Lecture 11: System-Level Testing

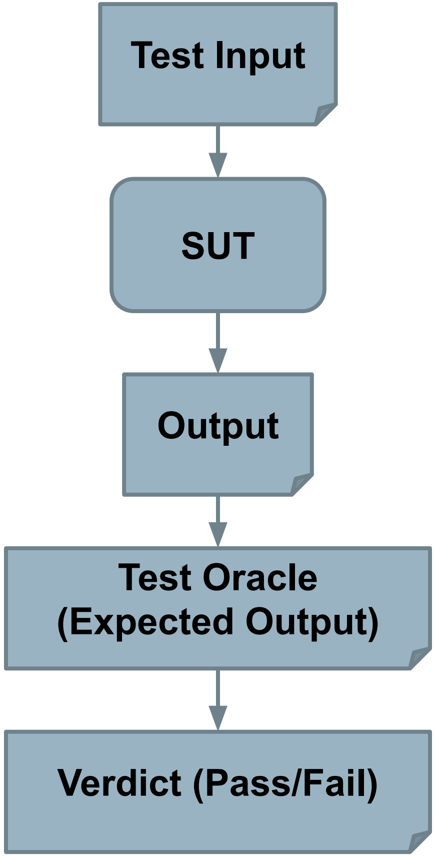
Gregory Gay
TDA594 - December 8, 2020

Today's Goals

- Discuss testing at the system level.
  - UI and Integration versus Unit Testing.
- Introduce process for creating System-Level Tests.
  - Identify Independently Testable Functionality
  - Identify Representative Values
  - Generate Test Case Specifications
  - Generate Concrete Test Cases

Software Testing

- An investigation into system quality.
- Based on sequences of **stimuli** and **observations**.
  - **Stimuli** that the system must react to.
  - **Observations** of system reactions.
  - **Verdicts** on correctness.

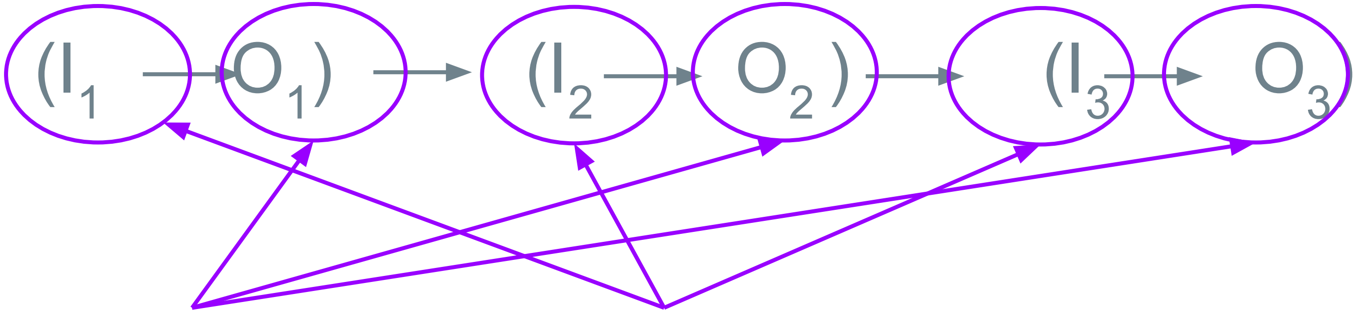


Axiom of Testing

"Program testing can be used to show the presence of bugs, but **never their absence**."

- Dijkstra

Anatomy of a Test Case



Test Inputs

How we "stimulate" the system (method call, API request, GUI event).

Test Oracle

How we check the correctness of the resulting observation (assertions).

if O_n = Expected(O_n)

then… Pass

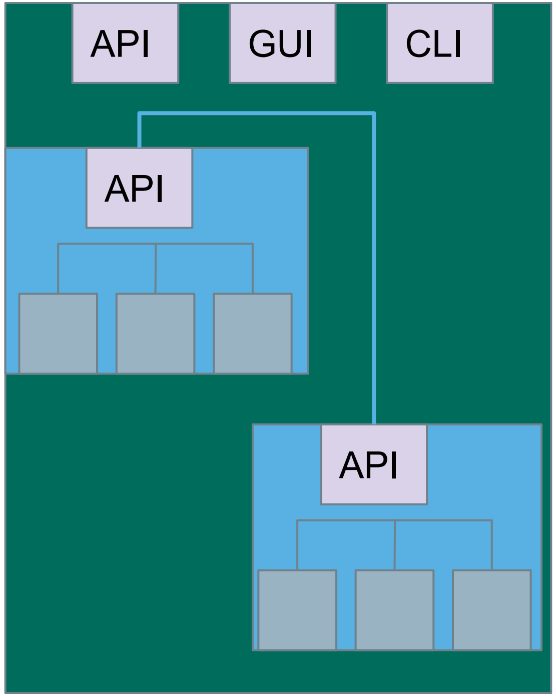
else… Fail

Anatomy of a Test Case

- **Initialization**
  - Any steps that must be taken before test execution.
- **Test Steps**
  - Interactions with the system, and comparisons between expected and actual values.
- **Tear Down**
  - Any steps that must be taken after test execution.

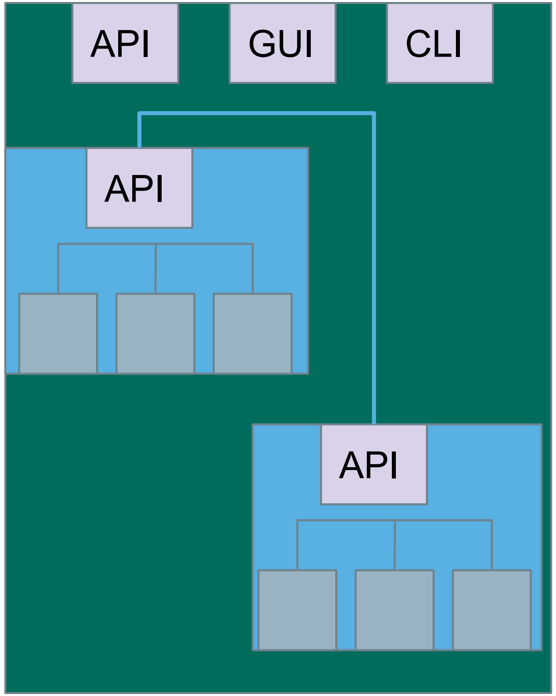
Testing Stages

- We interact with **systems** through **interfaces**.
- Systems built from **subsystems**.
  - With their own interfaces.
- Subsystems built from **units**.
  - Classes work with other classes through methods (interfaces).



Testing Stages

- Unit Testing
  - Do the methods of a class work?
- System Testing
  - Subsystem Integration Testing
    - Do the collected units work?
  - System Integration Testing
    - Do the collected subsystems work?
  - UI Testing
    - Does interaction through UIs work?



Unit Testing

- Testing the smallest “unit” that can be tested.
  - Often, a class and its methods.
- Tested in **isolation** from all other units.
  - **Mock** the results from other classes.
- Test input = method calls.
- Test oracle = assertions on output/class variables.

Unit Testing

- For a unit, tests should:
  - Test all "jobs" associated with the unit.
    - Individual methods belonging to a class.
    - Sequences of methods that can interact.
  - Set and check value of all class variables.
    - Examine how variables change after method calls.
    - Put the variables into all possible states (types of values).

Unit Testing - WeatherStation

WeatherStation
identifier temperature pressure
checkLink() reportWeather() reportInstrumentStatus() restart(instrumentName) shutdown(instrumentName) reconfigure(instrumentName, commands)

Unit tests should cover:

- Set and check class variables.
    - Can any methods change identifier, temperature, pressure?
- Each “job” performed by the class.
    - Single methods or method sequences.
    - Vary the order methods are called.
    - Each outcome of each “job” (error handling, return conditions).

Writing a Unit Test

Java-based unit testing (JUnit).

- Choose a target unit.
  - Ex. Calculator to right.
- Create a test class.
  - Unit tests are methods marked with @Test.

```
public class Calculator {
  public int evaluate (String
                expression) {
    int sum = 0;
    for (String summand:
                expression.split("\\+"))
      sum += Integer.valueOf(summand);
    return sum;
  }
}
```

JUnit Test Skeleton

@Test annotation defines a single test:

Type of scenario, and expectation on outcome.
I.e., `testEvaluate_GoodInput()` or `testEvaluate_NullInput()`

```
@Test
public void test<Feature or Method Name>_<Testing Context>() {
    //Define Inputs
    try{ //Try to get output.
    }catch(Exception error){
        fail("Why did it fail?");
    }
    //Compare expected and actual values through assertions or through
    //if-statements/fail commands
}
```

Writing JUnit Tests

```
public class Calculator {
  public
         expression) {
    int sum = 0;
    for (String summand:
              expression.split(
      sum += Integer.valueOf(summand);
    return sum;
  }
}
```

Each test is denoted with keyword **@test**.

Initialization

Test Steps

```
import static
org.junit.jupite
import org.junit

public class CalculatorTest {
  @Test
  void testEvaluate_Valid_ShouldPass(){
    Calculator calculator = new Calculator();
    int sum = calculator.evaluate("1+2+3");
    assertEquals(6, sum);
     calculator = null;
  }
}
```

Convention - name the test class after the class it is testing or the functionality being tested.

Input

Oracle

Tear Down

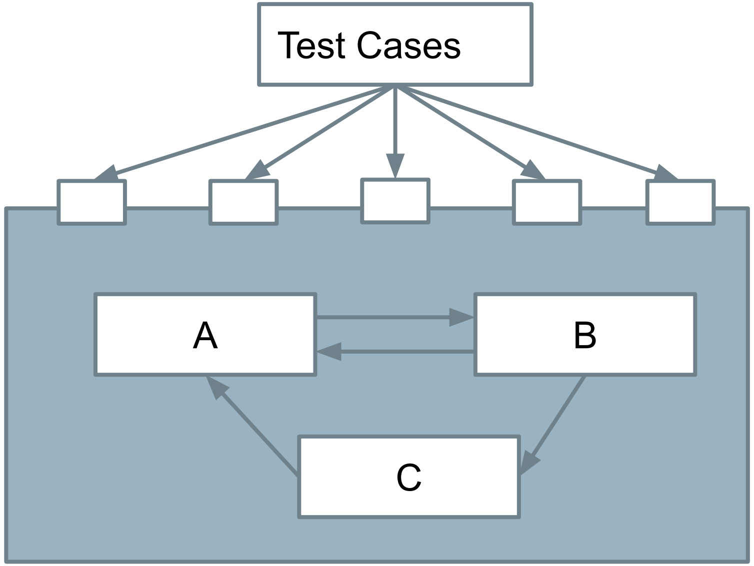
Integration Testing

- After testing units, test their **integration**.
  - Integrate units in one subsystem.
  - Then integrate the subsystems.
- Test input through a defined interface.
  - Focus on showing that functionality accessed through interfaces is correct.
  - Subsystems: “Top-Level” Class, API
  - System: API, GUI, CLI, …

Integration Testing

Subsystem made up classes of A, B, and C. We have performed unit testing...

- Classes work together to perform subsystem functions.
- Tests applied to the interface of the subsystem they form.
- Errors in combined behavior not caught by unit testing.

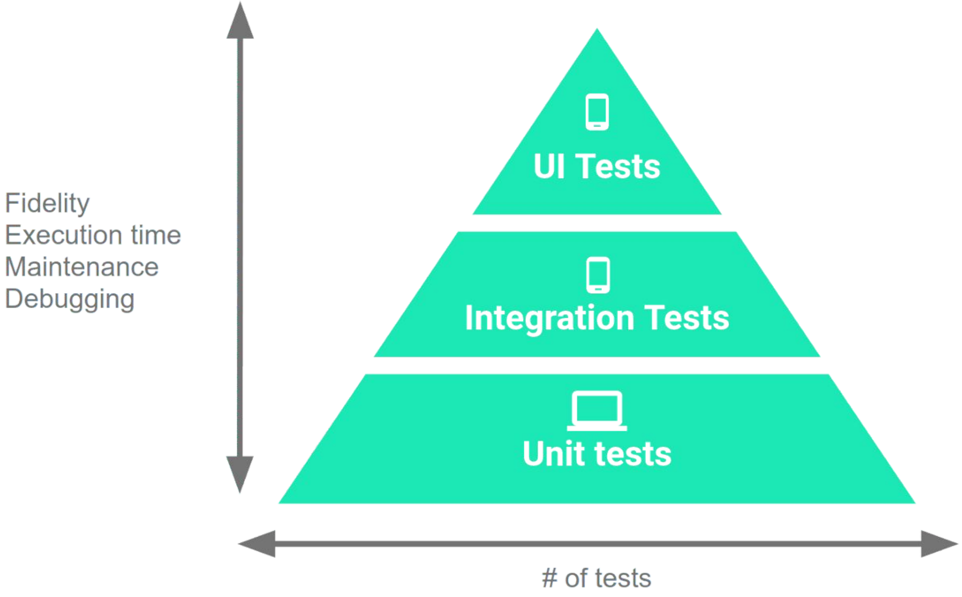


Interface Errors

- Interface Misuse
  - Malformed data, order, number of parameters.
- Interface Misunderstanding
  - Incorrect assumptions made about called component.
  - A binary search called with an unordered array.
- Timing Errors
  - Producer of data and consumer of data access data in the wrong order.

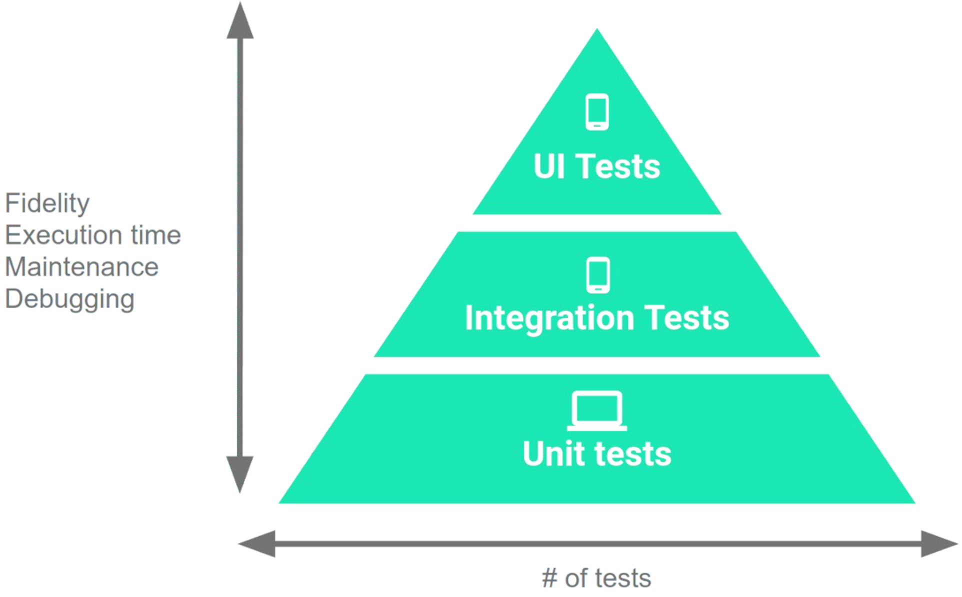
Testing Percentages

- Unit tests verify behavior of a single class.
  - 70% of your tests.
- Integration tests verify class interactions in a portion of the app.
  - 20% of your tests.
- UI tests verify end-to-end journey over the app.
  - 10% of your tests.



Testing

- 70/20/10 recommended.
- Unit tests execute quickly, without emulator or devices.
- UI tests must run in Android, are very slow.
- Well-tested units reduce likelihood of integration issues, making high levels of testing easier.



Writing Integration and UI Tests

- Testing framework depends on language and interface type.
  - Android: JUnit (Integration - AndroidX, UI - Espresso)
  - RESTful API: Postman
  - Browser-based GUI: Selenium

Android UI Test

Uses Espresso testing libraries to interact with Views and Intents. (Part of AndroidX)

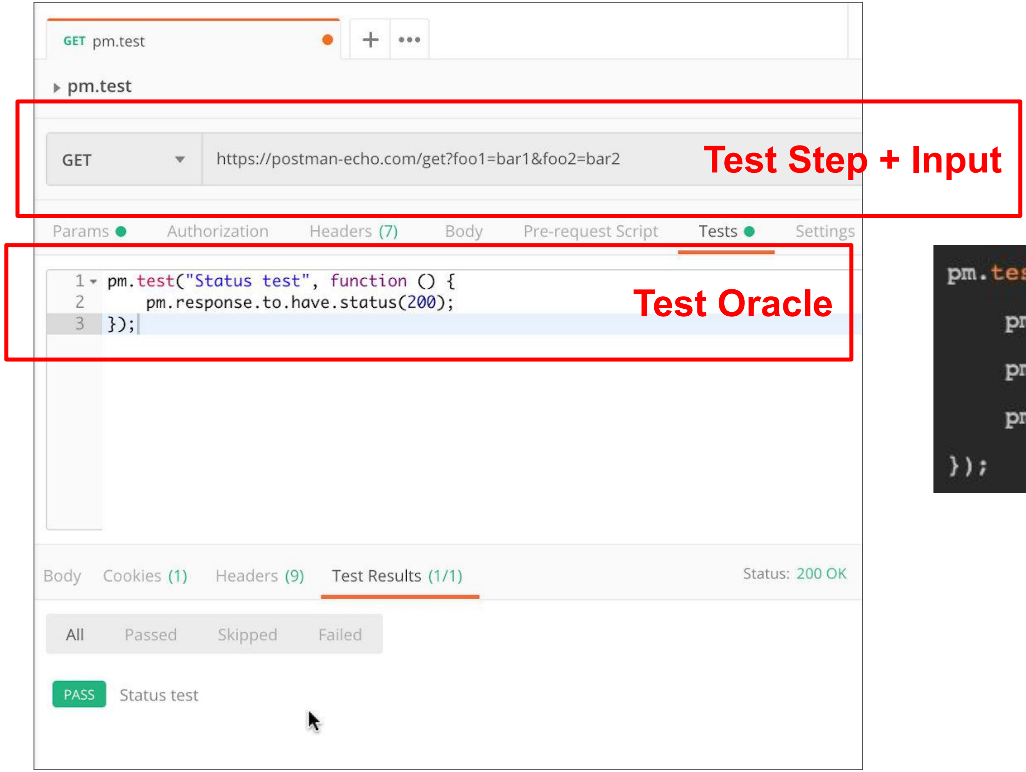
```
@Test
public void successfulLogin() {
    LoginActivity activity =
        ActivityScenario.launch(LoginActivity.class);
    onView(withId(R.id.user_name)).perform(typeText("test_user"));
    onView(withId(R.id.password))
       .perform(typeText("correct_password"));
    onView(withId(R.id.button)).perform(click());
    assertThat(getIntents().first())
        .hasComponentClass(HomeActivity.class);
}
```

Setup

Test Steps + Input

Test Oracle

RESTful API Test - Postman




```
pm.test("response should be okay to process", function () {
    pm.response.to.not.be.error;
    pm.response.to.have.jsonBody("");
    pm.response.to.not.have.jsonBody("error");
});
```

System-Level Tests and SPLs

- Variability is a ***system-level concept***.
  - Feature options tend to be entire classes or subsystems.
- **Unit testing during domain engineering.**
  - Assets tested in isolation.
- Many interaction errors between features, depending on chosen options.
  - **System testing during application engineering.**

Creating System-Level Test Cases

Creating System-Level Tests

Identify Independently Testable Functionality

Identify functionality that can be tested in (relative) isolation.

Identify Choices

Identify the choices you control when testing.

Identify Representative Input Values

Identify values for each choice that lead to different function outcomes.

Generate Test Case Specifications

Identify abstract test cases based on choice combinations.

Generate Test Cases

Identify concrete input/expected output pairs.

Independently Testable Functionality

- **A well-defined function that can be tested in (relative) isolation.**
  - Based on the “verbs” - what can we do with this system?
  - The high-level functionality offered by an interface.
  - UI - look for user-visible functions.
    - Web Forum: Sorted User List can be accessed.
    - Accessing the list is a testable functionality.
    - Sorting the list is not (low-level, unit testing target)

Units and "Functionality"

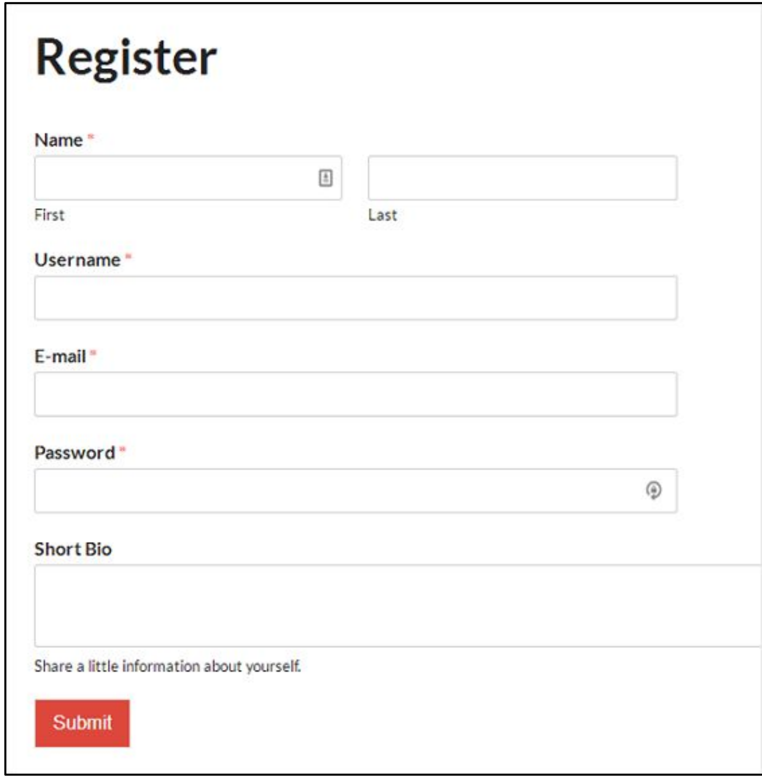
- Many tests written in terms of "units" of code.
- An independently testable function is a *capability* of the software.
  - Can be at class, subsystem, or system level.
  - **Defined by an interface.**

Identify the Choices

- What choices do we make when using a function?
  - Anything we control when we test.
- What are the inputs to that feature?
- What choices did we make for variation points?
- Are there environmental factors we can vary?
  - Networking environment, file existence, file content, database connection, database contents, disk utilization, …

Ex: Register for Website

- What are the inputs to that feature?
  - `(first name, last name, date of birth, e-mail)`
- Website is part of product line with different database options.
  - `(database type)`
- Consider implicit environmental factors.
  - `(database connection, user already in database)`



Parameter Characteristics

- Identify choices by understanding how parameters are used by the function.
- Type information is helpful.
  - `firstName` is string, database contains `UserRecords`.
- … but context is important.
  - Reject registration if in database.
  - … or database is full.
  - … or database connection down.

Parameter Context

- Input parameter split into multiple "choices" based on contextual use.
  - "Database" is an implicit input for User Registration, but it is not **one** choice.
  - "Database Connection Status", "User Record in Database", "Percent of Database Filled" influence function outcome.
    - **The Database input results in three choices.**
    - **Test cases will be based on these choices.**

Examples

Class Registration System

What are some independently testable functions?

- Register for class
- Drop class
- Transfer credits from another university
- Apply for degree

Example - Register for a Class

What are the choices?

- Course number to add
- Student record
- What about a course database? Student record database?
- **What else influences the outcome?**

Example - Register for a Class

- Student Record is an implicit input.
- How is it used?
  - Have you already taken the course?
  - Do you meet the prerequisites?
  - What university are you registered at?
  - Can you take classes at the university the course is offered at?

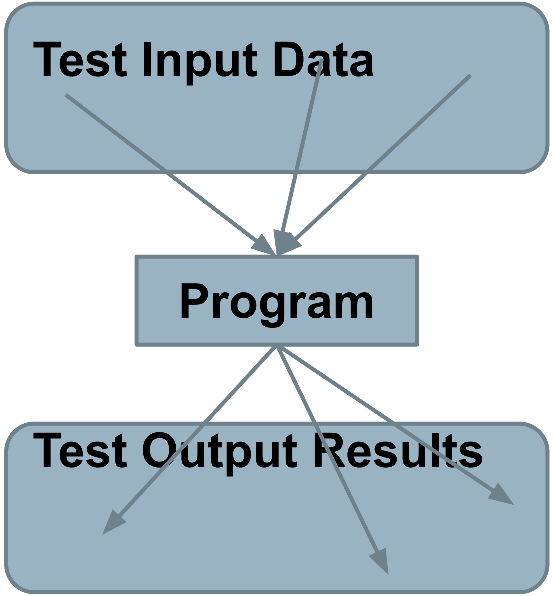
Example - Register for a Class

- Choices:
  - Course to Add
  - Does course exist?
  - Does student record exist?
  - Has student taken the course?
  - Which university is student registered at?
  - Is course at a valid university for the student?
  - Can student record be retrieved from database?
  - Does the course exist?
  - Does student meet the prerequisites?

Let’s take a break.

Identifying Representative Values

- We know the functions.
- We have a set of choices.
- What values should we try?
  - For some choices, finite set.
  - For many, near-infinite set.
- **What about exhaustively trying all options?**



Exhaustive Testing

Take the arithmetic function for the calculator:

```
add(int a, int b)
```

- How long would it take to exhaustively test this function?

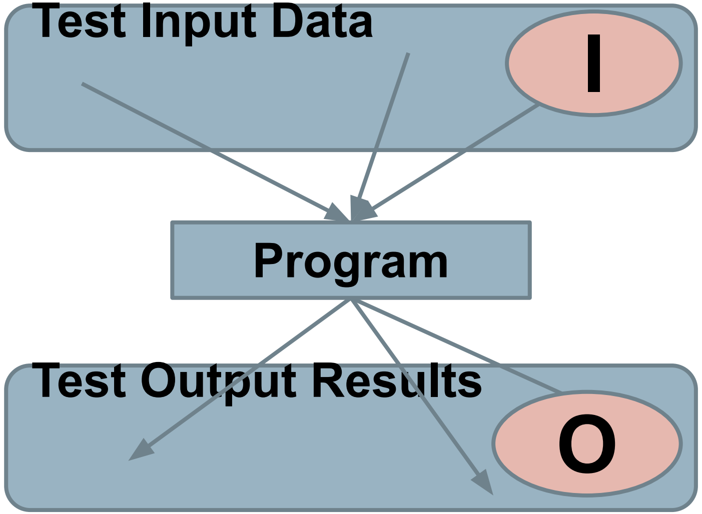
2^{32} possible integer values for each parameter.
= 2^{32} x 2^{32} = 2^{64} combinations = 10^{13} tests.

1 test per nanosecond
= 10^{5} tests per second
= 10^{10} seconds

or… about 600 years!

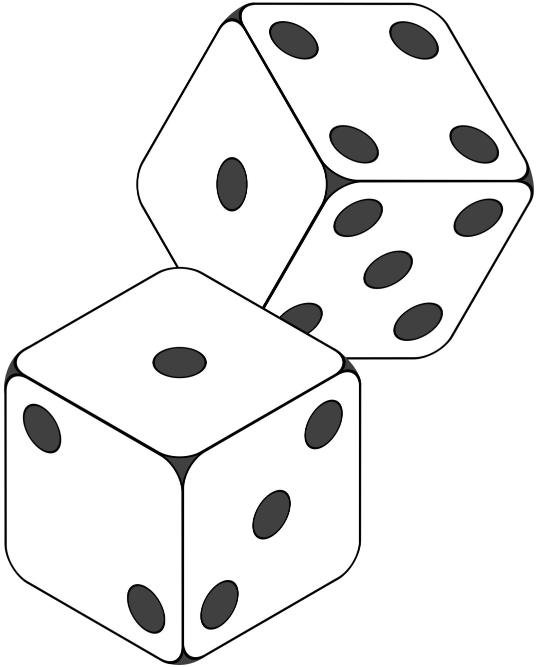
Not all Inputs are Created Equal

- Many inputs lead to same outcome.
- Some inputs better at revealing faults.
  - We can't know which in advance.
  - Tests with different input better than tests with similar input.



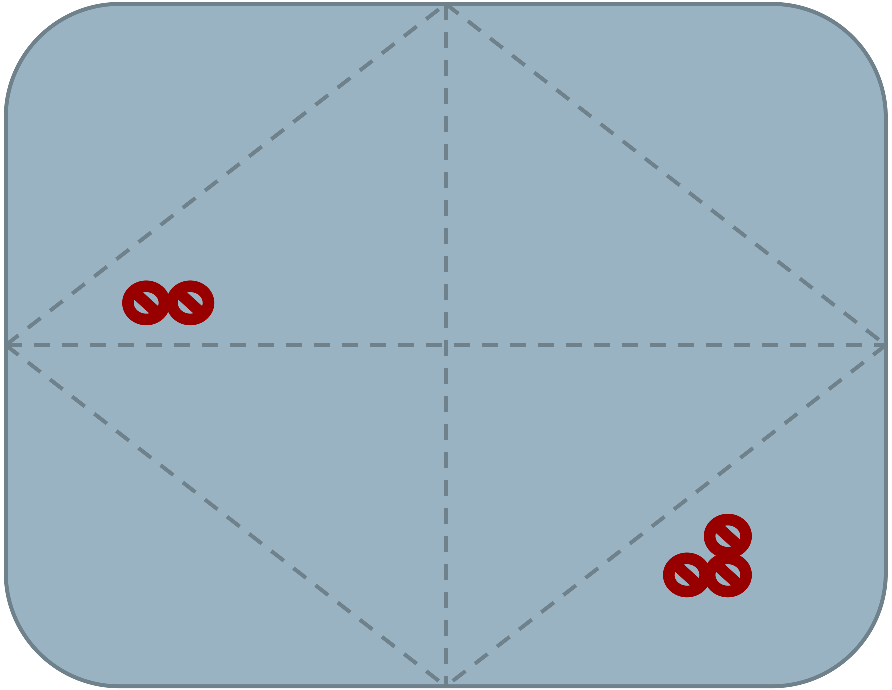
Random Testing

- Pick inputs uniformly from the distribution of all inputs.
- All inputs considered equal.
- Keep trying until out of time.
- No designer bias.
- Removes manual tedium.



Why Not Random?

Input Partitioning



- Consider possible values for a variable.
- Faults sparse in space of all inputs, but dense in parts where they appear.
  - Similar input to failing input also likely to fail.
- Try input from partitions, hit dense fault space.

Equivalence Class

- Divide the input domain into **equivalence classes**.
  - Inputs from a group interchangeable (trigger same outcome, result in the same behavior, etc.).
  - If one input reveals a fault, others in this class (probably) will too. In one input does not reveal a fault, the other ones (probably) will not either.
- Partitioning based on intuition, experience, and common sense.

Example

`substr(string str, int index)`

What are some possible partitions?

- index < 0
- index = 0
- index > 0
- str with length < index
- str with length = index
- str with length > index
- ...

Choosing Input Partitions

- Equivalent output events.
- Ranges of numbers or values.
- Membership in a logical group.
- Time-dependent equivalence classes.
- Equivalent operating environments.
- Data structures.
- Partition boundary conditions.

Look for Equivalent Outcomes

- Look at the outcomes and group input by the outcomes they trigger.
- Example: **`getEmployeeStatus(employeeID)`**
  - Outcomes include: Manager, Developer, Marketer, Lawyer, Employee Does Not Exist, Malformed ID
  - Abstract values for choice `employeeID`.
    - Can potentially break down further.

Look for Ranges of Values

- Divide based on data type and how variable used.
  - Ex: Integer input. Intended to be 5-digit:
    - < 10000, 10000-99999, >= 100000
    - Other options: < 0, 0, max int
    - Can you pass it something non-numeric? Null pointer?
- Try “expected” values and potential error cases.

Look for Membership in a Group

Consider the following inputs to a program:

- A floor layout
- A country name.

- All can be partitioned into groups.
  - Apartment vs Business, Europe vs Asia, etc.
- Many groups can be subdivided further.
- Look for context that an input is used in.

Timing Partitions

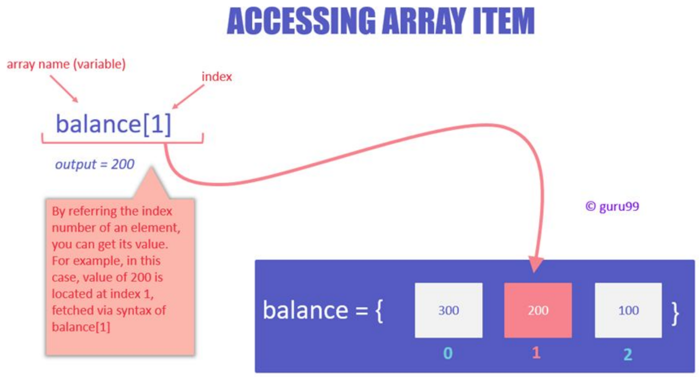
- Timing and duration of an input may be as important as the value.
  - Timing often implicit input.
    - Trigger an electrical pulse 5ms before a deadline, 1ms before the deadline, exactly at the deadline, and 1ms after the deadline.
    - Close program before, during, and after the program is writing to (or reading from) a disc.

Operating Environments

- Environment may affect behavior of the program.
- Environmental factors can be partitioned.
  - Memory may affect the program.
  - Processor speed and architecture.
  - Client-Server Environment
    - No clients, some clients, many clients
    - Network latency
    - Communication protocols (SSH vs HTTPS)

Data Structures

- Data structures are prone to certain types of errors.
- For arrays or lists:
  - Only a single value.
  - Different sizes and number filled.
  - Order of elements: access first, middle, and last elements.



Input Partition Example

What are the input partitions for:

```
max(int a, int b) returns (int c)
```

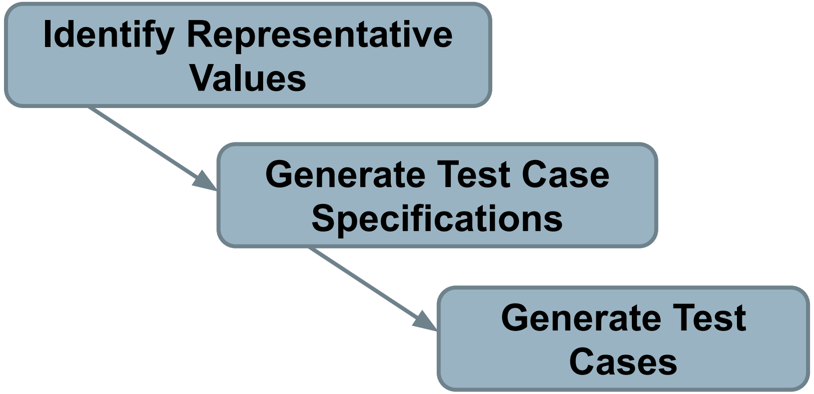
We could consider `a` or `b` in isolation:

```
a < 0, a = 0, a > 0
```

Consider combinations of `a` and `b` that change outcome:

```
a > b, a < b, a = b
```

Revisit the Roadmap



For each independently testable function, we want to:

1. Partition each choice into representative values.
2. Choose one partition for each choice to form a complete abstract test specification.
3. Assigning concrete values from each partition.

Forming Specification

Function `insertPostalCode(int N, list A)`.

- Partition choices into equivalence classes.
  - int N is a 5-digit integer between 10000 and 99999.
    - Possible partitions: <10000, 10000-99999, >100000
  - list A is a list of length 1-10.
    - Possible partitions: Empty List, List of Length 1, List Length 2-10, List of Length > 10

From Partitions to Test Case

Choose concrete values for each combination of input partitions:

`insertPostalCode(int N, list A)`

`int N`

< 10000
10000 - 99999
> 99999

`list A`

Empty List
List[1]
List[2-10]
List[>10]

Test Specifications:

(3 * 4 = 12 abstract specifications)

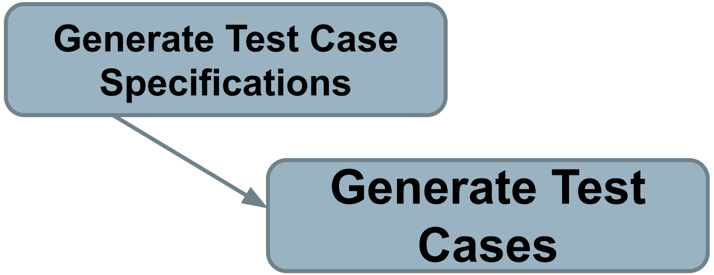
```
insert(< 10000, Empty List)
insert(10000 - 99999, list[1])
insert(> 99999, list[2-10])
...
```

Test Cases:

(Each specification = 1000s of potential test cases)

```
insert(5000, {})
insert(96521, {11123})
insert(150000, {11123, 98765})
...
```

Generate Test Cases



`substr(string str, int index)`

Specification:

`str`: length >=2, contains special characters

`index`: value > 0

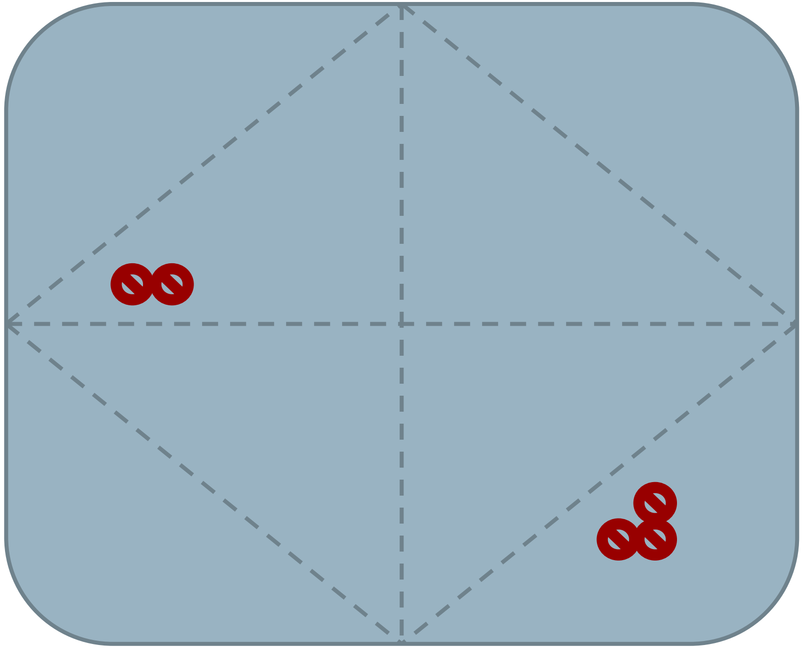
Test Case:

`str` = “ABCC!\n\t7”

`index`= 5

Boundary Values

- Errors tend to occur at the boundary of a partition.
- Remember to select inputs from those boundaries.

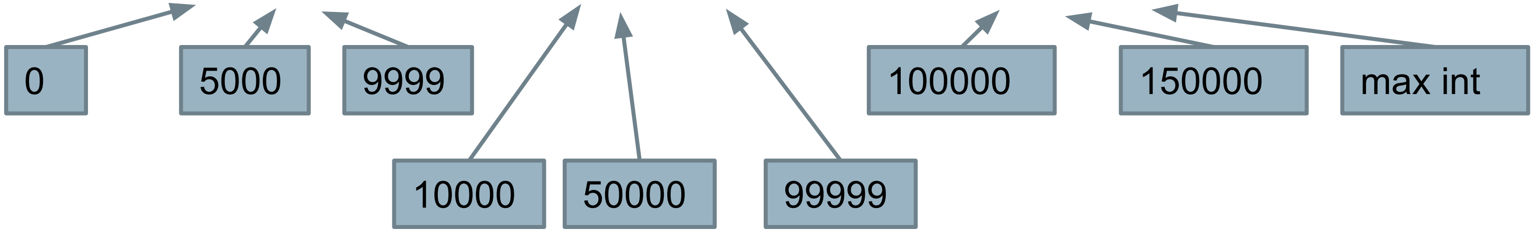


Choosing Test Case Values

Choose test case values at the boundary (and typical) values for each partition.

- If an input is intended to be a 5-digit integer between 10000 and 99999, you want partitions:

<10000, 10000-99999, >100000



Activity - System-Level Testing

- Microservice related to Sets:
  - void insert(Set set, Object obj)
  - Boolean find(Set set, Object obj)
  - void delete(Set set, Object obj)
- For each function, identify choices.
- For each choice, identify the representative values.
- Create abstract test specifications with expected outcomes.

Activity - System-Level Testing

- **`insert(Set set, Object obj)`**
  - Choices **`(Number of Items)`** and **`(Object Status)`**.
- One test specification might be:
  - **Input:** Set with One Item/Object Already in Set
  - **Expected output:** Object not Added
  - You can omit redundant test specifications.

Solution - Choices and Values

- (Number of Items in Set)
  - Empty
  - 1
  - 2 +
  - 100 + (may be slower - make sure it still works)
- (Object Status)
  - In set already
  - Not in set
  - Null pointer

Solution - Test Specifications

Insert	*Empty/ Object not in Set*	*obj in container*
	One element / Object not in Set	*obj in container*
	Multiple elements / Object not in Set	*obj in container*
	100+ / Object not in Set	*obj in container*
	(any choice) / Object in Set	*Error or no change*
	(any choice) / Null Object	*Error*
Exists	*One element / Object in Set*	*True*
	Empty / Object not in Set	*False*
	100 + / Object in Set	*True*
	100 + / Object not in Set	*False*
	(any choice) / Null Object	*Error*

Delete	*One element / Object in Set*	*obj no longer in set*
	One element / Object not in Set	*no change (or error)*
	(any choice) / Null Pointer	*error*
	100 + / Object in Set	*obj no longer in set*
	Empty / Object not in Set	*no change (or error)*

We Have Learned

- Unit testing centered around a single class.
- System-level tests centered around integration of components, through an interface.
  - Identifying independently testable functionality.
  - Identify choices that influence function outcome.
  - Partitioning choices into representative values.
  - Combining choice values into test specifications.
  - Choosing concrete values for specifications.

Next Time

- System-level testing and feature interactions
  - Handling infeasible combinations.
  - Selecting a valid subset of representative values.

- Assignment 3 due tomorrow!
  - Questions?
  - Assignment 4 posted. Due December 20.

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY