Software Testing, Quality Assurance & Maintenance—Lecture 5

Patrick Lam
University of Waterloo

January 19, 2026

Part I

The Oracle Problem

What's the right answer?

cop-out: "ask a human"

Begging the Question

Taking the answer the system computes as the right answer.

(is the basis for regression testing, though)

Simple example: add

```
def add(x, y):
  return x + y
```

We all agree about the output of `add(1,1)`.

Right?

(Almost)

(What about `add("3", 5)` in JavaScript?)

High school math

Consider function `solve_quadratic()` for

$$x^2 - 2x - 4 = 0.$$

Need to read the function name and remember high school math.

Also, edge cases: no solutions;
floating-point shenanigans.

Human Oracles: source of truth

Unit level: Mainly use the function name, plus any function documentation (if it exists).

More generally: You use your human experience to say what the answer should be.

Part II

Helping Human Oracles

Sometimes there is no alternative

You may have to
ask a human.

Basis for judgment

Does the output meet the system requirements?

(Eliciting requirements not in scope for this course.)

Easy versus hard inputs

Setting: function `calculate_days_between()`.

What's the answer for:

- 12/24/2025 and 12/25/2025?

What about

- -5455/23195/-30879 and -5460/24100/-30800?

Even if we sanitize/declare negative numbers invalid, some inputs are still easier to check.

Input Profiles

Generate inputs that fit expected input profiles:

Start with developers' sanity-check inputs
(like 12/24/2025 and 12/25/2025 etc).

Also, sanitizing checks inside the code are good places to start.

Months: valid months, 0, -1, 13.

Other options

1. Start from normal inputs,
   use genetic algorithms,
   or generate from distributions.

2. Reuse partial inputs, manually modified;
   change one thing at a time,
   thus, easier to reason about changes in the output.

   e.g. go from 0/1/2010 to 1/1/2010, etc.

Integers vs strings

Even though some integers aren't very good (e.g. 23195),
it's still easier to create integers than strings,
and easier to create strings than e.g. trees.

Space of strings is bigger;
space of sensible strings is proportionally smaller.

Can use random strings as fuzzed inputs,
but also want strings that pass sanity checks.

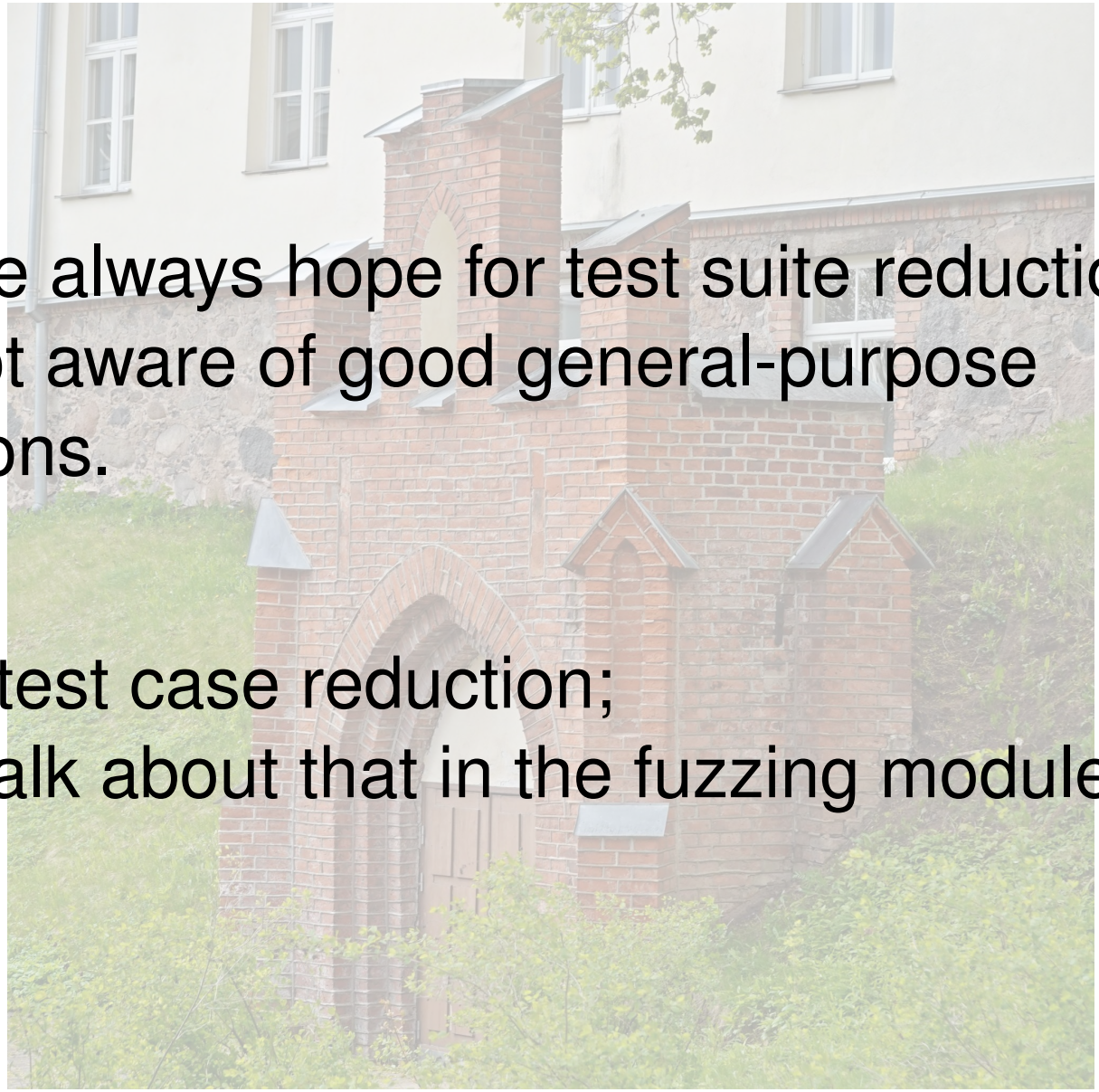
Can mine the web for strings, or generate strings using heuristics (or LLMs).

Crowdsourcing inputs

People have tried to use Mechanical Turk.

Apparently it's hard to get good results.

Reducing volume of work for human testers



People always hope for test suite reduction.
I'm not aware of good general-purpose solutions.

Also: test case reduction;
we'll talk about that in the fuzzing module.

Part III

Implicit Oracles

Segfaults: always bad

```
plam@poneke ~/c/s/n/c/L05 (main)> cat segfault.c
#include <stdlib.h>

int main()
{
  int * q = malloc(5*sizeof(int));
  q[2000000] = 4;
}
plam@poneke ~/c/s/n/c/L05 (main)> gcc segfault.c -o segfault
plam@poneke ~/c/s/n/c/L05 (main)> ./segfault
fish: Job 5, './segfault' terminated by signal SIGSEGV (Address boundary error)
plam@poneke ~/c/s/n/c/L05 (main) [SIGSEGV]>
```

Can always label as bad: Buffer overflows, segfaults, etc.

Might need a tool (e.g. Valgrind, AddressSanitizer) to identify some undefined behaviour.

Don’t need any domain knowledge here.

Also bad (what is “etc”)

- other crashes
  (but: uncaught exception vs
  quiet quitting?)
- race conditions, livelock, deadlock
- memory leaks, performance degradations

Fuzzing & Implicit Oracles

Preview of fuzzing lectures:

because we autogenerate heaps of inputs, we have no way of checking their outputs.

Can only check implicit oracles: no broken assertions, no crashes, use other sanitizers.

Even less evidence for correctness than artisanal tests.

Part IV

Specification-based Oracles

Enabling Verification

What should the implementation do?

One answer: what the specification says.

Specification: a description of [a part of] a system.

Model-based specification

Use modelling languages to describe system behaviour.

e.g. an Alloy action predicate:

```
pred upload [f : File] {
  f not in uploaded              // guard
  uploaded' = uploaded + f       // effect on uploaded
  trashed' = trashed             // no effect on trashed
  shared' = shared               // no effect on shared
}
```

says what changes in the model when a file is uploaded—acts as an oracle.
System state is modelled by sets or relations.

Using the Model

Can write test cases verifying the specified behaviour:

- what happens if we upload an already-uploaded file?
- when we upload a file, does the file appear in the uploaded set?

Need to translate between the model's sets and the implementation's data structures.

Modelling in the Implementation Language

Instead of a dedicated modelling language, use the program's data structures.

This is back to writing assertions.

Also: preconditions, postconditions, invariants (possibly with specialized syntax e.g. @ensures).

Modelling example

```
    # partially self-verifying
    # appendToList implementation
def appendToList(l,elem):
    l.append(elem)
    assert elem in l
```

Implementation Language Modelling Challenges

e.g. for a disk-based filesystem, can't verify using in-memory info.

some conditions are expensive to verify, e.g. is a linked list acyclic?

Assertions as Explicit Oracles

The conditions embedded in the assertions are explicit oracles (not implicit).

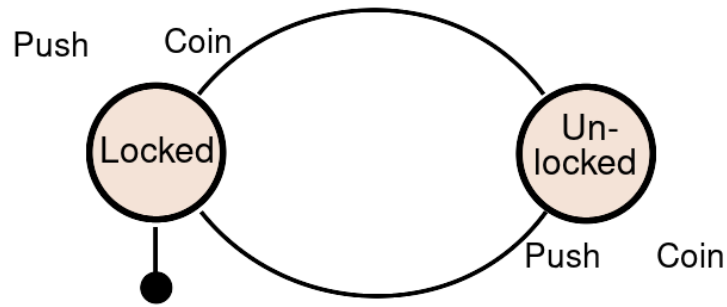
Your test case arranges and acts, and then uses the embedded explicit oracles.

Embedded assertion failure = test case failure (like implicit oracles): a bit more assurance.

Model-based specifications still better.

State Transition Systems

Kind of similar to model-based specification; turnstile example from Wikipedia:



Use the FSM as a test oracle—e.g. inserting a coin in a locked turnstile should result in an unlocked turnstile.

You can write a test to check this.

Part V

Derived Test Oracles

Not Enough Specs

Often, there are no specifications, or the specifications are insufficient or unhelpful.

Derived test oracles of various sorts:

- pseudo-oracles;
- regression testing;
- textual documentation;
- specification mining;
- metamorphic testing.

Pseudo-Oracles

You have to test this imperative implementation:

```
def fib_imperative(n):
    a = 1
    b = 1
    next = b
    count = 1
    seq = [1, 1]

    while count <= n:
        count += 1
        a, b = b, next
        next = a + b
        seq.append(next)
    return seq
```

Recursive fib

The recursive implementation matches the definition of fib closely:

```
def fib_recursive(n):
    if n == 0:
        return 0
    elif n == 1 or n == 2:
        return 1
    return fib(n-1) + fib(n-2)
```

Create tests (perhaps automatically) that compare outputs of the two versions.

Generalization: *N*-version programming, vote on the most popular answer.

Sort of an oracle, though if all versions implement the wrong spec, you still lose.

Regression Testing as Pseudo-Oracles

One can consider the earlier version to be a sort of pseudo-oracle.

But a perfective change in the software means that the earlier version was actually wrong, or maybe the specifications have changed.

Textual documentation

Words also serve as a source of truth.

Historically: then deciphered by humans, and converted into specifications.

Specification mining

Lots of work on observing program executions and deriving specifications (invariants and more generally).

Won't talk about that in this course,
but Daikon is one research tool that does this.

Part VI

Metamorphic Testing

Background: schist—metamorphic rock.

No Oracle? Now what?

Want to run zillions of tests.
What's the expected output?

We had implicit oracles, e.g. "doesn't crash".

Can we do better?

Consider min

Suspend disbelief & pretend we have no oracle:

```
def min(a,b):
    if a < b:
        return a
    else:
        return b
```

How can we generate heaps of tests?

“New tests from old”

“New tests from old” II

Say `min(3, 5) = X`.

Can we construct another test case with known output?

“New tests from old” III

`min(5, 3) = X` also.

A simple example of metamorphic testing: even if you don’t know X, it’s the same X:

`min(5, 3) = min(3, 5).`

Always domain-specific.

Observations on metamorphic testing

You are testing f and you have input x_0.

1. given one input x_0, generate another input x_1;
2. you know some property of $f(x_1)$ in relation to $f(x_0)$;
3. you don't necessarily know $f(x_1)$;
4. in fact, you don't need to know $f(x_0)$ in advance!

min and metamorphic testing: no outputs

You can randomly generate 1 zillion test inputs for `min`.

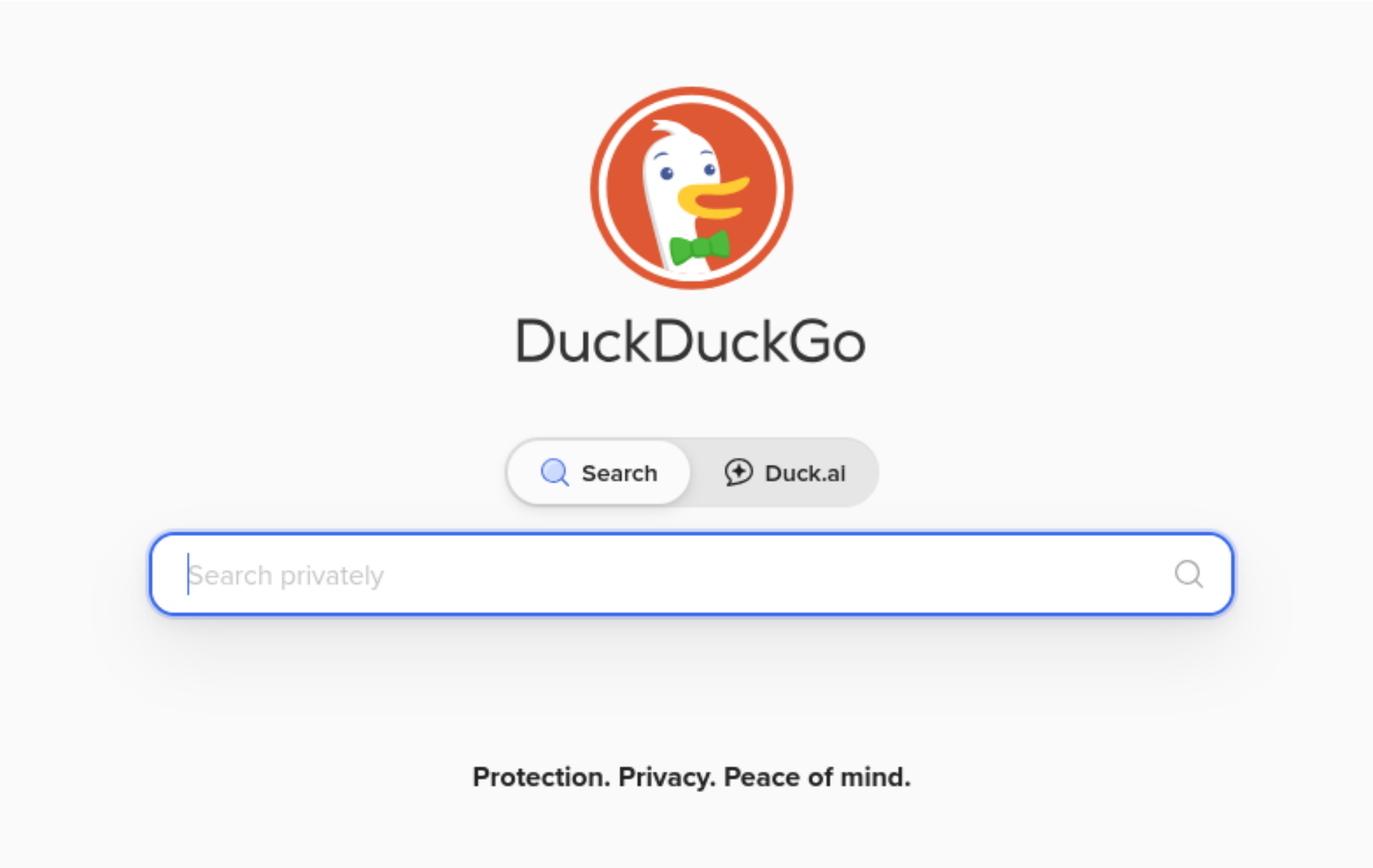
By our assumption, we don't have the corresponding outputs,
just that `min` shouldn't crash.

min and metamorphic testing: what we have

- can generate another 1 zillion test cases, by inverting parameter order.
- also don't have outputs for these cases, but
- we know that it should be the same as the uninverted input!

That's something, which is better than nothing.

Another example: search



Exclusion

1. Search for “metamorphic”.
2. Get, say, 3300 results.
3. Search for “metamorphic -testing
   (i.e. exclude tesing).
4. Now get 4200 results.
5. ???

How do we use this insight to get new tests from old?

Another example: speech-to-text

Speech-to-text scenario

You are writing a speech-to-text processor.

Input: audio file.
Output: text.

Testing speech-to-text

1. manual test generation: you record a text and compare to known good output (oracle).
2. you (or I) get 130 students to do that.
3. you generate audio using text-to-speech.
4. you use Mechanical Turk to get audio.

Still not even close to covering the space of valid inputs, e.g. accents.

Getting lots of inputs. . .

Well, there are a lot of audio files on the Internet. . .

. . . but there is no ground truth, and human oracles are expensive.

What about property testing?

"It doesn't crash on any input." OK...
"It doesn't transcribe acoustic music into words." I guess?

These properties don't really validate the system.

Transforming inputs

Let's say you have one input, which transcribes to `out`. You can:

- double the volume; or,
- raise the pitch; or,
- increase the tempo; or,
- add background static; or,
- add traffic noises; or,
- combine any of these.

Implications of transforming inputs

"add traffic noises" has a lot of freedom;
you can add 10 different traffic noises.

Then, double the volume on each of them;
now have 22 test cases.

etc.

Further implications

What's more, you know the output for all these cases.
Your tests cover much more of the input space.
Still no accents, though.

What if you didn't have the known output?
Can download any audio from the Internet.
Still have the equality relation with the transformed inputs.

Example: YouTube search

Request

```
GET https://www.googleapis.com/youtube/v3/search?part=snippet&
q=winter+pentathlon+1949&key={YOUR_API_KEY}
```

Response

```
200 OK
```

- SHOW HEADERS -


```
-{
  "kind": "youtube#searchListResponse",
  "etag": "\"5g01s4-wS2b4VpScndqCYc5Y-8k/m8dcw2gAjxK5kmiuptFaNM6sOlI\"",
  "nextPageToken": "CAUQAA",
  "regionCode": "ES",
 -"pageInfo": {
    "totalResults": 15,
    "resultsPerPage": 5
  },
 -"items": [
  -{
     "kind": "youtube#searchResult",
     "etag": "\"5g01s4-wS2b4VpScndqCYc5Y-8k/clrd20PVqvoHdka25QfP_RB_F0o\"",
```


(a) Source test case

Request

```
GET https://www.googleapis.com/youtube/v3/search?part=snippet&order=date&
q=winter+pentathlon+1949&key={YOUR_API_KEY}
```

Response

```
200 OK
```

- SHOW HEADERS -

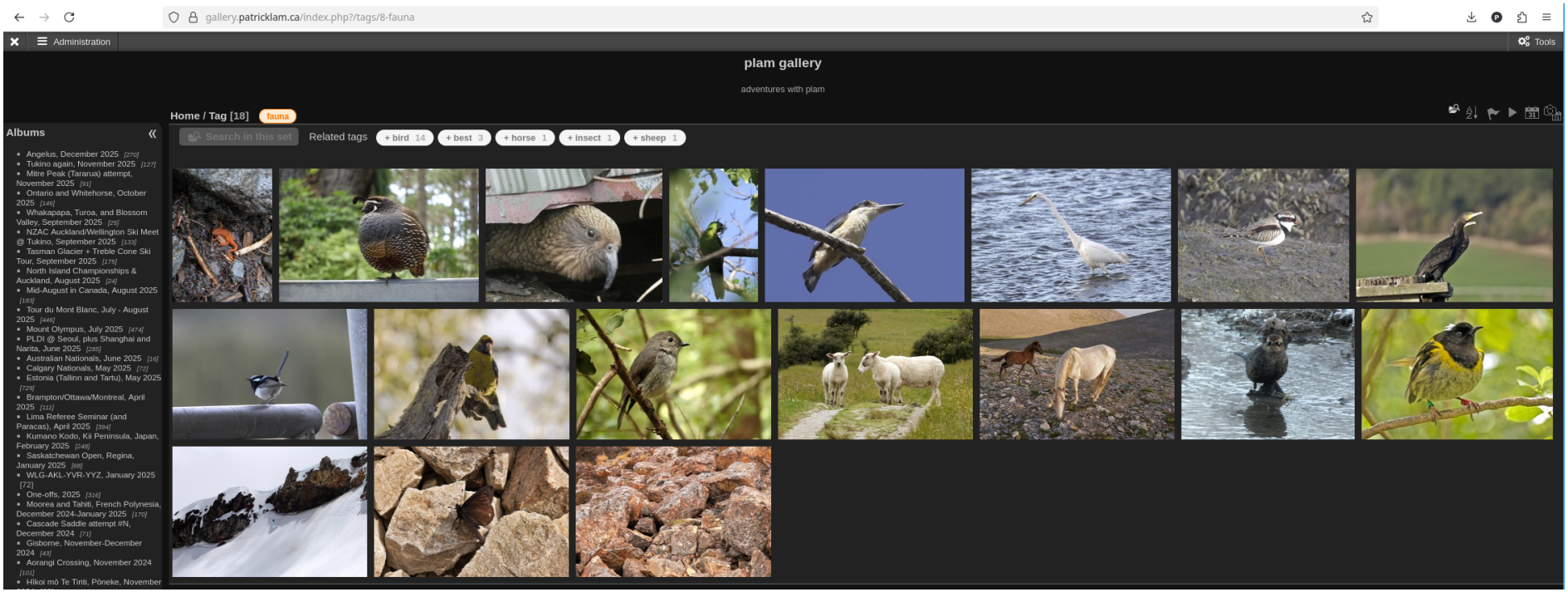

```
-{
  "kind": "youtube#searchListResponse",
  "etag": "\"5g01s4-wS2b4VpScndqCYc5Y-8k/Gl4O7EC7yUbnszssUgAphQFT1GI\"",
  "nextPageToken": "CAUQAA",
  "regionCode": "ES",
 -"pageInfo": {
    "totalResults": 14,
    "resultsPerPage": 5
  },
 -"items": [
  ]
}
```


Empty result set

(b) Follow-up test case (empty result set)

Figure 4: Metamorphic test revealing a bug in Youtube

Example: tagged image search



Example: tagged image search

For our example: tags “red” and “blue”.
An image may be tagged both “red” and “blue”.

Say there are 4 images.
$\{1, 2, 3\}$ are tagged “red”.
$\{1, 2, 4\}$ are tagged “blue”.

Some tagged image search queries

Query: “has any tag”: $\{$ 1, 2, 3, 4$\}$ (n=4).
Query: “red”: $\{1, 2, 3\}$ (n=3).
Query: “blue”: $\{1, 2, 4\}$ (n=3).

It must be that

$$3 + 3\ (red + blue) \geq 4\ (any).$$

Metamorphic relation output patterns

A list of patterns from the paper:

- equivalence: same output items, perhaps in different order;
- equality: same output items, same order;
- subset: follow-up output has a subset of original output;
- disjoint: source and follow-up outputs have no items in common;
- complete: union of follow-up outputs completely make up source output;
- difference: source and follow-up outputs differ by a specific set D.

Metamorphic relation output pattern: equivalence

Source input is a query.
Follow-up input is the same query but with some different ordering requested, like “sort by date”.

Expect: same items in outputs, but different order.

Metamorphic relation output pattern: equality

Had this in the `min` example at start; also speech-to-text.

Another example:

Source input is a query.

Follow-up input is the same query but explicitly requesting the default ordering (e.g. sort by relevance).

Expect: same items in outputs, with same order.

Metamorphic relation output pattern: subset

Search engine example with exclusions was like this.

Another example:

Source input is a geolocation query with radius of 50km.
Follow-up input is the same query but a smaller radius.

Expect: follow-up output is subset of source output.

Metamorphic relation output pattern: disjoint

Example:

source input = Spotify albums of “michael buble” from 2012,
follow-up input = Spotify albums of “michael buble” from 2014.

Expect: there should be no items in both the source and the follow-up output sets.

Metamorphic relation output pattern: complete

This is a relation between the source output and the set of follow-up outputs.

Example context: There are short, medium, and long YouTube videos.

If the source input is for keyword “testing”, then one can make three follow-up inputs: short “testing”, medium “testing”, and long “testing”.

Expect: combined, the results for short, medium, and long “testing” videos should be the same as the results for just “testing”.

Metamorphic relation output pattern: difference

Example:

I upload two videos which I know to be similar except for the length and title.
The create operation returns properties of the uploaded video.

Expect: the source and follow-up metadata only differ on length and title,
other properties may be the same.