CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Lecture 7: Functional Testing

Gregory Gay
DIT635 - February 12, 2020

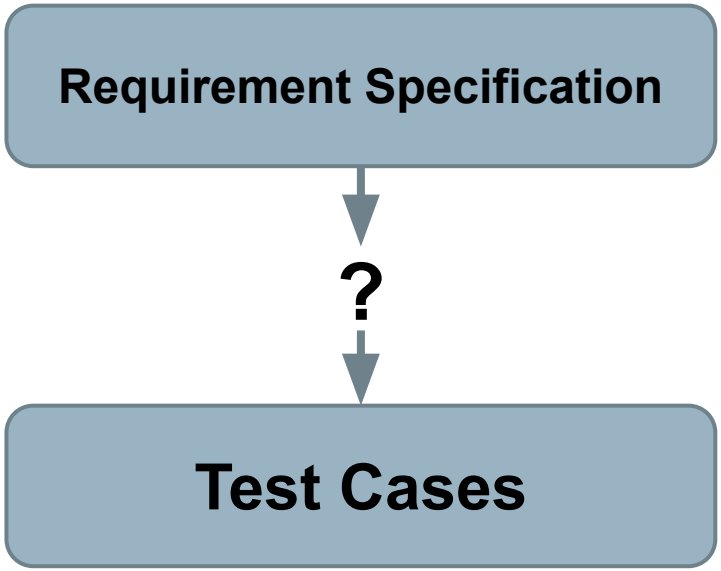
Where Do Tests Come From?

- Many are derived from requirement specifications.
  - The specification defines “correct” behavior.
  - The specification exists in some form before code is written, and guides development.
  - Test plans and cases can be developed and refined as the code is built.
- **Functional Testing:** The process of deriving tests from the requirement specifications.

Functional Testing

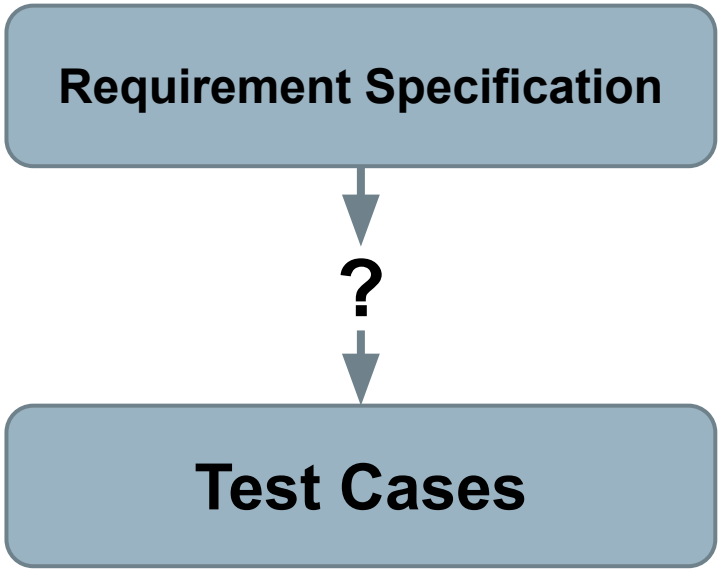
- Deriving tests from the requirement specifications.
  - Typically the baseline technique for designing test cases. Can begin as part of requirements specification, and continue through each level of design and implementation.
  - Basis of verification - builds evidence that the implementation conforms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

Partitioning



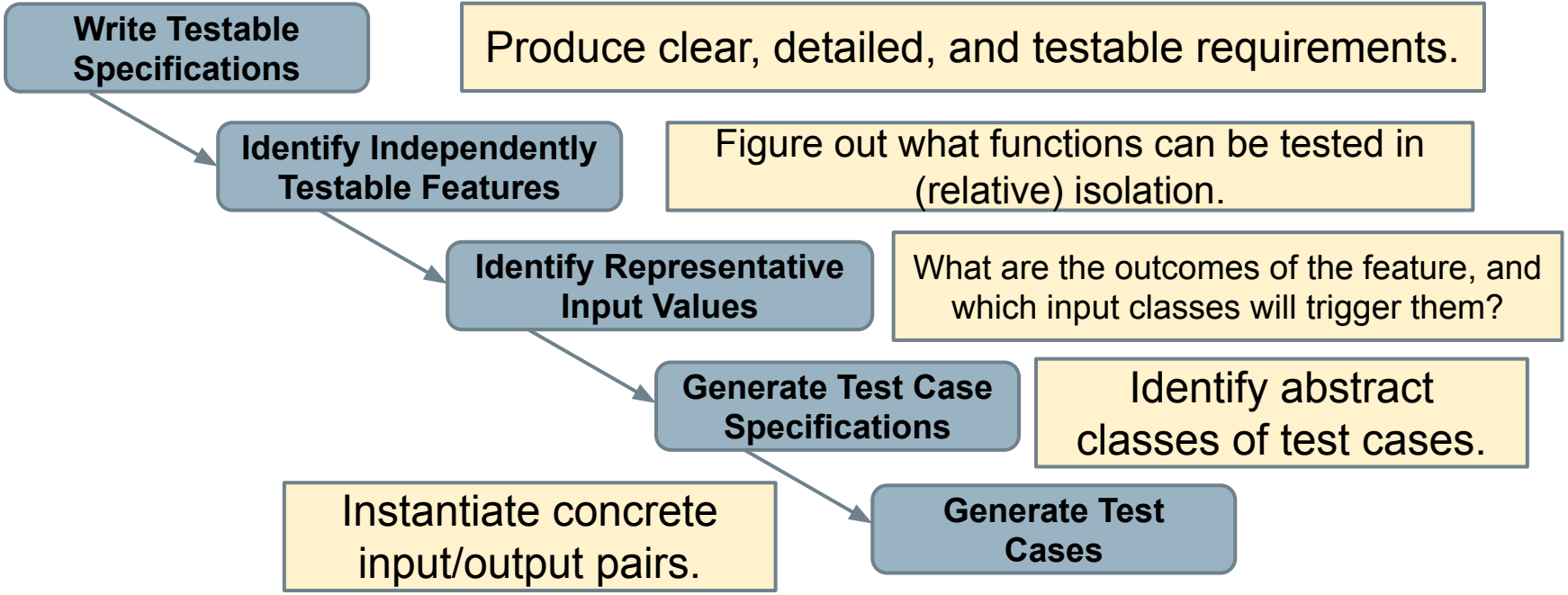
- Functional testing is based on the idea of partitioning.
  - You can't test individual requirements in isolation.
  - Instead, we need to partition the specification and software into features that can be tested.

Partitioning



- Not all inputs have the same effect.
- 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 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 “units” of code.
  - Usually a class or method.
  - Until we have a design, we do not have units.
- An independently testable feature is a *capability* of the software.
  - May not correspond to unit(s).
  - Can be at the class, subsystem, or system level.

Features and Parameters

Tests for features must be described in terms of parameters and environmental factors that influence its 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.
    - 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 string, database contains `UserRecord`s.
- … 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.

Parameter Context

- An input for a feature might be split into multiple “variables” based on contextual use.
  - The database may or may not contain a record for that user.
    - In either case, issues may emerge based on the size of the database.
    - The program may also have issues if a database connection cannot be established.
  - This is three “parameters” for a feature.

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?

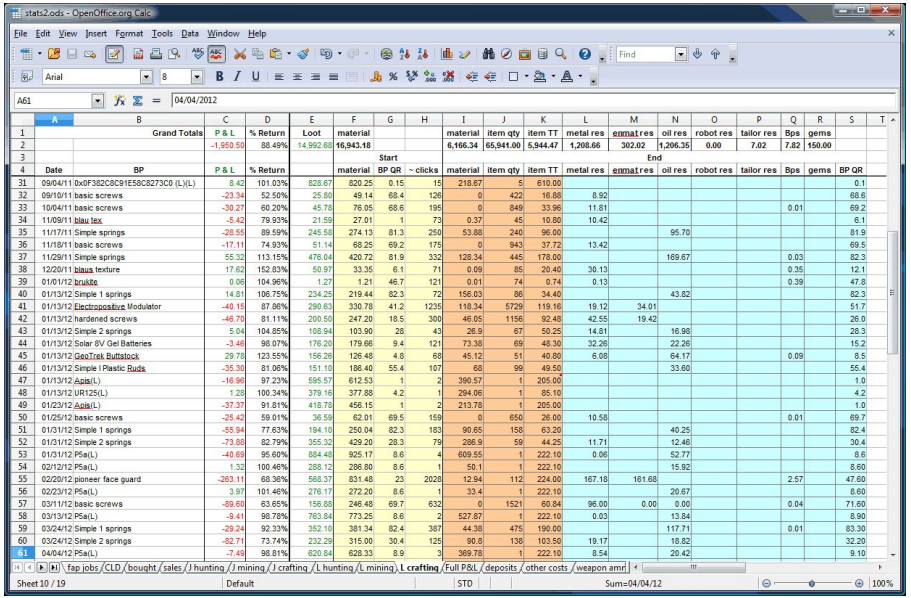
- Course number to add
- Grading basis
- Student record
- What about a course database? Student record database?

Examples

- Student Record
- Context - how is it used?
  - Have you already taken the course?
  - Are there holds on your record?
  - Do you meet the prerequisites?
  - …
  - Each of these can be varied when testing.

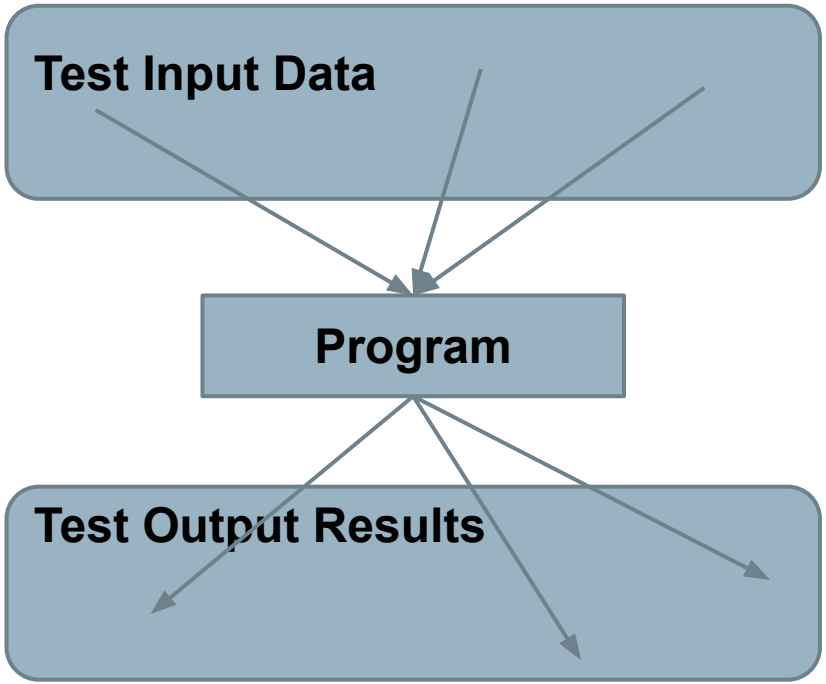
Independently Testable Features

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?

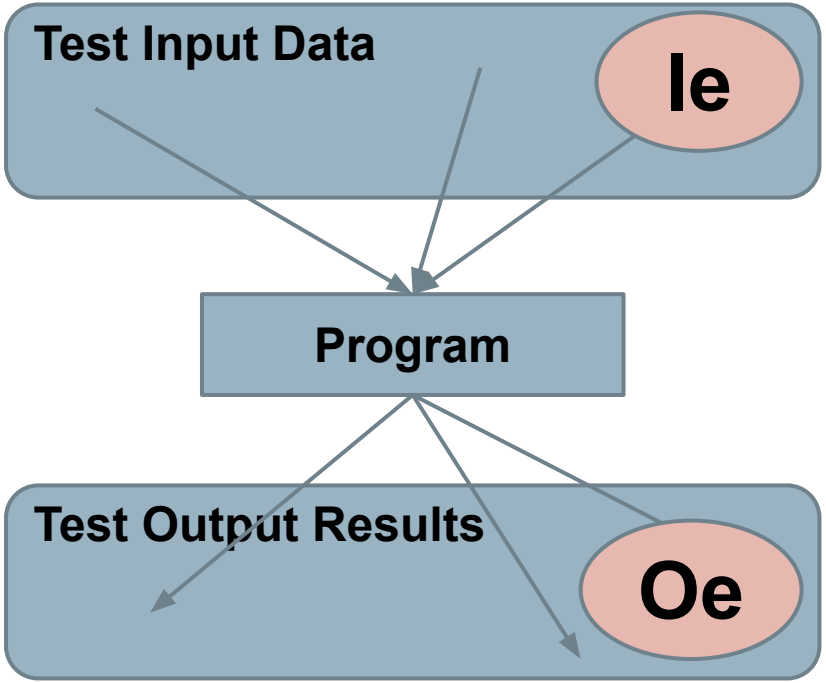
2^{32} possible integer values for each parameter.
= 2^{32} x 2^{32} = 2^{64} combinations = 10^{13} tests.

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

or… about 600 years!

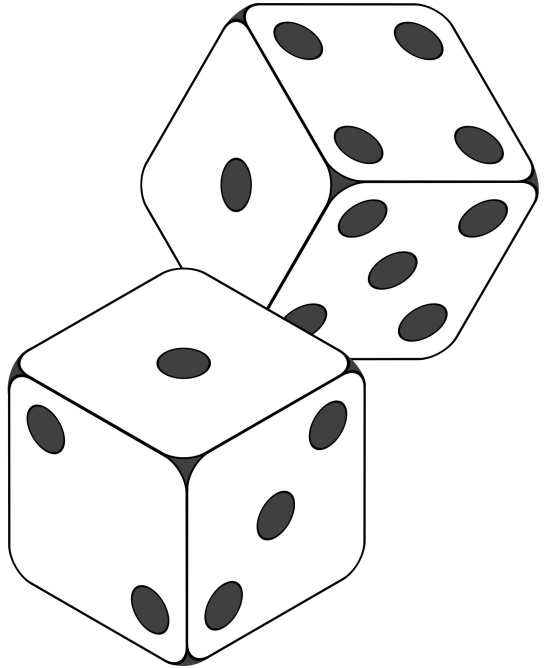
Not all Inputs are Created Equal

- 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.



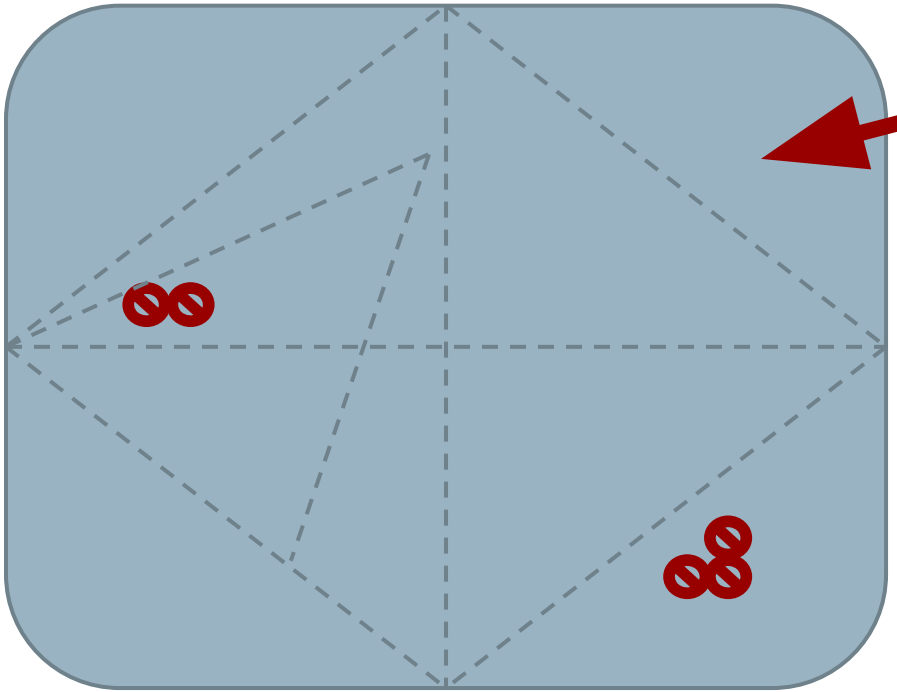
Random Testing

- Pick inputs uniformly from the distribution of all inputs.
- All inputs considered equal.
- Keep trying until 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 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 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: getEmployeeStatus(employee ID)
  - Manager, Developer, Marketer, Lawyer
  - Employee Does Not Exist
  - Malformed Employee ID

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 floor layout
- A country name.

- All can be partitioned into groups.
  - Numeric vs Other data types, Apartment vs Business, Europe vs Asia, etc.
- 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

- Environment may affect behavior of the program.
- Environmental factors can be partitioned.
  - 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 Structures

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)
```

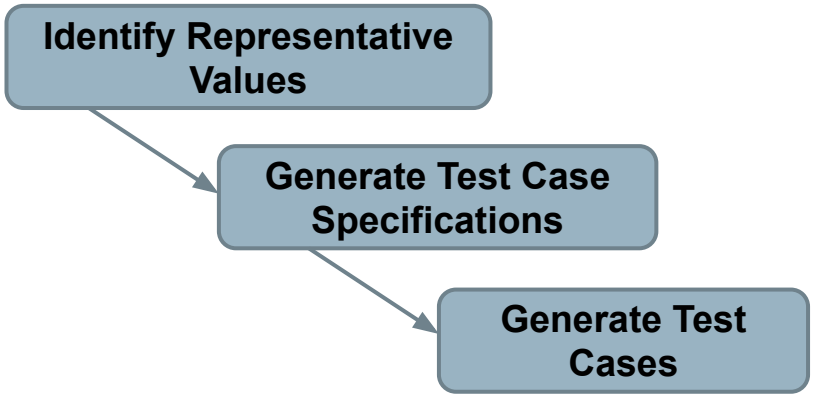
We could consider `a` or `b` in isolation:

```
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]

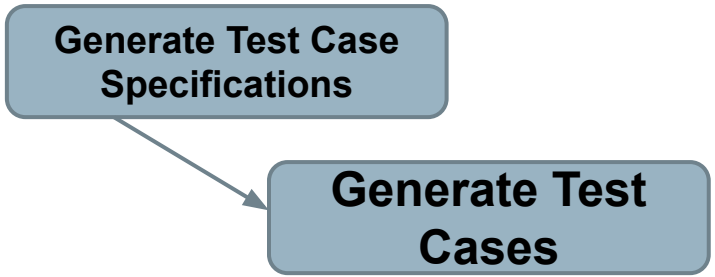
Test Specifications:

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)
```

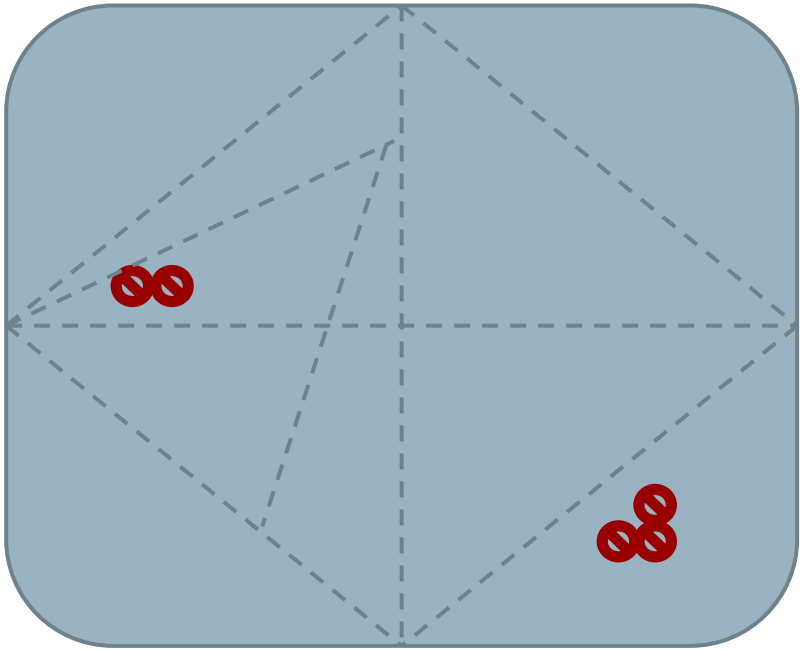
Specification:
`str`: length >=2, contains special characters
`index`: value > 0

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.

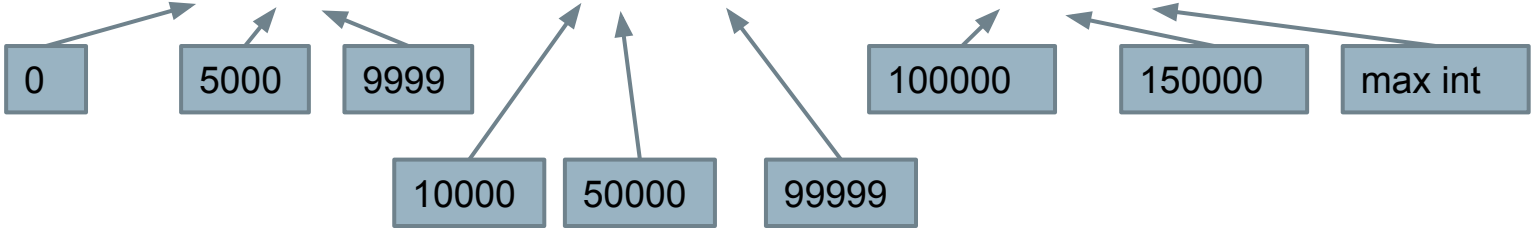


Choosing Test Case Values

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



Let’s take a break.

Activity - Functional Testing

You are asked to develop a simple C++ container class SetOfE containing elements of type E with the following methods:

- `void insert(E e)`
- `Bool find(E e)`
- `void delete(E e)`

Using domain partitioning, develop functional test cases for the methods. You can define your test cases as input/output pairs.

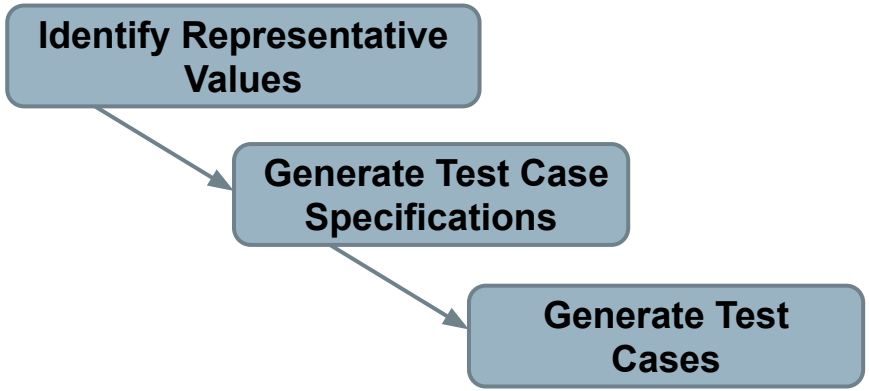
For example, to test `insert(E e)`, one test case could be:
Input: Empty Container/any e **Expected output:** e in Container.

Activity Solution

Insert	*Empty/ any e*	*e in container*
	E with one element / any e	*e in container*
	E with multiple elements / any e	*e in container*
	Very large E/ any e	*e in container*
	E containing e/ e	*Error or no change*
	Any E/ malformed e	*Error*
Exists	*E containing e/ e*	*True*
	E not containing e/ e	*False*
	Very large E containing e/ e	*True*
	E with only element e/ e	*True*
	Any E / malformed e	*Error*
	Empty / e	*False*

Delete	*E containing e/ e*	*e no longer in list*
	E not containing e/ e	*no change (or error)*
	Any E / malformed e	*error*
	Very large E containing e/ e	*e no longer in list*
	Empty / e	*no change*

Building a Test Suite



Smarter process than random testing, but still comes down to brute force:

- May still be an infeasibly high number of test specifications.
- Each specification can be transformed into MANY concrete test cases. How many should be tried?

How do we arrive at an effective, reasonably-sized test suite?

Category-Partition Method

Category-Partition Method

A method of generating test specifications from requirement specifications.

- Adds a small number of additional steps on the process discussed today.
- Requires identifying *categories*, *choices*, and *constraints*.
- Once identified, these can be used to automatically generate a list of test specifications to cover.

Identify Independently Testable Features and Parameter Characteristics

- Identify features and their parameters.
- Identify **characteristics** of each parameter.
  - What are the controllable attributes?
  - What are their possible values?
    - May be defined partially by other parameters and their characteristics.
    - May not correspond to variables in the code.
- Parameter characteristics are called ***categories***.

Example: Computer Configurations

- Your company sells custom computers.
- A *configuration* is a set of options for a *model*.
  - Some combinations are invalid (i.e., VGA monitor with HDMI video output).
- Testing feature:
  - `checkConfiguration(model,components)`
  - What are the parameters?
  - Next - what are the choices to be made for each parameter?

Parameter Characteristics

- Turn to the requirements specifications.
- **Model:** A model identifies a specific product and determines a set of constraints on available components. Models are identified by a model number. Models are characterized by logical slots on a bug. Slots may be required (must be filled) or optional (may be left empty).
- **Set of Components:** A set of <slot, component> pairs, which must correspond to the required and optional slots associated with the model. A component is a choice that can be varied within a model. Available components and a default for each slot is determined by the model. The special value “empty” is allowed and may be the default for optional slots. In addition to being compatible or incompatible with a model, components may be compatible or incompatible with each other.

Categories

- **Model**
  - Model number
  - Number of required slots (must have a selection)
  - Number of optional slots (may or may not have a selection)
- **Components**
  - Selected component valid for model
  - Number of required/optional slots with non-empty selections
  - Selected components for required/optional slots OK/not OK
- **Product Database**
  - Number of models in database
  - Number of components in database

Identify Representative Values

- For each category, many values that can be selected for concrete test cases.
- We need to identify *classes of values*, called ***choices***, for each category.
  - A test specification is a combination of choices for all categories.
- Consider all outcomes of a feature.
- Consider boundary values.

Choices for Each Category

Model

- Model number
  - malformed
  - not in database
  - valid
- Number of required slots
  - 0
  - 1
  - many
- Number of optional slots
  - 0
  - 1
  - many

Product Database

- Number of models in database
  - 0
  - 1
  - many
- Number of components in database
  - 0
  - 1
  - many

Components

- Correspondence of selection with model slots
  - omitted slots
  - extra slots
  - mismatched slots
  - complete correspondence
- Number of required (or optional) components with non-empty selections
  - 0
  - < number required (or optional)
  - = number required (or optional)
- Selected components for required (or optional) slots
  - some default
  - all valid
  - >= 1 incompatible with slot
  - >= 1 incompatible with another component
  - >= 1 not in database

Generate Test Case Specifications

- Test specifications are formed by combining choices for all categories.
- Number of possible combinations may be impractically large, so:
  - Eliminate impossible pairings.
  - Identify constraints to remove unnecessary options.
  - From the remainder, choose a subset of specifications to turn into concrete tests.

Choices for Each Category

Model

- Model nu
  - ma
  - not
  - vali
- Number
  - 0
  - 1
  - ma
- Number
  - 0
  - 1
  - ma

Components

ion with model

ndence
tional)
oty selections

(or optional)
(or optional)
required (or

with slot
- >= 1 incompatible with another component
- >= 1 not in database

- Seven categories with three choices.
- Two categories with 6 choices.
- One category with 4 choices.
- Results in $3^7 \times 6^2 \times 4 = 314928$ test specifications
- However… not all combinations correspond to reasonable specifications.

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.

Applying Constraints

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

Str length

length 0 — PROPERTY zeroLen

length 1

length >= 2

Str contents

contains special characters — if !zeroLen

contains lower case only — if !zeroLen

contains mixed case — if !zeroLen

empty — if zeroLen

Input index

value <0 — ERROR

value = 0

value = 1

value > 1

value = MAXINT — SINGLE

Applying Constraints

Model

- Model number
  - malformed **[error]**
  - not in database **[error]**
  - valid
- Number of required slots
  - 0 **[single]**
  - 1 **[property RS]** **[single]**
  - many **[property RS]** **[property RSMANY]**
- Number of optional slots
  - 0 **[single]**
  - 1 **[single]** **[property OS]**
  - many **[property OS]** **[property OSMANY]**

Product Database

- Number of models in database
  - 0 **[error]**
  - 1 **[single]**
  - many
- Number of components in database
  - 0 **[error]**
  - 1 **[single]**
  - many

Components

- Correspondence of selection with model slots
  - omitted slots **[error]**
  - extra slots **[error]**
  - mismatched slots **[error]**
  - complete correspondence
- Number of required (or optional) components with non-empty selections
  - 0 **[error]** **[if RS]**
  - < number required (or optional) **[error]** **[if RS]** / **[if OS]**
  - = number required (or optional) **[if RSMANY]** / **[if OSMANY]**
- Selected components for required (or optional) slots
  - some default **[single]**
  - all valid
  - >= 1 incompatible with slot
  - >= 1 incompatible with another component
  - >= 1 not in database **[error]**

Example - Find Command

Bash command: find

`find <pattern> <file>`

- Finds instances of a pattern in a file
  - `find john myFile`
    - Finds all instances of `john` in the file
  - `find “john smith” myFile`
    - Finds all instances of `john smith` in the file
  - `find ““john” smith” myFile`
    - Finds all instances of `john” smith` in the file

Example - Find Command

- Parameters: pattern, file
- What can we vary for each?
  - Our categories.
  - What can we control about the pattern? Or the file?
- What choices can we make for each category?
  - Our categories
  - **File name:**
    - Name of an existing file provided
    - File does not exist

Example - Find Command

1944 tests if we consider all combinations.

- Pattern size:
  - Empty
  - single character
  - many character
  - longer than any line in the file
- Quoting:
  - pattern is quoted
  - not quoated
  - improperly quoated
- Embedded spaces:
  - No spaces
  - One space
  - Several spaces

- Embedded quotes:
  - no quotes
  - one quote
  - several quotes
- File name:
  - Existing file name
  - no file with this name
- Number of occurrence of pattern in file:
  - None
  - exactly one
  - more than one
- Pattern occurrences on target line:
  - One
  - more than one

IF Constraints

678 Tests

- Pattern size:
  - Empty
  - single character **[not empty]**
  - many character **[not empty]**
  - longer than any line in the file **[not empty]**
- Quoting:
  - pattern is quoted **[quoted][if not empty]**
  - not quoted **[if not empty]**
  - improperly quoted **[if not empty]**
- Embedded spaces:
  - No spaces
  - One space **[if not empty and quoted]**
  - Several spaces **[if not empty and quoted]**
- Embedded quotes:
  - no quotes
  - one quote **[if not empty]**
  - several quotes **[if not empty and quoted]**
- File name:
  - Existing file name
  - no file with this name
- Number of occurrence of pattern in file:
  - None **[if not empty]**
  - exactly one **[match] [if not empty]**
  - more than one **[match] [if not empty]**
- Pattern occurrences on target line:
  - One **[if match]**
  - more than one **[if match]**

ERROR and SINGLE Constraints

- Pattern size:
  - Empty
  - single character
  - many character
  - longer than any line in the file **[error]**
- Quoting:
  - pattern is quoted
  - not quoted
  - improperly quoted **[error]**
- Embedded spaces:
  - No spaces
  - One space
  - Several spaces

40 Tests!

- Embedded quotes:
  - no quotes
  - one quote
  - several quotes **[single]**
- File name:
  - Existing file name
  - no file with this name **[error]**
- Number of occurrence of pattern in file:
  - None **[single]**
  - exactly one
  - more than one
- Pattern occurrences on target line:
  - One
  - more than one **[single]**

67

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

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.
- IF/ERROR/SINGLE constraints can remove unnecessary combinations of input.

Next Time

- Structural Testing
  - Optional Reading: Pezze and Young, Chapters 5.3 and 12
- Homework
  - Assignment 1 Due Sunday, February 16
  - Any questions?

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY