Software Testing, Quality Assurance and Maintenance Winter 2026

Lecture 6 — January 23, 2026

Patrick Lam *version 1*

Metamorphic Testing

We've just talked about the oracle problem—how it's difficult to know what is the right answer for a test input.

One proposed approach to getting around the oracle problem is via *metamorphic testing*. Sometimes we just don't have an oracle, but we want to run zillions of test cases, and we'd like to know something about whether the output is correct or not—we want to do better than just checking that the program doesn't crash.

The original reference to metamorphic testing is [CCY98]; but [SPTRC18] is a more modern reference, and contains Web examples like those in the assignment. And, maybe a better writeup than mine by Hillel Wayne: `https://www.hillelwayne.com/post/metamorphic-testing/`.

The general idea is that we can run the program a bunch of times. We control the parameters and so we can craft related inputs, such that the outputs have to be related in a certain way. Metamorphic testing is always domain-specific, so we'll need to provide concrete examples.

Example: min. Consider, then,

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

Let's suspend disbelief and say that we don't have an oracle for this function. But, we know that $\min(a, b) = \min(b, a)$ should hold. So if we have any input for $\min(a, b)$, say $(3, 5)$, then we can create another input, $(5, 3)$, and we know that the result on both of these inputs should be equal.

Let the function you're testing be f. Three observations:

1. if you have one input x_0, you can generate another input x_1;
2. you only know (properties of) the second expected output $f(x_1)$ in terms of the first actual output $f(x_0)$—you don't need to have a way of calculating the correct $f(x_1)$;
3. you don't necessarily know what $f(x_0)$ should be either.

So, you can randomly generate 1 zillion test inputs for min. By assumption, you don't know what the corresponding outputs should be, just that min shouldn't crash (implicit oracles from last time).

You can, however, given your 1 zillion test inputs, generate a second zillion test cases, by inverting the parameter order. And you also know that the inverted test case should produce the same output as the original test case, and you can assert on that. Well, that's something, which is better than nothing.

Example: search engine. One can argue that of course one knows what min should return. It's a contrived example. Here's a less contrived example. Consider a search engine. You can ask it a query q. What's more, you can ask it for all matches of q which do not contain word w. Clearly, the number of hits for q—call that $\mathit{Count}(q)$ should be no less than the number of hits for q excluding w—call that $\mathit{Count}(q - w)$. In [SPTRC18], they propose that searching for "metamorphic" may return 3.3K results. If searching for all pages with "metamorphic" but not containing "testing" returns 4.2K results, then something is wrong. There can't be more results if you exclude a term!

You don't know what the right answer is, but in this example, you know that at least one of the answers you got was wrong.

How can you use this insight to get new tests from old?

Example: speech-to-text. Here's another example, from a blogpost by Hillel Wayne. This one is harder to write concrete code for, but you can definitely imagine it. Let's take a step back. The situation is that you are writing an English speech-to-text processor. You feed it an audio file, and you get text.

This is certainly much more open-ended than anything we see in this course. I can read out something, and then compare the output to what I expect. You can all do that. But this doesn't even come close to covering the space of valid inputs.

(You could imagine using a system like Mechanical Turk to get people to read known-text inputs, but it would be expensive to get thousands of audio files. Or you can generate output with text-to-speech, but that's also quite limited.)

On the other hand, you can download audio files from the Internet, but what output should you expect? You don't know—there is no oracle, and it's super expensive to use human oracles.

Let's use property testing. As usual, "it doesn't crash on any input" is a baseline property. The blogpost also suggests "it doesn't turn acoustic music into words", but that's trickier to verify. In any case, such properties don't really tell you that the system is working properly.

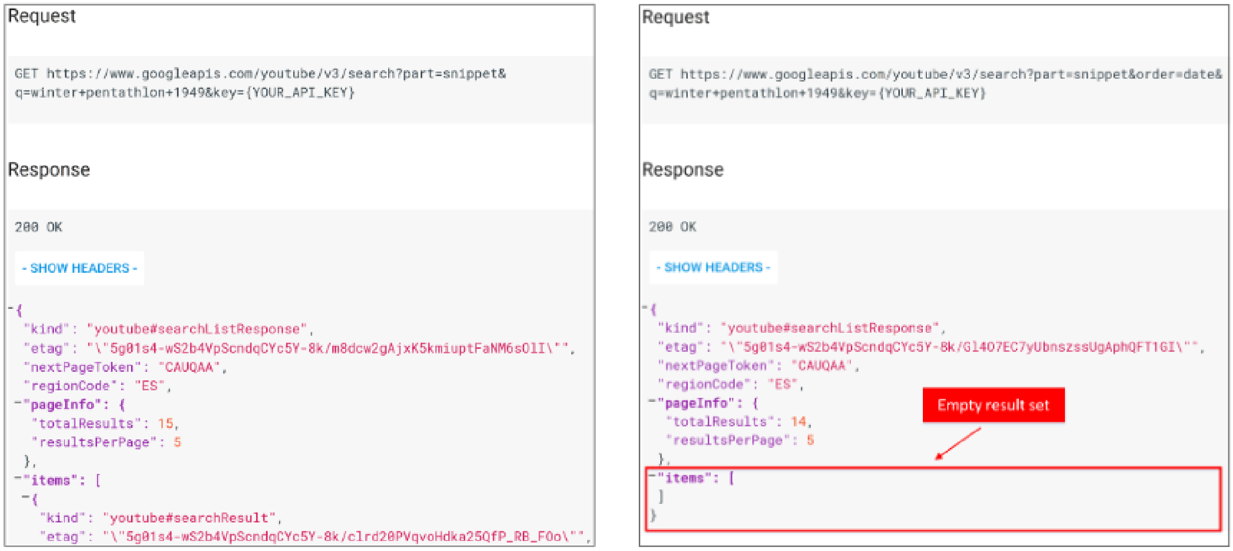
So, imagine that you have one input, and it transcribes to output `out`. We know how to transform audio files. The examples are:

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

So if you take our input and apply transformations, you can still expect the transformed audio file to transcribe to `out`. Indeed, you can take 10 different traffic noises, and then you have 11 (10 + the original) cases to test. You can then double the volume, and you have 22 cases. You know the correct answer for all of these, and your tests now cover much more of the input space.

But wait, there's more. In this case, you knew the output. But you can actually run your system on a collection of related tests without knowing the expected output. The output is supposed to be the same on all of them, but you don't care what it is. This allows you to vastly scale your testing to cases where you don't have an oracle: the blogpost suggests, for instance, downloading an episode of *This American Life*[1] (or, all of them), transforming it, and seeing if the output on all the transformed episodes match.

Example: real-life YouTube search. From [SPTRC18], here's an example from the YouTube search API in particular. They found that a search for "winter pentathlon 1949" returned 15 items; but issuing the same query and asking for ordering by date returned 0 items. That's not right[2]. Here is a figure from the paper I've cited.



Request

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

Response

```
200 OK
```

```
-{
  "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_FOo\"",
```

(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
```

```
-{
  "kind": "youtube#searchListResponse",
  "etag": "\"5g01s4-wS2b4VpScndqCYc5Y-8k/Gl407EC7yUbnszssUgAphQFT1GI\"",
  "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. This example is the one I've put on the assignment. Maybe I can explain it more clearly this time. Consider an image gallery, where you can tag images. Let's say that there are tags "red" and "blue", for pictures that contain something red, or something blue, respectively. A picture may be tagged both "red" and "blue".

Let's also say that there are 4 images, numbered 1 through 4. Images $\{1, 2, 3\}$ are tagged "red", while images $\{1, 2, 4\}$ are tagged "blue". There are no other images and no other tags.

If you request all images with any tag, then you'll get all 4 images: $\{1, 2, 3, 4\}$.

If you request all images tagged "red" and, separately, all images tagged "blue", then you get 3 results for each of the requests. If you add $3+3$ then you get 6, and this number 6 has to be greater than or equal to 4, the total number of images which have any tag.

Symbolically, let T be the set of tags, and let $\#(S)$ count the number of images with tags in the

[1] Canadian content plug: The Debaters on CBC might be better: `https://www.cbc.ca/radio/thedebaters`

[2] YouTube today returns things you didn't ask for.

set S. This has to be true:

$$\sum_{t \in T} \#(\{t\}) \geq \#(T).$$

Metamorphic Relations

The paper [SPTRC18] provides a list of metamorphic relation output patterns. Here's a brief explanation.

- Equivalence: the source and follow-up inputs produce equivalent outputs—same items, though perhaps in a different order. Example: source input is a query, follow-up input is the same query but with some different ordering requested, like "sort by date".
- Equality: source and follow-up inputs produce equal output—same items, same order. Example: source input uses the default value, follow-up input specifically requests the default. Specific example: default order is relevance; initial input omits the order, follow-up input asks for relevance order.

  One can also have relations where the source output is equal to at least one of the follow-up outputs.
- Subset: follow-up output is a (possibly strict) subset of the source output. Example: filtering a set of query results. Specific example: YouTube videos and narrowing the geolocation further between the source and the follow-up, say from 50km from a point to 25km to a point.
- Disjoint: source and follow-up outputs should be disjoint. Specific example: source input = Spotify albums of "michael buble" from 2012, follow-up input = Spotify albums of "michael buble" from 2014. There should be no items in both output sets.
- Complete: the union of the follow-up outputs completely make up the source output. Specific example: 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". Put together, the results for short, medium, and long "testing" videos should be the same as the results for just "testing".
- Difference: source and follow-up differ in a specific set of items D. Often occur in create and update operations. Specific example: I upload two videos which I know to be similar except for the length and title. The create operation returns the video uploaded. One can check that the source and follow-up output metadata only differ on items in D: for instance, they should have the same resolution, but may have different length and title.

Code Review

Code review is a powerful tool for improving code quality. Today's lecture is based on the following references:

- course reading on code review (main source):

  `http://web.mit.edu/6.031/www/sp17/classes/04-code-review/`
- how code review works in an MIT course on software construction:

http://web.mit.edu/6.031/www/sp17/general/code-review.html

- blog post by Vadim Kravcenko on code review: https://vadimkravcenko.com/shorts/code-reviews/
- Fog Creek code review checklist:

  https://github.com/godber/software_dev_templates_and_checklists/blob/master/code_review/fog_creek-code_review-checklist.md

The MIT course has more comprehensive coverage of code reviews; it requires students to respond to code reviews as well.

Code review is a communication-intensive activity. A reviewer needs to 1) read someone else's code and 2) communicate suggestions to the author of that code. We sometimes think that communicating with the computer is our primary goal when programming, but communicating with other people is at least as important over the long run.

From the blog post:

> The real problem was that no one could understand what the hell anyone else had written; we had duplicate logic in many places and different code styles in our modules. It was really bad

Purpose of code review. The communication inherent in code review aims to improve both the code itself as well as the author of the code. Good code review can give timely information to developers about the context in which their code operates, particularly the project and best-practices uses of the language.

We'll continue with a list of items that you are inspecting when you do a code review.

Formatting. Consistency in formatting helps avoid preventable errors. Positioning of { }s isn't something that necessarily has one right answer. Spaces are probably better than tabs. But the most important thing is to be self-consistent with yourself and within your project. Use a tool to check this.

Code smell example 1.

The following code has a number of bad smells:

```
public static int dayOfYear(int month, int dayOfMonth, int year) {
    if (month == 2) {
        dayOfMonth += 31;
    } else if (month == 3) {
        dayOfMonth += 59;
    } else if (month == 4) {
        dayOfMonth += 90;
    } else if (month == 5) {
        dayOfMonth += 31 + 28 + 31 + 30;
    } else if (month == 6) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31;
    } else if (month == 7) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30;
    }
    // ... through month == 12
    return dayOfMonth;
}
```

Let's go through some of the bad smells.

- **Don't Repeat Yourself**. Code cloning isn't always bad. Sometimes it is bad, as in the code above. The usual reason for it being bad is that fixes in one place may remain unfixed in the other place. (Recall: it wasn't always bad when used for forking and templating). For instance, if February actually had 30 days, you'd need to change a lot of code.
- **Fail Fast.** In the language of MIT course 6.031, we mean that a defect should be caught closest to when it's written. Static checks, as performed in compilers, catch defects earlier than dynamic checks, which catch defects earlier than letting wrong values percolate in the program state. In this particular example, there are no checks ensuring that a user had not permuted `month` and `dayOfMonth`.
- **Avoid Magic Numbers.** The above code is full of magic numbers. Particularly magical numbers include the `59` and `90` examples, as well as the month lengths and the month numbers. Instead, use names like `FEBRUARY` etc. Enums are a good way to encode months, and days-of-months should be in an array. The `59` should really be `31 + 28`, or better yet, `MONTH_LENGTH[JANUARY] + MONTH_LENGTH[FEBRUARY]`.
- **One Purpose Per Variable.** The specific variable that's being re-used in the above example is `dayOfMonth`, but this also applies to variables that you might use in your method. Use different variables for different purposes. They don't cost anything. Best to make method parameters `final` and hence non-modifiable.

Comments and code documentation

Code should, ideally, be self-documenting, with good names for classes, methods, and variables. Methods should come with specifications in the form of Javadoc comments, e.g.

```
/**
 * Compute the hailstone sequence.
 * See http://en.wikipedia.org/wiki/Collatz_conjecture#Statement_of_the_problem
 * @param n starting number of sequence; requires n > 0.
 * @return the hailstone sequence starting at n and ending with 1.
 *         For example, hailstone(3)=[3,10,5,16,8,4,2,1].
 */
public static List<Integer> hailstoneSequence(int n) {
    ...
}
```

Note how this comment describes what the method does, in one sentence, provides context, and then describes the parameters and return values.

Also, when you incorporate code from other sources, cite the sources. For instance, in last year's A2 `index.html` file:

```
    // adapted from Eli Bendersky's Lexer: http://eli.thegreenplace.net/2013/07/16/hand-written-lexer-in-javascript-compared-to-the-regex-based-ones
    // public domain according to author
    // modifications by Patrick Lam
```

This helps, for instance, when the source is later updated, and is the right thing to do in terms of IP (assuming, of course, that your use of the software is allowed by its license).

Don't write comments that don't contribute to code understanding. If it's blatantly obvious from the code, it shouldn't be a comment. Such comments can mislead and hence do more harm than good.

Chaos Monkey

Instead of thinking about bogus inputs, consider instead what happens in a distributed system when some instances (components) randomly fail (because of bogus inputs, or for other reasons). Ideally, the system would smoothly continue, perhaps with some graceful degradation until the instance can come back online. Since failures are inevitable, it's best that they occur when engineers are around to diagnose them and prevent unintended consequences of failures.

Netflix has implemented this in the form of the Chaos Monkey[3] and its relatives. The Chaos Monkey operates at instance level, while Chaos Gorilla disables an Availability Zone, and Chaos Kong knocks out an entire Amazon region. These tools, and others, form the Netflix Simian Army[4].

Jeff Atwood (co-founder of StackOverflow) writes about experiences with a Chaos Monkey-like system[5]. Why inflict such a system on yourself? "Sometimes you don't get a choice; the Chaos Monkey chooses you." In his words, software engineering benefits of the Chaos Monkey included:

- "Where we had one server performing an essential function, we switched to two."
- "If we didn't have a sensible fallback for something, we created one."
- "We removed dependencies all over the place, paring down to the absolute minimum we required to run."
- "We implemented workarounds to stay running at all times, even when services we previously considered essential were suddenly no longer available."

[3] `http://techblog.netflix.com/2012/07/chaos-monkey-released-into-wild.html`

[4] `http://techblog.netflix.com/2011/07/netflix-simian-army.html`

[5] `http://blog.codinghorror.com/working-with-the-chaos-monkey/`

References

[CCY98] T. Y. Chen, S. C. Cheung, and S. M. Yiu. Metamorphic testing: A new approach for generating next test cases. Technical Report HKUST-CS98-01, Department of Computer Science, The Hong Kong University of Science and Technology, 1998. https://arxiv.org/abs/2002.12543.

[SPTRC18] Sergio Segura, José A. Parejo, Javier Troya, and Antonio Ruiz-Cortés. Metamorphic testing of RESTful Web APIs. *IEEE Transactions on Software Engineering*, 44(11):1083–1099, 2018.