Assignment 3

In this assignment, our goal is to execute the "Boundary Value Analysis" in functions of our open source project. The objective of this method is to derive tests by identifying and testing the boundaries of the parameters, thus selecting the most critical scenarios. For this purpose, we selected a subset of 3 functions from the set of functions that we selected in the last assignment (Category Partition).

Selected functions:

`Strings.insertPadded`

- **Why**: It is a function with a very straight forward behaviour, clearly identifiable input characteristics.
- **Purpose**: Inserts a string into another string, padding it with spaces. It is aware if the insertion start and insertion end has spaces, and does not add more spaces.

`Strings.isBlank`

- **Why**: It is a simple but important function to detect blank inputs, that can be particurlaly common in GUI applications were the user might send blank data.
- **Purpose**: Detects if a string is null, empty or blank.

`TaskIo.loadTasksFromFile`

- **Why**: It deals with file handling and it is an important feature in the application.
- **Purpose**: This function reads a file in order to retrieve stored tasks and return an array with them.

Step-by-step of the Bound-value analysis for each function.

`String insertPadded(String s, int insertAt,String stringToInsert)`

- Parameters:
  - **s**: base string
  - **insertAt**: insertion point
  - **stringToInsert**: string to insert into `s` at `insertAt` index
- Returns: A string with *stringToInsert* at *insertAt* index of *s* and between whitespaces
- Input characteristics:
  - **s**: String object or null
  - **insertAt**: integer | 0 <= insertAt < s.length()

  - **strintToInsert**: String object or null

- Partitions:
  - E1: **s** is null
  - E2: **s** is empty
  - E3: **s** has a space before or after the insertion point.
  - E4: **s** has a space in both ends.
  - E5: **s** has no spaces in neither end of the insertion point.
  - E6: **insertAt** is in bounds, 0 <= **insertAt** <= len(**s**).
  - E7: **insertAt** is outside bounds, **insertAt** <= -1 || **insertAt** >= len(**s**) + 1.
  - E8: **stringToInsert** is null
  - E9: **stringToInsert** is empty
  - E10: **stringToInsert** is has at least one character

- Values On-point:
  - E1: s = null
  - E2: s = ""
  - E3: s = "12 4" && insertAt = 1 ; s = "12 4" && insertAt = 3
  - E4: s = "1 2 4" && insertAt = 2;
  - E5: s = "124" && insertAt = 1;
  - E6: insertAt = {0, len(s)}
  - E7: insertAt = {-1, len(s) + 1}
  - E8: stringToInsert = null
  - E9: stringToInsert = ""
  - E10: stringToInsert = "1"

- Values Off-point:
  - E1: s = {"","1"}
  - E2: s = {null, "1"}
  - E3: s = "12 4" && insertAt = 3 ; s = "12 4" && insertAt = 1
  - E4: s = "1 2 4" && insertAt = 1
  - E5: s = "12 4" && insertAt = 1
  - E6: insertAt = {-1, len(s) + 1}
  - E7: insertAt = {0, len(s)}
  - E8: stringToInsert = {"","1"}
  - E9: stringToInsert = {null, "1"}
  - E10: stringToInsert = {null, ""}

- Generated unit tests: In this function, we decided to include one test per partition and, because of the interdependence of the input variables, we also created tests with a combination of these partitions. For the *insertAt* integer variable, we also created tests using the boundaries of its related partitions. The strategy for the string variables was to test its on-point values together with different on-points and off-points of the other stringToInsert variable.

- *baseStringIsNull()* -> Tests passing a null baseString value.
  - Failed. Unexpected NullPointerException
- *baseStringIsEmpty()* -> Tests passing an empty baseString value with an empty and also a not empty stringToInsert.
  - Failed. Resulting string was not padded by spaces before and after stringToInsert value, this is not clear in the documentation but we assumed it is an unexpected behaviour.
- *spaceInInsertAtTest()* -> Tests passing a string with a space in the insertAt position.
  - Passed. Resulting string had the expected value.
- *spaceInBothEndsTest()* -> Tests passing the insertAt before and after a space.
  - Passed. Resulting string had the expected value.
- *noSpacesInBothEnds()* -> Tests passing the insertAt to a position within no spaces.
  - Passed. Resulting string had the expected value.
- *inBoundInsertionPoint()* -> Tests if the lower and upper boundary of the E6 partition works as expected.
  - Failed. Resulting string when using the lower boundary is not padded with a space before the actual value.
- *outBoundsInsertionPoint()* -> Tests the boundaries of the E7 partition.
  - Passed. Function correctly throws IndexOutOfBoundsException as expected.
- *stringToInsertIsNull()* -> Tests passing stringToInsert with null value.
  - Passed. Resulting string had the expected value.
- *stringToInsertIsEmpty()* -> Tests passing stringToInsert with empty value.
  - Passed. Resulting string had the expected value.
- *stringToInsertIsNotEmpty()* -> Tests passing stringToInsert with length greater than 0.
  - Passed. Resulting string had the expected value.

`boolean isBlank(String s)`

- Parameters:
  - **s**: base string
- Returns:
  - True if **s** is an empty string, a whitespace-only string, or null.
- Input characteristics:
  - **s**: String object or null
- Partitions:
  - E1: **s** is null.
  - E2: **s** is empty.
  - E3: **s** has only whitespaces.

  - E4: **s** has at least one character that is not a whitespace.
- Values On-point:
  - E1: **s** = null
  - E2: **s** = ' '
  - E3: **s** = ' '
  - E4: **s** = 'a'
- Values Off-point:
  - E1: **s** = {Empty string, Whitespace only string, Any valid string}
  - E2: **s** = {Null string, Whitespace only string, Any valid string that has at least one character}
  - E3: **s** = {Empty string, Null string, Any valid string with at least one non-whitespace character}
  - E4: **s** = {Null string, Whitespace only string, Empty string}
- Generated unit tests: As there is only one input variable, and the partitions are fairly straight forward, we can create a test for each partition. Fortunately, the tests created for the last assignment (Category-partition) already cover our range of possibilities here:
  - *nullIsBlankTest()* -> Tests "input is null" category.
    - Passed. The input is considered blank.
  - *emptyIsBlankTest()* -> Tests "input is empty" ccategory.
    - Passed: The input is considered blank.
  - *whitespaceOnlyIsBlankTest()* -> Tests "input has only whitespaces" category.
    - Passed: The input is considered blank.
  - *regularStringIsNotBlankTest()* -> Tests "has at least one character that is not a whitespace" category.
    - Passed: The input is considered not blank.

`ArrayList<Task> loadTasksFromFile(File file)`

- Parameters:
  - **file**: File storing the tasks
- Returns: An array of the stored tasks in the file.
- Input characteristics:
  - **file**: File object or null
- Partitions:
  - E1: **file** is null.
  - E2: **file** object is not null but file does not exist.
  - E3: **file** exists and it is not empty.
  - E4: **file** exists but it is empty.
- Values On-point:
  - E1: file = null

  - E2: file = Non-existent File
  - E3: file = File stream
  - E4: file = Empty File stream

- Values Off-point:
  - E1: file = {Unknown File, File Stream, Empty File Stream}
  - E2: file = {null, File Stream, Empty File Stream}
  - E3: file = {null, Unknown File, Empty File Stream}
  - E4: file = {null, File Stream, Unknown File}

- Generated unit tests: There is only one input variable, so we create one test for each partition related to this variable. Given the type of the variable, it is not possible to derive multiple boundaries.
  - *loadNullFile()* -> Tests passing a null file.
    - Failed, Throws an unexpected NullPointerException.
  - *loadNonExistentFile()* -> Tests passing a unknown file.
    - Passed, returns an empty array.
  - *loadNotEmptyFile()* -> Tests passing a file with tasks stored.
    - Passed, returns an array with the correct tasks that were stored in the file.
  - *loadEmptyFile()* -> Tests passing an empty file.
    - Passed, returns an empty array.

Conclusion

In the case of more straight forward partitions and single arguments, we did not have to produce tests different from those produced for the last assignment, and given the lack of numeric variables in most of the functions, we did not observe the obvious advantage of this technique (deriving numeric test cases that check the bounds of the numeric variable), however, for more complex partitions (insertPadded, for example), we could also verify the advantages of defining the boundaries to produce test cases that can accurately test edge cases in a scenario in which these might not be so obvious.