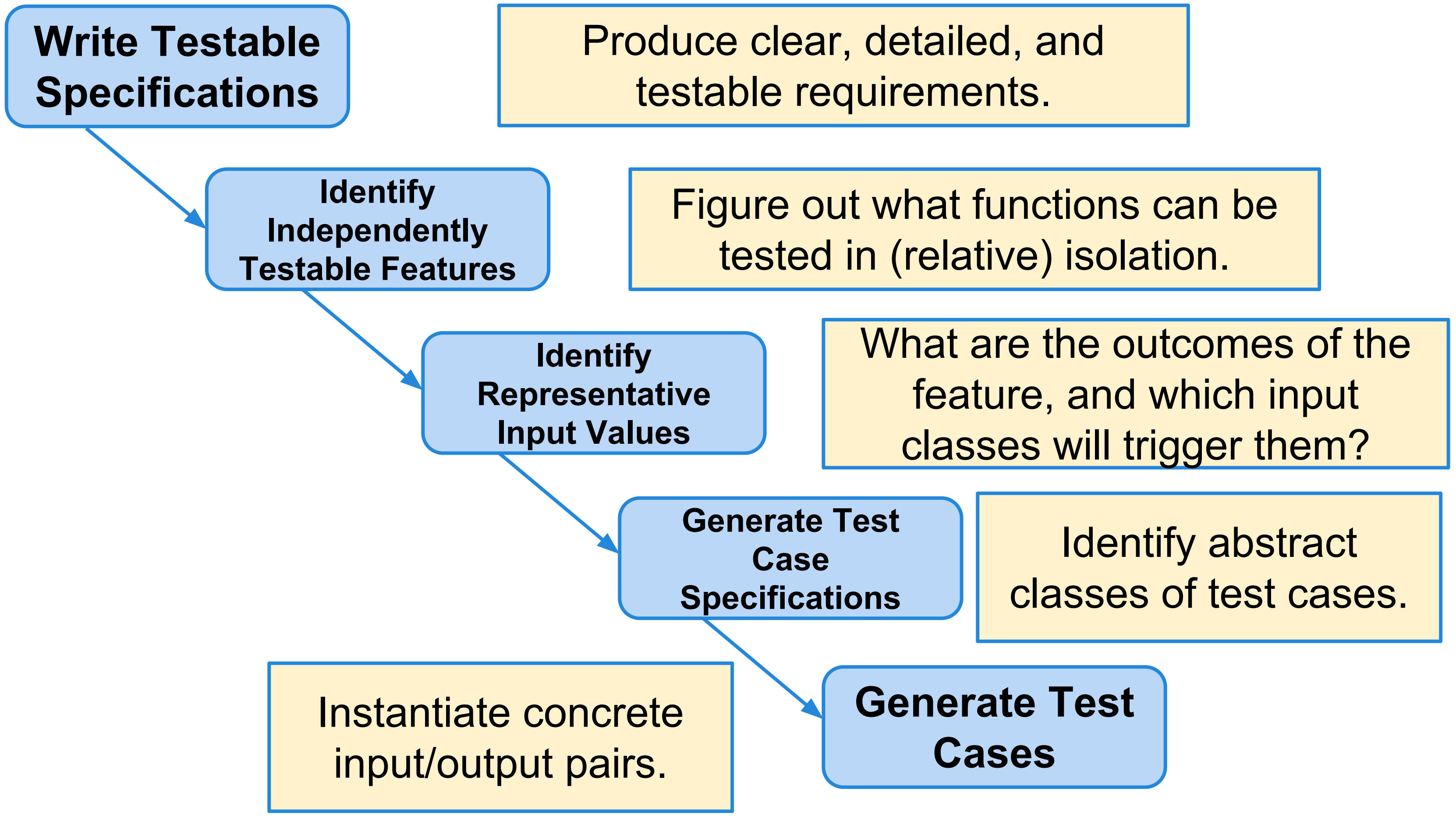
Combinatorial Testing

CSCE 747 - Lecture 5 - 01/24/2017

Creating Requirements-Based Tests


Write Testable Specifications
Identify Independently Testable Features
Identify Representative Input Values
Generate Test Case Specifications
Generate Test Cases

Produce clear, detailed, and testable requirements.

Figure out what functions can be tested in (relative) isolation.

What are the outcomes of the feature, and which input classes will trigger them?

Identify abstract classes of test cases.

Instantiate concrete input/output pairs.

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.

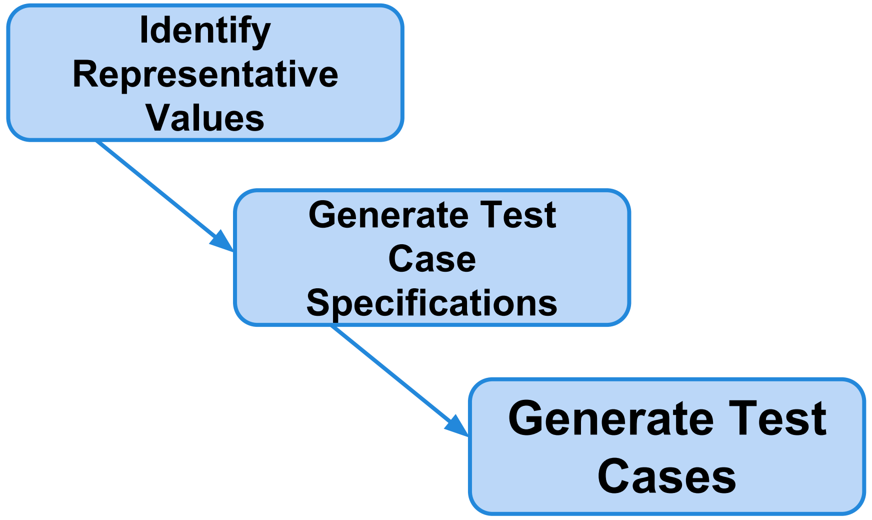
For example, to test `insert(E e)`, one test case could be:
Input: Empty Container/any e **Expected output:** e in Container.

Question 6 (2) - 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


Identify Representative Values
Generate Test Case Specifications
Generate Test Cases

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?

Today's Goals:

- Category-Partition Method
    - Assists in identifying test specifications, estimating the number of tests, and forming a subset that meets your budget
- Combinatorial Interaction Testing
    - Method of covering n-way combinations of parameter values with a small number of tests.
- Catalog-Based Testing
    - Makes identifying attributes and representative values more systematic and enables some automation.

Category-Partition Method

Category-Partition Method

A method of generating test specifications from requirement specifications.

- A small number of additional steps on the process discussed last class.
- 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.
- The parameter characteristics are called ***categories***.

Example: Computer Configurations

- Your company sells custom computers.
- A *configuration* is a set of options for a *model* of computer.
  - 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
  - Number of optional slots
- **Components**
  - Correspondence of selection with model slots
  - Number of required components with non-empty selections
  - Number of optional components with non-empty selections
  - Selected components for required slots
  - Selected components for optional slots
- **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.

Categories

- **Model**
  - Model number
  - Number of required slots
  - Number of optional slots
- **Components**
  - Correspondence of selection with model slots
  - Number of required components with non-empty selections
  - Number of optional components with non-empty selections
  - Selected components for required slots
  - Selected components for optional slots
- **Product Database**
  - Number of models in database
  - Number of components in database

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(optional) components with non-empty selections
    - 0
    - < number required (optional)
    - = number required (optional)
  - Selected components for required (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 that can remove unnecessary options.
  - From the remainder, choose a subset of specifications to turn into concrete tests.

Choices for Each Category

- **Model**
  - Model number
    - malformed
    - not in d
    - valid
  - Number of re
    - 0
    - 1
    - many
  - Number of op
    - 0
    - 1
    - many
- **Product Database**
  - Number of m
    - 0
    - 1
    - many
  - Number of co
    - 0
    - 1
    - many

- **Components**
  - Correspondence of selection with model slots
  - s
  - pondence
  - tional)
  - empty
  - ed (optional)
  - ed (optional)
  - for required
  - e with slot
  - e 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:

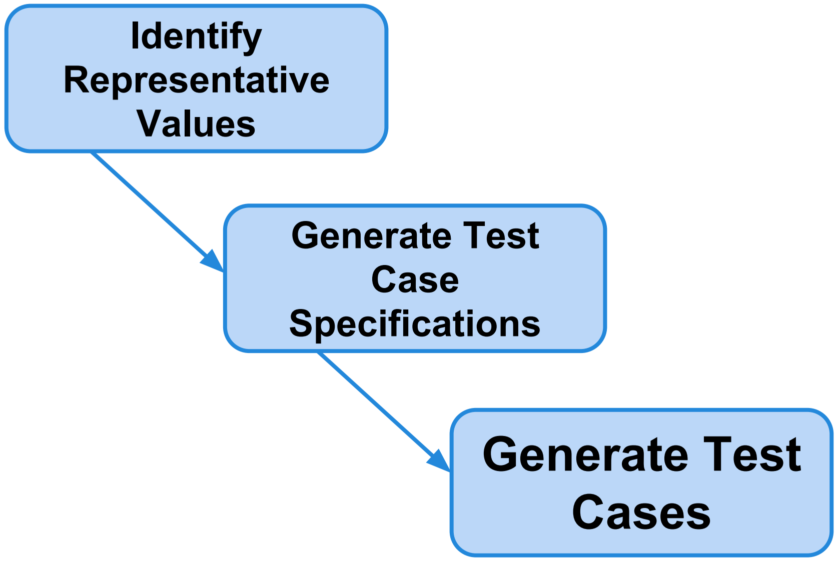
- IF
  - This partition only needs to be considered if another property is true.
- ERROR
  - This partition should cause a problem no matter what value the other input variables have.
- SINGLE
  - Only a single test with this partition is needed.

Applying Constraints

- **Model**
  - Model number
    - malformed **[error]**
    - not in database **[error]**
    - valid
  - Number of required slots
    - 0 **[single]**
    - 1 **[property RSNE] [single]**
    - many **[property RSNE], [property RSMANY]**
  - Number of optional slots
    - 0 **[single]**
    - 1 **[property OSNE][single]**
    - many **[property OSNE], [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 components with non-empty selections
    - 0 **[if RSNE] [error]**
    - < number required **[if RSNE] [error]**
    - = number required **[if RSMANY]**
  - Number of optional components with non-empty selections
    - 0
    - < number optional **[if OSNE]**
    - = number optional **[if OSMANY]**
  - Selected components for required (optional) slots
    - some default **[single]**
    - all valid
    - >= 1 incompatible with slot
    - >= 1 incompatible with another component
    - >= 1 not in database **[error]**

Combinatorial Interaction Testing

Dealing With All of These Test Specifications


Identify Representative Values
Generate Test Case Specifications
Generate Test Cases

- Category-partition testing takes exhaustive enumeration as a base approach and adds constraints to reduce the number of tests.
- This is only reasonable when constraints reflect real conditions.
- If constraints are added solely to reduce the number of combinations, then you will produce bad tests.

Website Display Options

- Display Mode
  - full-graphics
  - text-only
  - limited-bandwidth
- Color
  - monochrome
  - color-map
  - 16-bit
  - true-color
- Language
  - English
  - French
  - Spanish
  - Portuguese
- Screen Size
  - Handheld
  - Laptop
  - Full-size
- Fonts
  - Minimal
  - Standard
  - Document-loaded

Combinatorial Interaction Testing

- Some parameter combinations may cause faults, so we can't just try each choice once.
- But we do not need all combinations either - many will be redundant.
- Instead, pick a number *k* < *n* (n = number of parameters), and generate all *k*-way combinations.
    - Exhaustive enumeration grows exponentially with the number of parameters.
    - Pairwise combinations grow logarithmically.

Combinatorial Interaction Testing

- Choose two parameters, enumerate all combinations.
- Adding the third is multiplicative.
- Instead, consider all n-way combinations of values
    - (2-way in this case)
- Each tuple contains three pairings. Careful selection of those pairings covers more combinations.

Display-Mode	**Screen Size**	**Fonts**
Full-graphics	Handheld	Minimal
Full-graphics	Laptop	Standard
Full-graphics	Fullsize	Document-Loaded
Text-Only	Handheld	Standard
Text-Only	Laptop	Document-Loaded
Text-Only	Fullsize	Minimal
Limited-Bandwidth	Handheld	Document-Loaded
Limited-Bandwidth	Laptop	Minimal
Limited-Bandwidth	Fullsize	Standard

Covering Arrays

- In functional testing, we want to *cover* a large number of the parameter combinations.
  - We seek coverage of strength *k*.
    - *k=n* means we have covered all combinations.
    - *k < n* means all *k-way* combinations are covered.
- A covering array of strength *k* is the smallest array that covers all *k-way* combinations.
- Selecting the smallest array is NP-hard.
  - However, greedy and heuristic searches can produce near-optimal solutions.

Constraining the Combinations

- Some combinations may not be possible in practice. Constraints can be used to remove invalid combinations.
    - Monochrome is only an option for handheld displays.
    - So, we remove any pairing of monochrome with laptop or full-size displays.

Catalog-Based Testing

Learning from Experience

- Generating functional tests requires human judgement.
  - Identifying features and parameters is fairly straightforward.
  - Selecting representative value requires creativity.
    - How do you best partition the input space?
    - What boundary values need covered?
    - How do I hit all outcomes of a function?
- Lessons learned from testing one system can improve testing of new systems.

Catalog-Based Testing

- Catalogs encode checklists of input classes for particular types of variables.
  - Example: A computation uses an integer variable that is supposed to fall in a range. The catalog recommends the following partitions:
    - Value immediately preceding the lower bound of the interval.
    - Lower bound of the interval.
    - A value within the interval.
    - The upper bound of the interval.
    - The value immediately following the upper bound.
  - Covers normal, erroneous, and boundary cases.

Catalog-Based Testing

- The catalog-based approach requires:
  - Decomposing the requirement specification into elementary items related to testing that specification.
    - Features, parameters, and conditions on both.
  - Deriving an initial set of test specifications from these elementary items.
  - Completing test specifications using a suitable catalog.

Identify Elementary Items

- We need information about what we are testing.
- From the requirement specification, identify:
  - Preconditions - conditions that must be satisfied before test execution.
  - Postconditions - the result of executing this feature.
  - Variables - input, output, and intermediate values that the system operates on.
  - Operations - calculations performed using the variables.
  - Definitions - other facts offered by the specification.

Example: cgi_decode

cgi_decode(e
the er
alpha
“%xy”
other alphanu

PRE 1.

se

INPUT: encoded
characters, ‘+’, an

OUTPUT: decode
the input sequenc
corresponding pos
character with hex

OUTPUT: return_

OP 1. Scan the input string Encoded.

POST 2. If the input string Encoded contains ‘+’ characters, they are replaced by ASCII space characters in the corresponding positions in the output string.
POST 3. If the input string Encoded contains CGI-hexadecimals, they are replaced by the corresponding ASCII characters.
POST 4. If the input string Encoded is a valid sequence, cgi_decode returns 0.
POST 5. Id the input string Encoded contains a malformed CGI-hexadecimal, i.e., a substring “%xy” where either x or y are absent and not hexadecimal digits, cgi_decode returns 1.
POST 6. If the input string Encoded contains any illegal characters, cgi_decode returns 1.

Elementary Items

- **DEF 1.** Hexidecimal digits are 0-9, A-F, a-f.
- **DEF 2.** a CGI hexidecimal is a sequence of three characters “%xy” where x and y are hexadecimal digits.
- **DEF 3.** a CGI item is either an alphanumeric character, ‘+’, or a CGI hexadecimal.
- **VAR 1.** Encoded: string of CGI characters.
- **VAR 2.** Decoded: string of ASCII characters.
- **VAR 3.** return value: boolean.
- **PRE 1.** (assumed) the input string Encoded is a null-terminated string of characters.
- **PRE 2.** (validated) the input string Encoded is a sequence of CGI items.
- **POST 1.** If the input string Encoded contains alphanumeric characters, they are copied to the corresponding position in the output string.
- **POST 2.** If the input string Encoded contains ‘+’ characters, they are replaced by ASCII space characters in the corresponding positions in the output string.
- **POST 3.** If the input string Encoded contains CGI-hexadecimals, they are replaced by the corresponding ASCII characters.
- **POST 4.** If the input string Encoded is a valid sequence, cgi_decode returns 0.
- **POST 5.** Id the input string Encoded contains a malformed CGI-hexadecimal, i.e., a substring “%xy” where either x or y are absent and not hexadecimal digits, cgi_decode returns 1.
- **POST 6.** If the input string Encoded contains any illegal characters, cgi_decode returns 1.
- **OP 1.** Scan the input string Encoded.

Derive Initial Test Specifications

- Now, we want to partition the input domain. We can use the elementary items to do so.
- Validated Preconditions:
  - Simple preconditions (true/false) divide input into two classes.
  - Complex preconditions (involving and/or) can add additional input classes.
- Assumed Preconditions:
  - Not responsible for checking, but make sure that they are checked elsewhere.

Derive Initial Test Specifications

- Postconditions:
  - If the postcondition is given in a conditional form, it is treated as a validated precondition.
- Definitions:
  - Any that refer to variables and are given in conditional form should be evaluated as validated preconditions.
- Scan elementary items and derive test specifications.
  - An incremental process of discovery and refinement.

Deriving Initial Test Specifications

- **DEF 1.** Hexidecimal digits are 0-9, A-F, a-f.
- **DEF 2.** a CGI hexidecimal is a sequence of three characters “%xy” where x and y are hexadecimal digits.
- **DEF 3.** a CGI item is either an alphanumeric character, ‘+’, or a CGI hexadecimal.
- **VAR 1.** Encoded: string of CGI characte
- **VAR 2.** Decoded: string of ASCII
- **VAR 3.** return value: boolean.
- **PRE 1.** (assumed) the input strin
- **PRE 2.** (validated) the input strin
- **POST 1.** If the input string Encod corresponding position in the out
- **POST 2.** If the input string Encod characters in the corresponding
- **POST 3.** If the input string Encod corresponding ASCII characters.
- **POST 4.** If the input string Encod
- **POST 5.** Id the input string Enco “%xy” where either x or y are abs
- **POST 6.** If the input string Encoded contains any illegal characters, cgi_decode returns 1.
- **OP 1.** Scan the input string Encoded.

TC-POST1-1: Encoded contains 1+ alphanumeric characters.
TC-POST1-2: Encoded does not contain any alphanumeric

TC-POST2-1: Encoded contains 1+ ‘+’ characters.
TC-POST2-2: Encoded does not contain any ‘+’ characters.

TC-POST3-1: Encoded contains 1+ CGI hexadecimals.
TC-POST3-2: Encoded does not contain any CGI hexadecimals.

TC-POST5-1: Encoded contains 1+ malformed CGI

TC-POST6-1: Encoded contains 1+ illegal characters.

Derive Initial Test Specifications

- **TC-PRE2-1:** Encoded is a sequence of CGI items.
- **TC-PRE2-2:** Encoded is not a sequence of CGI items.
- **TC-POST1-1:** Encoded contains 1+ alphanumeric characters.
- **TC-POST1-2:** Encoded does not contain any alphanumeric characters.
- **TC-POST2-1:** Encoded contains 1+ ‘+’ characters.
- **TC-POST2-2:** Encoded does not contain any ‘+’ characters.
- **TC-POST3-1:** Encoded contains 1+ CGI hexadecimals.
- **TC-POST3-2:** Encoded does not contain any CGI hexadecimals.
- **TC-POST5-1:** Encoded contains 1+ malformed CGI hexadecimals.
- **TC-POST6-1:** Encoded contains 1+ illegal characters.

Complete Test Specifications Using Catalogs

- Final step is to generate additional test specifications from variables and operations using catalogs.
- Use variable type and context to add additional test specifications.
- In a catalog, value partitions are labeled based on whether that variable is an input, output, or either.

Common Catalogs

- Boolean
  - [in/out] true
  - [in/out] false
- Enumeration
  - [in/out] each enumerated value
  - [in] values outside of the enumerated set
- Range L..U
  - [in] L-1
  - [in/out] L
  - [in/out] A value between L and U
  - [in/out] U
  - [in] U+1
- Numeric Constant C
  - [in/out] C
  - [in] C - 1
  - [in] C + 1
  - [in] Any other constant in the same data type.
- Non-Numeric Constant C
  - [in/out] C
  - [in] Any other constant in the same data type
  - [in] Some other value of the same data type
- Sequence
  - [in/out] Empty
  - [in/out] A single element
  - [in/out] More than one element
  - [in/out] Maximum length (in bounded) or very large
  - [in] Longer than max length (if bounded)
  - [in] Incorrectly terminated
- Scan with action on element P
  - [in] P occurs at beginning of sequence
  - [in] P occurs in interior of sequence
  - [in] P occurs at end of sequence
  - [in] P appears twice in a row
  - [in] P does not occur in sequence

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

We Have Learned

- Catalogs can be used to come up with input and output partitions that make sense given the type and context we use variables in.
- We may have too many test specifications to realistically implement.
    - Can impose constraints through category-partition testing.
    - Can use combinatorial interaction testing to cover all *n-way* pairs efficiently.

Next Class

- Assessing test suite adequacy
  - How do we measure “good enough” testing?
- Structural testing
  - Deriving tests from the source code of the system.

- Reading: Chapter 9, 12
- Homework:
  - Assignment 1 is out. Due February 2.
  - Any questions?