11
Property-based Testing

We have previously gone through unit testing, which sometimes can be described more as a design and development tool than an actual testing tool. We have gone through integration testing, which provides tests to see whether the code actually works with external interfaces such as databases, web services, and filesystems. Then, we have functional testing, where we use tests as specifications of the system and have them provide regression, which tells us whether the system changes its behavior between code changes.

In the first chapter, we quickly touched the subject of manual testing. The purpose of manual testing is to challenge the system and explore how the system actually works. In this, manual testing and property-based testing have the same purpose but with widely different approaches.

In property-based testing, you use properties to describe facts about your system, which should hold true for any universal value. All the given values belongs to a domain where the property holds true.

$$\forall i \in Z$$

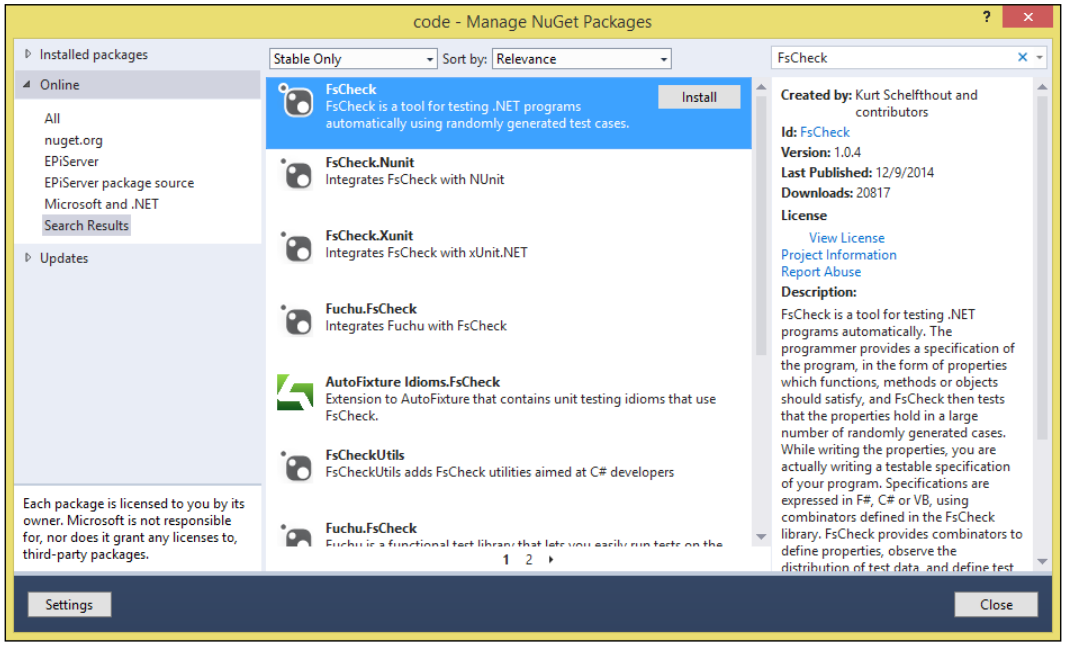
Contrary to unit tests, the property is not an effect of the implementation but rather an exploratory statement about the function. It is not a specification as in functional testing and it is not a verification of the actual function. Instead, it could be considered an experiment that you run in order to find edge cases where your code is not doing what was intended.

Property-based testing is about writing down our assumptions about the system and exploring these assumptions with data—lots of data.

Getting started with FsCheck

In order to get started with property-based testing on F#, we're going to utilize a framework called FsCheck. This is a conversion of the popular QuickCheck on the **Haskell** platform. It can simply be described as a testing framework that takes your defined properties, which describe some truths about your system and send generated data to test these truths. If the property holds for the whole sequence of generated data, then the test passes. Otherwise, you will be presented by a counter example when the property is not true.

Let's add **FsCheck** from **NuGet Package Manager** in order to start testing:



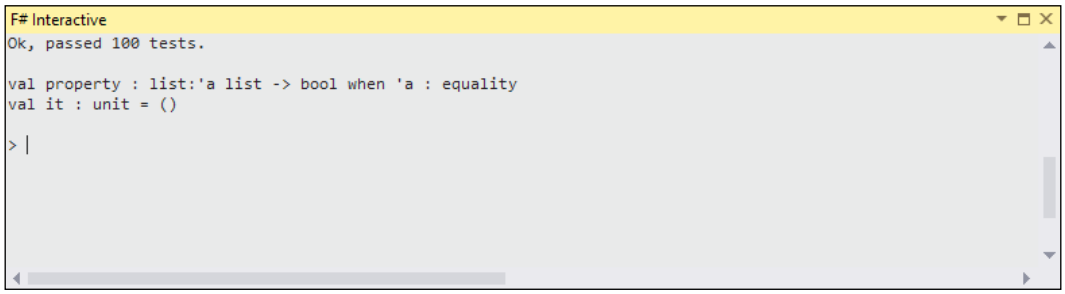
The easiest way to start playing with this is to create an F# script file (`fsx`) in your project and write your code and tests from there.

Right-click on the reference to FsCheck and **Send to F# Interactive**. This ensures that FSI knows about FsCheck and you can execute your tests there.

Write the following code in the text editor and **Send to F# Interactive**:

```
open FsCheck
let property list = list = List.rev (List.rev list)
Check.Quick property
```

You will get the following output in the **F# Interactive** window:



This is the hello world equivalent in FsCheck. This means that we have the test framework up and running, but the code and its output will need some explaining.

A property is a function that takes one argument and returns `bool` or `unit` arguments. The Boolean value returns the `true` parameter if the test is a success and the `false` parameter if it's not. A property returning unit will only fail when an exception is thrown:

```
let property list = list = List.rev (List.rev list)
```

This property states that reversing a list twice will result in the original list, and this is true for every list. This is our stated fact about the system. We execute tests on this property with the following command:

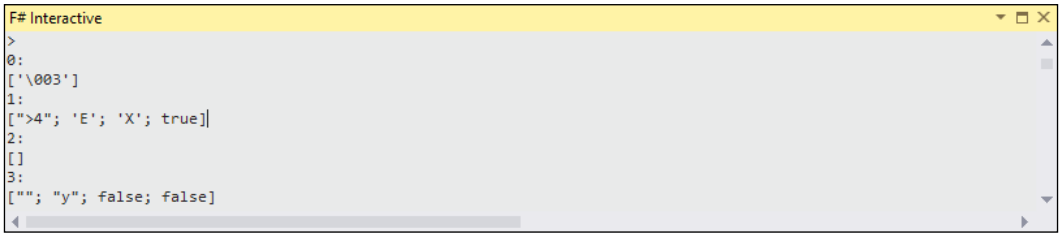
```
Check.Quick property
```

This statement will take 100 generated lists and input into the property and expect to get a true result from each of the 100 lists. The output in F# Interactive clearly states that the property has passed these 100 tests.

If we want to see the test data, we can run the following command instead:

```
Check.Verbose property
```

It will output all the different tests that were executed. This is not very interesting to us unless one of them fails:

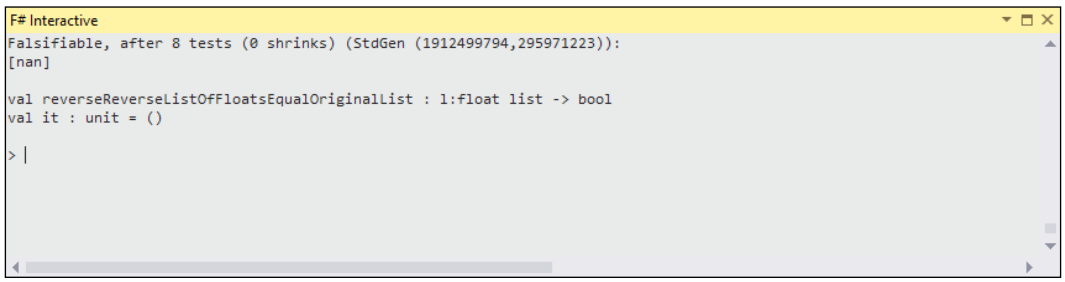


The reverse of a list is not always equal to the original list. A list of floating point numbers may contain infinity or `nan`, and because `nan <> nan`, the comparison will return a `false` parameter.

Here is an example:

```
let reverseReverseOfFloatsEqualOriginalList (l : float list) =
    l = List.rev (List.rev l)

Check.Quick reverseReverseListOfFloatsEqualOriginalList
```

Running this in **F# Interactive** will give the following output:



After eight tests, FsCheck finds that the `[nan] = List.rev (List.rev [nan])` command equals `false` and fails the test. Now we can take this newly found information and use it to improve our system under test.

Sorting out sorting

We define properties that explore given facts about our system. This is a different way of thinking compared to our previous methods of testing. It's a way of thinking that is closer to manual testing, where we explore and challenge our system's boundaries.

Let's look at a closer to real-world scenario. Here is an implementation of `selection sort`:

```
// remove first occurance of item from list
let rec remove item = function
| [] -> []
| hd :: tl when hd = item -> tl
| hd :: tl -> hd :: (remove item tl)

// naive implementation of selection sort
let rec sort = function
```

```
    | [] -> []
    | l ->  let value = (List.min l)
            value :: (sort (l |> remove value))
```

I chose `selection sort` as an example because it is very easy to understand and implement. It works by removing the smallest value from the input list and placing it last in the return list until the input list is empty. This way, the result list will come out sorted.

Here is how we will implement a unit test for this `sort()` function:

```
[<Test>]
let ``should sort [1; 3; 5; 3; 1] as [1; 1; 3; 3; 5]`` () =
    sort [1; 3; 5; 3; 1] |> should equal [1; 1; 3; 3; 5]
```

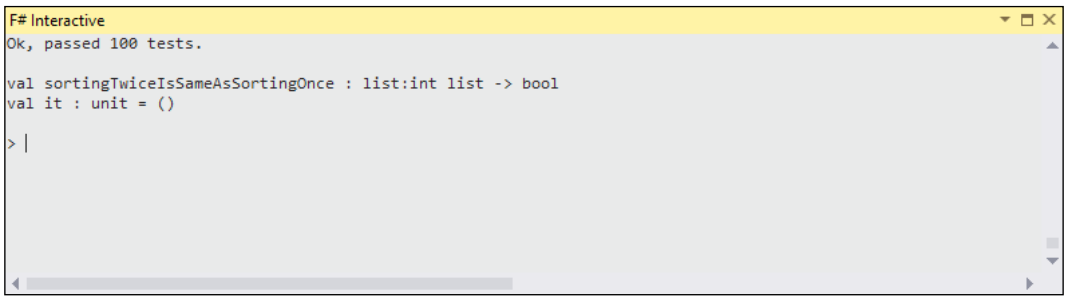
We could be happy about this test and claim that it covers all the lines of code in the system under test. When it turns green, the SUT must work. This false sense of security is one of the dangers of thinking about coverage. What we need in this case is to test some assumptions of the system with different kinds of data.

What kinds of properties will we define for this code?

One good place to start is finding the **idempotency** property. This is a property of a pure function, meaning applying the function twice is the same as applying it once. This is very easy in our case, as sorting a sorted list would result in the original result:

```
// property
let sortingTwiceIsSameAsSortingOnce (list : int list) =
    (sort list) = sort (sort list)

Check.Quick sortingTwiceIsSameAsSortingOnce
```

Running this, we will get a confirmation that this property is true for 100 datasets:



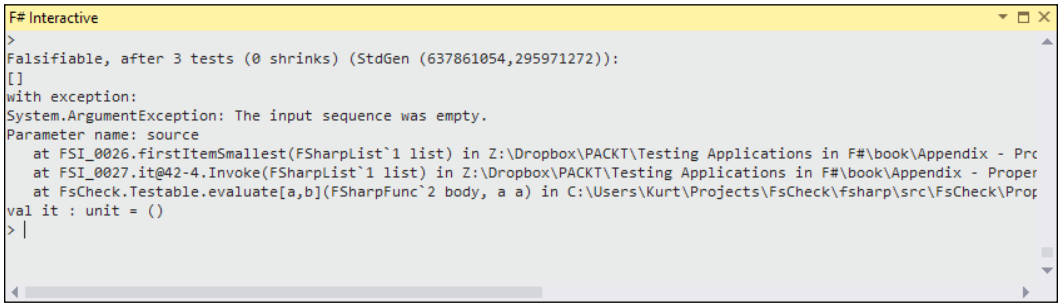
Awesome! What other properties of a sort algorithm can we come up with? Here are some of them:

- The first item should always be the smallest in the result
- The last item should always be the largest in the result
- The result should be a permutation of the input result
- The result should always be ordered

Finding out whether the first item in the list is always the smallest one should be a simple task. Consider the following implementation:

```
let firstItemSmallest (list : int list) =
    (List.min list) = (List.head (sort list))

Check.Quick firstItemSmallest
```

This does, however, fail on the third dataset because the property can't handle empty lists:



We need to add a condition to the property so that it is not run with empty lists. For this, we have a conditional operator:

```
<condition> ==> <property>
```

With this knowledge, we can now implement these two properties:

```
let firstItemSmallest (list : int list) =
    (not list.IsEmpty) ==>
        lazy(List.min list = List.head (sort list))

Check.Quick firstItemSmallest

let lastItemLargest (list : int list) =
    (not list.IsEmpty) ==>
        lazy(List.max list = List.head (List.rev (sort list)))

Check.Quick lastItemLargest
```

The property here is marked as `lazy` because we do not want the property to be evaluated before the condition. This is important as the property will throw an exception on empty lists.

Check whether the sorted list is a permutation of the original list:

```
// lists are of equal length
// all elements in sorted list exists in original list
let permutationOf (list : int list) =
    let exists item = List.exists ((=) item)
    let sorted = sort list

    (sorted.Length = list.Length)
        |@ "same length" .&.
    (sorted |> List.forall (fun x -> list |> (exists x)))
        |@ "all elements exists"

Check.Quick permutationOf
```

Here, we use the AND operator to include a separate assumption in the property and give each fact a label. The syntax for this looks as follows:

```
<property> |@ <label> .&.
```

When a fact doesn't validate as true, we will get a direct output of what label was failing. This is very important so we know what problem we should start debugging:

```
Falsifiable, after 4 tests (0 shrinks) (StdGen
(1646695793,295971284)):
Label of failing property: all elements exists
[0]
```

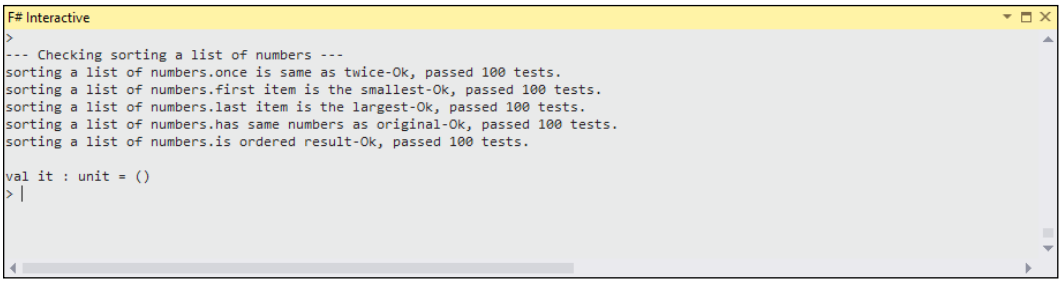
The last property we're going to explore is that the sorted list is actually ordered. For this, we make the following implementation:

```
let rec ordered (list : int list) =
    let rec _ordered = function
    | [] -> true
    | hd :: [] -> true
    | fst :: snd :: tl -> (fst <= snd) && (_ordered (snd :: tl))

    _ordered (sort list)

Check.Quick ordered
```

With these assumptions expressed as properties, we do cover the sort functionality pretty well, or at least better than we did with the unit test. To have this make any sense, we can group these properties and run them all simultaneously as they are testing the same SUT:

```
type ``sorting a list of numbers`` =
    static member ``once is same as twice`` () =
        sortingTwiceIsSameAsSortingOnce
    static member ``first item is the smallest`` () =
        firstItemSmallest
    static member ``last item is the largest`` () =
        lastItemLargest
    static member ``has same numbers as original`` () =
        permutationOf
    static member ``is ordered result`` () =
        ordered

Check.QuickAll typeof<``sorting a list of numbers``>
```

This will verify all the properties and present the result in a pleasant way:



For more advanced usages of FsCheck properties, you should visit the FsCheck project page on GitHub, where the full documentation can be found: `https://fsharp.github.io/FsCheck/`.

Generators

The example of sorting is still very simple because it requires very little configuration of the framework. We can write our properties and make it work at that.

Let's try an example where we need to look at generators. First, we have an implementation of FizzBuzz:

```
let fizzbuzz = function
| d when d % 15 = 0 -> "fizzbuzz"
```

```
    | d when d % 3 = 0 -> "fizz"
    | d when d % 5 = 0 -> "buzz"
    | d -> sprintf "%d" d
```

In FizzBuzz, you are going to write an algorithm that returns the `fizz` string value for every number divisible by 3, the `buzz` string value for every number divisible by 5, and the `fizzbuzz` string value for every number divisible by both 3 and 5.

Here's a naïve implementation of our first property:

```
let ``fizz when evenly divisible by 3`` d =
    (d % 3 = 0) ==> ("fizz" = fizzbuzz d)

Check.Quick ``fizz when evenly divisible by 3``
```

Executing this in the interactive does not return a successful result:

```
Falsifiable, after 1 test (0 shrinks) (StdGen
(1632633969,295971627)):
0
```

Of course, it fails because of the `fizzbuzz 0 = "fizzbuzz"` parameter, and this is because of the `0 % x = 0`. We can add this to our property condition:

```
let ``fizz when evenly divisible by 3`` d =
    (d > 0 && d % 3 = 0) ==> ("fizz" = fizzbuzz d)
```

We have forgotten that the `fizz` string value is the result that is returned only when the `d` parameter is not evenly divisible by 5. FsCheck finds the error in our assumption:

```
Falsifiable, after 11 tests (1 shrink) (StdGen
(562665355,295971629)):
15
```

This is the result from the `fizzbuzz 15 = "fizzbuzz"` line, and we need to modify our property to take this into account:

```
let ``fizz when evenly divisible by 3`` d =
    (d > 0 && d % 3 = 0 && d % 5 <> 0) ==> ("fizz" = fizzbuzz d)
```

Now finally it passes 100 tests, but how many of these data items actually passed the criteria? Since it seems unnecessary to generate any data points where the `d < 1` parameter, we could implement a generator that makes sure this is the case:

```
let posNumberGenerator = Gen.sized <| fun s -> Gen.choose(1, s)
```

The `Gen.choose()` function will randomly pick a number between `1` and `s` arguments. The `Gen.sized` function makes sure that this number is gradually increasing, starting low and getting bigger and bigger.

In order to use the generator, we need to bundle it with a `shrinker` function. And we do this by adding them to an `Arbitrary` instance. A `shrinker` is a function that will reduce a value once an error is found. We do not need it in this case:

```
let positiveIntegers =
    {new Arbitrary<int>() with
        override x.Generator = Gen.sized <| fun s -> Gen.choose(1, s)
        override x.Shrinker t = Seq.empty }
```

We can also use this shorthand function on the `Arbitrary` class:

```
let positiveIntegers =
    Arb.Default.Int32()
    |> Arb.mapFilter abs (fun n -> n > 0)
```

This generates the same result where we use the `Arb.mapFilter` parameter to create a `Generator` value with only positive numbers.

In order to use the `Arbitrary` instance together with our property, we need to put them in the same static class and register them. In the end, the complete setup looks like this:

```
type FizzBuzz =
    // positive integers generator
    static member positiveIntegers =
        Arb.Default.Int32()
        |> Arb.mapFilter abs (fun n -> n > 0)

    // property
    static member ``fizz when evenly divisible by 3`` d =
        (d % 3 = 0 && d % 5 <> 0) ==> ("fizz" = fizzbuzz d)

    // property
    static member ``buzz when evenly divisible by 5`` d =
        (d % 5 = 0 && d % 3 <> 0) ==> ("buzz" = fizzbuzz d)

    // property
    static member ``fizzbuzz when evenly divisible by 3 and 5`` d =
        (d % 3 = 0 && d % 5 = 0) ==> ("fizzbuzz" = fizzbuzz d)

Arb.registerByType (typeof<FizzBuzz>)
Check.QuickAll (typeof<FizzBuzz>)
```

If you want to verify that only positive integers are being used, you can run the tests with the `Check.VerboseAll` command instead of the `Check.QuickAll` command. You will get the full test output.

The output from these tests is a bit interesting:

```
--- Checking FizzBuzz ---
FizzBuzz.fizz when evenly divisible by 3-Ok, passed 100 tests.
FizzBuzz.buzz when evenly divisible by 5-Ok, passed 100 tests.
FizzBuzz.fizzbuzz when evenly divisible by 3 and 5-Arguments exhausted
after 59 tests.
```

In the last property, arguments were exhausted after 59 tests. This means that after 59 tests, FsCheck couldn't find any more test cases that would satisfy the condition of the property. In order to avoid infinite looping, the test run breaks and reports that 59 test cases were a success before the test run was aborted.

You can use the `config` parameter's `MaxTest` to reduce the number of test cases and avoid the error message:

```
Check.All({ Config.Quick with MaxTest = 50 }, (typeof<FizzBuzz>))
```

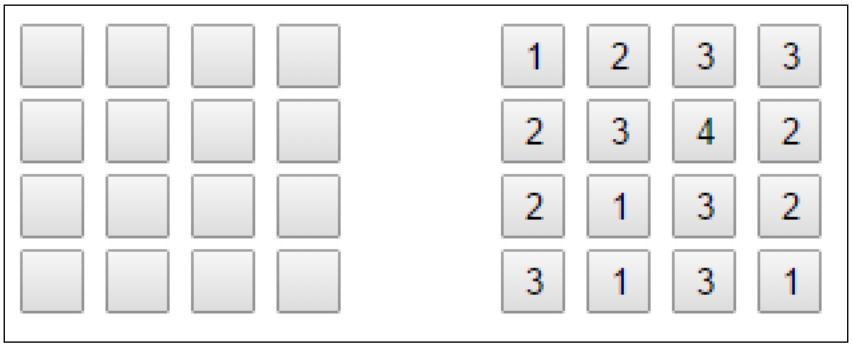
You can read more about generators in the FsCheck documentation on GitHub:

`https://fsharp.github.io/FsCheck/TestData.html`.

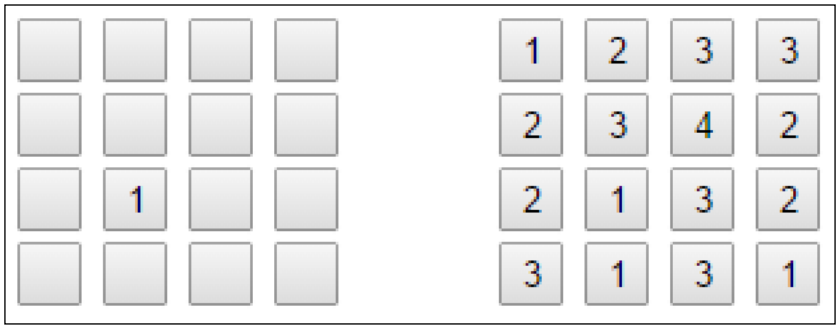
Model-based testing

When I was young, I played a game called Brains. I still don't know who its original creator was, but the puzzle game was so mind-bending that it kept me occupied for hours, working out rules on how to beat the game in the fastest possible time.

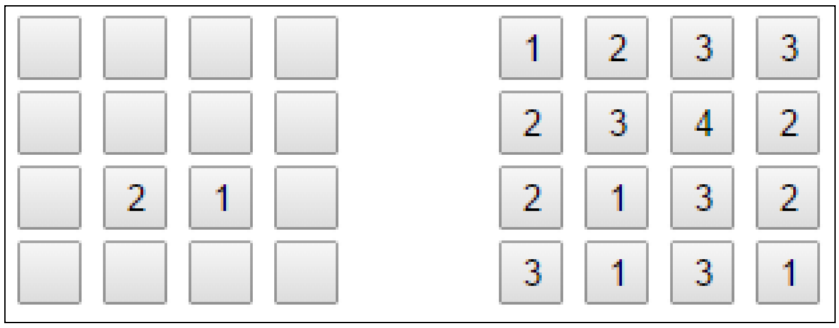
The basic concept of this game is so simple you will understand it in 1 minute:



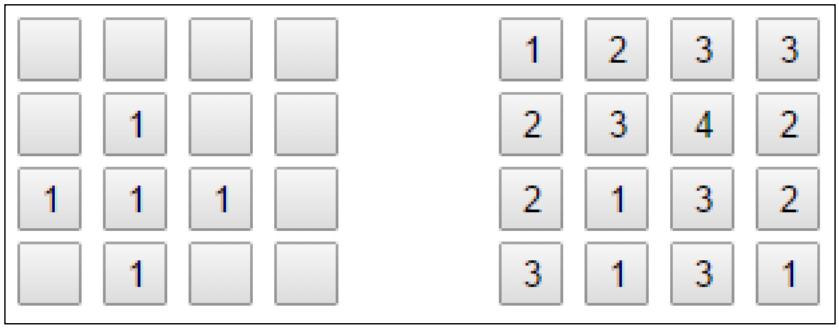
The task is to click on the buttons on the left-hand side so that you can replicate the button grid to the right. Each time you click on a button, you increase its value by 1, but a button can only be clicked once:



Each time you click on a button next to a button that has already been clicked, the button will be increased by 1:



If a button has its value increased beyond 4, it will overturn and become **1**:



This is all you need to know about the game. You can put a timer on it and record a high score. Define different grid sizes to have varying difficulties. Now we're going to look at how we can test this using model-based testing, which is an extended concept of property-based testing.

Let's start with defining the model for a single button:

```
type Button (value : int) =
    new () = Button(0)
    member __.Click () = Button(value + 1 % 4)
    member __.Value = value
    override __.ToString () = value.ToString()
```

I have sneaked in an error to make this exercise a bit more interesting.

In order to test this model, we need to use an extension to FsCheck called `Command`. This works by implementing the `ISpecification<'T>` parameter and the `ICommand<'T>` parameter for each property that we want to test:

```
let spec =
    let click =
        { new ICommand<Button,int>() with
            member x.RunActual button = button.Click()
            member x.RunModel value = (value % 4) + 1
            member x.Post (button, value) = value = button.Value |>
Prop.ofTestable
            override x.ToString() = "click" }

    { new ISpecification<Button,int> with
      member x.Initial() = (new Button(), 0)
      member x.GenCommand _ = Gen.elements [click] }

Check.Quick (asProperty spec)
```

When executing this, FsCheck will find our bugs for us:

```
Falsifiable, after 10 tests (4 shrinks) (StdGen (57267399,295971699)):
[click; click; click; click; click]
```

Clicking on the button five times will expose the error. Let's try it:

```
> (new Button()).Click().Click().Click().Click().Click();;
val it : Button = 5 {Value = 5;}
```

One of the rules was that a button could never increase in a value beyond 4; it would overturn and become `1`. This is, of course, because of my deliberate error in the model. Let's fix it:

```
member __.Click () = Button((value % 4) + 1)
```

Now the test passes with the following report:

```
Ok, passed 100 tests.
77% long sequences (>6 commands).
17% short sequences (between 1-6 commands).
3% trivial.
```

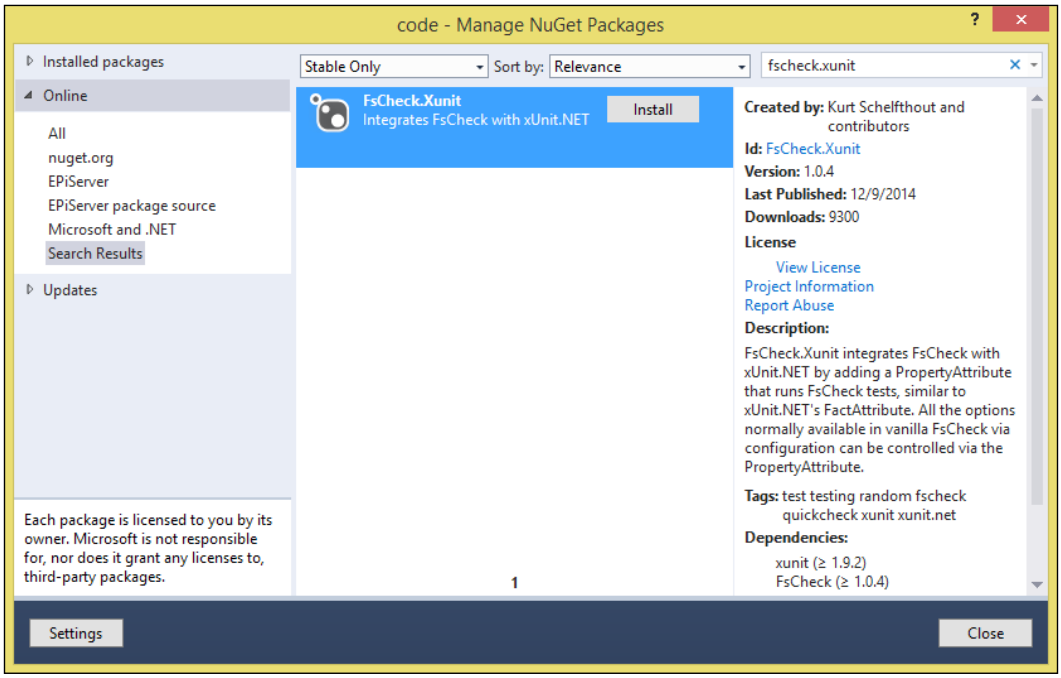
From here, you can try to implement the complete board and use model-based testing to state some assumptions that you'd like to try out.

Integration with unit testing frameworks

So far, we have only used FsCheck from an F# script file (`fsx`) and fed it into F# Interactive. This is great for experimenting and discovery, but not so good when you want to formalize your assumptions and have them repeatedly verified, that is, by continuous integration.

In order to achieve this, we can utilize our existing testing frameworks, where FsCheck has extensions for both NUnit and xUnit and the support for the latter is greatest.

Start with adding **FsCheck.xUnit** from **NuGet Package Manager** to the project:

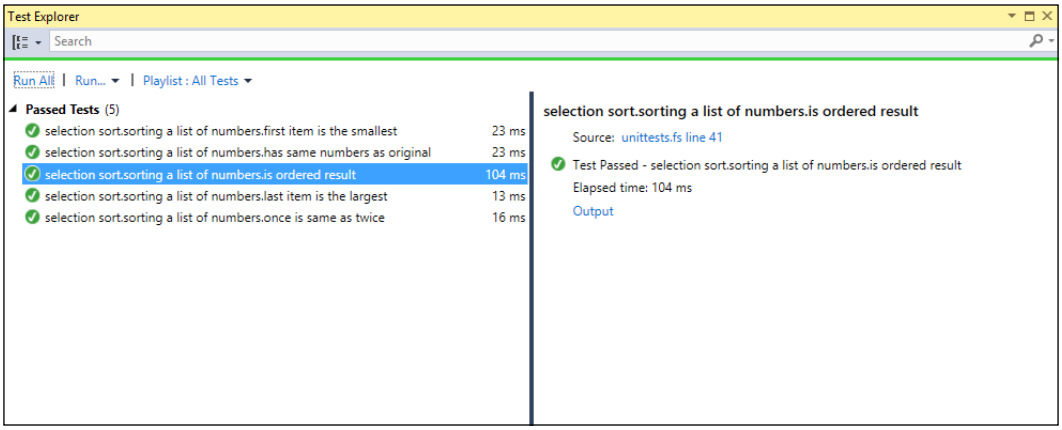


We can now implement properties as unit tests using the `[<Property>]` attribute. This will allow the test to act as a property and receive an argument to its function.

Here are our properties, for selection-sort-implemented, as unit tests:

```
module ``sorting a list of numbers`` =

    open FsCheck
    open FsCheck.Xunit

    [<Property>]
    let ``once is same as twice`` (list : int list) =
        (sort list) = sort (sort list)

    [<Property>]
    let ``first item is the smallest`` (list : int list) =
        (not list.IsEmpty) ==>
            lazy(List.min list = List.head (sort list))

    [<Property>]
    let ``last item is the largest`` (list : int list) =
        (not list.IsEmpty) ==>
            lazy(List.max list = List.head (List.rev (sort list)))

    [<Property>]
    let ``has same numbers as original`` (list : int list) =
        let exists item = List.exists ((=) item)
        let sorted = sort list

        (sorted.Length = list.Length)
            |@ "same length" .&.
        (sorted |> List.forall (fun x -> list |> (exists x)))
            |@ "all elements exists"

    [<Property>]
    let rec ``is ordered result`` (list : int list) =
        let rec _ordered = function
        | [] -> true
        | hd :: [] -> true
        | fst :: snd :: tl -> (fst <= snd) && (_ordered (snd :: tl))

        _ordered (sort list)
```

The `[<Property>]` attribute is a subclass of the xUnit Fact attribute, which makes it discoverable by the **Test Explorer** in Visual Studio, as long as you have the xUnit Visual Studio Runner installed.



The same functionality is available to you if you choose to use NUnit as the testing framework, but with a few limitations on how you can configure FsCheck from the `Property` attribute.

Summary

Property-based testing is an interesting approach to testing where you, as a programmer, does not control what the computer does to verify the functionality. All you do is describe the properties or assumptions about your system and what should be true independent of the values you throw at it.

It is a method of exploring what your system does instead of verifying it straight up. You will find edge cases that the code you're writing doesn't take care of, and will be able to fill those gaps.

I think that property-based testing is more suitable to algorithmic kind of functionality, but not exclusively so. Wherever there is crunching of data, there will be a need for FsCheck.