Software Testing, Quality Assurance and Maintenance Winter 2026

Lecture 12 — February 13, 2026

Patrick Lam *version 1*

The last topic for the first half of the course is property-based testing. In some ways, property-based testing overlaps with fuzzing, and also with metamorphic testing. It turns out that property-based testing appears in a first-year course[1] at the University of Toronto, though somewhat superficially. We know more than first-year students, so we're going to go into a lot more detail.

I couldn't find a clean definition of property-based testing. It is begging the question to say that it is testing using properties. But that's what it is.

The question, then, is what kind of properties? Various references say that you need to specify the "shape" of inputs that are "interesting"[2]; or that you have to know the system well enough to specify properties of its abstractions[3]; or that property-based testing triggers failures that could not have been revealed by direct fuzzing[4].

Another answer[5] is that properties are things that are true for *any* correct implementation.

OK, so I can't really say what it is. Let's try saying what it does. Maybe that might bring some insight. In property-based testing, we write test cases that verify system properties, ideally deep properties. Furthermore, property-based tests usually use a fuzzer to generate inputs that test these properties. OK, here's an example. It'll remind you of metamorphic testing.

The `map()` function takes a function and a list, and applies the function to every element of the list. In imperative-style Python, we can write:

```
1 def map(fn, xs):
2   ys = []
3   for x in xs:
4     ys.append(fn(x))
5   return ys
```

and so, as we might expect, `list(map(lambda x: x+1, [3, 4]))` returns the list `[4, 5]`.

Apparently a popular example, then, is the property that passing `map` the identity function $\lambda x.\ x$ and a list ℓ will give you back the same list ℓ. We can write this as a property test using the Python Hypothesis library.

```
1 @given(st.lists(st.integers()))
2 def map_identity_yields_self(xs):
3     id = lambda x: x
4     assert list(map(id, xs)) == xs
```

[1] https://www.cs.toronto.edu/~david/course-notes/csc110-111/04-function-specification-and-correctness/04-testing-functions-2.html

[2] https://www.tedinski.com/2018/12/11/fuzzing-and-property-testing.html

[3] https://www.tedinski.com/2018/04/24/design-and-property-tests.html

[4] https://hypothesis.works/articles/what-is-property-based-testing/

[5] https://fsharpforfunandprofit.com/posts/property-based-testing/

The body of the test looks like a normal test: there is an arrange part, where we set up `id`. The act part happens in the assert body, which could be bad style, but is OK here, especially since we're not modifying any state.

What's unusual about this test is that it's not concrete. Property-based test people call concrete tests example-based tests. Here, we just have a parameter `xs`. We signal to Hypothesis that this is a property-based test with the `@given` decorator, and we can invoke the test:

```
>>> map_identity_yields_self()
```

Hypothesis intercepts calls to functions decorated with `@given`, picks values for the specified list of integers, and invokes the function repeatedly with different values. The mechanics of these calls are similar to fuzzing, but the purpose is to ensure that the properties specified by assertions hold. Specifically, here is some F# code that does what Hypothesis does:

```
let propertyCheck property =
  // property has type: int -> int -> bool
  for _ in [1..100] do
    let x = randInt()
    let y = randInt()
    let result = property x y
    Assert.IsTrue(result)
```

To check out what Hypothesis is doing, specify `@settings(verbosity=Verbosity.verbose)` as an additional decorator. Otherwise, if you don't specify the verbosity, no news is good news: if you see nothing, then Hypothesis thinks that the property holds on a bunch of inputs. When you do get a failure report, then the property testing framework will also simplify the values in the report, for the same reason that we talked about test case reduction.

You can also put calls to these property-based tests in functions with names starting with `test_` and run them with `pytest`, but then you don't get verbose output.

More List Examples

Next, a bunch of examples[6].

Sorting a List. One function that we can check using property-based testing is a sort function. In Python we'll check `sorted()`, which returns a sorted version of its input. The source I'm using talks about the Enterprise Developer From Hell (EDFH), who writes implementations that pass tests without being correct (aka malicious compliance).

One type of property that we can check is called "different paths, same destination": is it the case that doing X and then Y yields the same as doing Y and then X? Obviously, that depends on what X and Y are. Concretely, though, we can pick X as "add one to each list element" and Y as "sort", and a moment's thought should convince you that should work. Even more concretely:

$$\texttt{list}(\texttt{map}(\texttt{add\_one}, \texttt{sorted}([2,3,1]))) == \texttt{sorted}(\texttt{list}(\texttt{map}(\texttt{add\_one}, [2,3,1])))$$

[6] https://fsharpforfunandprofit.com/posts/property-based-testing-3/

because we have, on the left, $[2,3,1] \to [1,2,3] \to [2,3,4]$; and, on the right, $[2,3,1] \to [3,4,2] \to [2,3,4]$. (You can draw a so-called commutative diagram to illustrate this.) In Hypothesis, we can write:

```
@given(st.lists(st.integers()))
def add_then_sort_eq_sort_then_add(sort_fn, xs):
    def add1(x):
        return x + 1
    result1 = list(map(add1, sort_fn(xs)))
    result2 = sort_fn(list(map(add1, xs)))
    assert result1 == result2
```

which is quite close to what we have seen. The parameter `sort_fn` allows us to pass in the sorting function, and we can use library function `sorted`:

```
def test_add_then_sort():
    add_then_sort_eq_sort_then_add(sorted)
```

OK, so what can the EDFH do? Can they write a so-called "sort" routine that doesn't actually sort, but that does pass this test? Sure. Here's one:

```
def edfhSort1(xs):
    return xs
```

and if you run that you'll see that our property-based test passes, even though `edfhSort1` doesn't do any sorting.

Forcing a sort. OK, how do we force the function being tested to actually sort? We can put a known value into the list and see if it ends up in the right place. For instance, if we put `-inf` at the end of the list, and then sort, it should be at the beginning of the list.

```
@given(st.lists(st.integers()))
def min_value_then_sort_eq_sort_then_min_value(sort_fn, xs):
    append_then_sort = sort_fn(xs + [float('-inf')])
    sort_then_prepend = [float('-inf')] + sort_fn(xs)
    assert append_then_sort == sort_then_prepend
```

Again, we can try this with the concrete $[2,3,1]$ test; `append_then_sort` yields $[2,3,1] \to [2,3,1,\texttt{-inf}] \to [\texttt{-inf},1,2,3]$, while `sort_then_append` yields $[2,3,1] \to [1,2,3] \to [\texttt{-inf},1,2,3]$, and they match.

We find that, with this property-based test, we can detect the brokenness of `edfhSort1`:

```
>>> min_value_then_sort_eq_sort_then_min_value(edfhSort1)
Traceback (most recent call last):
   [...]
Falsifying example: min_value_then_sort_eq_sort_then_min_value(
    sort_fn=edfhSort1,
    xs=[0],
)
```

Indeed, the append-then-"sort" path on `edfhSort1` yields $[0] \to [0,\texttt{-inf}] \to [0,\texttt{-inf}]$ while the "sort"-then-prepend path yields $[0] \to [0] \to [\texttt{-inf},0]$, so we've detected a property violation.

The EDFH, though, can special-case the test which ends with `-inf`:

```python
def edfhSort2(xs):
    if xs == []:
        return []
    if xs[-1] == float('-inf'):
        return [xs[-1]] + xs[:-1]
    else:
        return xs
```

and this gets by this test case.

OK, no magic numbers. On one branch, negate then sort. On the other branch, sort, then negate, and then reverse.

```python
@given(st.lists(st.integers()))
def negate_then_sort_eq_sort_then_negate_then_reverse(sort_fn, xs):
    def negate(x):
        return x * -1

    negate_then_sort = sort_fn(list(map(negate, xs)))
    sort_then_negate_then_reverse = list(reversed(list(map(negate, sort_fn(xs)))))
    assert negate_then_sort == sort_then_negate_then_reverse

negate_then_sort_eq_sort_then_negate_then_reverse(sorted)
```

This property-based test does defeat the above tests. However, it doesn't defeat the following test.

```python
def edfhSort3(xs):
    return []
```

Let's put this aside for a while. Actually, we won't get back to it, but the original source for this lecture does get back to it.

List Reversal. We'll instead talk about verifying list reversal using property-based testing. We can use the same append/prepend property as we used for sorting.

```python
@given(st.integers(), st.lists(st.integers()))
def append_then_reverse_eq_reverse_then_prepend(rev_fn, x, xs):
    append_then_reverse = list(rev_fn(xs + [x]))
    reverse_then_append = [x] + list(rev_fn(xs))
    assert append_then_reverse == reverse_then_append

append_then_reverse_eq_reverse_then_prepend(reversed)
```

This property-based test takes a reverse function as well as an element `x` and a list `xs`. One way is to append `x` to the end of `xs` and then reverse the resulting list. The other way is to reverse `xs` and then prepend `x`. For instance, if $x = 0$, then we have $[1, 2, 3] \to [1, 2, 3, 0] \to [0, 3, 2, 1]$ one way, and $[1, 2, 3] \to [3, 2, 1] \to [0, 3, 2, 1]$; these match.

You can convince yourself, or you can try it to see, that these EDFH attempts don't get past this property-based test.

```python
def edfhReverse1(xs):
    return []

```

```
def edfhReverse2(xs):
    return xs
```

There and back again. A different type of property is about inverses. It's hard to invert sorting, but we can reverse list reversal. It turns out that the inverse of list reversal is itself list reversal.

```
@given(st.lists(st.integers()))
def reverse_then_reverse_eq_original(rev_fn, xs):
    reverse_then_reverse = list(rev_fn(list(rev_fn(xs))))
    assert reverse_then_reverse == xs

reverse_then_reverse_eq_original(reversed)
```

But bad implementations can pass too:

```
reverse_then_reverse_eq_original(lambda x: x)
```

Other examples of inverses: serialization/deserialization.

Hard to prove, easy to verify. In the source material, there's a diagram of a maze. Path-finding is harder than checking if a path works. It then verifies a string split function; in Python there is `split`. We can write a property-based test by asking hypothesis to generate a list of strings, concatenating them, splitting them, and concatenating them again.

```
@given(st.lists(st.text()))
def concat_elements_of_split_string_eq_original_string(xs):
    input_string = ",".join(xs)
    tokens = input_string.split(",")
    recombined_string = ",".join(tokens)
    assert input_string == recombined_string
```

There are a lot more example properties in the source material, but we'll stop here. Hopefully this gives you an inventory of properties that you can use.

Common Patterns

These are similar to metamorphic relations, but let's talk about them in the context of property-based testing[7]. Perhaps the main challenge with property-based testing is coming up with properties; as that webpage says,

> Everyone who sees a property-based testing tool like FsCheck or QuickCheck thinks that it is amazing but when it comes time to start creating your own properties, the universal complaint is: "What properties should I use? I can't think of any!"

We saw some examples of these properties above.

Here's a list from the page.

[7] source: `https://fsharpforfunandprofit.com/posts/property-based-testing-2/`

- Different paths, same destination: do X and then Y and check that this gives the same thing as doing Y and then X; e.g. sort, add-1.
- There and back again: do X and then do the inverse X^{-1}; e.g. serialize/deserialize.
- Some things never change: is an invariant preserved? e.g collection size/contents.
- The more things change, the more they stay the same: idempotence—doing an operation twice is the same as doing it once, e.g. deduplicating a collection.
- Solve a smaller problem first: properties based on structural induction.
- Hard to prove, easy to verify: eg maze pathfinding vs verification, factorization into primes, string tokenization vs concatenation, and literally proof derivation vs verification.
- The test oracle: compare results with those from an oracle.

Note that for the test oracle, property-based testing generates interesting inputs to feed to the system and the oracle.

Commentary

Property testing versus metamorphic testing. These ideas are pretty similar; the properties we test using property-based testing often will qualify as metamorphic relations. I would say that the difference is that in property testing, you rely on the framework to generate test cases, while in metamorphic testing, you use the metammorphic relation to generate subsequent tests.

Property testing versus fuzzing. Property testing uses fuzzing or something very much like it to generate inputs. Fuzzing benefits from implicit oracles, including invariants embedded as asserts. But there is a difference here: I'd say that fuzzing is more focussed on just creating random inputs somehow, and finding crashes, especially with respect to security-critical inputs. It does not focus on the system's properties; they are incidental. Property-based testing relies on the developer describing "interesting" inputs, which may not be the security-critical ones.

The property-based testing advocate says:

> When it comes to testing almost any property outside of security, property testing will generally be superior.

but they would say that, wouldn't they.

Property testing and system design. One last comment, from `https://www.tedinski.com/2018/12/11/fuzzing-and-property-testing.html`. The claim here is that property testing can give you a better understanding of the system design, and its invariants, as you are designing the system. It's like Test-Driven Development (TDD), but with deeper properties. Indeed, in TDD, sometimes people say to write the minimal working code that passes the test cases. With property testing, you write minimal code (the EDFH code) and then you think of properties that break this code.