

CS608

Software Testing

Dr. Stephen Brown
Room Eolas 116
stephen.brown@mu.ie

CS608

Equivalence Partitions in More Detail

(Essentials of Software Testing, Chapter 2.4-2.7)

Fault Model

- The equivalence partition fault model is where entire ranges of values are not processed correctly
- These faults can be associated with incorrect decisions in the code, or missing sections of functionality
- By testing with at least one value from every equivalence partition, where every value should be processed in the same way, equivalence partition testing attempts to find these faults

Description

- Each EP for each of the parameters is a test coverage item
- Both the inputs and the output should be considered
- Use representative values of each parameter from the EPs
- As few test cases as possible: each new test case should include as many uncovered partitions as possible
- Test coverage items for errors treated separately to avoid error hiding
- The goal is to achieve 100% coverage of the equivalence partitions

Recap Error Hiding

```
1  // return true if both x and y are valid
2  //           (within the range 0..100)
3  // otherwise return false to indicate an error
4  public static boolean valid(int x, int y) {
5      if (x<0 || x>100) return false;
6      if (y<0 || y>1000) return false;
7      return true;
8  }
```

Recap Error Hiding

```
1  // return true if both x and y are valid
2  //      (within the range 0..100)
3  // otherwise return false to indicate an error
4  public static boolean valid(int x, int y) {
5      if (x<0 || x>100) return false;
6      if (y<0 || y>1000) return false;
7      return true;
8  }
```

- Fault on line 6 – should use 100, not 1000

Recap Error Hiding

```
1  // return true if both x and y are valid
2  //      (within the range 0..100)
3  // otherwise return false to indicate an error
4  public static boolean valid(int x, int y) {
5      if (x<0 || x>100) return false;
6      if (y<0 || y>1000) return false;
7      return true;
8  }
```

- Fault on line 6 – should use 100, not 1000
- Example Error TCs (representative value > 100)
 - x=500
 - y=500

Recap Error Hiding

```
1  // return true if both x and y are valid
2  //      (within the range 0..100)
3  // otherwise return false to indicate an error
4  public static boolean valid(int x, int y) {
5      if (x<0 || x>100) return false;
6      if (y<0 || y>1000) return false;
7      return true;
8  }
```

- Fault on line 6 – should use 100, not 1000
- Example Error TCIs
 - x=500, y=500
- If test with multiple error TCIs
 - `assertFalse(valid(500,500))` – PASSES
- Does NOT find the error on line 6 – line 5 returns false first

Recap Error Hiding

```
1  // return true if both x and y are valid
2  //      (within the range 0..100)
3  // otherwise return false to indicate an error
4  public static boolean valid(int x, int y) {
5      if (x<0 || x>100) return false;
6      if (y<0 || y>1000) return false;
7      return true;
8  }
```

- Fault on line 6 – should use 100, not 1000
- Example Error TCIs: x=500, y=500
- Test with multiple error TCIs
 - `assertFalse(valid(500,500))` – PASSES
- Does NOT find the error on line 6 – line 5 returns false first
- **The Error TCI “y=500” is NOT tested**

Recap Error Hiding

```
1  // return true if both x and y are valid
2  //      (within the range 0..100)
3  // otherwise return false to indicate an error
4  public static boolean valid(int x, int y) {
5      if (x<0 || x>100) return false;
6      if (y<0 || y>1000) return false;
7      return true;
8  }
```

- And obviously error case ($x > 100$) hides the normal case where (y in range $0..100$)
 - Can't return an error and a non-error value
- And the error case ($y > 100$) hides the normal case where (x in range $0..100$)
 - Can't return an error and a non-error value

Summary

- Analysis
 - Parameters
 - Value Ranges
 - Equivalence Partitions
 - Selecting Equivalence Partitions
- Test Coverage Items
- Test Cases

Analysis/Parameters

- Methods (and functions) have explicit and implicit parameters
- Explicit parameters are passed in the method call
- Implicit parameters are not:
 - in C may be global variables
 - in Java may be attributes
- Both types of parameter must be considered in testing
- A complete specification should include all inputs and outputs.

Analysis/Value Ranges

- All inputs and outputs have both natural values, and specification-based, ranges of values
- The natural range is based on the type
- The specification-based ranges, or partitions, are based on the specified processing
- It often helps in analysing ranges to use a diagram (value line)

Value Lines

| Integer.MIN_VALUE | Integer.MAX_VALUE |
|-------------------|-------------------|
| | |

- Value line for a Java int
- 32-bit value
- Minimum value: -2^{31} (Integer.MIN_VALUE)
- Maximum value: $2^{31} - 1$ (Integer.MAX_VALUE)

Natural Ranges

- Natural ranges for common types
 - **byte** [Byte.MIN VALUE..Byte.MAX VALUE]
 - **short** [Short.MIN VALUE..Short.MAX VALUE]
 - **int** [Integer.MIN_VALUE..Integer.MAX_VALUE]
 - **long** [Long.MIN VALUE..Long.MAX VALUE]
 - **char** [Character.MIN VALUE..Character.MAX VALUE]

Natural Ranges

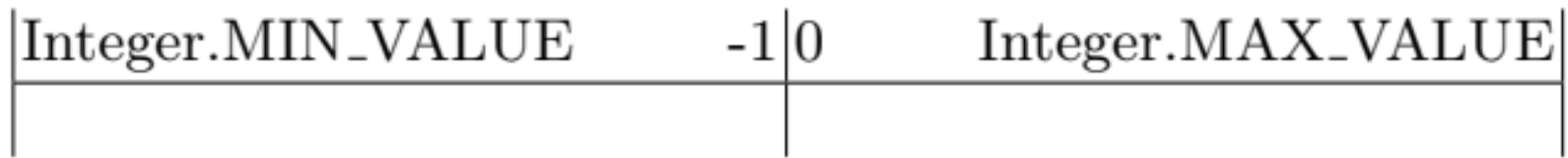
- For types with no natural ordering, each value is a separate range containing one value
 - **boolean** [true][false]
 - **enum Colour {Red, Blue, Green}** [Red][Blue][Green]
 - Treat numeric outputs with limited, specific values in the same way as enum's
- Consider compound types later...

Equivalence Partitions

- An Equivalence Partition is a range of values for a parameter for which the specification states *equivalent processing*
- Example:
 - boolean isNegative(int x)
 - Returns true if x is negative, otherwise false
- Two equivalence partitions for the parameter x can be identified:
 - Integer.MIN VALUE..-1
 - 0..Integer.MAX VALUE

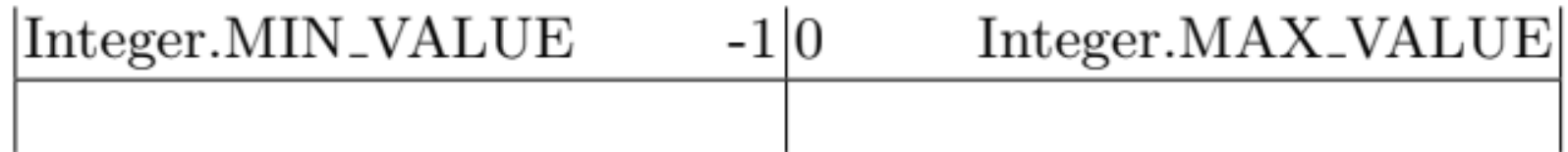
EP Example

- Two equivalence partitions for the parameter x can be identified:
 - Integer.MIN_VALUE..-1
 - 0..Integer.MAX_VALUE



- These specification-based ranges are called equivalence partitions
- According to the specification, any value in the partition is processed equivalently to any other value

EP Example



- Both inputs and outputs have natural ranges and equivalence partitions
- A boolean is an enumerated type with the values true and false
 - Each enumerated value is a separate range
- In this example, the two different values are produced by different processing, and so the return value has two equivalence partitions:
 - true
 - false

Guidelines for Selecting EPs

- Every value for every parameter must be in one equivalence partition
- There are no values between partitions
- The natural range of the parameter provides the upper and lower limits for partitions where not specified otherwise

Selecting EPs

- Equivalence partitions are used in testing as, according to the specification, any **one** value can be selected to represent any other value
- Instead separate tests for every value in the partition, a single test can be executed using a single value from the partition
- Equivalent to any other value, picked from anywhere in the partition
- Traditionally a value in the middle of the partition is picked
- Useful for testing the fundamental operation of the software: if the software fails using equivalence partition values, then it is not worth testing with more sophisticated techniques until the faults have been fixed

Test Coverage Items (TCI)

- Each partition for each input and output is a test coverage item
- It is good practice to give each test coverage item (for each test item) a unique identifier
- It is often useful to use the prefix “EP-” for EP test coverage items
- So, for example, for two methods, you can have:
 - method1(): EP1, EP2, EP3, etc.
 - method2(): EP1, EP2, EP3, etc.
- Selected values used in the Test Cases

Test Cases: normal and error

- Input test cases are selected, based on test coverage items which are not yet covered
- Ideally, each normal test cases will include as many additional normal test coverage items as possible
- Each test case that represents an error must only include one error test coverage item

Test Cases: expected results

- Values of expected results are derived from the specification
- **However, the tester must ensure that all the test coverage items related to the output parameters are covered**
- It may be necessary to read the specification “backwards” to determine input values that will result in an output value being in the required equivalence partition
 - Backwards means working from the return value to the matching inputs

Test Cases: Hint

- Identify test cases by going through the test coverage items in order, selecting the next uncovered test coverage item for each parameter in order (and finally for the return value)
- Selecting “fixed” equivalence partition values
- There is no reason to use different values from the same partition:
 - in fact it is easier to review the test case for correctness if one particular value is chosen from each partition, and then used throughout

Pitfalls

- The technique calls for a minimal number of tests cases

Pitfalls

- The technique calls for a minimal number of tests cases
- No tests for every combination of inputs

Pitfalls

- The technique calls for a minimal number of tests cases
- No tests for every combination of inputs
- No separate test case for each test coverage item

DUPLICATE TEST CASES

- Examine an example of unnecessary (duplicate) test cases in detail
- I've generated an alternative set of Test Cases that provide full EP coverage
- Using X1.1 etc. for "candidate" test cases
- And T1.1 etc. for final test cases

Unnecessary Test Cases (Test Duplication)

| ID | TCI Covered | Inputs | | Exp. Results |
|------|-------------|-------------|--------------|--------------|
| | | bonusPoints | goldCustomer | return value |
| X1.1 | EP2,5,7 | 40 | true | FULLPRICE |
| X1.2 | EP2,6[7] | 40 | false | FULLPRICE |
| X1.3 | EP[3]6,8 | 100 | false | FULLPRICE |
| X1.4 | EP4[6]8 | 200 | false | DISCOUNT |
| X1.5 | EP[4,6,8] | 1000 | false | DISCOUNT |
| X1.6 | EP1*,9 | -100 | false | ERROR |

Duplicate Test Cases

| ID | TCI Covered | Inputs | | Exp. Results |
|------|-------------|-------------|--------------|--------------|
| | | bonusPoints | goldCustomer | return value |
| X1.1 | EP2,5,7 | 40 | true | FULLPRICE |
| X1.2 | EP2,6[7] | 40 | false | FULLPRICE |
| X1.3 | EP[3]6,8 | 100 | false | FULLPRICE |
| X1.4 | EP4[6]8 | 200 | false | DISCOUNT |
| X1.5 | EP[4,6,8] | 1000 | false | DISCOUNT |
| X1.6 | EP1*,9 | -100 | false | ERROR |

- Duplicate test cases
- X1.5 covers the same TCIs as X1.4
 - EP4, EP6, EP8
- Either can be deleted

Duplicate Test Cases

| ID | TCI Covered | Inputs | | Exp. Results |
|------|-------------|-------------|--------------|--------------|
| | | bonusPoints | goldCustomer | return value |
| X1.1 | EP2,5,7 | 40 | true | FULLPRICE |
| X1.2 | EP2,6[7] | 40 | false | FULLPRICE |
| X1.3 | EP[3]6,8 | 100 | false | FULLPRICE |
| X1.4 | EP4[6]8 | 200 | false | DISCOUNT |
| X1.5 | EP[4,6,8] | 1000 | false | DISCOUNT |
| X1.6 | EP1*,9 | -100 | false | ERROR |

- Unnecessary test cases
- X1.2 can be deleted,
- X1.1 and X1.3 together cover all the test coverage items
 - X1.1 covers: EP and EP7
 - X1.3 covers: EP6

Evaluation

- Testing with equivalence partitions provides a minimum level of black-box testing
- At least one value has been tested from every input and output partition...
- ...using a minimum number of tests cases
- Ensure: basic data processing
- Do not exercise: different decisions made in the code
- Decisions frequent source of mistakes – generally reflect boundaries of input partitions, or combinations of inputs requiring particular processing
- These issues will be addressed in later techniques

Limitations

- The software has passed all the equivalence partition tests
- So is it fault free?

Is giveDiscount() Fault Free?

- Only exhaustive testing can answer this question
- Faults may remain
- Explore limitations of equivalence partition testing by injecting two different categories of fault into the code

Correct Code

```
22      public static Status giveDiscount(long bonusPoints, boolean
        goldCustomer)
23      {
24          Status rv = FULLPRICE;
25          long threshold=120;
26
27          if (bonusPoints<=0)
28              rv = ERROR;
29
30          else {
31              if (goldCustomer)
32                  threshold = 80;
33              if (bonusPoints>threshold)
34                  rv=DISCOUNT;
35          }
36
37          return rv;
38      }
```

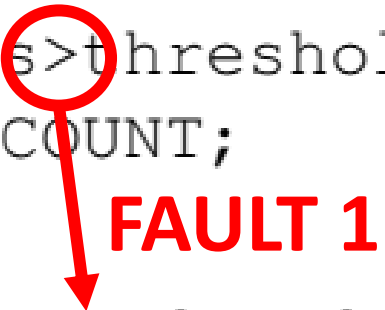
NOTE: If
bonusPoints is
greater than the
threshold, return
DISCOUNT

Fault 1

- Equivalence partition tests are designed to find faults associated with entire ranges of values
- If we inject a fault on line 33, which in effect disables the processing that returns DISCOUNT, we expect to see at least one test fail

```
33      if (bonusPoints>threshold)
34          rv=DISCOUNT;

33      if (bonusPoints==threshold) // fault 1
34          rv=DISCOUNT;
```



Run EP Tests Against Fault 1



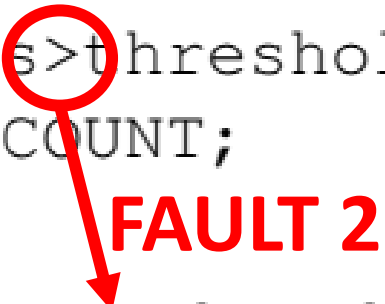
```
PASSED: test_giveDiscount("T1.1", 40, true, FULLPRICE)
PASSED: test_giveDiscount("T1.2", 100, false, FULLPRICE)
PASSED: test_giveDiscount("T1.4", -100, false, ERROR)
FAILED: test_giveDiscount("T1.3", 200, false, DISCOUNT)
java.lang.AssertionError: expected [DISCOUNT] but found [FULLPRICE]
=====
Command line suite
Total tests run: 4, Passes: 3, Failures: 1, Skips: 0
=====
```

Fault 2

- Equivalence partition tests are not designed to find faults at the values at each end of an equivalence partition
- If we inject a fault which moves the boundary value for the processing that returns DISCOUNT, then we do not expect to see any failed tests

```
33         if (bonusPoints > threshold)
34             rv=DISCOUNT;

33         if (bonusPoints >= threshold) // fault 2
34             rv=DISCOUNT;
```



Run EP Tests Against Fault 2



```
PASSED: test_giveDiscount("T1.1", 40, true, FULLPRICE)
PASSED: test_giveDiscount("T1.2", 100, false, FULLPRICE)
PASSED: test_giveDiscount("T1.3", 200, false, DISCOUNT)
PASSED: test_giveDiscount("T1.4", -100, false, ERROR)
Total tests run: 4, Passes: 4, Failures: 0, Skips: 0
```

=====

Strengths & Weaknesses

- Strengths
 - Provides a good basic level of testing
 - Well suited to data processing applications where input variables may be easily identified and take on distinct values allowing easy partitions
 - Provides a structured means for identifying basic test coverage items
- Weaknesses
 - Correct processing at the edges of partitions is not tested
 - Combinations of inputs are not tested

Key Points

- Equivalence partition testing tests basic software functionality
- Each range of values with *equivalent processing* is a test coverage item
- A representative value from each partition is selected as test data

Notes for Experienced Testers

- Probably reduce the number of formal steps
- For example, not use value lines for:
 - boolean and enumerated data types – list TCI directly
 - Simple integer parameters with few partitions – list TCI directly
- Method with a few simple parameters, directly develop the Test Cases
- Or even write the test code directly from the specification
- Reasons to document in detail:
 - High quality required – embedded systems, life-critical systems
 - Quality review
 - Legal challenges

Tutorial

- Complete the EP Tutorial on Moodle
 - Under topic DAY 3 – BVA Testing