Assignment 2: KWIC Extensions

Code Repository URL:	https://github.com/TIC3001/tic3001-ay2122s2-13/releases/tag/Assignment2		
Team number:	13		
Name:	Roshan Kumar	Name:	Noel Lim Xian
Student Number:	A0094621H	Student Number:	A0211270Y

1. Introduction

This assignment is a sequel to Assignment 1 and builds additional functionality using one of the two implementations as scaffold. We approached the solution using pipe and filter design pattern. As features is extended on existing codebase, we practiced iterative development with comprehensive testing to sustain productivity. We clearly define the sequence of action to be performed as follows:

i. Read data from file.

ii. Parse data as an array of lines, with each line an array of words.

iii. Perform circular shift of lines.

iv. Sort lines by alphabetical order.

v. Filter by omitting lines starting with *ignore* keywords.

vi. Filter by selecting lines starting with *require* keywords.

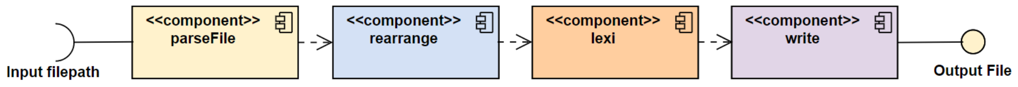
vii. Combine lines as text.

viii. Buffer test into output file.

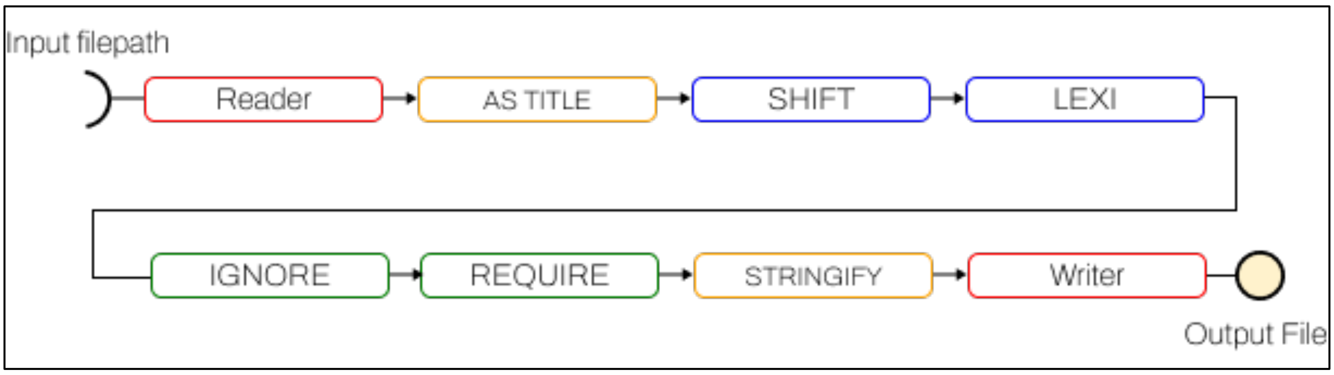
2. Applying Design Principles

Using Pipe and Filter design pattern, the sequence of action in the pipeline is composed of independent pipe and filter components. To meet additional requirements and achieve its expected behavior, we inject new components into the sequence. The design pattern allows us to write new components without affecting existing components. Testing components are also simpler as each component can be demarcated as a unit for unit testing.

Component Diagram - Assignment 1



Component Diagram - Assignment 2



2.1 From Principles to Implementation

To emphasize on pipe and filter pattern and its design principles, we adhere to the following implementation principles:

ix. No class instantiation of pipes and filters and all methods are static and unary. Each pipe and filter correspond to a method and have an input interface which takes in output of the previous pipe or filter. The workflow processes inputs, and operation and information state are self-contained in the filter's scope. Application should not require utilizing objects to store extraneous state. //Just to compare to OOP

x. As the application is a synchronous routine, pipes are transient in the design. Output of the previous filter will be immediately fed into the next filter.

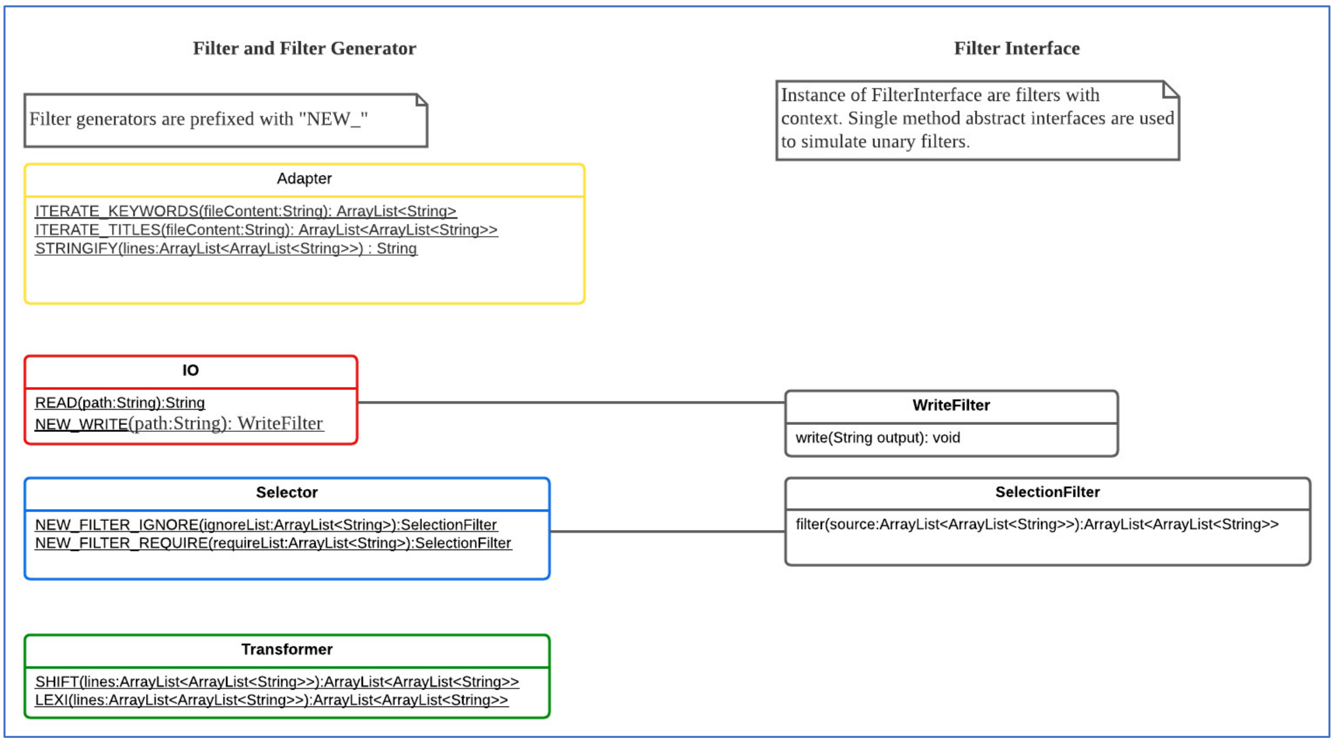
xi. Filters are categorized into four subgroups:

   i. Adapters – Each filter has a typed parameter. Adapters are placed in between filters to support type compatibility of previous filter's return value for the next filter to receive.

   ii. Transformers – Data inputs are manipulated with a defined computational procedure.

   iii. Selectors – Performs set operations and filter a set based on some predicate. Selectors does not evaluate and change data values.

   iv. IO – Read and write with file path as target.


Filter and Filter Generator
Filter Interface
Filter generators are prefixed with "NEW_"
Instance of FilterInterface are filters with context. Single method abstract interfaces are used to simulate unary filters.
Adapter
ITERATE_KEYWORDS(fileContent:String): ArrayList<String>
ITERATE_TITLES(fileContent:String): ArrayList<ArrayList<String>>
STRINGIFY(lines:ArrayList<ArrayList<String>>) : String
IO
READ(path:String):String
NEW_WRITE(path:String): WriteFilter
WriteFilter
write(String output): void
Selector
NEW_FILTER_IGNORE(ignoreList:ArrayList<String>):SelectionFilter
NEW_FILTER_REQUIRE(requireList:ArrayList<String>):SelectionFilter
SelectionFilter
filter(source:ArrayList<ArrayList<String>>):ArrayList<ArrayList<String>>
Transformer
SHIFT(lines:ArrayList<ArrayList<String>>):ArrayList<ArrayList<String>>
LEXI(lines:ArrayList<ArrayList<String>>):ArrayList<ArrayList<String>>

xii. Filters may be contextual and derived from user input. These filters need to be prepared in advance and generated for the main routine prior to execution. Generation of filters should also follow pipe and filter design pattern, with the filter as an end product.

xiii. Like Assignment 1, main pipeline execution is written in a single line:

```
write(output_path,lexi(rearrange(parseFile(input_path)));   //assignment 1

pWriter.write(Adapter.STRINGIFY(fRequire.filter(fIgnore.filter(Transformer.LEXI((Transformer.SHI
FT(Adapter.ITERATE_TITLES(READ(pathTitle)))))))); //assignment 2
```

As it can be seen, both lines of code include a input, processing of the input and returning an output. The key difference here is that the code in assignment 2 contains more functions, which can be easily added into the pipe and filter design due to its modular design.

3. CI

3.1. Integration Test

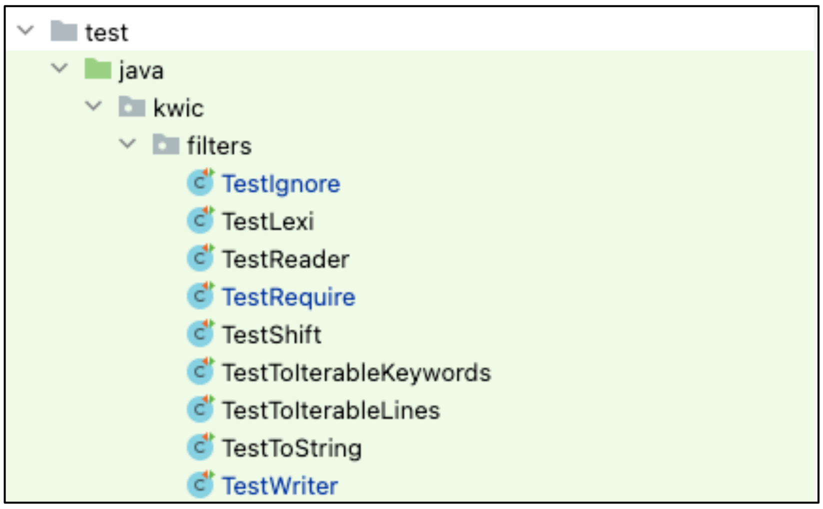
In development phase, we attempt to test often to verify and validate codebase with most recent change. We write an authoritative script (./run-integration-test.sh) that can meet final test requirements on any environment. It also ensures uniformity of build steps, scope of test coverage and package. The general flow for the script is as follows:

```
1. Cleans build directory to avoid using stale artefacts.
2. Runs source unit test.
3. Install dependencies.
4. Compile to bytecode.
5. Package into jar.
6. System test against jar.
```

3.2. Test Methodology

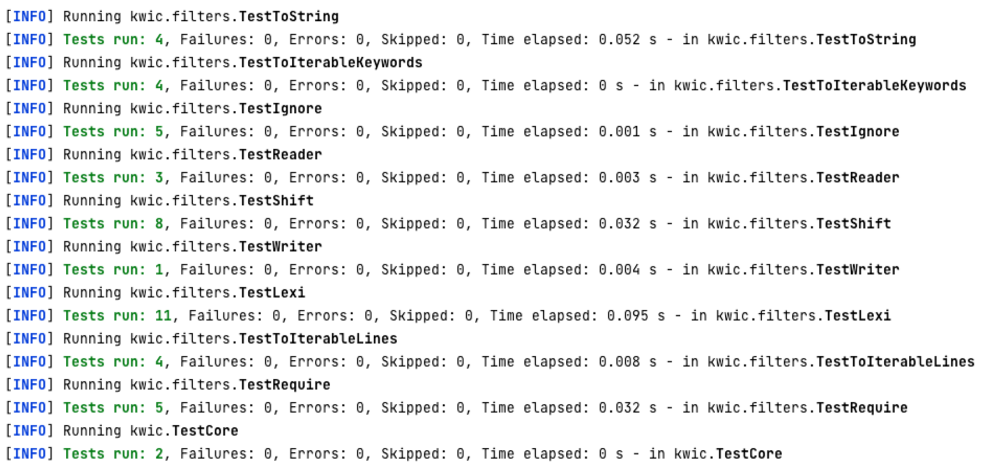
Test assertions is the primary method to reflect objectives of the codebase. Tests are categorized into two types – unit test and system test. For unit test, a test file is created for each filter to contain tests against possible inputs and expected outputs based on our understanding of the requirement. We will make simpler test cases work and incrementally progress to more complexity. System test will build the executable and run with test input file arguments. We then compare and check for discrepancy between actual output and expected output.

Unit Test Folder



```
test
  java
    kwic
      filters
        TestIgnore
        TestLexi
        TestReader
        TestRequire
        TestShift
        TestToIterableKeywords
        TestToIterableLines
        TestToString
        TestWriter
```

Unit Test Result from Integration Test Step 2



```
[INFO] Running kwic.filters.TestToString
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.052 s - in kwic.filters.TestToString
[INFO] Running kwic.filters.TestToIterableKeywords
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0 s - in kwic.filters.TestToIterableKeywords
[INFO] Running kwic.filters.TestIgnore
[INFO] Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.001 s - in kwic.filters.TestIgnore
[INFO] Running kwic.filters.TestReader
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.003 s - in kwic.filters.TestReader
[INFO] Running kwic.filters.TestShift
[INFO] Tests run: 8, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.032 s - in kwic.filters.TestShift
[INFO] Running kwic.filters.TestWriter
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.004 s - in kwic.filters.TestWriter
[INFO] Running kwic.filters.TestLexi
[INFO] Tests run: 11, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.095 s - in kwic.filters.TestLexi
[INFO] Running kwic.filters.TestToIterableLines
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.008 s - in kwic.filters.TestToIterableLines
[INFO] Running kwic.filters.TestRequire
[INFO] Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.032 s - in kwic.filters.TestRequire
[INFO] Running kwic.TestCore
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0 s - in kwic.TestCore
```

System Test Code Sample

```
java -jar NoelLim/KWIC.jar "$./sample_test_cases/Test1/Titles1.txt"
"./sample_test_cases/Test1/Ignored1.txt" "./sample_test_cases/Test1/Required1.txt"
cmp --print-chars "./sample_test_cases/Test1/Titles1/Output1.txt"
"./sample_test_cases/Test1/Titles1-output.txt"
```

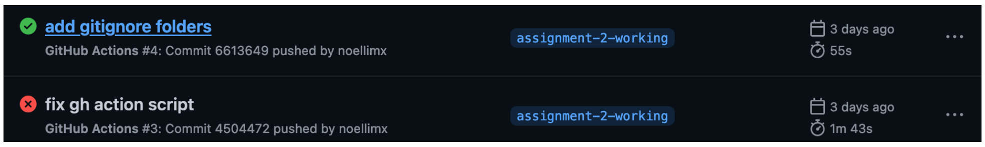
3.3. Test Strategy

We maximize test frequency to detect error early. The events which will trigger integration test is as follows:

i. Just-In-Time – Integration test can be manually carried out by executing the authoritative script.

ii. Pre-commit (Local) – We created a git-hook to include automate integration test run. Test failure will abort the commit and rectification must be made proactively. This will help to keep the codebase healthy.

iii. Github push/merge (Remote) – We add the integration test in Github Action. Github will perform the test remotely and the logs are readily available for viewing.



✓ add gitignore folders
GitHub Actions #4: Commit 6613649 pushed by noellimx
assignment-2-working
3 days ago
55s

✗ fix gh action script
GitHub Actions #3: Commit 4504472 pushed by noellimx
assignment-2-working
3 days ago
1m 43s

4. Contributions

Both members contributed equally within this assignment.