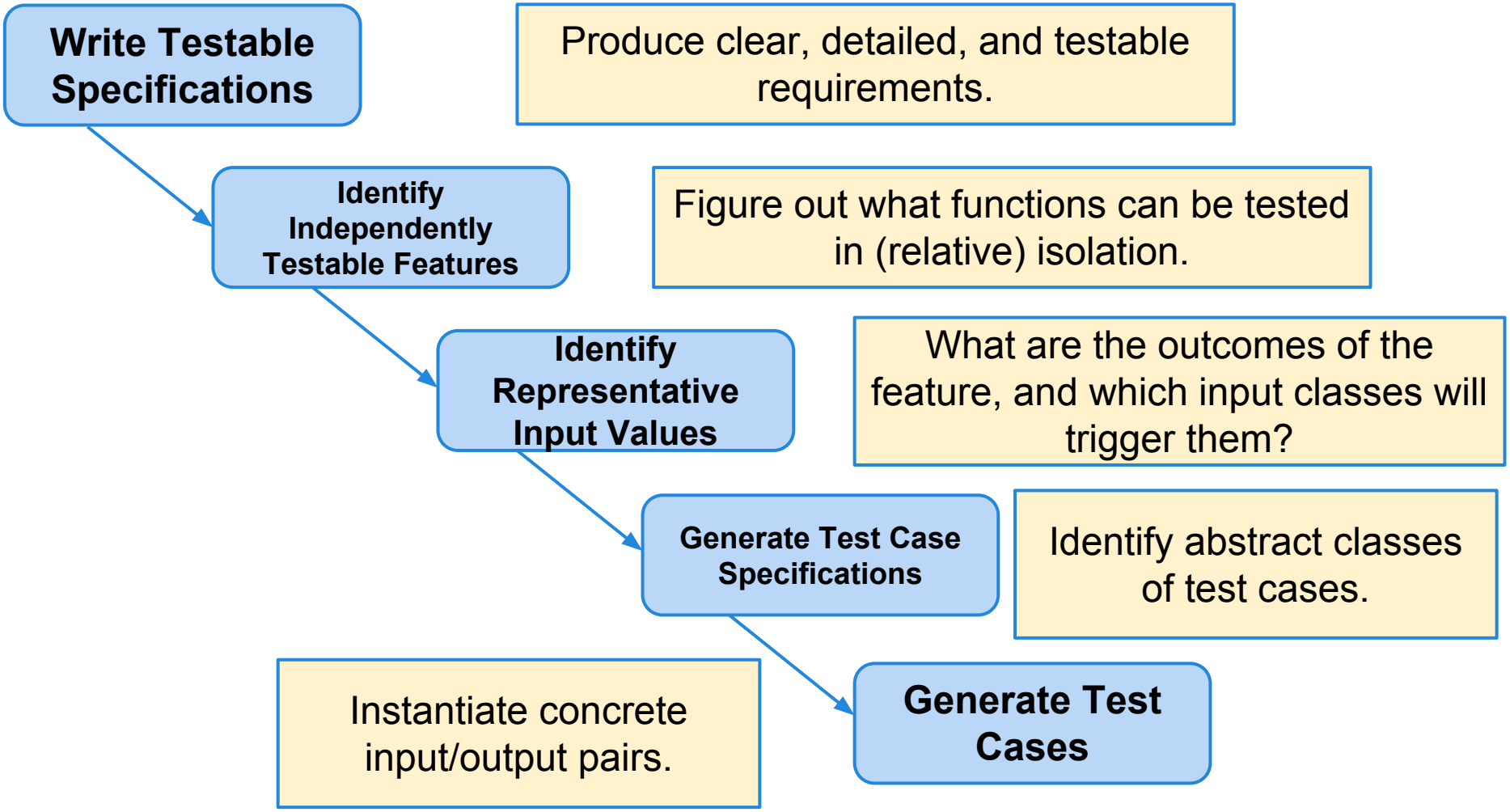
Developing Requirements-Based Tests

CSCE 740 - Lecture 8 - 09/21/2015

Creating Requirements-Based Tests



Today's Goals

- How to define and select requirements-based tests
  - Choosing representative input values.
  - Creating abstract test case "specifications"
  - Filling in the concrete input values.

Calculator Requirement

- Requirement 7.63: Divide-By-Zero
  - When a 0 is provided as input, it should be intercepted. Division-by-zero indicates an unsolvable expression.

Any problems?

- Input to what? Anything?
- Intercepted?

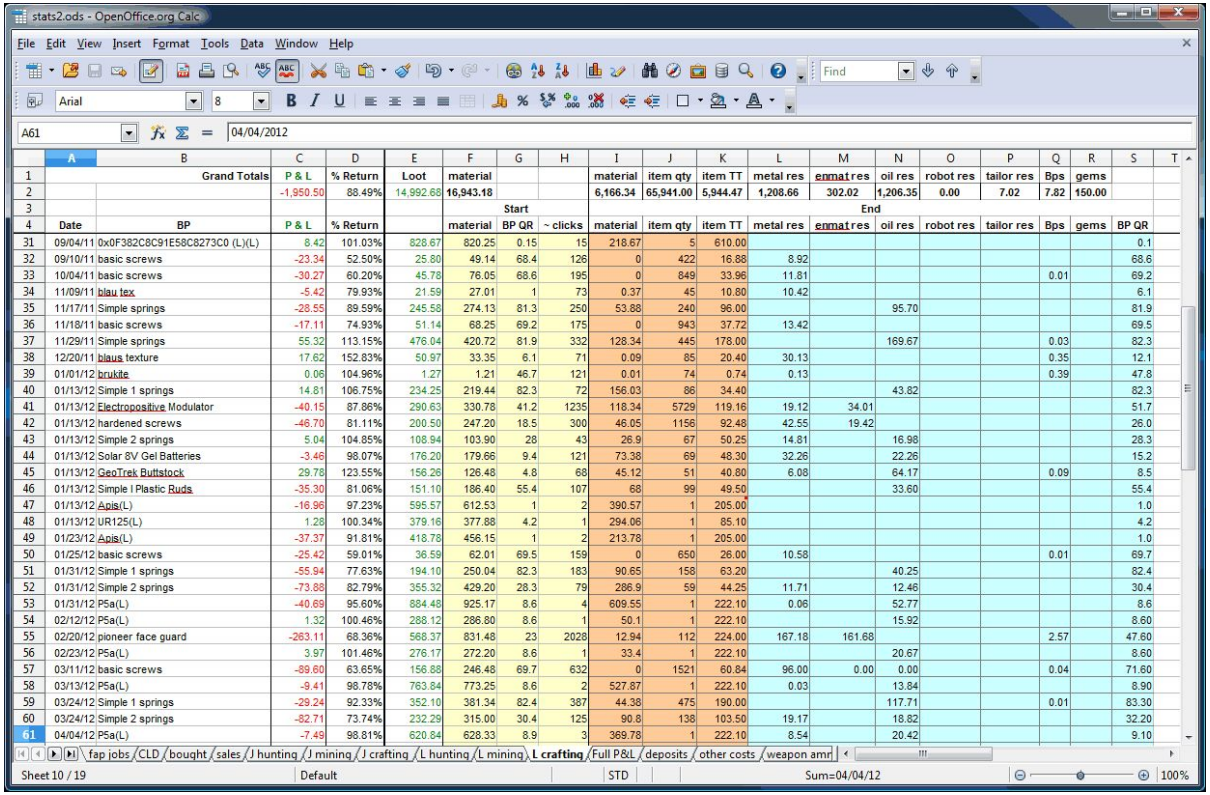
Calculator Requirement (Take 2)

- Requirement 7.63: Divide-By-Zero
  - When a 0 is provided as input as the divisor in any use of the division function, the software shall stop and issue an error message instead of performing the operation. Division-by-zero indicates an unsolvable expression.

- What are the independently testable features of a calculator?
- What are the parameters of the division feature? Their characteristics?
- How would you test that this requirement is fulfilled?

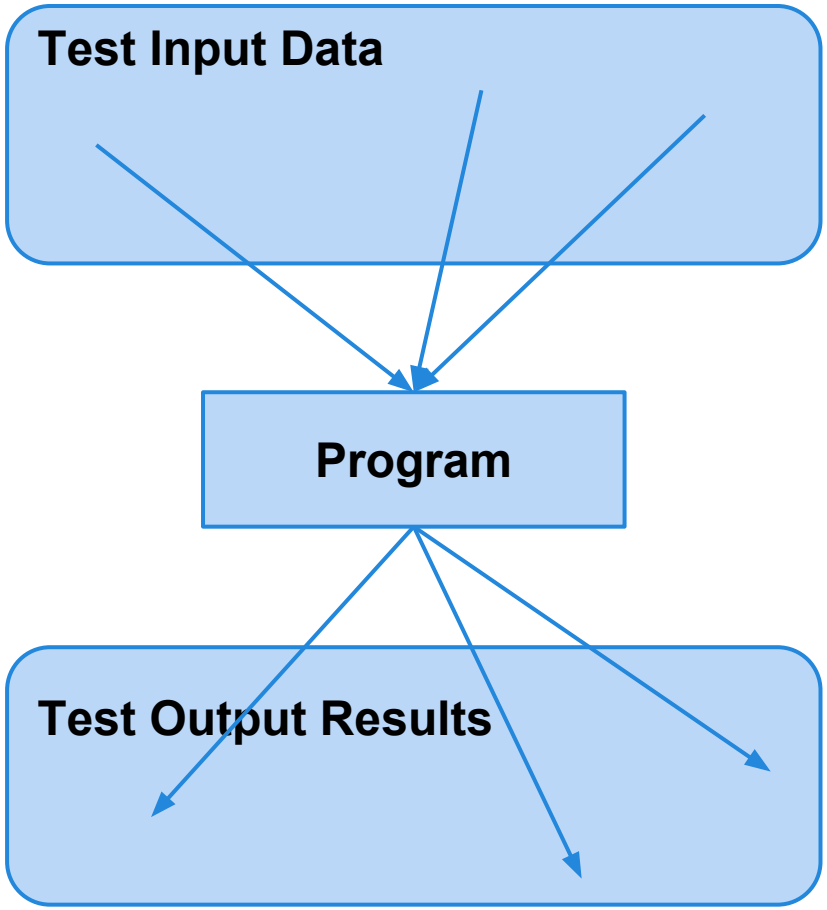
Independently Testable Features

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?

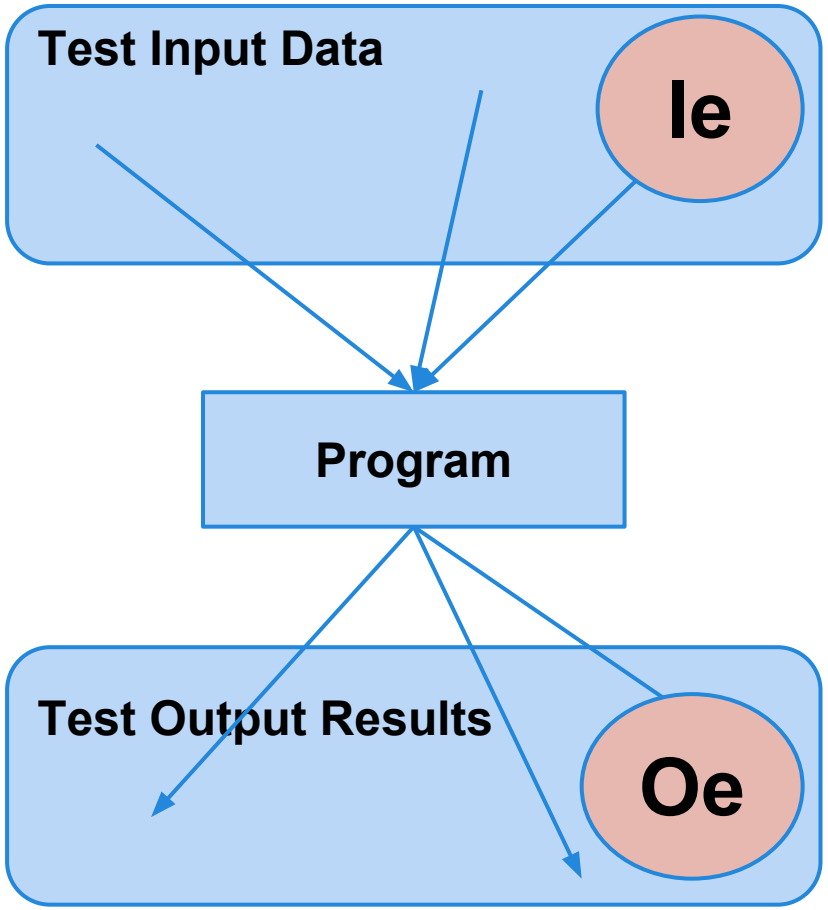
2^{32} possible integer values for each parameter.
$= 2^{32} \times 2^{32} = 2^{64}$ combinations
$= 10^{13}$ tests.

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

or… about 600 years!

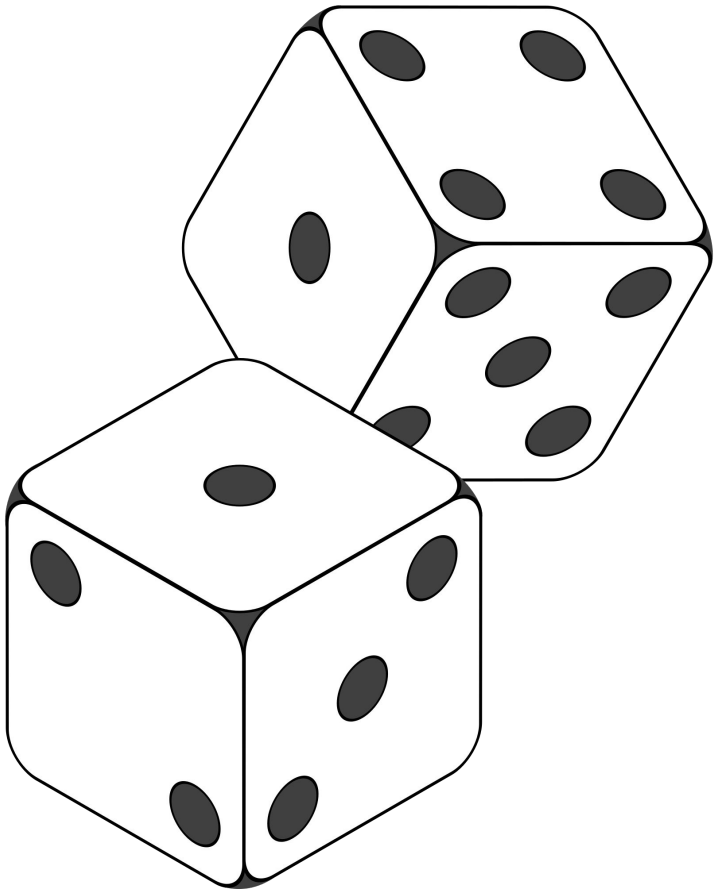
Not all Inputs are Created Equal

- 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.



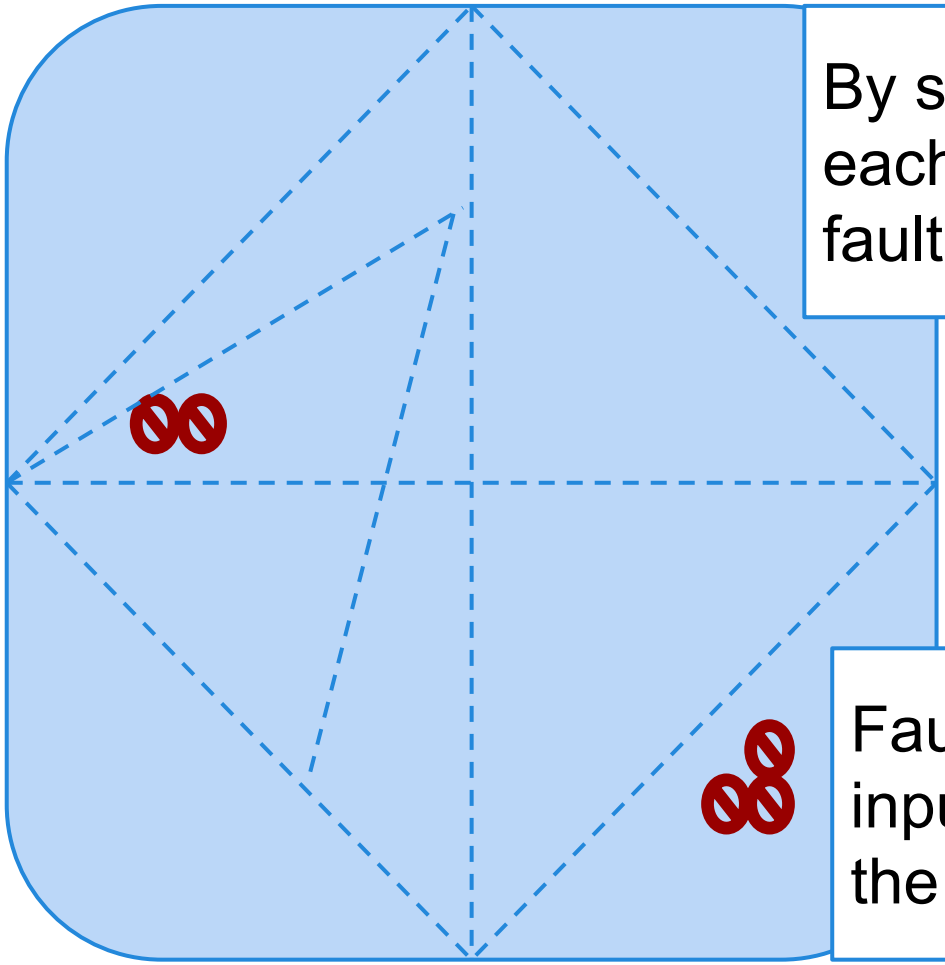
Random Testing

- 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



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

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

Equivalence Class

- We want to divide the input domain into *equivalence classes*.
- A group of tests form an equivalence class if:
  - They test the same thing (outcome, input type, 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 into equivalence classes 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 ranges of numbers or values.
- Look for membership in a group.
- Look for time-dependent equivalence classes.
- Look at the data structures involved.
- Look for equivalent output events.
- Look for equivalent operating environments.
- Look at invalid inputs and boundary conditions.

Look for Ranges of Numbers

- 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.

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.

Look for Equivalent Outputs

- It is often easier to find good tests by looking at the outputs (working backwards).
  - Instead of “I want input from these partitions”...
    - “I want input that results in output from those partitions.”
- Example: A graphics routine that draws lines on a canvas. Output partitions include:
  - No line
  - Thin, short line
  - Thin, long line
  - Thick, short line
  - … etc.

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)

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)
```

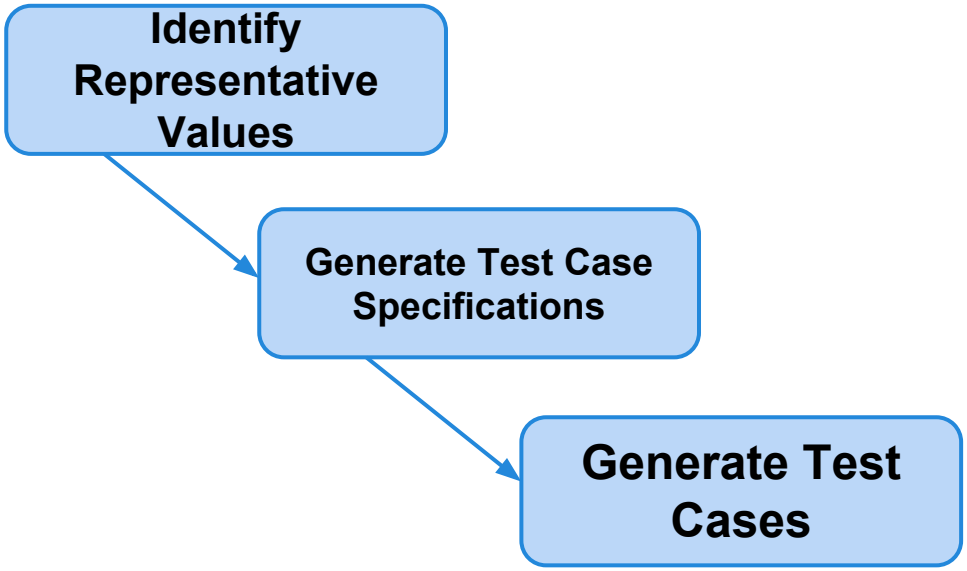
We could consider `a` or `b` in isolation:

```
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

Partition system inputs (or outputs) into equivalence classes.

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

   <10000, 10000-99999, >100000

2. If an input for an insertion function is a list of length 1-10, you want 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

Identify Constraints Among Choices

- Test specifications are formed by combining partitions for all inputs of a feature.
- Number of possible combinations may be impractically large, so:
  - Eliminate impossible pairings.
  - Identify constraints that can remove unnecessary options.
  - From the remainder, choose a practical subset.
  - (called “category partition testing”)

Identify Constraints Among Choices

Three types of constraint:

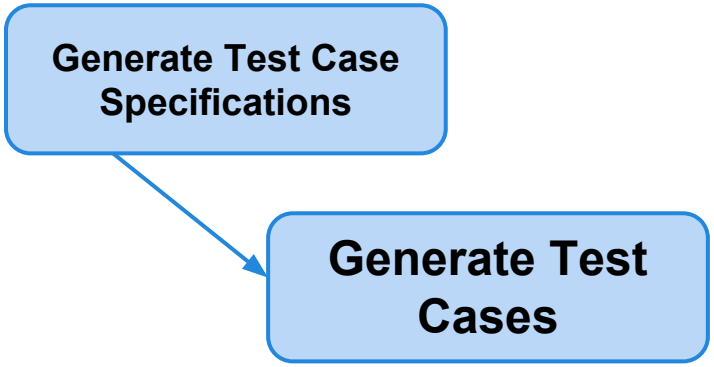
- IF
  - This partition only needs to be considered if another property is true.
- ERROR
  - This partition should cause a problem no matter what value the other input variables have.
- SINGLE
  - Only a single test with this partition is needed.

Constraint Example - Split

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

Input str (combine a length and contents choice)		Input index	
length 0	PROPERTY zeroLen	value <0	ERROR
length 1		value = 0	
length >= 2		value = 1	
contains special characters	if !zeroLen	value > 1	
contains lower case only	if !zeroLen	value = MAXINT	SINGLE
contains mixed case	if !zeroLen		

Generate Test Cases




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

Specification:

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

`index`: value > 0

Test Case:

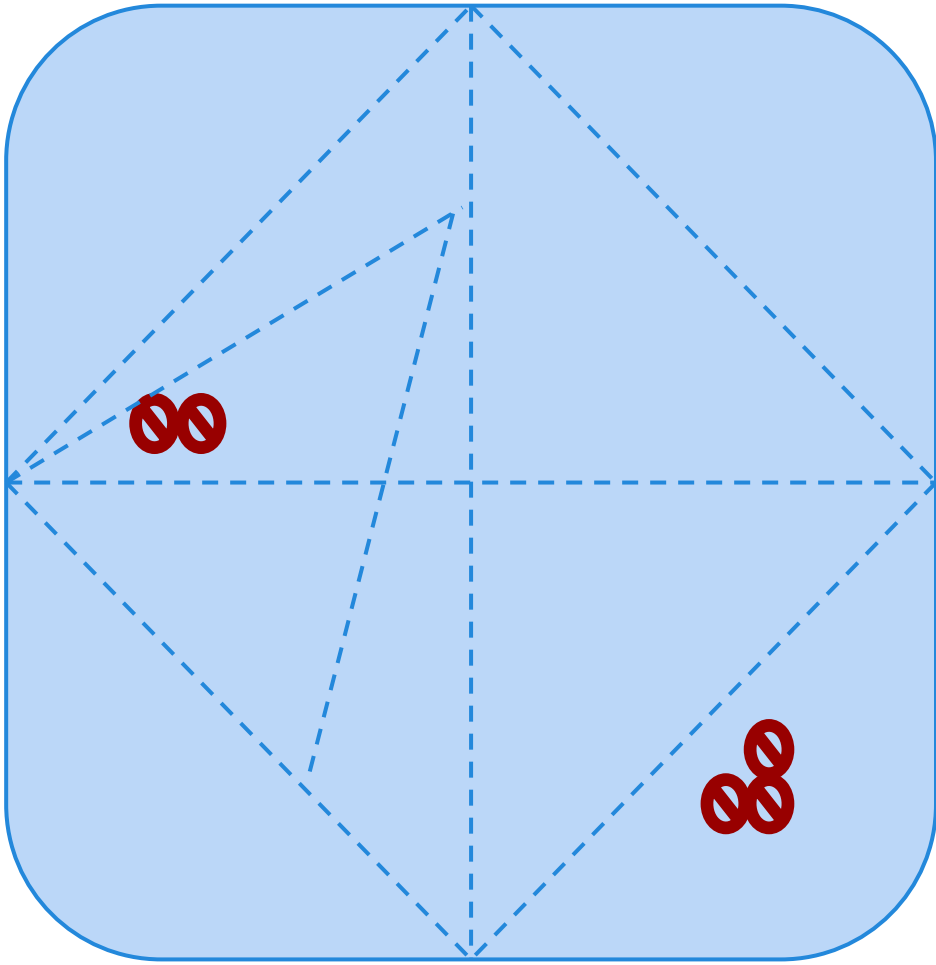
`str` = “ABCC!\n\t7”

`index`= 5

Boundary Values

Basic Idea:

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

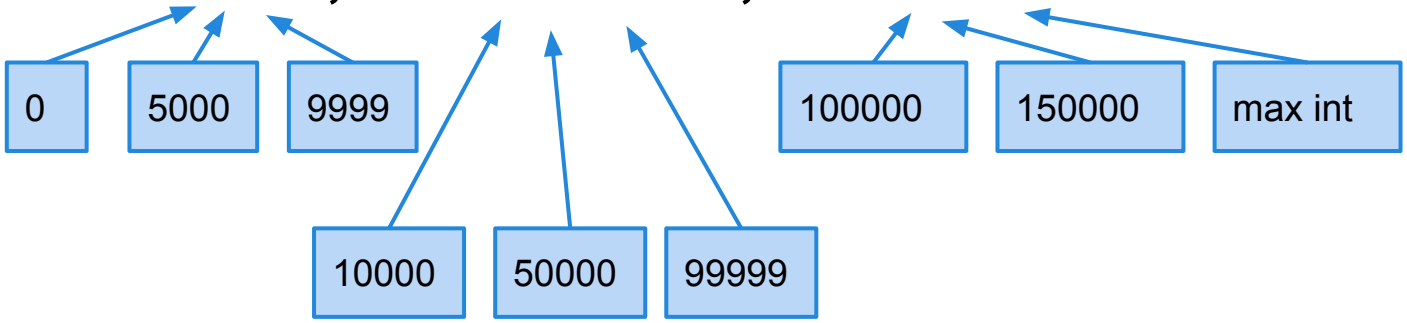


Choosing Test Case Values

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



Cost of Test Scaffolding

- Each test case specification must be converted into one or more concrete tests, and checked for correctness.
- This requires writing code to execute the test and system.
  - If generic code can be written once and used for all tests, then a combinatorial approach may be affordable.
  - If all test code must be written by hand, then care must be invested in choosing a subset of test specifications that is realistic and powerful.

Activity: GRADS Partitioning

Consider the GRADS system you are designing for your homework.

1. **What are the independently testable features of GRADS?**
2. **Choose one - how would you partition the input domain? Define the inputs and outputs for at least one of the independently testable features and identify partitions for each input.**

Activity: GRADS Partitioning

1. **How would you partition the GRADS functionality? What are the independently testable features?**

- Generate Progress Summary
- Generate Hypothetical Progress Summary after future courses
- Log in
- Generate list of students
- Add note

Activity: GRADS Partitioning

2. **Choose one - how would you partition the input domain? Define the inputs and outputs for at least one of the independently testable features and identify partitions for each input.**

Generate Progress Summary
Inputs: ID of current user, ID of requested student record, collection of student records (database).
How would we partition these?

We Have Learned

- Requirements-based tests are derived by
  - identifying independently testable features
  - partitioning their input/output to identify equivalence partitions
  - combining inputs into test specifications
    - and removing impossible combinations
  - then choosing concrete test values for each specification

Next Time

- Arguing for the correctness of our specifications.
  - The World and Machine Model
- Reading:
  - Paper: “Will it Work?”
    - Available on Moodle
- Homework
  - Due Wednesday!
  - Any questions?