COMS W4156 Advanced Software Engineering (ASE)

November 30, 2021

shared google doc for discussion during class

Test Oracles

The entity that knows the correct outputs, and/or can check that the actual outputs are the same as or consistent with the expected outputs, is called the "*test oracle*"

Often the human tester is the test oracle - either knows the expected output in advance or, given an actual output, can determine whether or not it is correct

For manual testing, the human tester checks the results

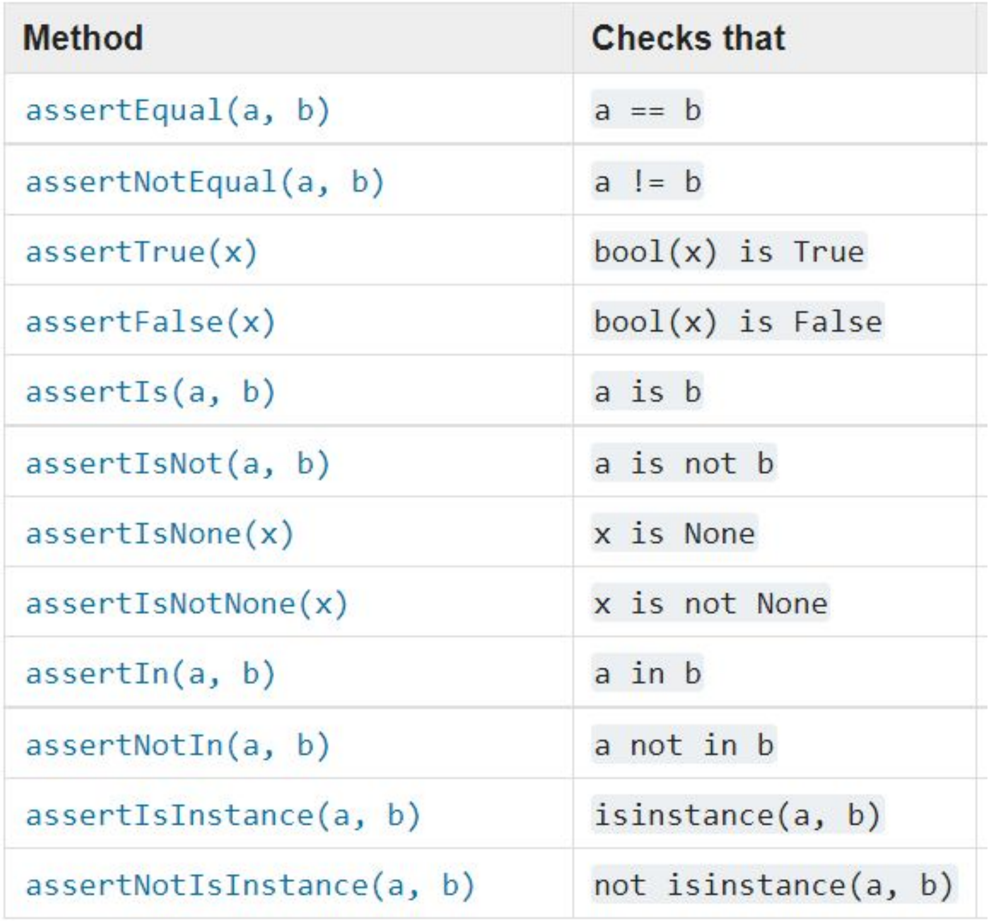
For automated testing, the tester writes assertions to check the results

Refresher: Assertions

Notation for checking the results of executing a test (test oracle)

Usually supports checking for true/false result, comparing an output to a known value, and checking whether an exception has been raised

```
assert 1 != 1, "This is a great Python Tutorial!"
```



Method	Checks that
`assertEqual(a, b)`	`a == b`
`assertNotEqual(a, b)`	`a != b`
`assertTrue(x)`	`bool(x) is True`
`assertFalse(x)`	`bool(x) is False`
`assertIs(a, b)`	`a is b`
`assertIsNot(a, b)`	`a is not b`
`assertIsNone(x)`	`x is None`
`assertIsNotNone(x)`	`x is not None`
`assertIn(a, b)`	`a in b`
`assertNotIn(a, b)`	`a not in b`
`assertIsInstance(a, b)`	`isinstance(a, b)`
`assertNotIsInstance(a, b)`	`not isinstance(a, b)`

Writing Better Assertions: Example

Assume that we test a Java method whose signature is

```
List<Employee>
getEmployees(Predicate<Employee> constraint)
```

This method accepts a predicate on `Employee` and retrieves from some database the list of all employees that match the given predicate.

Let's say we want to check that the function retrieves all employees older than 25 when it is called with a predicate of the form "*the age of an employee is greater than 25*". We could do so in a JUnit test like this one:

```
@Test
public void getEmployeesGreaterThan25() {
 // arrange
 Employee mark = new Employee ("Mark", 19);
 Employee nina = new Employee ("Nina", 26);
 Employee jules = new Employee ("Jules", 31);
clearDatabase();
 insertEmployee(mark);
 insertEmployee(nina);
 insertEmployee(jules);
 Predicate<Employee> constraint = e -> e.getAge() > 25;
```

Example continued

```
// act
  List<Employee> result = getEmployees(constraint);
  // assert
  assertEquals(result.size(), 2);
  assertEquals(result.get(0).getName(), "Nina");
  assertEquals(result.get(0).getAge(), 26);
  assertEquals(result.get(1).getName(), "Jules");
  assertEquals(result.get(1).getAge(), 31);
     }
```

We used five assertions to check that the function returns all employees older than 25.

But, looking closely, these assertions actually check something else: they check that the result list always contains Nina in the first entry and Jules in the second one.

This particular detail could change as we update the implementation of getEmployees, or could even be altered by external sources of non-determinism.

Assert requirements, not implementation details

Assert the exact desired behavior; not more, not less

Alternatively, we could have written the following lines in place of those 5 assertions:

```
for (Employee e : result) {
  assertTrue(e.getAge() > 25);
    }
```

This version has problems, too. We now check that every employee returned by the function is over 25.

But we are not checking that *all* such employees are returned. If getEmployees was modified to always return an empty list, this test would still pass.

Good assertions check what is precisely requested from the code under test, as opposed to checking an overly precise or an overly loose condition.

It is easy to write overly precise assertions, but the resulting test is often not maintainable or might be affected by slight non-deterministic variations in the environment.

It is also tempting to write overly loose assertions, but the resulting test might not catch regressions that it should catch.

One assertion, one condition

It may be tempting to aggregate multiple checks in one assertion. We could have written something like:

```
boolean first = result.get(0).getAge() > 25;
boolean second = result.get(1).getAge() > 25;
    assertTrue(first && second);
```

If that assertion fails, we will not immediately know if it is because the first or the second employee. Splitting into two assertions gives us a better clue about the cause:

```
assertTrue(result.get(0).getAge() > 25);

assertTrue(result.get(1).getAge() > 25);
```

Don’t write assertions that check conditions that could be split into multiple assert statements.

Always check the simplest condition possible.

Any gain in speed when writing complex conditions is lost when someone needs to debug why such assertion fails.

How Does the Tester Know the Correct Answer to Assert?

If it doesn't crash, hang, or produce obviously nonsensical output, it works!

Stick to math and string processing libraries, with well-understood functions

Wait until users complain that the software doesn't do what they expected, then write test cases to check for what they say they expected

Wouldn't it be better if the users told us what they want before we wrote the software?

They Do! (Or Think They Do)

User Stories Conditions of Satisfaction
(or Acceptance Criteria) -
list known situations where the code
needs to work, but may not explain what “working” means

Use Cases Basic and Alternative Flows - step by step guide to the required interaction between the code and the actor, including handling of special cases, but may not include details of expected results

Asserting Conditions of Satisfaction

As a user, I am required to login before using the CONTAIN app to protect student privacy against unauthorized access.

- user is logged in only when proper credentials are provided
- a "remember me" option is available
- user can request a password reminder
- user is locked out after three failed attempts

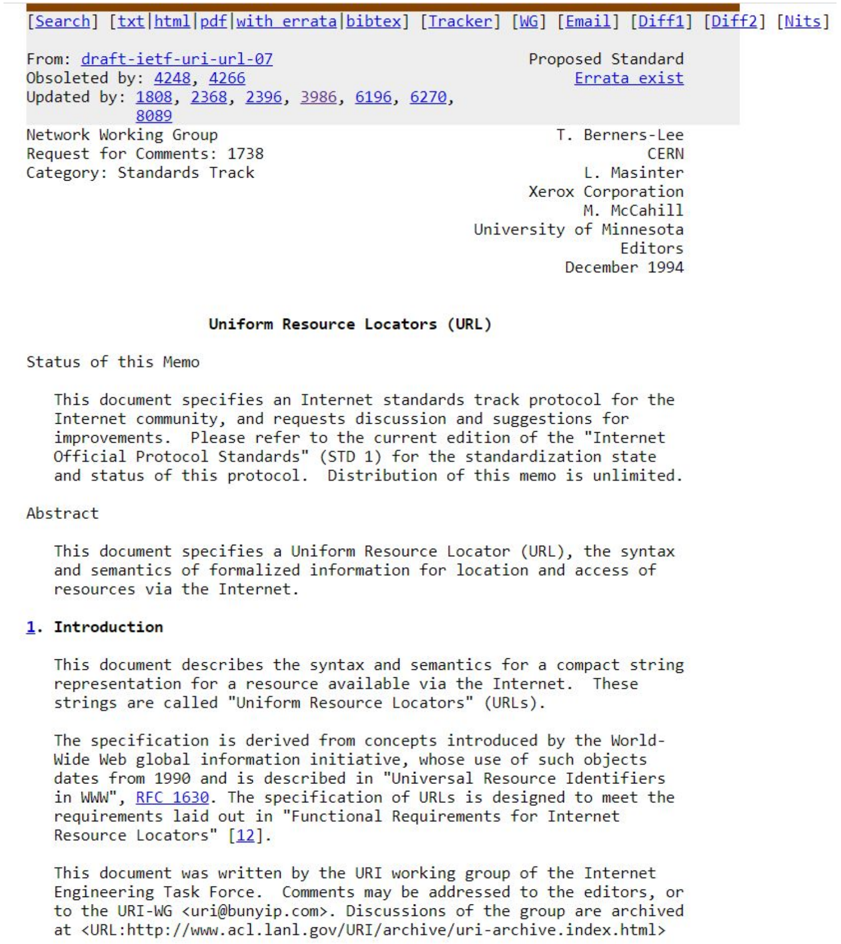
How might we write test assertions for these conditions of satisfaction?

“Promise of Conversation”

Agile Concept - User Story (or Use Case) “comes with” a conversation with the customer or product owner

Specifications

Detailed prose specification - common for networking, database, file system and other standard protocols necessary for interoperability, rare for user-facing functionality of business/consumer applications



```
[Search] [txt|html|pdf|with errata|bibtex] [Tracker] [WG] [Email] [Diff1] [Diff2] [Nits]

From: draft-ietf-uri-url-07                               Proposed Standard
Obsoleted by: 4248, 4266                                      Errata exist
Updated by: 1808, 2368, 2396, 3986, 6196, 6270,
            8089
Network Working Group                                      T. Berners-Lee
Request for Comments: 1738                                           CERN
Category: Standards Track                                     L. Masinter
                                                        Xerox Corporation
                                                              M. McCahill
                                                University of Minnesota
                                                                  Editors
                                                            December 1994


                     Uniform Resource Locators (URL)

Status of this Memo

   This document specifies an Internet standards track protocol for the
   Internet community, and requests discussion and suggestions for
   improvements.  Please refer to the current edition of the "Internet
   Official Protocol Standards" (STD 1) for the standardization state
   and status of this protocol.  Distribution of this memo is unlimited.

Abstract

   This document specifies a Uniform Resource Locator (URL), the syntax
   and semantics of formalized information for location and access of
   resources via the Internet.

1. Introduction

   This document describes the syntax and semantics for a compact string
   representation for a resource available via the Internet.  These
   strings are called "Uniform Resource Locators" (URLs).

   The specification is derived from concepts introduced by the World-
   Wide Web global information initiative, whose use of such objects
   dates from 1990 and is described in "Universal Resource Identifiers
   in WWW", RFC 1630. The specification of URLs is designed to meet the
   requirements laid out in "Functional Requirements for Internet
   Resource Locators" [12].

   This document was written by the URI working group of the Internet
   Engineering Task Force.  Comments may be addressed to the editors, or
   to the URI-WG <uri@bunyip.com>. Discussions of the group are archived
   at <URL:http://www.acl.lanl.gov/URI/archive/uri-archive.index.html>
```

Specifications

Formal Specification - expensive to write and verify, mostly used for safety-critical systems

$$\frac{P \to P' \quad \{P'\}\, C\, \{Q'\} \quad Q \to Q'}{\{P\}\, C\, \{Q\}}\,\text{WEAK}$$

$$\frac{}{\{A[E/x]\}\; x := E\; \{A\}}\,\text{ASG}$$

$$\frac{\{P\}\, C_1\, \{A\} \quad \{A\}\, C_2\, \{Q\}}{\{P\}\, C_1; C_2\, \{Q\}}\,\text{SEQ}$$

$$\frac{\{P \wedge B\}\, C_1\, \{Q\} \quad \{P \wedge \neg B\}\, C_2\, \{Q\}}{\{P\}\ \textbf{if}\ B\ \textbf{then}\ C_1\ \textbf{else}\ C_2\ \{Q\}}\,\text{IF}$$

$$\frac{\{I \wedge B\}\, C\, \{I\}}{\{I\}\ \textbf{while}\ B\ C\ \{I \wedge \neg B\}}\,\text{While}$$

Sometimes There Isn't Any Test Oracle

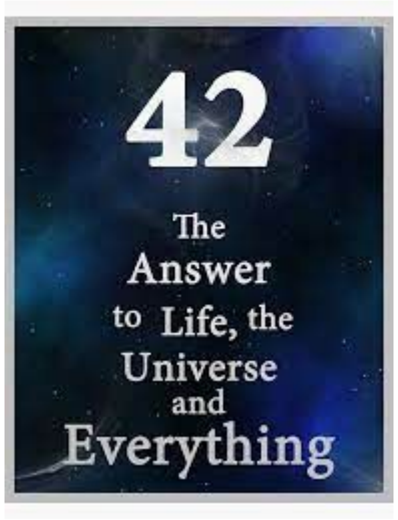
Consider this search: https://www.google.com/search?q=gail+kaiser

Is the answer correct? Are the first ten answers the best ones?

Now consider this: https://www.google.com/search?q=gail+kaiser&hl=en&tbm=isch

Is the answer correct? Are the first ten answers the best ones?

Non-testable program = *"Programs which were written in order to determine the answer in the first place. There would be no need to write such programs, if the correct answer were known."* (Weyuker 1982)

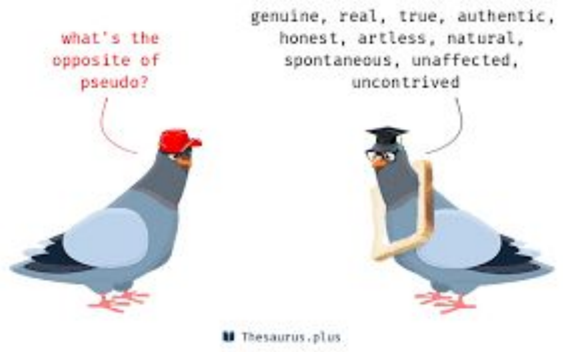


Pseudo-Oracles

Pseudo-oracle = one test case acts as an oracle for another test case

You may not know whether the result of either test case is right, but if the results are inconsistent you know (at least) one of them is wrong

They could both be wrong


what's the opposite of pseudo?
genuine, real, true, authentic, honest, artless, natural, spontaneous, unaffected, uncontrived
Thesaurus.plus

Pseudo-Oracles: Regression Testing

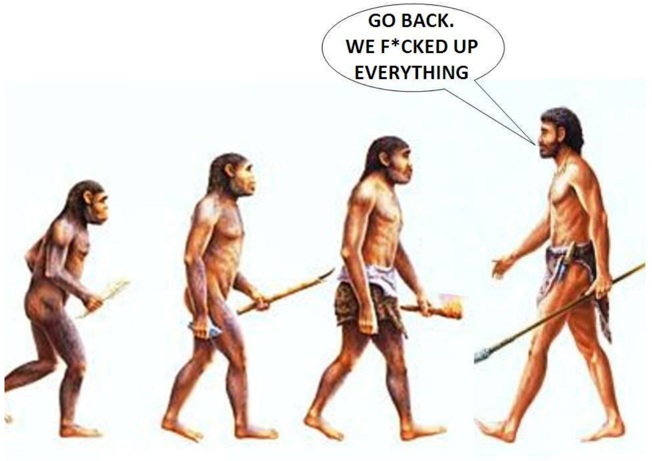
Regression testing is the most common form of pseudo-oracle

The test results for the old version are known and we re-run the tests on the new version to see if they have the same or different results

Some test cases *should* change results on the new version ⇒ when we try to fix a bug that caused a test case to fail on the old version, we want that test case to pass on the new version

But unrelated test cases should not be affected ⇒ if they are, that indicates a hidden dependency and (probably) a newly introduced bug

```
program_v1(input)->output_v1

program_v2(input)->output_v2

output_v1 =? output_v2
```



Pseudo-Oracles: Differential Testing

Differential testing requires a second *independently developed* implementation of the same functionality.

Practical for standard protocols, e.g., there are many different implementations of SSL/TLS and HTTP

- ➢ If they disagree, one or both is wrong
- ➢ If they agree, that does not guarantee bot are right, but does provide some confidence that they are right

```
program_1(input)->output_1

program_2(input)->output_2

output_1 =? output_2
```

It's critical that the implementations are truly independent, with no common third-party libraries or other common factors

But "independent" developers can make the same mistakes

Pseudo-Oracles: Metamorphic Testing

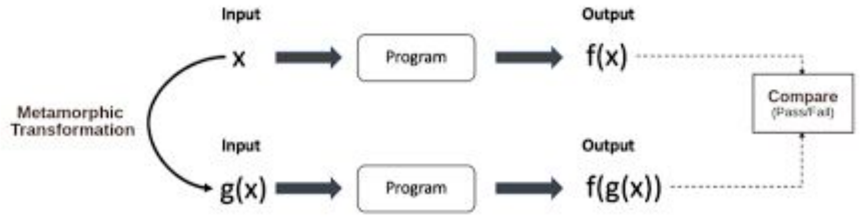
Metamorphic testing requires only one implementation

Starts with some original input (possibly chosen randomly) and its known original output

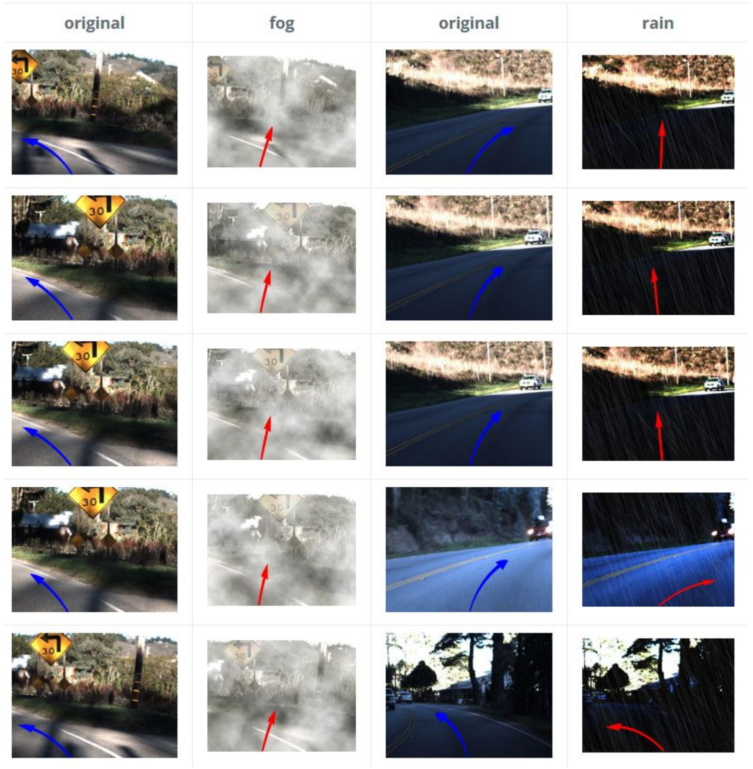
Create a new test case by *deriving* a new input from the original input, and *predicting* the expected new output from the original output - the prediction is usually that the output is the same, nearly the same, or changed in a simple way

If the actual new output deviates too much from the predicted output, there is (probably) a bug

```
program(input_1)->output_1

deriveInput(input_1)->input_2

predictOutput(input_1, input_2,
output_1) -> predicted_output_2

program(input_2)->output_2

check(predicted_output_2, output_2)
-> do they match?
```



Example Metamorphic Testing



Say we have a machine learning application that controls the steering of a self-driving car, so it obeys traffic laws and doesn't cause accidents

How do we know what is "correct" steering for a given real-world driving scenario? How can we write test cases?

Different human drivers will not necessarily steer at exactly the same angle and the same human driver will not necessarily steer at the same angle every time

DeepTest: Synthetic but realistic changes to driving images such as adding snow, fog, rain, more sunlight, less sunlight, etc. should not (in most cases) change automobile steering

Predicted Output is not Necessarily the Same Output

Metamorphic relations for a sorting program (lowest first) applied to an array of numbers

sort(5,2,3,1,4) -> 1,2,3,4,5

Permutative: if we shuffle the order of the input array, the sorted output array should be the same

sort(3,4,1,2,5) -> 1,2,3,4,5 😀

Additive: if we add N to every element of the input array, the sorted output array should be in the same order

sort(25,22,23,21,24) -> 21,22,23,24,25 😀

Multiplicative: if we multiply every element of the input array by (positive) N, the sorted output array should be in the same order

sort(15,6,9,3,12) -> 3,6,9,12,15 😀

Invertive: if we multiply every element of the input array by (negative) N, the sorted output array should be in the opposite order

sort(-5,-2,-3,-1,-4) -> -5,-4,-3,-2,-1 😀

Inclusive: if we add one new element to the input array, the sorted output array should be the same except for the placement of that new element

sort(5,2,42,3,1,4) -> 1,2,3,4,5,42 😀

Exclusive: if we remove one element from the input array, the sorted output array should be the same except it is missing that one element

sort(5,3,1,4) -> 1,3,4,5 😀

Using Metamorphic Relations to Detect Bugs

Original

buggysort(5,2,3,1,4) -> 1,2,3,4,5

Permutative: if we shuffle the order of the input array, the sorted output array should be the same

buggysort(3,4,1,2,5) -> 2,4,1,5,3 😞

Additive: if we add N to every element of the input array, the sorted output array should be in the same order

buggysort(25,22,23,21,24) -> 25,25,25,25,25 😲

Multiplicative: if we multiply every element of the input array by (positive) N, the sorted output array should be in the same order

buggysort(15,6,9,3,12) -> 6,12,3,15,9 😟

Invertive: if we multiply every element of the input array by (negative) N, the sorted output array should be in the opposite order

buggysort(-5,-2,-3,-1,-4) -> -1,-4,-3,-2,-5 😕

Inclusive: if we add one new element to the input array, the sorted output array should be the same except for the placement of that new element

buggysort(5,2,42,3,1,4) -> ⌛ 😖

Exclusive: if we remove one element from the input array, the sorted output array should be the same except it is missing that one element

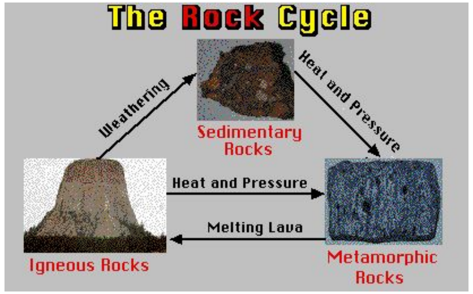
buggysort(5,3,1,4) -> 🌋 😳

Where Do Metamorphic Relations Come From?

Some are generic, e.g., permutation can be applied to any ordered input

Others are specific to domain or application

Metamorphic Testing automates generation of additional tests from existing test suite whether or not there is a test oracle - it’s a test generation technique as well as a pseudo-oracle technique



In-Class Exercise

Recall the CONTAIN app, **CON**tact **T**racing for instruction **A**ss**I**sta**N**ts, which does “contact tracing” based on IA (instruction assistant) connections. An IA for xxx course is a contact of all the other IAs for xxx as well as for every student who takes xxx. Since an IA is also a student, they are a contact for all other students in every course they take as well as for all the IAs of those courses.

The input to CONTAIN is a uni, the set of courses where that uni is an IA (from courseworks), and the set of courses that uni is taking as a student (from SSOL). Version 1.0 of CONTAIN does not have an attendance-recording drone. CONTAIN’s output is a list of unis, which is supposed to be the input uni’s possible contacts. We do not have internal access to courseworks or ssol to manually determine whether that list is correct, so we do not have a test oracle. (Although metamorphic testing is useful even if we did.)

What are some possible metamorphic relations we could use to derive test cases?

One example to get you started: Adam is an IA for COMS 5123 and is taking two courses as a student, COMS 5321 and COMS 5555. We run CONTAIN with Adam’s uni as input and get the list L as output. These are the source inputs and output.

In this sample metamorphic relation, we derive new inputs as follows: we keep the same top-level input (Adam’s uni) and the same input from courseworks (he’s an IA for 5123), but derive a new input from SSOL (now 5321, 5555, and 5678). The predicted output is that CONTAIN produces a new list of unis M that includes all the original unis from L plus possibly some additional unis. If any of the original unis from L are missing from the new output M, we know there’s a bug.

timer

Automated Test Generation

Metamorphic testing generates both the input and the assertion (“is it consistent with the predicted output?”)

Most test generation techniques only generate the input

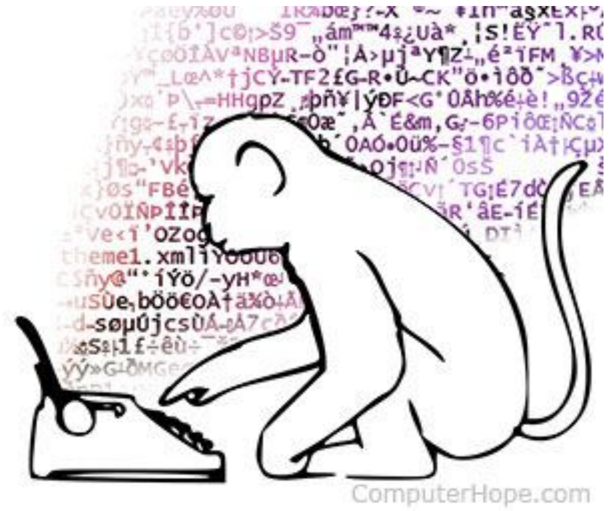
The assertions must be written by human developers OR an easy to check generic assertion is used, like “did it crash?”

Automated Test Generation Techniques

Random - most random testing techniques aren't truly random, they use some guidance mechanism to avoid redundancy and focus on inputs likely to find bugs

Fuzzing - start with a "seed" input and randomly modify it (or parts of it), also not truly random, often used for security and robustness

Symbolic - useful for increasing coverage, use automated constraint solving to identify inputs that will force a previously untested branch



SUCCEED IN IMPLEMENTATION YOU WANT
START WITH END IN MIND YOU SHOULD

Testing the Test Suite

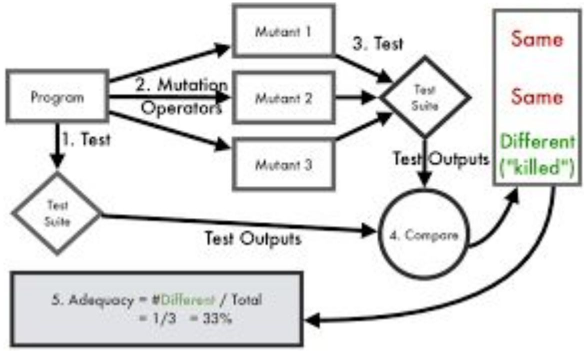
How do we know if we have a “good” or “good enough” test suite?

Equivalence Partitioning and Boundary Analysis - outputs as well as inputs

Control Flow Coverage - high percentage of statement, branch, MC/DC & MCC

Data Flow Coverage - high percentage of intraprocedural paths from each definition (write) to everywhere it is used (read)

Mutation Testing - use mutation operators to inject bugs into the code (e.g., switch > to <, insert return statements at arbitrary locations in the middle of functions) and run the regular test suite,
did it “kill” (detect) all the mutants?



Team Project

Assignment T5: Second Iteration due December 11

Assignment T6: Second Iteration Demo due December 15

Assignment T7: Demo Day on December 20