Functional Testing

CSCE 747 - Lecture 4 - 01/21/2016

How do you come up with test cases?

Test Plans

- Plan for how we will test the system.
  - **What** is being tested (units of code, features).
  - **When** it will be tested (required stage of completion).
  - **How** it will be tested (what scenarios do we run?).
  - **Where** we are testing it (types of environments).
  - **Why** we are testing it (what purpose does this test serve?).
  - **Who** will be responsible for writing test cases (assign responsibility).

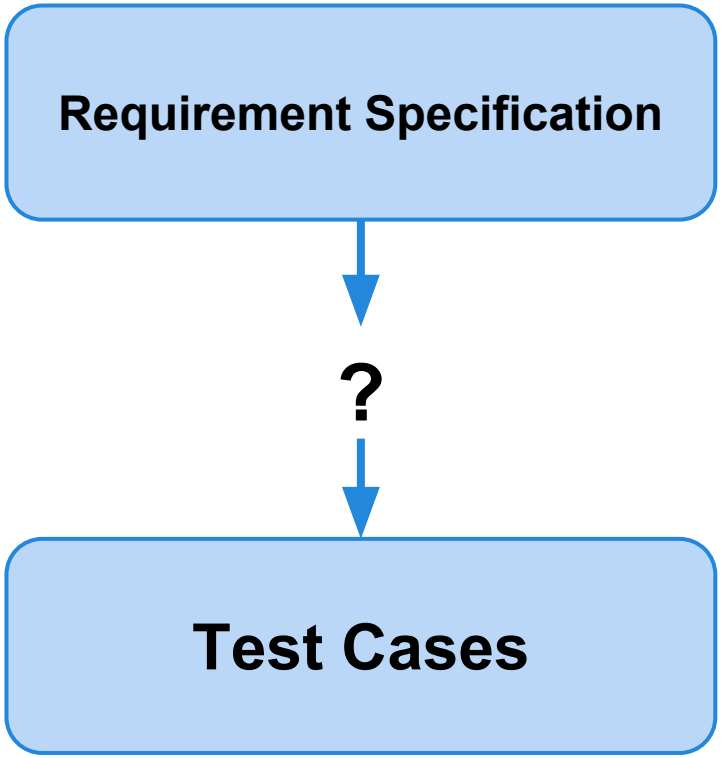
Where Does a Test Plan Come From?

- The first stage of software development is requirements specification.
  - Requirements = Properties that must be met by the final program.
  - Requirement Specification = How we the program will fulfill those properties.
- Verification ensures that the program conforms to its requirement specifications.
- Tests can be derived directly from these specifications.

Functional Testing

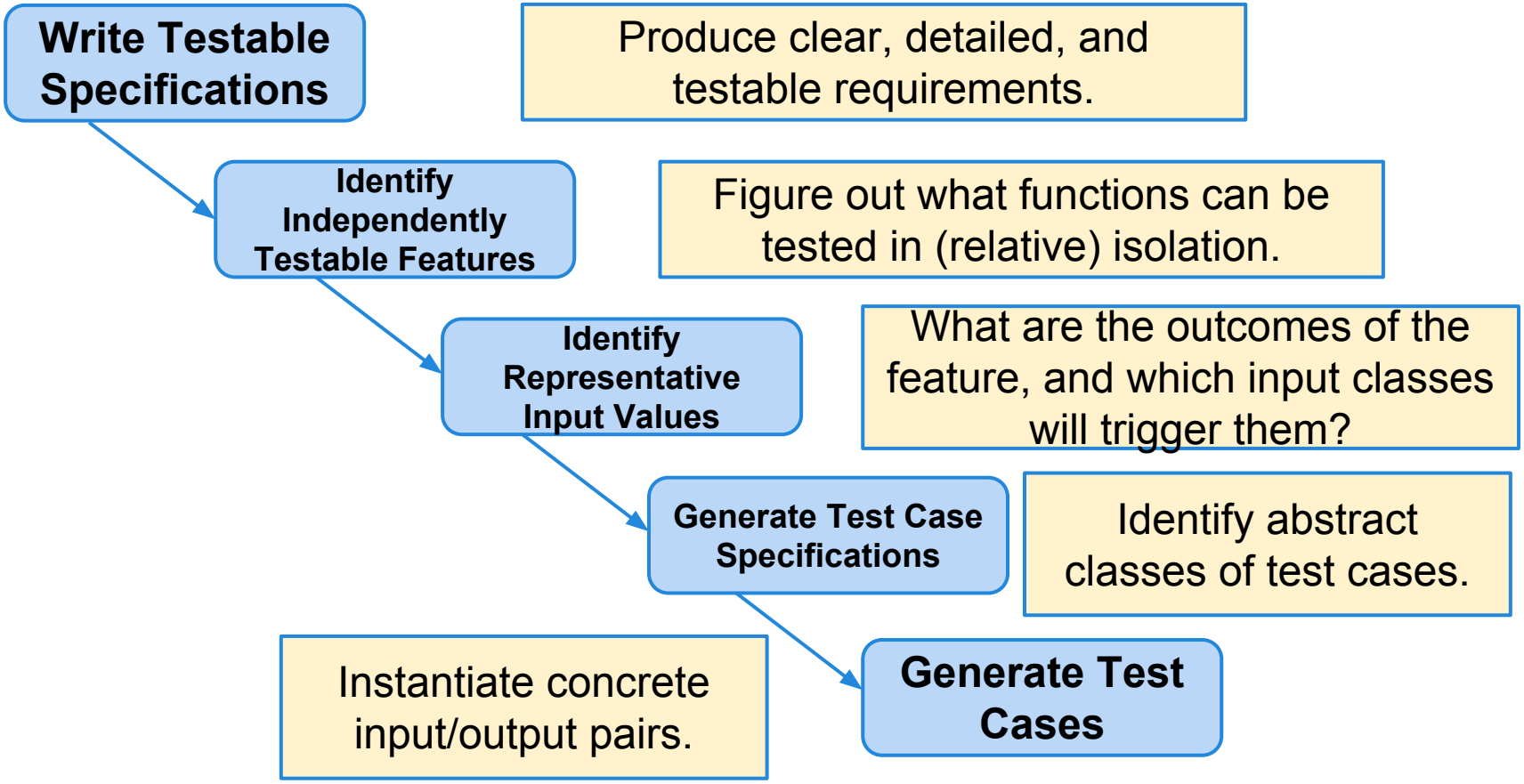
- Process of deriving tests from the requirement specifications.
  - Typically the baseline technique for designing test cases. Can begin as part of requirements specification, and continue through each level of design and implementation.
  - Basis of verification - builds evidence that the implementation conforms to its specification.
  - Effective at finding some classes of faults that elude code-based techniques.
    - i.e., incorrect outcomes and missing functionality

Partitioning



- Functional testing is based on the idea of **partitioning**.
  - You can't actually test individual requirements in isolation.
  - First, we need to partition the specification and software into features that can be tested.
  - Not all inputs have the same effect.
  - We can partition the outputs of a feature into the possible outcomes.
    - and the inputs, by what outcomes they cause (or other potential groupings).

Creating Requirements-Based Tests



Specification Verifiability

"The system should be easy to use by experienced engineers and should be organized in such a way that user errors are minimized."

- Problem is the use of vague terms such as "errors shall be minimized."
- The error rate must be quantified

Example Specifications

- After a high temperature is detected, an alarm must be raised quickly.
- Novice users should be able to learn the interface with little training.

How in the world do you make these specifications verifiable?

Test the Requirement

After a high temperature is detected, an alarm must be raised quickly.

Test Case 1:

- Input:
  - Artificially raise the temperature above the high temperature threshold.
- Procedure:
  - Measure the time it takes for the alarm to come on.
- Expected Output:
  - The alarm shall be on within 2 seconds.

Test the Requirement

Novice users should be able to learn the interface with little training.

Test Case 2:

- Input:
  - Identify 10 new users and put them through the training course (maximum length of 6 hours)
- Procedure:
  - Monitor the work of the users for 10 days after the training has been completed
- Expected Output:
  - The average error rate over the 10 days shall be less than 3 entry errors per 8 hours of work.

“Fixed” Specifications

- **Original:** After a high temperature is detected, an alarm must be raised quickly.
- **New:** When the temperature rises over the threshold, the alarm must activate within 2 seconds.

- **Original:** Novice users should be able to learn the interface with little training.
- **New:** New users of the system shall make less than 2 entry mistakes per 8 hours of operation after 6 hours of training.

Detailed is Not Always Testable

- Number of invalid attempts to enter the PIN before a user is suspended.
  - This count is reset when a successful PIN entry is completed for the user.
  - The default is that the user will never be suspended.
  - The valid range is from 0 to 10 attempts.

Problem: “never” is not testable.

(same for “always”)

How Many Tests Do You Need?

Testing a single requirement specification does not mean writing a single test.

- You normally have to write several tests to ensure that the requirement holds.
  - What are the different conditions that the requirement must hold under?
- Maintain traceability links from tests to the requirements they cover.

Independently Testable Feature

- Requirements are difficult to test in isolation. However, the system can usually be decomposed into the functions it provides.
- **An independently testable feature is a well-defined function that can be tested in (relative) isolation.**
- Identified to “divide and conquer” the complexity of functionality.

Units and Features

- Executable tests are typically written in terms of blocks of code (small “units” that can be executed).
  - Until we have code, we do not know what the units are.
- An independently testable feature is a *capability* of the software.
  - May not correspond to any one unit of code.
  - Can be at the class, subsystem, or system level.

Features and Parameters

Tests for features must be described in terms of all of the parameters and environmental factors that influence the feature's execution.

- What are the inputs to that feature?
  - User registration on a website might take in:
    - `(firstName, lastName, dateOfBirth, eMail)`
- Consider implicit environmental factors.
  - Registration also requires a user database.
    - The existence and contents of that database influence execution.

Parameter Characteristics

The key to identifying tests is in understanding *how* the parameters are used by the feature.

- Type information is helpful.
  - `firstName` is a string, the database contains `UserRecord` structs.
- … but context is important.
  - If the database already contains an entry for that combination of fields, registration should be rejected.
  - `dateOfBirth` is a collection of three integers, but those integers are not used for any arithmetic operations.

Examples

Class Registration System

What are some independently testable features?

- Add class
- Drop class
- Modify grading scale
- Change number of credits
- Graphical interface of registration page

Examples

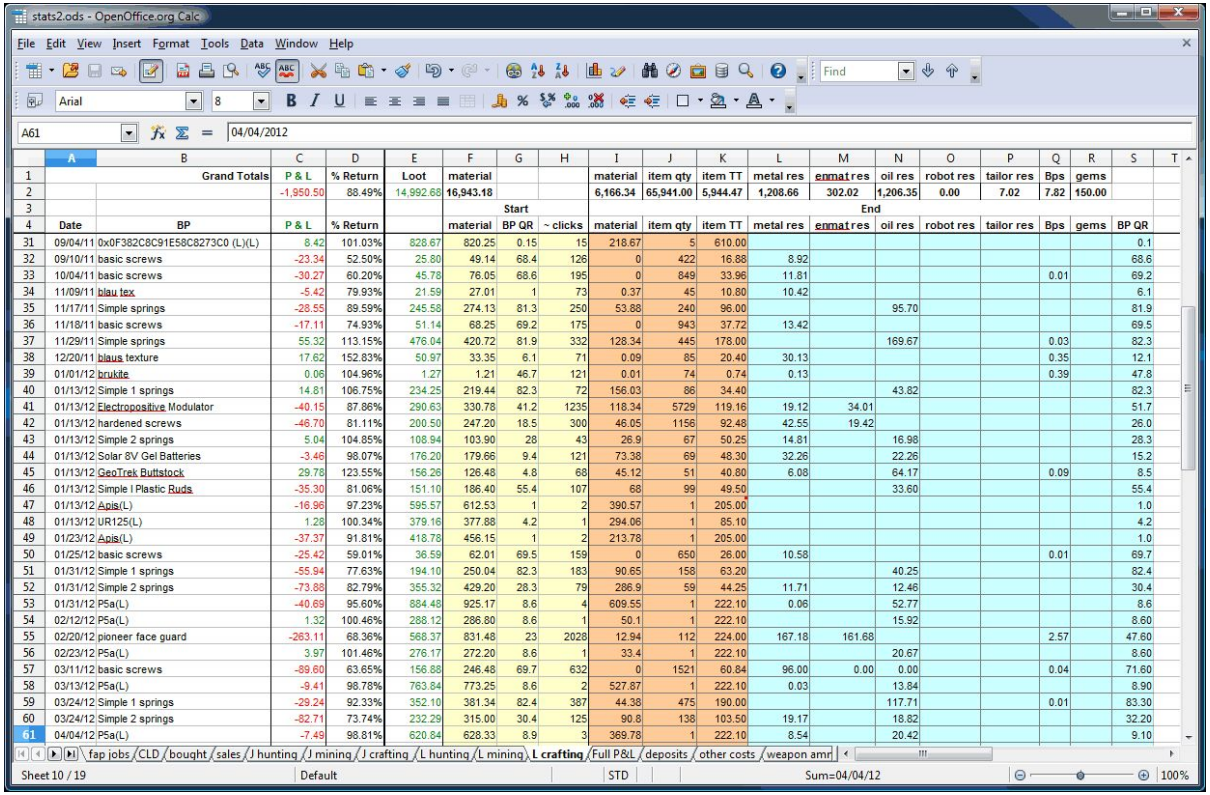
Adding a class

What are the parameters?

- Course number to add
- Grading basis
- Student record
- What about a course database? Student record database?

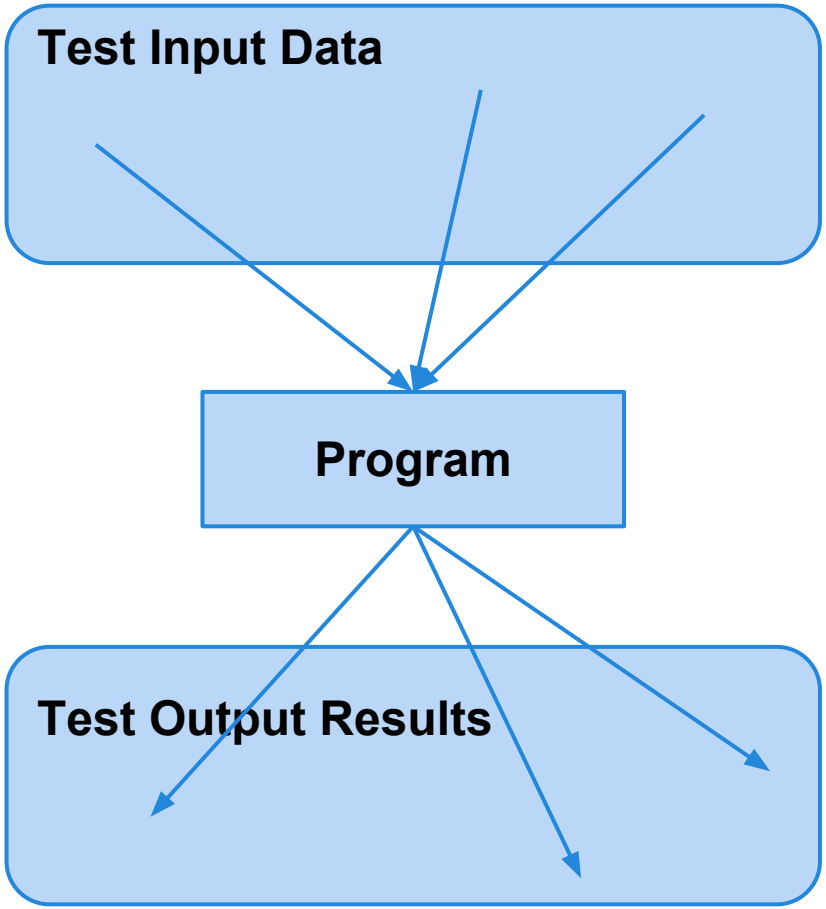
Independently Testable Features

What are three independently testable features of a spreadsheet?



Identifying Representative Values

- We know the features. We know their parameters.
- What input values should we pick?
- **What about exhaustively trying all inputs?**



Exhaustive Testing

Take the arithmetic function for the calculator:

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

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

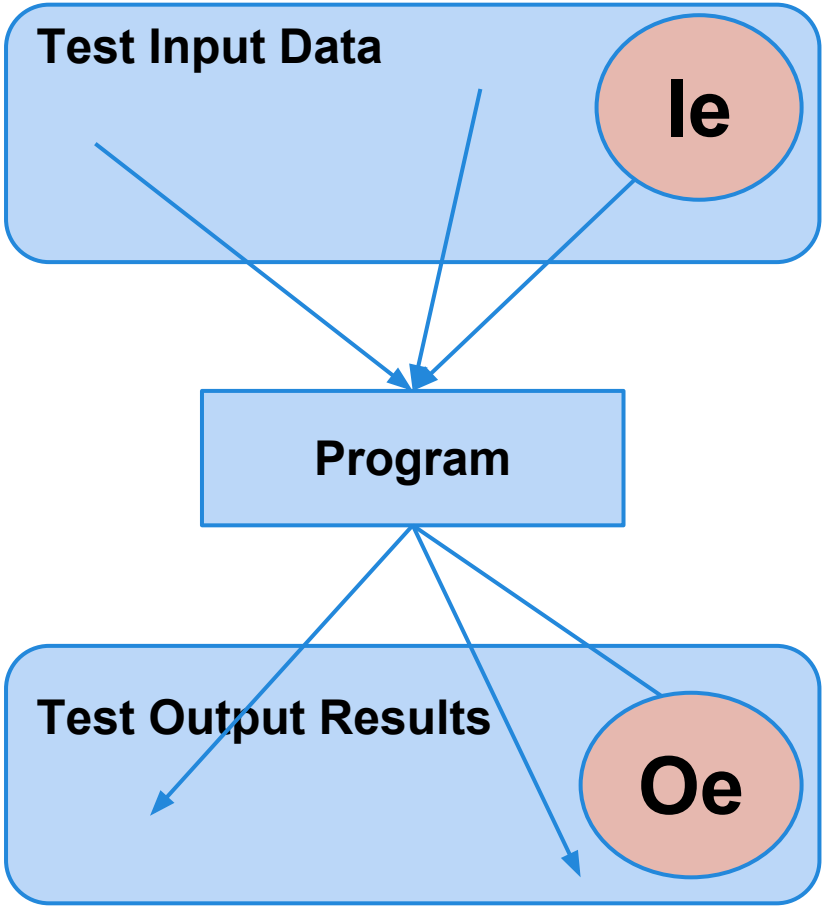
2^{32} possible integer values for each parameter.
$= 2^{32} \times 2^{32} = 2^{64}$ combinations $= 10^{13}$ tests.

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

or… about 600 years!

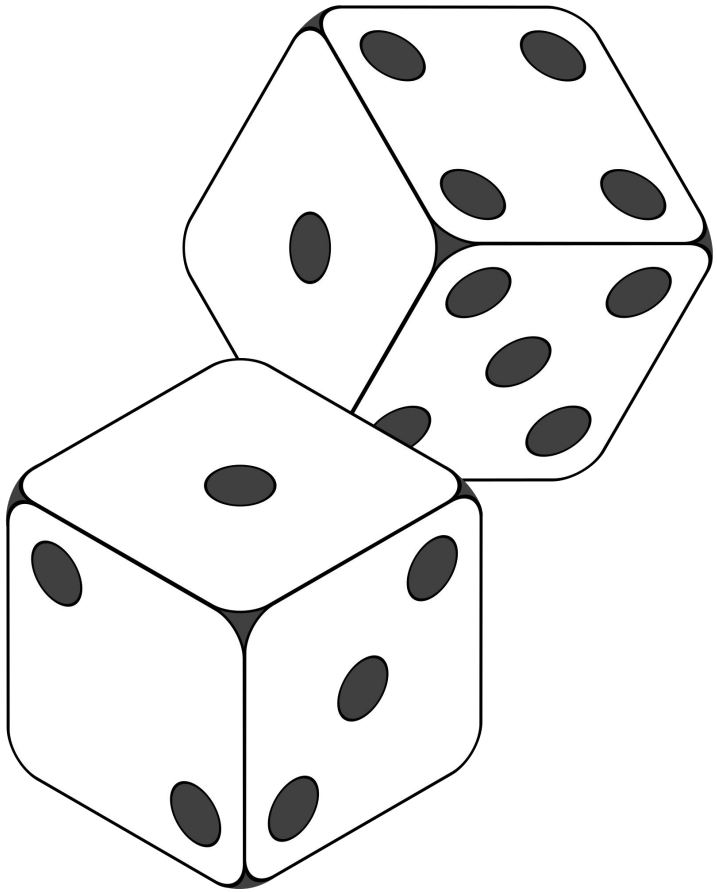
Not all Inputs are Created Equal

- We can't exhaustively test any real program.
  - **We don't need to!**
- Some inputs are better than others at revealing faults, but we can't know which in advance.
- Tests with different input than others are better than tests with similar input.



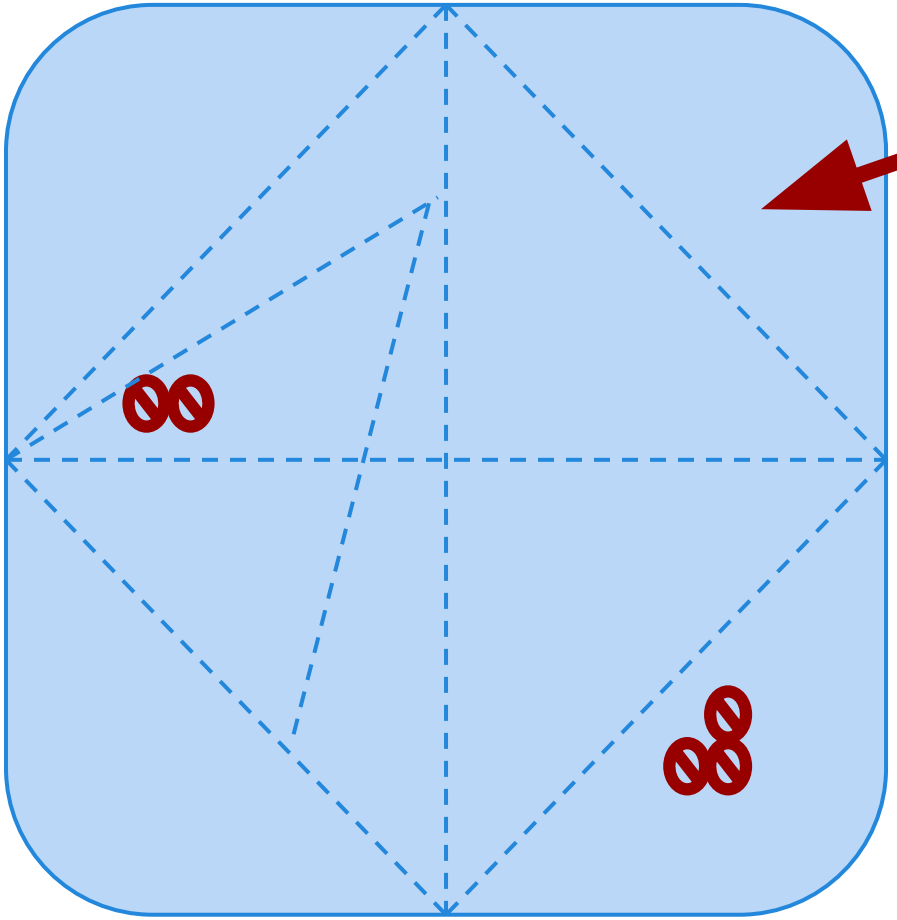
Random Testing

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



Why Not Random?

Input Partitioning



Faults are sparse in the space of all inputs, but dense in some parts of the space where they appear.

Program

By systematically trying input from each partition, we will hit the dense fault space.

Equivalence Class

- We want to divide the input domain into *equivalence classes*.
  - Inputs from a group can be treated as the same thing (trigger the same outcome, result in the same behavior, etc.).
  - If one test reveals a fault, others in this class (probably) will too. In one test does not reveal a fault, the other ones (probably) will not either.
- Perfect partitioning is difficult, so grouping based largely 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

- Look for equivalent output events.
- Look for ranges of numbers or values.
- Look for membership in a logical group.
- Look for time-dependent equivalence classes.
- Look for equivalent operating environments.
- Look at the data structures involved.
- Remember invalid inputs and boundary conditions.

Look for Equivalent Outcomes

- It is often easier to find good tests by looking at the outputs and working backwards.
  - Look at the outcomes of a feature and group input by the outcomes they trigger.
- Example: A graphics routine that draws lines on a canvas. Outcomes include:
  - No line
  - Thin, short line
  - Thin, long line
  - Thick, short line
  - … etc.

Look for Ranges of Values

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

  <10000, 10000-99999, >100000
- Other options: < 0, max int, real-valued numbers
- You may want to consider non-numeric values as a special partition.

Look for Membership in a Group

Consider the following inputs to a program:

- The name of a valid Java data type.
- A letter of the alphabet.
- A country name.

- All make up input partitions.
- All groups can be subdivided further.
- Look for context that an input is used in.

Timing Partitions

The timing and duration of an input may be as important as the value of the input.

- Very hard and very crucial to get right.

- Trigger an electrical pulse 5ms before a deadline, 1ms before the deadline, exactly at the deadline, and 1ms after the deadline.
- Push the “Esc” key before, during, and after the program is writing to (or reading from) a disc.

Equivalent Operating Environments

- The environment may affect the behavior of the program. Thus, environmental factors can be partitioned and varied when testing.

- Memory may affect the program.
- Processor speed and architecture.
  - Try with different machine specs.
- Client-Server Environment
  - No clients, some clients, many clients
  - Network latency
  - Communication protocols (SSH, FTP, Telnet)

Data Structure Can Suggest Partitions

Certain data structures are prone to certain types of errors. Use those to suggest equivalence classes.

For sequences, arrays, or lists:

- Sequences that have only a single value.
- Different sequences of different sizes.
- Derive tests so the first, middle, and last elements of the sequence are accessed.

Do Not Forget Invalid Inputs!

- Likely to cause problems. Do not forget to incorporate them as input partitions.
  - Exception handling is a well-known problem area.
  - People tend to think about what the program should do, not what it should protect itself against.

- Take these into account with all of the other selection criteria already discussed.

Input Partition Example

What are the input partitions for:

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

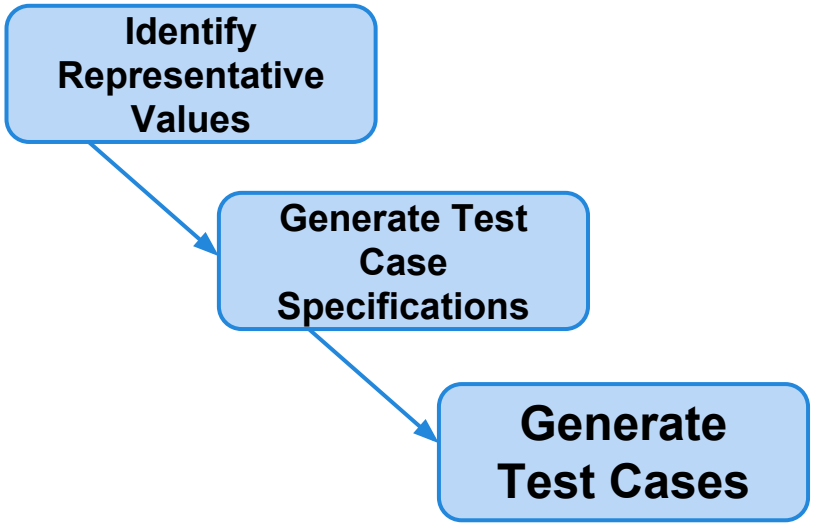
We could consider `a` or `b` in isolation:

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

We should also consider the combinations of `a` and `b` that influence the outcome of `c`:

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

Creating Requirements-Based Tests



For each independently testable feature, we want to:

1. Identify the representative value partitions for each input or output.
2. Use the partitions to form abstract test specifications for the combination of inputs.
3. Then, create concrete test cases by assigning concrete values from the set of input partitions chosen for each possible test specification.

Equivalence Partitioning

Feature `insert(int N, list A)`.

Partition inputs into equivalence classes.

1. `int N` is a 5-digit integer between 10000 and 99999. Possible partitions:

   <10000, 10000-99999, >100000

2. `list A` is a list of length 1-10. Possible partitions:

   Empty List, List of Length 1, List of Length 2-10, List of Length > 10

From Partition to Test Case

Choose concrete values for each combination of input partitions: `insert(int N, list A)`

`int N`

< 10000
10000 - 99999
> 99999

`list A`

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

Test Specifications:

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

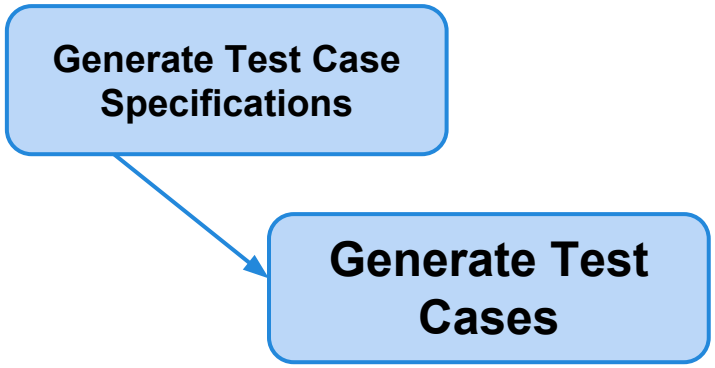
etc

Test Cases:

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

etc

Generate Test Cases




```
substr(string str, int index)
```

Specification:

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

`index`: value > 0

Test Case:

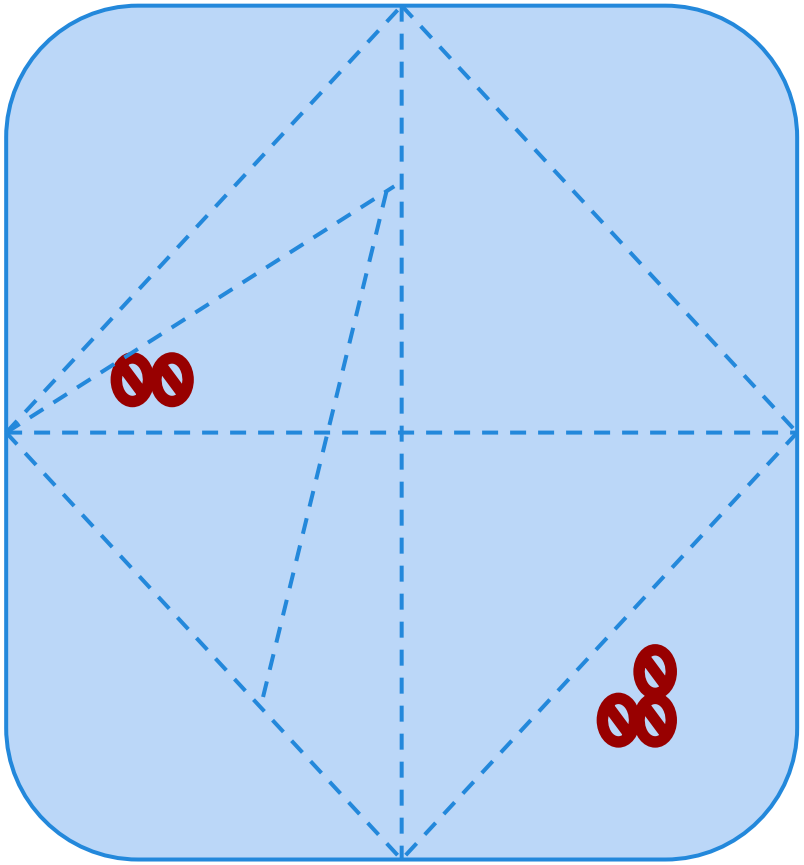
`str` = “ABCC!\n\t7”

`index`= 5

Boundary Values

Basic Idea:

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

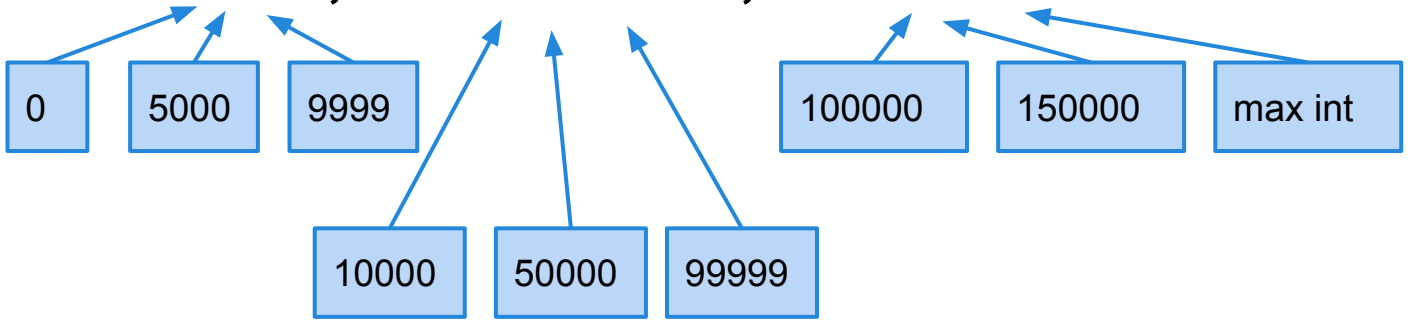


Choosing Test Case Values

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



Key Points

- The requirement specifications define the correct behavior of the system.
  - Therefore, the first step in testing should be to derive tests from the specifications.
- If the specification cannot be tested, you most likely have a bad requirement.
  - Rewrite it so it is testable.
  - Remove the requirement if it can't be rewritten.
- Tests must be written in terms of independently testable features.

Key Points

- Not all inputs will have the same outcome, so the inputs should be partitioned and test cases should be derived that try values from each partition.
- Input partitions can be used to form abstract *test specifications* that can be turned into 1+ concrete test cases.

Next Time

- Combinatorial Testing
  - How to come up with a reasonable number of requirements-based test cases.
  - Reading: Chapter 11

- Homework:
  - Assignment 1 Posted
  - Reading Assignment due Tuesday (11:59 PM)
    - James Whittaker. *The 10-Minute Test Plan*.
      - One page write-up:
        - summary + thoughts + suggestions for improvement