POLITECHNIKA
GDAŃSKA

Property based testing

Jak testować zachowanie, a nie przypadki

Paulina Brzęcka Marek Borzyszkowski

15 grudnia 2024

Wstęp

Istnieje wiele koncepcji testowania oprogramowania, jedną z nich jest testowanie na podstawie właściwości [1].

Wstęp

Istnieje wiele koncepcji testowania oprogramowania, jedną z nich jest testowanie na podstawie właściwości [1]. Dziś dowiecie się:

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.

Testowanie na przykładach

- Testowanie funkcji na podstawie konkretnych wartości wejściowych i oczekiwanych rezultatów.

Kod testujący

```
[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...

```
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, dopóki test nie będzie zielony :)

Ograniczenia podejścia example-based

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

Automatyczne generowanie danych

```
[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)
```

Testowanie na podstawie właściwości

- Zamiast testować pojedyncze przypadki, definiujemy ogólne właściwości, które funkcja powinna spełniać.
- Narzędzia, np. `FsCheck`, generują dane wejściowe i sprawdzają właściwości.

Przykładowe właściwości:

- Przemienność.
- Dodanie liczby 1 dwukrotnie jest równe dodaniu liczby 2 raz.
- Dodanie liczby 0 nie zmienia wartości liczby.

```
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
```

Pozostałe właściwości

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

[Test]
let addOneTwiceIsSameAsAddTwo() =
    Check.Quick add1TwiceIsAdd2Property

let identityProperty x _ =
    let result1 = x |> add 0
    result1 = x

[Test]
let addZeroIsSameAsDoingNothing() =
    Check.Quick identityProperty
```

Zalety podejścia property-based

- Większa pewność poprawności implementacji.
- Lepsze zrozumienie wymagań i istoty problemu.
- Automatyczne generowanie danych pozwala na testowanie różnych scenariuszy.

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

1. Różne ścieżki, ten sam wynik

Strategie

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

1. Różne ścieżki, ten sam wynik
2. Tam i z powrotem

Strategie

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

1. Różne ścieżki, ten sam wynik
2. Tam i z powrotem
3. Są rzeczy niezmienne

Strategie

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

1. Różne ścieżki, ten sam wynik
2. Tam i z powrotem
3. Są rzeczy niezmienne
4. Z czasem rzeczy są niezmienne

Strategie

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

1. Różne ścieżki, ten sam wynik
2. Tam i z powrotem
3. Są rzeczy niezmienne
4. Z czasem rzeczy są niezmienne
5. Dziel i rządź

Strategie

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

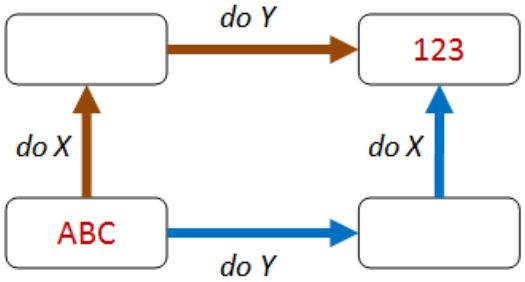
1. Różne ścieżki, ten sam wynik
2. Tam i z powrotem
3. Są rzeczy niezmienne
4. Z czasem rzeczy są niezmienne
5. Dziel i rządź
6. Łatwiej zweryfikować niż zaimplementować

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

1. Różne ścieżki, ten sam wynik
2. Tam i z powrotem
3. Są rzeczy niezmienne
4. Z czasem rzeczy są niezmienne
5. Dziel i rządź
6. Łatwiej zweryfikować niż zaimplementować
7. Testowanie z wyrocznią

Strategie - Różne ścieżki, ten sam wynik

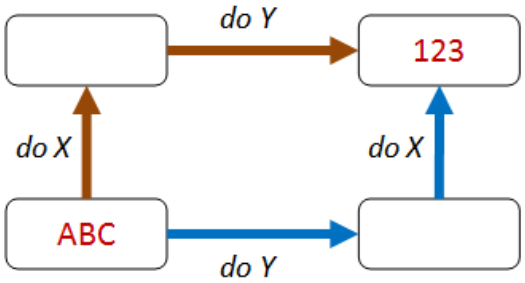
Jedną z podstawowych strategii skorzystanie z komutatywności niektórych operacji. Można to zrobić poprzez wykonanie operacji w różnej kolejności.



Rysunek: Strategia - komutatywność

Strategie - Różne ścieżki, ten sam wynik

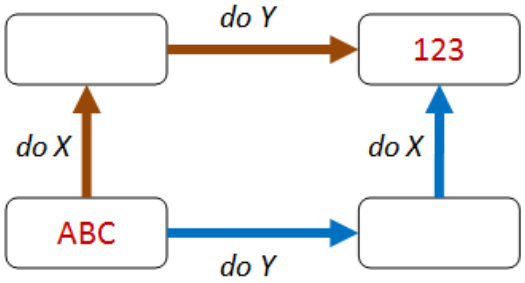
Jedną z podstawowych strategii skorzystanie z komutatywności niektórych operacji. Można to zrobić poprzez wykonanie operacji w różnej kolejności.
Przykładem takiej strategii może być komutatywność dodawania, gdzie wykorzystano `add x y`, jak i odwrotność tej operacji `add y x`.



Rysunek: Strategia - komutatywność

Strategie - Różne ścieżki, ten sam wynik

Jedną z podstawowych strategii skorzystanie z komutatywności niektórych operacji. Można to zrobić poprzez wykonanie operacji w różnej kolejności.
Przykładem takiej strategii może być komutatywność dodawania, 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.

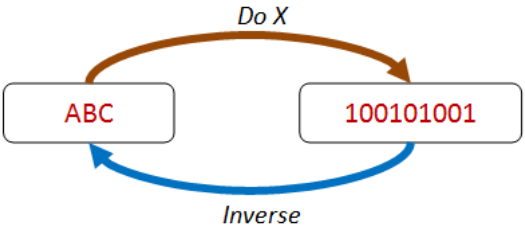


Rysunek: Strategia - komutatywność

Strategie - Tam i z powrotem

Przykładem takiego testu mogą być przeciwne operacje matematyczne jak:

- `dodawanie/odejmowanie`

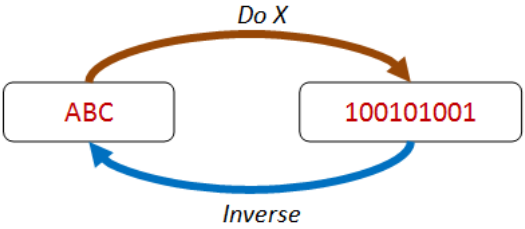


Rysunek: Strategia - inwersja

Strategie - Tam i z powrotem

Przykładem takiego testu mogą być przeciwne operacje matematyczne jak:

- `dodawanie/odejmowanie`
- `mnożenie/dzielenie`

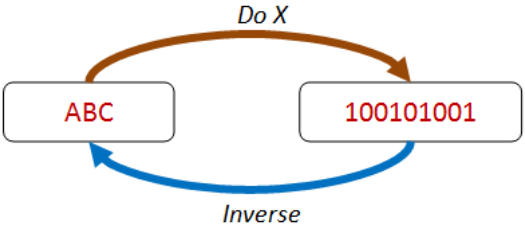


Rysunek: Strategia - inwersja

Strategie - Tam i z powrotem

Przykładem takiego testu mogą być przeciwne operacje matematyczne jak:

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



Rysunek: Strategia - inwersja

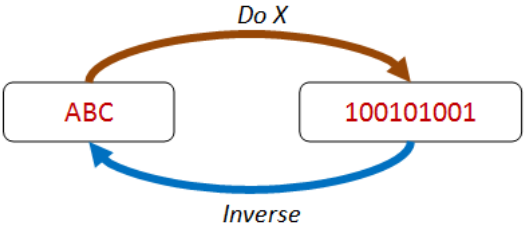
Strategie - Tam i z powrotem

Przykładem takiego testu mogą być przeciwne operacje matematyczne jak:

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

Innymi przykładami są operacje niekoniecznie matematyczne:

- `serializacja/deserializacja`



Rysunek: Strategia - inwersja

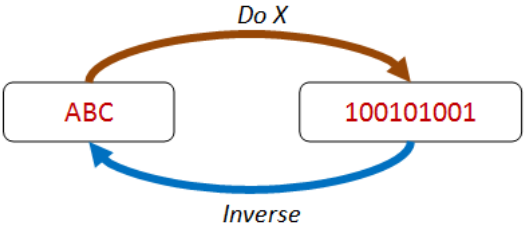
Strategie - Tam i z powrotem

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`



Rysunek: Strategia - inwersja

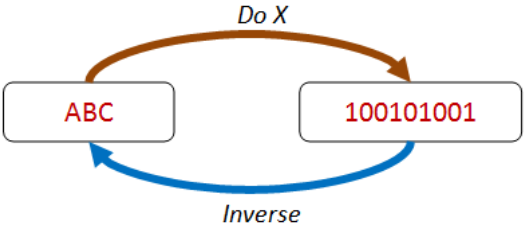
Strategie - Tam i z powrotem

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`.



Rysunek: Strategia - inwersja

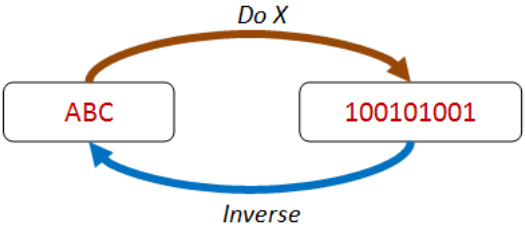
Strategie - Tam i z powrotem

Przykładem takiego testu mogą być przeciwne operacje matematyczne jak:

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

Innymi przykładami są operacje niekoniecznie matematyczne:

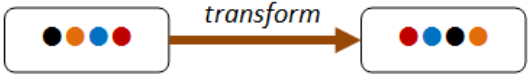
- `serializacja/deserializacja`
- `zapis/odczyt z pliku`
- `wstaw/sprawdź czy zawiera`.
- `odwrócenie listy/odwrócenie listy`



Rysunek: Strategia - inwersja

Strategie - Są rzeczy niezmienne

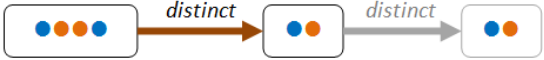
Czasami testowana funkcja przetwarzając dane zachowuje część ich właściwości. Chociażby funkcje `sort` lub `map` wykonane na liście `n` elementów, zwracaja odpowiednio zmodyfikowaną listę `n` elementową.



Rysunek: Strategia - niezmienność

Strategie - Z czasem rzeczy są niezmienne

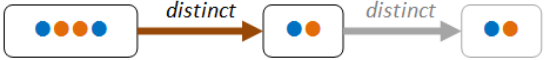
Inną właściwością funkcji może być niezmienność wyniku funkcji po ponownym jej zaaplikowaniu. Innymi słowy, wykonanie funkcji 2 razy daje taki sam efekt, jak jednokrotne jej zaaplikowanie.



Rysunek: Strategia - idempotentność

Strategie - Z czasem rzeczy są niezmienne

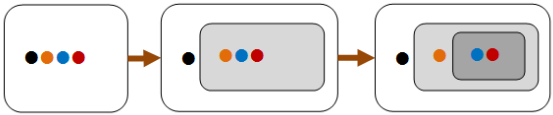
Inną właściwością funkcji może być niezmienność wyniku funkcji po ponownym jej zaaplikowaniu. Innymi słowy, wykonanie funkcji 2 razy daje taki sam efekt, jak jednokrotne jej zaaplikowanie. Przykładami takich operacji, dla których taki typ testu miałby zastosowanie to metoda `distinct` wykonana na danej liście, lub wykonanie `update` na danej bazie danych.



Rysunek: Strategia - idempotentność

Strategie - Dziel i rządź

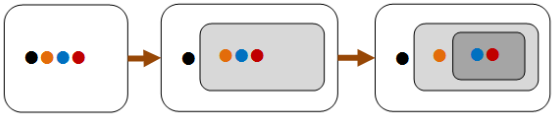
Istnieją sposoby na testowanie na podstawie właściwości jest wykorzystanie rekursywności struktur przekazywanych do funkcji, takich jak `listy`, `drzewa`.



Rysunek: Strategia - rekursywna

Strategie - Dziel i rządź

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



Rysunek: Strategia - rekursywna

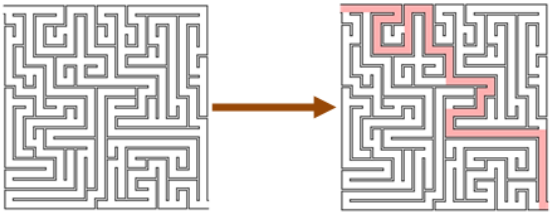
Strategie - Dziel i rządź

```
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
```

POLITECHNIKA GDAŃSKA

Strategie - Łatwiej zweryfikować niż zaimplementować

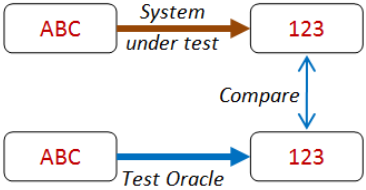
Niekiedy testowana funkcja jest skomplikowana, ale jej rezultat da się łatwo sprawdzić. Przykładem może być funkcja wyszukująca wyjście z labiryntu, gdzie sam algorytm wyszukiwania odpowiedniej ścieżki jest skomplikowany, natomiast samo sprawdzenie, czy ścieżka dobrze prowadzi do wyjścia można w łatwy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



Rysunek: Strategia - łatwe sprawdzenie

Strategie - Testowanie z wyrocznią

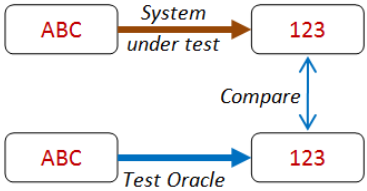
Zdaża się, że funkcjonalność została już napisana i trzeba ją zrefactorować, przepisać, napisać od nowa. Warto wtedy wykorzystać wartości zwracane przez oryginalnie zaimplementowany algorytm jako pewną wartość wyniku, pewnego rodzaju wyrocznię, uznając go jako prawdę.



Rysunek: Strategia - wyrocznia

Strategie - Testowanie z wyrocznią

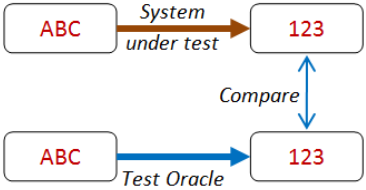
Zdaża się, że funkcjonalność została już napisana i trzeba ją zrefactorować, przepisać, napisać od nowa. Warto wtedy wykorzystać wartości zwracane przez oryginalnie zaimplementowany algorytm jako pewną wartość wyniku, pewnego rodzaju wyrocznię, uznając go jako prawdę. W taki sposób można sprawdzić, czy 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.



Rysunek: Strategia - wyrocznia

Strategie - Testowanie z wyrocznią

Zdaża się, że funkcjonalność została już napisana i trzeba ją zrefactorować, przepisać, napisać od nowa. 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.



Rysunek: Strategia - wyrocznia

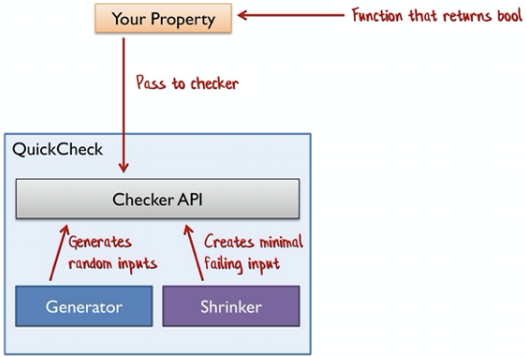
QuickCheck został stworzony w języku Haskell jako pierwsze narzędzie wspierające testowanie oparte na właściwościach. Jest inspiracją dla bibliotek w innych językach, takich jak FsCheck dla .NET.

Cechy QuickCheck

- Automatyczne generowanie danych testowych.
- Sprawdzanie, czy zdefiniowane właściwości funkcji są spełniane dla wielu losowych przypadków.
- Redukcja (*shrinking*) danych wejściowych w przypadku błędu.

Schemat działania QuickCheck

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.



Rysunek: Schemat działania QuickCheck

Funkcje QuickCheck

- **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.

- Odwrócenie listy dwukrotnie powinno zwrócić pierwotną listę.

```
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.

- Tworzenie generatora.
- Generowanie danych testowych dla różnych struktur.
- Wykorzystanie arbitrażu (Arb) do definiowania generatorów.

Generatory

```
let intGenerator = Arb.generate<int>
Gen.sample 100 3 intGenerator  // [-37; 24; -62]
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
let intListGenerator = Arb.generate<int list>
Gen.sample 5 10 intListGenerator
// [ []; []; [-4]; [0; 3; -1; 2]; [1]; [1]; []; [0; 1; -2]; [];]
let stringGenerator = Arb.generate<string>
Gen.sample 10 3 stringGenerator // [""; "eiX$a^"; "U%0Ika&r"]
```

Przykład generatora dla złożonych struktur

```
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

Proces redukcji danych wejściowych, który minimalizuje przypadki testowe prowadzące do błędu.

- Redukcja liczb do coraz mniejszych wartości.
- Redukcja ciągów znaków.

Shrinkowanie

```
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)
```

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
```

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.
```

Testowanie 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.
AdditionSpecification.Associative-Ok, passed 100 tests.
AdditionSpecification.Left Identity-Ok, passed 100 tests.
AdditionSpecification.Right Identity-Ok, passed 100 tests.
```

Pytania?

Bibliografia I

[1] S. Wlaschin, “The ”Property Based Testing” series”,, s. 6, grud. 2014. adr.: https://fsharpforfunandprofit.com/posts/property-based-testing/.

Podziękowania

Chcielibyśmy podziękować Panu dr. inż. Janowi Cychnerskiemu za stworzenie i udostępnienie stylu *pg-beamer*, co zostało wykorzystane do stworzenia tej prezentacji.
`https://github.com/jachoo/pg-beamer`

Dziękujemy za uwagę!

POLITECHNIKA
GDAŃSKA