Vibe analysis

17-313: Foundations of Software Engineering

https://cmu-313.github.io

Michael Hilton and **Josh Sunshine**

Spring 2026

Learning Goals

- Understand technical debt and its impact on maintainability
- Analyze the quality of a test suite
- Identify properties of test suites that are measurements of quality
- Develop input generators for composite input

Test Suite Quality

Code Coverage

This is the most common metric, measuring how much of your source code is executed when the suite runs. While high coverage doesn't guarantee quality, low coverage almost certainly guarantees risk.

- Statement/Line Coverage: The percentage of executable lines reached.
- Branch Coverage: Percentage of branches through control structures (like an if/else statement) that have been followed.
- Path Coverage: Percentage of path sequences tested.

Mutation Score

If code coverage tells you which lines were hit, mutation testing tells you if those lines were actually checked.

A tool automatically makes small changes to your source code (e.g., changing a > to a <). These "mutants" should cause your tests to fail.

Mutation Score = Mutants Killed / Total Mutants

If a mutant survives, it means your tests aren't sensitive enough to catch a logic change in that specific area.

Test Flakiness

A "flaky" test is one that provides inconsistent results (passing and failing) without any changes to the code.

Flake Rate: The percentage of tests that require a "retry" to pass.

High flakiness usually points to poor isolation, race conditions, or reliance on external state/APIs.

Test Execution Time

A suite that takes three months to run is a suite that cannot be run frequently.

- **Feedback Loop Speed:** Measures how long it takes from "code commit" to "test results."
- **Trend Analysis:** If the suite execution time is growing exponentially compared to the codebase, the tests may be poorly designed or too heavily reliant on slow integration-level checks.

Defect Leakage

This is a "post-mortem" metric that looks at how many bugs were found in production compared to how many were caught by the test suite.

- **High Leakage:** If users are finding bugs in areas that have "100% test coverage," it indicates that your test cases are checking for the wrong things or lack meaningful assertions.

In-class activity

https://tinyurl.com/4akxcz48

1. Use two different LLMs to create a testable JavaScript function (side effect free, deterministic, small input space, small output space)
2. Find an example where the function's behavior is very different from between the two LLMs.
3. Put your example in the form.

Conditional Properties

Conditional Properties

- **If** condition A is met, **then** property B must hold. Mathematical logic: $A \Rightarrow B$
- Examples:
  - **Wordle**: If the guess word matches the secret word the output should be (G,G,G,G,G).
  - **Compilers**: If the program includes an invalid keyword, the parser should throw a parse error.
  - **NodeBB** converts post titles into URL-safe "slugs" (e.g., "Hello World!" becomes hello-world). If the input string contains at least one alphanumeric character then the generated slug must never be empty and must only contain a-z, 0-9, and -.

Conditional Properties Approach 1: Filtering

Most PBT frameworks allow you to "discard" inputs that don't meet your criteria. In fast-check you use the fc.pre() method.

- **How it works:** The generator creates a random withdrawal amount. If that amount is ≤ the balance, the test framework simply throws that case away and tries again with a new random number.
- **The Pro:** It's very easy to write.
- **The Con:** It can be inefficient. If your condition is very rare (e.g., "only when the amount is exactly 1,000,000.03"), the framework might exhaust its retry limit before finding enough valid cases.

Conditional Properties Approach 2: Targeted Generators

Instead of generating *any* number and throwing away the bad ones, you create a custom generator that *only* produces "invalid" inputs.

- **How it works:** You define a generator specifically for the error case: amount = random_int(min=balance + 1).
- **The Pro:** Every single test run is a "hit." It's incredibly efficient and ensures you are testing the error handling thoroughly.
- **The Con:** It requires more setup code and a deeper understanding of your data constraints.

```
import * as fc from 'fast-check';
import { createHash } from 'crypto';


// 1. Define the internal data you want to vary
const rawDataArb = fc.record({
  id: fc.uuid(),
  amount: fc.integer({ min: 1, max: 10000 }),
timestamp: fc.date().map(d => d.toISOString())
});
```

```
// 2. Create the "Targeted" Generator using .map()
// This ensures every single output has a perfectly valid
// integrity constraint.
const signedJsonArb = rawDataArb.map(data => {
  const payload = JSON.stringify(data);

  // Integrity Constraint: Generate the signature based on the payload
  const signature = createHash('sha256').update(payload).digest('hex');

  // Return the final "rare" input format
  return JSON.stringify({ payload, signature });
});
```

```
// 3. The Property Test
fc.assert(
fc.property(signedJsonArb, (jsonInput) => {
  // Every input here is guaranteed to have a valid signature
  const result = mySecureService.process(jsonInput);
  return result.status === 'ACCEPTED’;
  })
);
```

Conditional Properties Approach 3: Branching Properties

- **How it works:** The test itself has branches (e.g. if and else) and the property is defined across the branches.
- **The Pro:** This creates a "complete" specification of the function in one place.
- **The Con:** It can make the test code more complex and harder to debug if it fails.