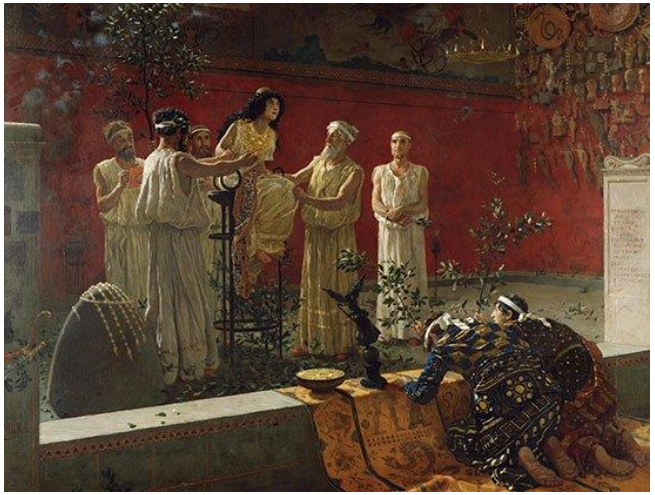
What's the connection?



Oracles

17-313: Foundations of Software Engineering

https://cmu-313.github.io

Michael Hilton and **Josh Sunshine**

Fall 2026

Course Announcements

- Problem Set 4 due Friday, Oct 2
- Midterm Review
  October 5 in Recitation
- Quiz next Tuesday, Oct 6
- Midterm 1
  October 8
- Quiz grades posted
- Review Answer Key

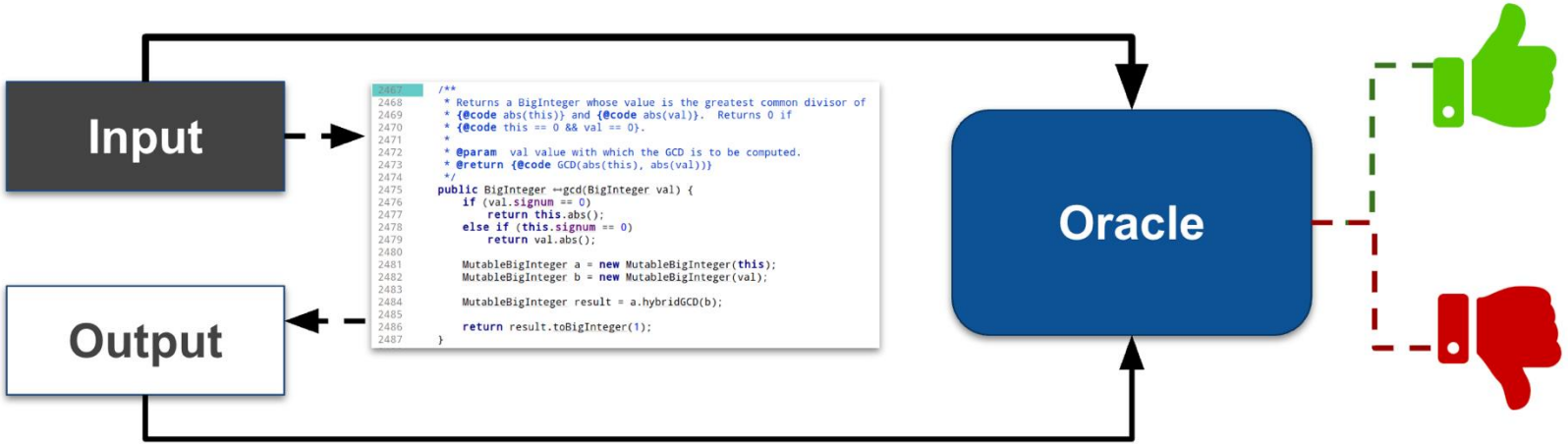
Gentle Reminder

- Please close laptops
- Tablets with stylus ok

What are the limitations of random input generation?

Testing is Only as Good as your Oracle

An oracle decides if behavior is correct for a given input



Example-based Oracles

- A human picks a specific input and works out the expected output in advance
- The test compares actual output to expected: assertEquals(6, factorial(3))
- **Strengths:** precise, easy to read, and doubles as documentation of intended behavior
- **Each test checks exactly one point** in a huge input space
- **The expected value is only as good as the person computing it:** hand calculations can be wrong
- **Doesn't scale:** you can't hand-compute answers for thousands of generated inputs

Implicit Oracles

- **These oracles are used by most fuzzing approaches**
- These oracles are **generic properties** that is not tied to any test inputs
  - allows us to automatically generate and test any input
  - but the oracle is weak (i.e., not crashing does not imply correct)

- Examples:
  - Crashes
  - Uncaught exceptions
  - **Sanitizer Errors**
  - Hangs
  - Timeouts

Sanitizers as Implicit Oracles

- We can make the oracle slightly stronger by using **sanitizers**
  - detects illegal program states that might not cause an immediate crash
  - instruments the program at compile time (e.g., `-fsanitize=address`)
  - finds more safety issues but slows down execution / fuzzing
  - doesn't reveal logic bugs
- Most common examples:
  - AddressSanitizer (https://clang.llvm.org/docs/AddressSanitizer.html)
  - MemorySanitizer (https://clang.llvm.org/docs/MemorySanitizer.html)
- Others: ThreadSanitizer, LeakSanitizer, DataFlowSanitizer, TypeSanitizer, ControlFlowIntegrity

Oracle: Assertions in Example-Based Tests

- **This is the most common type of oracle in traditional tests**
- These assertions are often hardcoded to a specific test input
  - tedious to write for complex outputs (e.g., documents, actions)
  - can be very brittle (e.g., formatting changes lead to test failures)
  - non-determinism and environment coupling lead to flaky tests

```
it('should redirect to login if user is not logged in', async () => {
    const { response, body } = await request.get(`${nconf.get('url')}/me/bookmarks`);
    assert.equal(response.statusCode, 200);
    assert(body.includes('Login to your account'), body.slice(0, 500));
});
```

Oracle: Assertions in Source Code

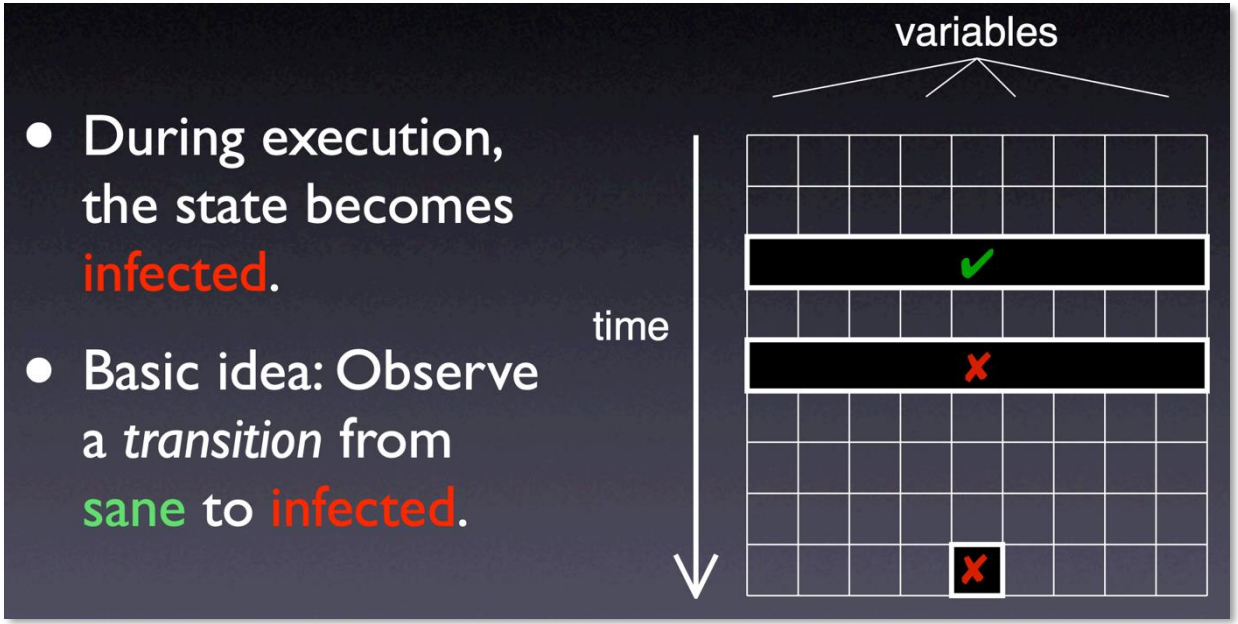
- Assertions are **executable specifications**
  - document intended behavior (pre/postconditions, invariants)
- This oracle is generic and **not tied to any test inputs**
  - if we add assertions, we can use fuzzing to find some logic bugs!

```
function toUSD(amountCents: number): string {
  assert(Number.isInteger(amountCents), 'amount must be integer cents');
  assert(amountCents >= 0, 'amount must be non-negative');
  const dollars = (amountCents / 100).toFixed(2);
  return `$${dollars}`;
}
```


https://blog.regehr.org/archives/1091
https://nullprogram.com/blog/2022/06/26

Assertions catch infections earlier

- Finds more bugs (e.g., during fuzzing) and helps to localize them



Assertions should always be true unless you have a bug in your code

- Assertions state invariants: conditions that must always hold if the program is correct (e.g., impossible states, internal consistency).
  - **Never rely on asserts for control flow or user-visible behavior**
  - Make sure that your assertions **don’t contain side effects**
- Use exceptions and returns for errors that can reasonably happen and should be handled (e.g., invalid inputs, failed API calls).

Assertions in the Wild: Apache Cassandra

- Used to enforce an **invariant** that must hold throughout sorting

```
218         *
219         * @param a the array in which a range is to be sorted
220         * @param lo the index of the first element in the range to be sorted
221         * @param hi the index after the last element in the range to be sorted
222         * @param start the index of the first element in the range that is
223         *        not already known to be sorted (@code lo <= start <= hi}
224         * @param c comparator to used for the sort
225         */
226        @SuppressWarnings("fallthrough")
227        private static void binarySort(long[] a, int lo, int hi, int start,
228                                       LongComparator c) {
229            if (DEBUG) assert lo <= start && start <= hi;
230            if (start == lo)
231                start++;
232            for ( ; start < hi; start++) {
233                long pivot = a[start];
234                // Set left (and right) to the index where a[start] (pivot) belongs
235                int left = lo;
236                int right = start;
237                if (DEBUG) assert left <= right;
```

Assertions in the Wild: SQLite & LLVM

- Used to enforce a **precondition** and find bugs at call sites

```c
/*
** Insert a new entry into the cache.  If the cache is full, expel
** the least recently used entry.  Return SQLITE_OK on success or a
** result code otherwise.
**
** Cache entries are stored in age order, oldest first.
*/
static int jsonCacheInsert(
  sqlite3_context *ctx,   /* The SQL statement context holding the cache */
  JsonParse *pParse       /* The parse object to be added to the cache */
){
  JsonCache *p;

  assert( pParse->zJson!=0 );
  assert( pParse->bJsonIsRCStr );
  assert( pParse->delta==0 );
  p = sqlite3_get_auxdata(ctx, JSON_CACHE_ID);
  if( p==0 ){
    sqlite3 *db = sqlite3_context_db_handle(ctx);
```

lldb / include / lldb / Interpreter / OptionValueUInt64.h

Code | Blame | 99 lines · 3.08 KB

```
69          m_current_value = value;
70          return true;
71        }
72        return false;
73      }
74
75      bool SetDefaultValue(uint64_t value) {
76        assert(value >= m_min_value && value <= m_max_value &&
77               "disallowed default value");
78        m_default_value = value;
79        return true;
80      }
```

Assertions in the Wild: Firefox

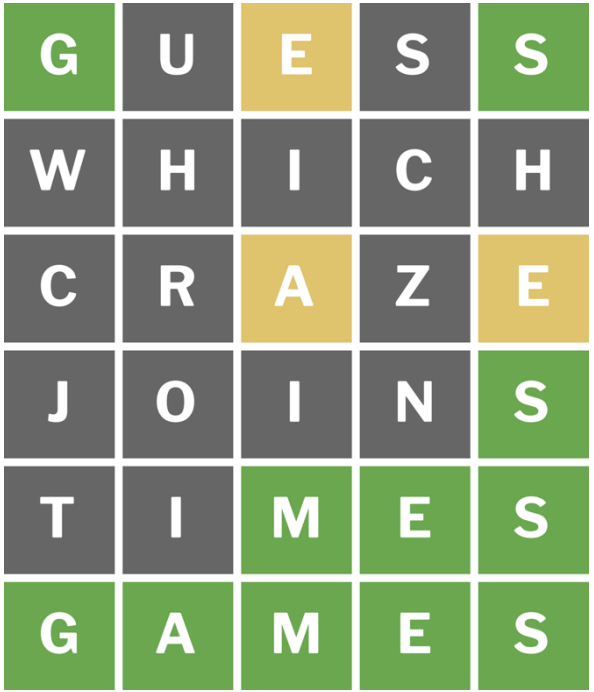
- Used to enforce a **postcondition** that makes sure audio packet is the correct size after processing

```
namespace mozilla {
void AudioInputProcessing::Process(AudioProcessingTrack* aTrack,

  // Postconditions of the audio-processing logic.
  MOZ_ASSERT(static_cast<uint32_t>(mSegment.GetDuration()) +
                 mPacketizerInput->FramesAvailable() ==
             mPacketizerInput->mPacketSize);
  MOZ_ASSERT(mSegment.GetDuration() >= 1);
  MOZ_ASSERT(mSegment.GetDuration() <= mPacketizerInput->mPacketSize);
}
```

Activity: Wordle Oracle

1. What precondition assertions could you add to `mark_guess`?
2. What postcondition assertions could you add?



Derived Oracle 1: Differential testing

- **Oracle:** another implementation of the same behavior. Run both on the same input and flag any disagreement.
- **Sources of the second implementation:**
  - a competing tool (GCC vs. Clang, two JSON parsers)
  - an older version of your own code
  - a slow but obviously correct reference version
- **Strength:** works with generated inputs at scale, since no human computes expected values
- **Famous example:** Csmith generated random C programs and compared compilers' outputs, finding hundreds of bugs in GCC and LLVM
- **A disagreement doesn't tell you which side is wrong**, only that at least one is
- **Shared blind spots:** independently written versions tend to fail on the same hard inputs
- **False alarms from legitimate differences**, such as unspecified behavior or floating-point rounding

Derived Oracle 2: Golden master

- **Oracle:** the system's own past output. Record it once, then compare every future run against it.
- **Also called** snapshot tests (Jest), approval tests, and characterization tests (Feathers, *Working Effectively with Legacy Code*)
- **Strength:** needs no spec and no second implementation, which makes it ideal before refactoring legacy code you don't fully understand
- **Assumes today's behavior is correct**, so recorded bugs become "expected" behavior
- **Breaks on intentional changes**, which trains people to click "update snapshot" without looking
- **Nondeterminism (timestamps, IDs, ordering)** has to be scrubbed or the test becomes flaky
- ***This is also what AI-generated tests often turn into:*** *they assert whatever the code currently does*

Derived Oracle 3: Metamorphic testing

- **Oracle:** a relation between the outputs of related inputs, used when you can't know the correct output for any single input
- **Examples:**
  - sin(x) == sin(π − x)
  - adding a search filter never returns more results
  - shortest path from A to B equals B to A in an undirected graph
  - an image classifier's label shouldn't change when brightness shifts slightly
- **Strength:** one relation checks unlimited generated inputs, and it works for "no known answer" domains like ML, simulations, and search
- **Finding good relations takes domain insight**
- **Partial oracle:** a buggy program can still satisfy every relation you wrote down