**Category-Partition Testing Method**

**Steps needed to Implement the Category-Partition Method**

1. Identify independently testable feature (use case, method)
2. \*Identify inputs/parameters for feature
3. Identify categories/characteristics for each input/parameter
4. Partition categories into choices
5. Identify constraints among choices
6. Produce and evaluate test case specifications (frames) using TSL as needed.
7. Generate test cases from test case specifications which should be validated and verified.
8. \*Identify what criteria must hold true for each test case to pass, *i.e.* what are you going to assert? Note that sometimes there are multiple side-effects so each one must be checked.

\*These steps were not listed in the video you watched, they were implicit. I list them here to make it clear that these items need to be addressed. And, in particular, that you address them when doing one of the homeworks.

Finally, to complete:

1. Write JUnit tests.
2. Run JUnit tests and assess output

**Example**

1. Suppose we have an algorithm that calculates the amount of health subsidy that is needed for an individual depending on their age, income, and support from government programs. Age is classified as either minor, adult, or senior. Income is below poverty level or above. Support is none, Medicaid, Medicare. Thus, the 3 inputs and their equivalence classes (choices) are:

|  |  |  |  |
| --- | --- | --- | --- |
| Input | Category | Choices | Coded Choices |
| Age | Value | Minor, Adult, Senior | M, A, S |
| Income | Value | Below Poverty, Above Poverty | B, A |
| Support | Value | None, Medicaid, Medicare | N, A, C |

Thus, we can that with no constraints that there are 18 = 3\*2\*3 test frames:

MBN, MBA, MBC, MAN, MAA, MAC, ABN, ABA, ABC, AAN, AAA, AAC, SBN, SBA, SBC, SAN, SAA, SAC

1. Suppose that only seniors who are below poverty level can receive Medicare. That removes the 5 combinations shown highlighted in yellow, leaving 13 test frames.

MBN, MBA, MBC, MAN, MAA, MAC, ABN, ABA, ABC, AAN, AAA, AAC, SBN, SBA, SBC, SAN, SAA, SAC

The statement that only seniors below poverty can receive Medicare is a *constraint*. We can represent this in TSL as:

Age:

Minor.

Adult.

Senior. [property old]

Income:

BP. [property poor]

AP.

Support:

None.

Aid.

Care. [if old && poor]

And when we run the TSL program on this data we get 13 frames:

|  |  |  |
| --- | --- | --- |
| Test Case 1 (Key = 1.1.1.)  Age : Minor  Income : BP  Support : None  Test Case 2 (Key = 1.1.2.)  Age : Minor  Income : BP  Support : Aid  Test Case 3 (Key = 1.2.1.)  Age : Minor  Income : AP  Support : None  Test Case 4 (Key = 1.2.2.)  Age : Minor  Income : AP  Support : Aid  Test Case 5 (Key = 2.1.1.)  Age : Adult  Income : BP  Support : None | Test Case 6 (Key = 2.1.2.)  Age : Adult  Income : BP  Support : Aid  Test Case 7 (Key = 2.2.1.)  Age : Adult  Income : AP  Support : None  Test Case 8 (Key = 2.2.2.)  Age : Adult  Income : AP  Support : Aid  Test Case 9 (Key = 3.1.1.)  Age : Senior  Income : BP  Support : None  Test Case 10 (Key = 3.1.2.)  Age : Senior  Income : BP  Support : Aid | Test Case 11 (Key = 3.1.3.)  Age : Senior  Income : BP  Support : Care  Test Case 12 (Key = 3.2.1.)  Age : Senior  Income : AP  Support : None  Test Case 13 (Key = 3.2.2.)  Age : Senior  Income : AP  Support : Aid |

1. Now, suppose that minors at any income level can also receive Medicare.

Age:

Minor. [property young]

Adult.

Senior. [property old]

Income:

BP. [property poor]

AP.

Support:

None.

Aid.

Care. [if (old && poor) || young]

Which yields 15 test frames.

1. Go back to the original version, with no constraints. Now, suppose that only minors and seniors with below poverty income can receive Medicare:

Age:

Minor.

Adult. [property adult]

Senior.

Income:

BP. [property poor]

AP.

Support:

None.

Aid.

Care. [if !adult && poor]

Which yields 14 test frames.

**Example from Video Lecture**

One of the most useful features of CPM is the idea of specifying *characteristics/categories* for each input. For the example below, the input, *str* has two characteristics: *length* and *content.* For the input to a mathematical function, f(x), the only characteristic is probably the value of x, which we may still need to partition into choices. However, for many situations, the input may have different characteristics that make the algorithm behave differently.

|  |
| --- |
| split(string str, int size) |
| Inputs:   1. str – Categories: 2. Length – Choices: 3. 0 4. size-1 5. size 6. size+1 7. 2\*size-1 8. ... 9. Content – Choices: 10. spaces 11. special characters 12. ... 13. size – Category: 14. Value – Choices: 15. 0 16. >0 17. <0 18. MAXINT 19. ... |

The CPM essentially treats each *characteristics/categories* as a separate “input”. In other words, all choices in each category are combined with all other choices in every other category (until constraints are added).

**Example from Fall 2017 Project**

|  |  |
| --- | --- |
| **Input for TSL** | **Output** |
| # Book  Book Status:  Available.  On Hold. [property onHold]  Invalid. [error]  # Library Card  Card Status:  Valid.  Invalid. [error]  # Member  Member Status:  Good Standing.  Suspended Overdue. [single]  Suspended Lost Book. [single]  Suspended General. [single]  # Num Books Checked Out  Num Books:  0-9.  10. [if !onHold] | Test Case 1 <error>  Book Status : Invalid  Test Case 2 <error>  Card Status : Invalid  Test Case 3 <single>  Member Status : Suspended Overdue  Test Case 4 <single>  Member Status : Suspended Lost Book  Test Case 5 <single>  Member Status : Suspended General  Test Case 6 (Key = 1.1.1.1.)  Book Status : Available  Card Status : Valid  Member Status : Good Standing  Num Books : 0-9  Test Case 7 (Key = 1.1.1.2.)  Book Status : Available  Card Status : Valid  Member Status : Good Standing  Num Books : 10  Test Case 8 (Key = 2.1.1.1.)  Book Status : On Hold  Card Status : Valid  Member Status : Good Standing  Num Books : 0-9 |

1. In some ways, I don’t like how the program handles “single” and “error” because it doesn’t specify what choices should be used for the other inputs. You could make things more explicit as the following example shows. This results in the same 8 test frames as above.

# Book

Book Status:

Available. [property available]

On Hold. [property onHold]

Invalid. [error]

# Library Card

Card Status:

Valid. [property valid]

Invalid. [error]

# Num Books Checked Out

Num Books:

0-9. [property 0to9]

10. [if !onHold]

# Member

Member Status:

Good Standing.

Suspended Overdue. [if available && valid && 0to9] #instead of single

Suspended Lost Book. [if available && valid && 0to9]

Suspended General. [if available && valid && 0to9]

Note that Test Frame 3 above was:

Test Case 3 <single>

Member Status : Suspended Overdue

Now it has become:

Test Case 4 (Key = 1.1.1.2.)

Book Status : Available

Card Status : Valid

Num Books : 0-9

Member Status : Suspended Overdue

1. Note that this might be too tedious with a larger example. Also, properties must be defined before they can be used, otherwise you will get an error! That is why I moved “Num Books” above “Member Status” so that I could use “property 0to9” in the if statements in “Member Status”. Another example appears later.

**Notes about using the TSL Program**

|  |
| --- |
| #Age of Person  Age:  Minor.  Adult.  Senior. [property old]  #Income of Person  Income:  BP. [property poor]  AP.  #Suppor reveived  Support:  None.  Aid.  Care. [if old && poor] |

1. A # indicates a comment. Optional.
2. In category-partition, we use: input variable -> characteristic -> choice to model a situation. However, with TSL there is no “variable”, you only specify characteristic and choices. The variable is indicated by the comment preceding each characteristic.
3. Characteristics end with a colon
4. Choices end with a period
5. Use &&, ||, and ! when constructing if statements on choices.
6. Use parentheses to group if needed, however no space between parenthesis and property. For example:

Valid: [if (x && y) || (x && z)]

Not valid: [if ( x && y ) || (x && z)]

1. To run TSL, display the number of test frames, and be prompted to save results:

tsl –c pathToSpecFile/specFile

For example: tsl –c Specs/getPay.txt

1. To run TSL and display test frames to the console:

tsl –s pathToSpecFile/specFile

1. To run TSL and save test frames to a text file:

tsl pathToSpecFile/specFile –o outFile.txt.tsl

1. If a test frame has “n/a” for a choice this means something about your constraints is wrong, leading to ambiguity.
2. Properties must be defined before they can be used, otherwise you will get an error. This seems to me to be a serious limitation. In other words, this will give an error:

Support:

None.

Aid.

Care. [if old]

Age:

Minor.

Adult.

Senior. [property old]