



## 3. Test Generation – Domain Partitioning

Andrea Polini

Advanced Topics on Software Engineering – Software Testing  
MSc in Computer Science  
University of Camerino

# ToC

- 1 Equivalence Partitioning
- 2 Boundary Value Analysis
- 3 Category Partition

# Software Requirements

## Requirements Specification

- ▶ informal
- ▶ semi-formal
- ▶ formal

Depending on the degree of formality more or less **automated strategies** can be applied

# The test selection problem

## Challenge

Construct a test set  $\mathcal{T} \subseteq \mathcal{D}$  that will reveal as many errors in  $p$  as possible (where  $\mathcal{D}$  is the input domain and  $\mathcal{T}$  is the set of tests)

To give an idea...

Consider a procedure that has to manage data of an employee defined as follows:

- `ID:int` – three digit long from 001 to 999
- `name:string` – name is a 20 character long. Each characters belong to the set of 26 letters and space
- `rate:float` – rate varies from \$5 to \$10 per hour and in multiple of a quarter
- `hoursWorked:int` – hoursWorked varies from 0 to 60

Therefore:

$$999 \times 27^{20} \times 21 \times 61 \approx 5.42 \times 10^{34}$$

If time to execute a test is equal to  $1\mu\text{s}$  total time to execute all test will be  $\approx 1,75 \times 10^{21}$  years

# The test selection problem

## Challenge

Construct a test set  $\mathcal{T} \subseteq \mathcal{D}$  that will reveal as many errors in  $p$  as possible (where  $\mathcal{D}$  is the input domain and  $\mathcal{T}$  is the set of tests)

To give an idea . . .

Consider a procedure that has to manage data of an employee defined as follows:

- `ID:int` – three digit long from 001 to 999
- `name:string` – name is a 20 character long. Each characters belong to the set of 26 letters and space
- `rate:float` – rate varies from \$5 to \$10 per hour and in multiple of a quarter
- `hoursWorked:int` – hoursWorked varies from 0 to 60

Therefore:

$$999 \times 27^{20} \times 21 \times 61 \approx 5.42 \times 10^{34}$$

If time to execute a test is equal to  $1\mu\text{s}$  total time to execute all test will be  
 $\approx 1,75 \times 10^{21}$  years

# Equivalence partitioning

## How to ...

using the **equivalence partitioning** strategy a tester should subdivide the input domain into “small numbers” of subdomains, which can be **disjoint**

## Assumption

Equivalence classes are built assuming that the program under test exhibits the **same behaviour** on all elements of the same subset. One element for each subset is selected to form  $\mathcal{T}$

## Results?

Quality of  $\mathcal{T}$  depends from experience, familiarity with requirements, access and familiarity with the code

# Equivalence partitioning

## How to ...

using the **equivalence partitioning** strategy a tester should subdivide the input domain into “small numbers” of subdomains, which can be **disjoint**

## Assumption

Equivalence classes are built assuming that the program under test exhibits the **same behaviour** on all elements of the same subset. One element for each subset is selected to form  $\mathcal{T}$

## Results?

Quality of  $\mathcal{T}$  depends from experience, familiarity with requirements, access and familiarity with the code

# Equivalence partitioning

## How to ...

using the **equivalence partitioning** strategy a tester should subdivide the input domain into “small numbers” of subdomains, which can be **disjoint**

## Assumption

Equivalence classes are built assuming that the program under test exhibits the **same behaviour** on all elements of the same subset. One element for each subset is selected to form  $\mathcal{T}$

## Results?

Quality of  $\mathcal{T}$  depends from experience, familiarity with requirements, access and familiarity with the code



# Faults targeted

Simple partitioning:

- set of **legal** and **not legal** input
- try to identify requirements explicitly referring to different sets
  - Consider an application related to the management of pensions and retirements – (Req1: $i \in [1, \dots, 15]$ , Req2: $i \in [16, \dots, 65]$  and Req3: $i \in [65, \dots, 120]$ )
- above and below

## Formalizing the approach

Relations helping a tester in partitioning are of the form:

$$R \subseteq \mathcal{I} \times \mathcal{P}(\mathcal{I})$$

where the union of the subset in the relation could constitute a partition of the input domain

$$R_i : \mathcal{I} \rightarrow \{0, 1\}$$

### The grocery (simple one)

Consider a method `getPrice` that takes the name of a grocery item consults a database of prices and return the unit price for the item.

How would you partition the input?

$$pFound : \mathcal{I} \rightarrow \{0, 1\}$$

Elements in the database  $pF$  and elements not in the database  $pNF$ . They constitute a partition of  $\mathcal{I}$

## Formalizing the approach

Relations helping a tester in partitioning are of the form:

$$R \subseteq \mathcal{I} \times \mathcal{P}(\mathcal{I})$$

where the union of the subset in the relation could constitute a partition of the input domain

$$R_i : \mathcal{I} \rightarrow \{0, 1\}$$

### The grocery (simple one)

Consider a method `getPrice` that takes the name of a grocery item consults a database of prices and return the unit price for the item.

How would you partition the input?

$$pFound : \mathcal{I} \rightarrow \{0, 1\}$$

Elements in the database  $pF$  and elements not in the database  $pNF$ . They constitute a partition of  $\mathcal{I}$

## Formalizing the approach

Relations helping a tester in partitioning are of the form:

$$R \subseteq \mathcal{I} \times \mathcal{P}(\mathcal{I})$$

where the union of the subset in the relation could constitute a partition of the input domain

$$R_i : \mathcal{I} \rightarrow \{0, 1\}$$

### The grocery (simple one)

Consider a method `getPrice` that takes the name of a grocery item consults a database of prices and return the unit price for the item.

How would you partition the input?

$$pFound : \mathcal{I} \rightarrow \{0, 1\}$$

Elements in the database  $pF$  and elements not in the database  $pNF$ . They constitute a partition of  $\mathcal{I}$

## Formalizing the approach

Relations helping a tester in partitioning are of the form:

$$R \subseteq \mathcal{I} \times \mathcal{P}(\mathcal{I})$$

where the union of the subset in the relation could constitute a partition of the input domain

$$R_i : \mathcal{I} \rightarrow \{0, 1\}$$

### The grocery (simple one)

Consider a method `getPrice` that takes the name of a grocery item consults a database of prices and return the unit price for the item.

How would you partition the input?

$$pFound : \mathcal{I} \rightarrow \{0, 1\}$$

Elements in the database  $pF$  and elements not in the database  $pNF$ . They constitute a partition of  $\mathcal{I}$

# Examples

## Printers

Consider an automatic printer testing application named `pTest`. The application takes the manufacturer name and the model of a printer as input and selects a test script from a list. The script is then executed to test the printer. Our goal is to test if the script selection part of the application is implemented correctly. Different types of printers available (B/W Inkjet, Color Inkjet, Color laserjet, Color multifunction).

How would you partition the input?

It is possible to have overlapping among equivalence classes

## Words Count I

The `wordCount` method takes a word `w` and a file name `f` and returns the number of occurrences of `w` in the text contained in the file. An exception is raised if there is no file with name `f`.

How would you partition the input?

# Examples

## Printers

Consider an automatic printer testing application named `pTest`. The application takes the manufacturer name and the model of a printer as input and selects a test script from a list. The script is then executed to test the printer. Our goal is to test if the script selection part of the application is implemented correctly. Different types of printers available (B/W Inkjet, Color Inkjet, Color laserjet, Color multifunction).

How would you partition the input?

It is possible to have overlapping among equivalence classes

## Words Count I

The `wordCount` method takes a word  $w$  and a file name  $f$  and returns the number of occurrences of  $w$  in the text contained in the file. An exception is raised if there is no file with name  $f$ .

How would you partition the input?

# Examples

## Printers

Consider an automatic printer testing application named `pTest`. The application takes the manufacturer name and the model of a printer as input and selects a test script from a list. The script is then executed to test the printer. Our goal is to test if the script selection part of the application is implemented correctly. Different types of printers available (B/W Inkjet, Color Inkjet, Color laserjet, Color multifunction).

How would you partition the input?

It is possible to have overlapping among equivalence classes

## Words Count I

The `wordCount` method takes a word  $w$  and a file name  $f$  and returns the number of occurrences of  $w$  in the text contained in the file. An exception is raised if there is no file with name  $f$ .

How would you partition the input?



# Examples

## Words Count II

Now suppose to have access to the code of `wordCount`:

```
1 begin
2   string w, f;
3   input (w, f);
4   if (!exists(f)) {raise exception; return(0)};
5   if (length(w)==0) return (0);
6   if (empty(f)) return (0);
7   return (getCount (w, f));
8 end
```

How would you partition the input, now?

Combination of `w`: null/non-null, `f`: exists/does not exist, empty/non empty

In some cases the equivalence classes are based on the output generated by the program

# Examples

## Words Count II

Now suppose to have access to the code of `wordCount`:

```
1 begin
2   string w, f;
3   input (w, f);
4   if (!exists(f)) {raise exception; return(0)};
5   if (length(w)==0) return (0);
6   if (empty(f)) return (0);
7   return (getCount (w, f));
8 end
```

How would you partition the input, now?

Combination of `w`: null/non-null, `f`: exists/does not exist, empty/non empty

In some cases the equivalence classes are based on the output generated by the program

# Examples

## Words Count II

Now suppose to have access to the code of `wordCount`:

```
1 begin
2   string w, f;
3   input (w, f);
4   if (!exists(f)) {raise exception; return(0)};
5   if (length(w)==0) return (0);
6   if (empty(f)) return (0);
7   return (getCount(w, f));
8 end
```

How would you partition the input, now?

Combination of `w`: null/non-null, `f`: exists/does not exist, empty/non empty

In some cases the equivalence classes are based on the output generated by the program

# Equivalence classes for variables

There are some guidelines to define equivalence classes on the base of variables domains and defined requirements. **They reflect possible implementation choices related to explicit knowledge or implicit one:**

- **Range** (implicitly or explicitly defined): one class with values inside the range and two with values outside the range
- **Strings**: at least one containing all legal strings and one containing all illegal strings
- **Enumerations**: each value in a separate class
- **Arrays**: one class containing all legal arrays, one the empty array, and one larger than the expected size
- **Compound Data Types** (e.g. age and name): combine the classes composing the compound type

# Unidimensional vs. Multidimensional partitioning

## Unidimensional

Each input variable is considered per-se and classes are combined to cover all the possible equivalence classes

## Multidimensional

The **Cartesian product** of equivalence classes is considered and test derived accordingly.

# Partitioning

## A Systematic Procedure

- ▶ **Identify input domains** – read requirements carefully, identify input and output variables and their types, as well as conditions related to them.
- ▶ **Equivalence classing** – partition the set of values of each variable
- ▶ **Combine equivalence classes** – combine equivalence classes
- ▶ **Identify infeasible equivalence classes** – combination of data that cannot be input to the application under test

# The Boiler Control System (BCS)

## BCS

The control system takes in input:

- ▶ A command:  $cmd = (temp|shut|cancel)$
- ▶ When  $temp$  is selected  $tempch = -10| -5|5|10$

Input can be provided via a GUI or via a configuration file.

How would you partition the input domain?

## BCS input domain

Variable	Type	Value(s)
$V$	Enumerated	{ <i>GUI, file</i> }
$F$	String	<i>A file name</i>
$cmd$	Enumerated	{ <i>temp, cancel, shut</i> }
$tempch$	Enumerated	{ <i>-10, -5, 5, 10</i> }

# The Boiler Control System (BCS)

## BCS

The control system takes in input:

- ▶ A command:  $cmd = (temp|shut|cancel)$
- ▶ When  $temp$  is selected  $tempch = -10| -5|5|10$

Input can be provided via a GUI or via a configuration file.

How would you partition the input domain?

## BCS input domain

Variable	Type	Value(s)
$V$	Enumerated	{ <i>GUI, file</i> }
$F$	String	<i>A file name</i>
$cmd$	Enumerated	{ <i>temp, cancel, shut</i> }
$tempch$	Enumerated	{ <i>-10, -5, 5, 10</i> }



# The Boiler Control System (BCS)

## BCS Equivalence Classes

Variable	Type	Value(s)
<i>V</i>	Enumerated	{{ <i>GUI</i> }, { <i>file</i> }, { <i>undefined</i> }}
<i>F</i>	String	<i>f_valid</i> , <i>f_invalid</i>
<i>cmd</i>	Enumerated	{{ <i>temp</i> }, { <i>cancel</i> }, { <i>shut</i> }, { <i>c_invalid</i> }}
<i>tempch</i>	Enumerated	{{ -10 }, { -5 }, { 5 }, { 10 }, { <i>c_invalid</i> }}

# ToC

- 1 Equivalence Partitioning
- 2 Boundary Value Analysis**
- 3 Category Partition

# Boundary-value analysis

Experience indicates that programmers make mistakes in processing values at and near the boundaries of equivalence classes

## Boundary-value analysis

test-selection techniques that targets faults in applications at the boundaries of equivalence classes.

Once the input domain has been identified:

- Partition the input domain using unidimensional partitioning
- Identify the boundaries of each partition
- Select test data such that each boundary value occurs in at least one test input

# Boundary-value analysis

Experience indicates that programmers make mistakes in processing values at and near the boundaries of equivalence classes

## Boundary-value analysis

test-selection techniques that targets faults in applications at the boundaries of equivalence classes.

Once the input domain has been identified:

- Partition the input domain using unidimensional partitioning
- Identify the boundaries of each partition
- Select test data such that each boundary value occurs in at least one test input

# Boundary-value analysis

Experience indicates that programmers make mistakes in processing values at and near the boundaries of equivalence classes

## Boundary-value analysis

test-selection techniques that targets faults in applications at the boundaries of equivalence classes.

Once the input domain has been identified:

- Partition the input domain using unidimensional partitioning
- Identify the boundaries of each partition
- Select test data such that each boundary value occurs in at least one test input

# BA Example

## The *findPrice* procedure

Two integer parameter *code* and *quantity* with the following input domain:

- $99 \leq \textit{code} \leq 999$
- $1 \leq \textit{quantity} \leq 100$

Which are the relevant partitions?

Which are the relevant boundary values?

# BA Example

## The *findPrice* procedure

Two integer parameter *code* and *quantity* with the following input domain:

- $99 \leq \textit{code} \leq 999$
- $1 \leq \textit{quantity} \leq 100$

Which are the relevant partitions?

Which are the relevant boundary values?

# BA Example

Consider the following test set:

$$T = \left\{ \begin{array}{l} t_1 : (code = 98, quantity = 0), \\ t_2 : (code = 99, quantity = 1), \\ t_3 : (code = 100, quantity = 2), \\ t_4 : (code = 998, quantity = 99), \\ t_5 : (code = 999, quantity = 100), \\ t_6 : (code = 1000, quantity = 101), \end{array} \right\}$$

Minimal but:

```
public void fP(int code, int quantity) {
    if (code < 99 && code > 999)
        {display_error("invalid code"); return;}
    // Validity check for quantity is missing!
    // Begin processing code and quantity
    ...
}
```



## BA Example

Consider the following test set:

$$T = \{ \begin{array}{l} t_1 : (code = 98, quantity = 0), \\ t_2 : (code = 99, quantity = 1), \\ t_3 : (code = 100, quantity = 2), \\ t_4 : (code = 998, quantity = 99), \\ t_5 : (code = 999, quantity = 100), \\ t_6 : (code = 1000, quantity = 101), \end{array} \}$$

Minimal but:

```
public void fP(int code, int quantity) {
    if (code < 99 && code > 999)
        {display_error("invalid code"); return;}
    // Validity check for quantity is missing!
    // Begin processing code and quantity
    ...
}
```

# On Combining Boundary Values

## `textSearch`

Consider the method `textsearch` that takes in input a string `s` and a text `txt` and checks if the string is a substring of the text. In such a case it returns in `p` the position of the first character, -1 otherwise.

“Vi veri universum vivus vici”

# On Combining Boundary Values

## Combining Boundary Values

Boundary values for an input set should be tested in isolation **avoiding interferences** from other input sets

## Relationship among input variables

- ▶ Relationships amongst the input variables must be examined carefully while identifying boundaries along the input domain.
- ▶ Additional tests may be obtained when using a partition of the input domain obtained by taking the product of equivalence classes created using individual variables

# ToC

- 1 Equivalence Partitioning
- 2 Boundary Value Analysis
- 3 Category Partition**

# Category Partition Method

## the findPrice procedure (2nd version)

```
findPrice (code, quantity, weight)
```

- ▶ `code`: string of eight digits
- ▶ `qty`: quantity purchased
- ▶ `weight`: weight of the purchased item

The procedure accesses a database to find and display **the unit price**, **the description**, and the **total price** of the item corresponding to `code`.

Leftmost digit	Interpretation
0	Ordinary grocery items such as bread, magazines soup
2	Variable-weight items such as meats, fruits, and vegetables
3	Health-related items such as tylenol, bandaids, and tampons
5	Coupon; digit 2(dollars), 3 and 4 (cents) specify the discounts
1, 6-9	unused

# Category Partition Method

## CP Method

Mixed **manual/automatic** approach consisting of **eight successive steps** based approach to go from **requirements to test scripts**

- Analyze specification
- Identify Categories
- Partition Categories
- Identify Constraints
- (Re)write test specification
- Process Specification
- Evaluate Generator Output
- Generate Test Scripts

# Analyze Specification

The tester identify **each functional unit** that can be tested separately

E.g. it could be the case that the `findPrice` procedures implements in a single component the functionalities related to the retrieval of information from the database

# Identify categories

For each testable unit the specification is analyzed and inputs isolated. Also objects in the environment are considered. Then the relevant characteristics (category) of each parameter are identified

*findPrice*

Categories:

- *code*: length, leftmost digits, remaining digits
- *quantity*: integer quantity
- *weight*: float quantity
- *database*: contents



# Partition Categories

For each category different cases (**choices**) against which to test the functional units are identified.

- *code*:
  - Length: Valid (8 digits), Invalid (< or > 8)
  - leftmost digit: 0,2,3,5,others
  - remaining digits: valid string, invalid string
- *quantity*: valid, invalid
- *weight*: valid, invalid
- **Environments**
  - *Database*: item exists, item does not exist

# Identify Constraints

Constraints among choices are specified and used to exclude infeasible tests

# (Re)write test specification

The tester write a complete test specification using a TSL, and taking into account the information derived in the previous steps

# Process Specification

TSL specification are processed to derive test frames (test template).

# Evaluate Generator Output

Generated tests are analyzed for redundancy of missing cases

# Generate Test Scripts

Test frames are finally grouped into test scripts

CP is mainly a systematization of the equivalence partitioning and boundary value analysis

# Generate Test Scripts

Test frames are finally grouped into test scripts

CP is mainly a systematization of the equivalence partitioning and boundary value analysis

# In Summary

Test derivation strategies based on characteristics of the input sets

- Partitioning
- Boundary analysis
- Category Partition