests (1 shrink):
    //         80
```

Warunki wstępne

Listing 12: Dodawanie warunków wstępnych

```
let preCondition x y =
    (x,y) <> (0,0)
    && (x,y) <> (2,2)

let additionIsNotMultiplication_withPreCondition x y =
preCondition x y ==> additionIsNotMultiplication x y
// Ok, passed 100 tests.
```

Jak widać, tego rodzaju rozwiązania mogą być użyte tylko w nielicznych przypadkach, gdy możemy zdefiniować niewielką liczbę "wyjątków od reguły".

Testowanie kilku właściwości na raz

W celu zapewnienia uporządkowania i ustrukturyzowania kodu istnieje możliwość testowania kilku właściwosci równocześnie.

Listing 13: Sprawdzanie wielu właściwości jednocześnie

```
type AdditionSpecification =
    static member ``Commutative`` x y =
      commutativeProperty x y
    static member ``Associative`` x y z =
      associativeProperty x y z
    static member ``Left Identity`` x =
      leftIdentityProperty x
    static member ``Right Identity`` x =
      rightIdentityProperty x

Check.QuickAll<AdditionSpecification>()

--- Checking AdditionSpecification ---
AdditionSpecification.Commutative-Ok, passed 100 tests.
```

```
15 AdditionSpecification.Associative—Ok, passed 100 tests.
16 AdditionSpecification.Left Identity—Ok, passed 100 tests.
17 AdditionSpecification.Right Identity—Ok, passed 100 tests.
```

5 Podsumowanie

TU JEST PODSUMOWANIE

Spis rysunków

Spis listingów