# Assignment 2

In this assignment, we decided to use the package `com.todotxt.todotxttouch.util` to select 5 distinct funtions to perform unit tests.

## 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.

**`Util.createImageIcon`**

- **Why**: Relies on specified resource actually existing in the filesystem.
- **Purpose**: Loads a resource for rendering/drawing.

**`Util.integerList2IntArray`**

- **Why**: Useful function which iterates through a list and creates an array.
- **Purpose**: Receives and Integer list and returns an equivalent array of ints.

**`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 'Category-Partition' algorithm for each function.

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

- Parameters:
  - **s**: base string
  - **insertAt**: insertion point

    - **stringToInsert**: string to insert into `s` at `insertAt` index
- Input characteristics:
    - **s**: String object or null
    - **insertAt**: integer | 0 <= insertAt < s.length()
    - **strintToInsert**: String object or null
- Categories:
    - **s** has a space before or after the insertion point.
    - **s** has a space in both ends.
    - **s** has no spaces in neither end of the insertion point.
    - **insertAt** is in bounds.
    - **insertAt** is outside bounds.
- Generated unit tests:
    - *inBoundInsertionPoint()* -> Tests "no spaces in neither end category" and "insertAt in bounds" category.
        * Passed. Resulting string had the expected value.
    - *outBounsInsertionPoint()* -> Tests "insertAt out of bounds" ccategory.
        * Failed: function does not deal correctly with this scenario.
    - *spaceInBothEndsTest()* -> Tests "space in both ends" category.
        * Passed. Resulting string had the expected value.
    - *spaceInStartTest()* -> Tests "space before or after insertion" category.
        * Passed. Resulting string had the expected value.

**`boolean isBlank(String s)`**

- Parameters:
    - **s**: base string
- Input characteristics:
    - **s**: String object or null
- Categories:
    - **s** is null.
    - **s** is empty.
    - **s** has only whitespaces.
    - **s** has at least one character that is not a whitespace.
- Generated unit tests:
    - *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.

**`ImageIcon createImageIcon(String path)`**

- Parameters:
  - **path**: string
- Input characteristics:
  - **r**: string specifying a resource in the filesystem.
- Categories:
  - **path** specifies a resource that exists.
  - **path** specifies a inexisting resource.
- Generated unit tests:
  - *resourceExistsTest()* -> Tests if function works properly with an existing resource.
    - Passed.
  - *resourceDoesNotExistsTest()* -> Tests if method handles inexisting resources as expected.
    - Passed.

**`int[] integerList2IntArray(List<Integer> integerlist)`**

- Parameters:
  - **integerlist**: list of Integer objects
- Input characteristics:
  - **integerlist**: is as List object or null
- Categories:
  - **integerlist** is null.
  - **integerlsit** is not null but is empty.
  - **integerlist** is not null and not empty.
- Generated unit tests:
  - nullIntegerList2IntArray() -> Tests if it handles a null list as input.
    - Failed, it throws an exception not documented in the docs.
  - emptyIntegerList2IntArray() -> Tests if it handles an empty array.
    - Passed, returns an empty array as expected.
  - notEmptyIntegerList2IntArray() -> Tests if it successfully converts the list to an array.
    - Passed, returns a non empty array of ints with equivalent elements as the input.

**`ArrayList<Task> loadTasksFromFile(File file)`**

- Parameters:
  - **file**: File storing the tasks
- Input characteristics:
  - **file**: File object or null
- Categories:
  - **file** is null.
  - **file** object is not null but file does not exist.
  - **file** exists and it is not empty.

    - **file** exists but it is empty.
- Generated unit tests:
    - *loadNullFile()* -> Tests if a null File object throws an expected exception.
        - Fail, throws unexpected exception.
    - *loadNonExistentFile()* -> Tests if it handles non existent files.
        - Passed, returns an empty array as expected.
    - *loadNotEmptyFile()* -> Tests if it returns the right tasks array from a file.
        - Passed, returns an array with the file tasks.
    - *loadEmptyFile()* -> Tests if the function handles an empty file and return an empty array.
        - Passed, returns an empty array.

## Conclusion

As a conclusion of the outcome, we found out that the code is very coupled, at least in the package `com.todotxt.todotxttouch.util`, so some tests are difficult to set up and perform without the use of mocks, such as the `getRelativeDate(Calendar d1, Calendar d2)` method (we decided not to include it in this assignment), which, besides appearing to do a very specific task, it is also responsible for formatting the output and thus needs to use the language preference of the user, so some refactoring would be necessary to maintain the tests clear and more efficient.