Assignment 2B

Matthew Visciglio

Student Identifier: 110374151
Email: Vismp001@mymail.unisa.edu.au

10/03/2023

University of
South Australia

UO System Design and Realisation SP1 2023 Assessment 2A

INFS2048 Assignment 2A Case Study: Facilities Monitoring System

Version	Date	Notes
V1.0.0	16/03/23	Final

Class Diagram deviations

The UML class diagram was modified to meet the expectations of the software standards. These changes were carefully chosen, and though thoroughly discussed, only slightly deviate from the previous proposed implementation, as it is assumed that the design process is iterative requires corrections. The updated Class diagram can be examined in the next following pages.

The first modification to the UML class diagram was regarding the orchestrating *WordStatsManager* class. Advice from assessment 2A, lead me to the realisation that an aggregation relationship would be appropriate to show the connection to the services.

One key change made to the UML class diagram was to have the DescriptorInfo dataclass returned by *getDescriptors()* from the *Summary* class*, instead of directly returning the class from addDescriptor()*. It was important to remove the return from *addDescriptor()* as maintaining this behaviour introduces unnecessary redundancy by providing two methods to do the same thing. Furthermore, code consistency was kept in mind, to prevent *addDescriptor()* from simultaneously becoming a setter and a getter method.

An essential behaviour in the summary class was also considered in order to detect duplicate descriptors. It was important to catch this exception in order to prevent it from interfering the *Formatting* component, which would have caused unstructured output. With the intention of detecting duplication, the *descriptors* attribute of *Summary* was made private to avoid accidental dependencies, while the *descriptorName* attribute was changed from private to public. Additionally, the check verifies that there are no *Descriptor* instances with the same *descriptorName* attribute within the descriptors list.

Other considered alternatives included

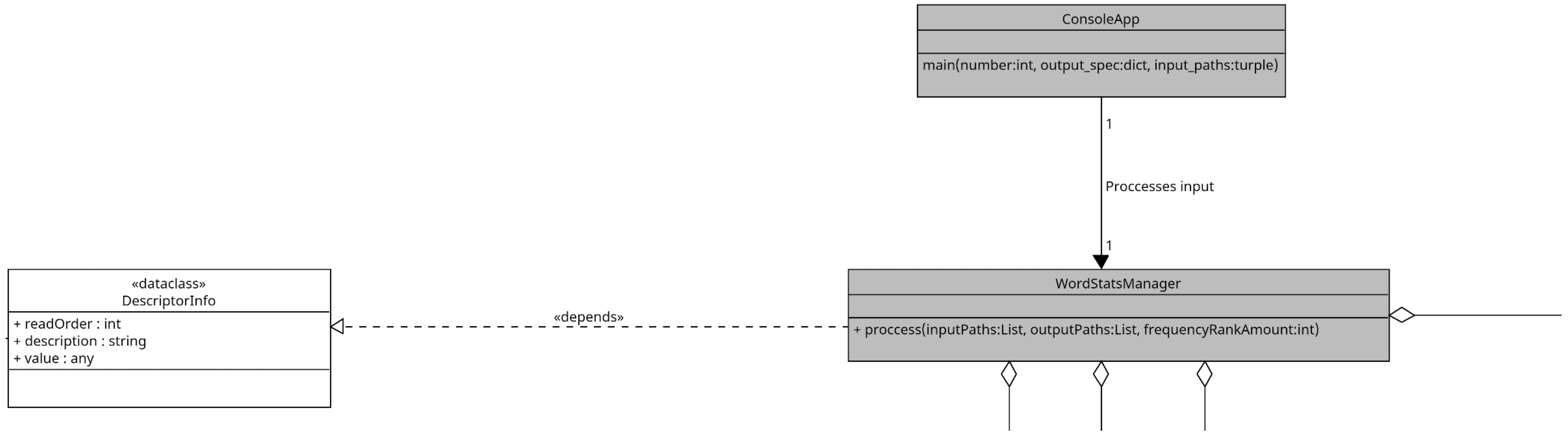
- Providing the interface with a getter/setter method
- Check if the DescriptorInfo instance was equal

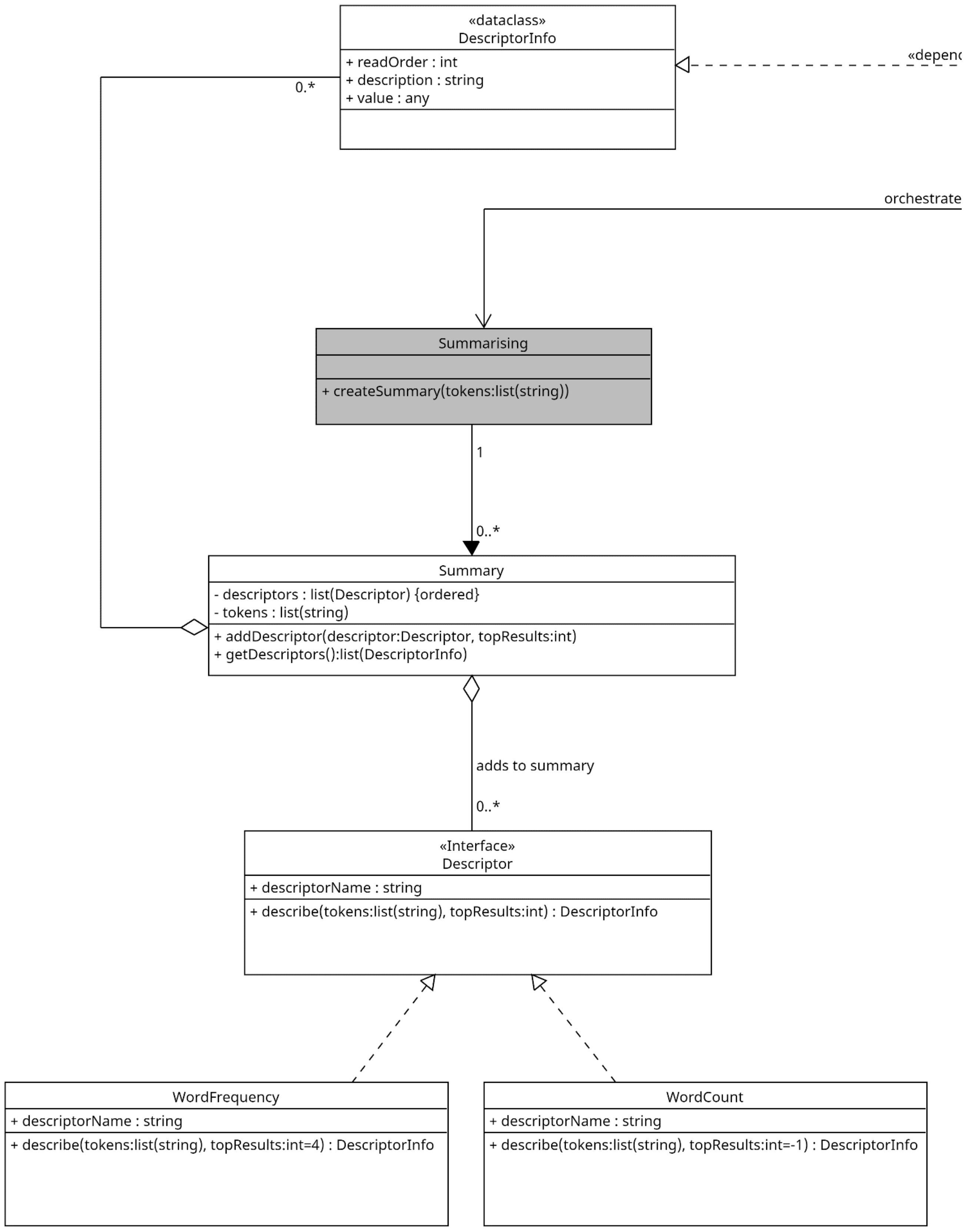
However, many of these solutions would introduce pitfalls, and/or vulnerabilities, in addition to code smell. These alternatives were ultimately put aside, and the discussed solution was implemented to maintain the existing abstractions.

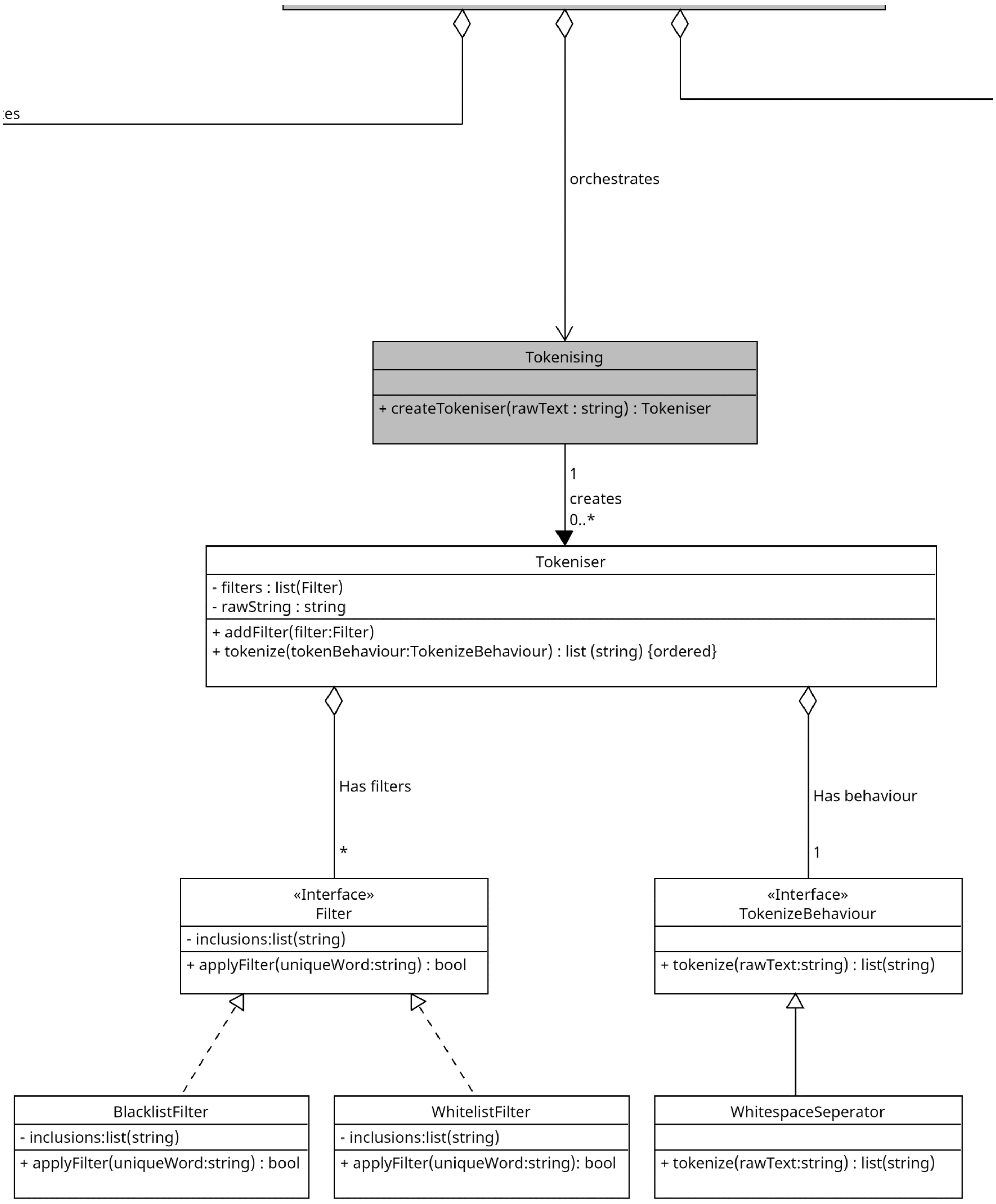
Smaller changes

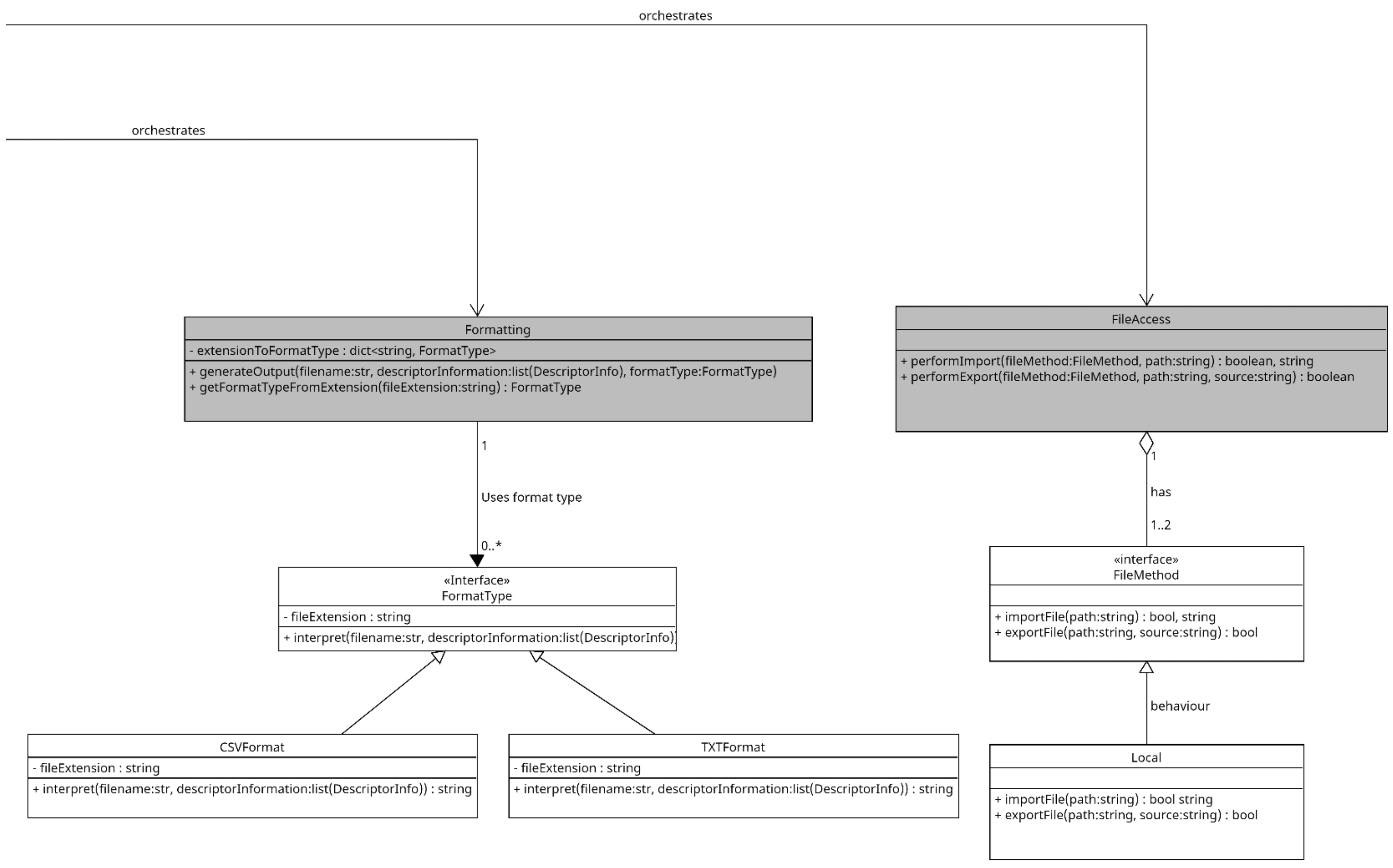
- Added filename argument to FormatType superclass, as the filename is used in the formatting examples for both TXT and CSV, this had not been considered in the original class diagram but was a small change.
- The private attribute Tokens were passed as a parameter into summary, but it was not included in the UML class diagram, this has now been rectified.

UML Class diagram (Updated)




«dataclass»
DescriptorInfo
+ readOrder : int
+ description : string
+ value : any
«depenc
0.*
orchestrate
Summarising
+ createSummary(tokens:list(string))
1
0..*
Summary
- descriptors : list(Descriptor) {ordered}
- tokens : list(string)
+ addDescriptor(descriptor:Descriptor, topResults:int)
+ getDescriptors():list(DescriptorInfo)
adds to summary
0..*
«Interface»
Descriptor
+ descriptorName : string
+ describe(tokens:list(string), topResults:int) : DescriptorInfo
WordFrequency
+ descriptorName : string
+ describe(tokens:list(string), topResults:int=4) : DescriptorInfo
WordCount
+ descriptorName : string
+ describe(tokens:list(string), topResults:int=-1) : DescriptorInfo


es
orchestrates
Tokenising
+ createTokeniser(rawText : string) : Tokeniser
1
creates
0..*
Tokeniser
- filters : list(Filter)
- rawString : string
+ addFilter(filter:Filter)
+ tokenize(tokenBehaviour:TokenizeBehaviour) : list (string) {ordered}
Has filters
*
Has behaviour
1
«Interface»
Filter
- inclusions:list(string)
+ applyFilter(uniqueWord:string) : bool
«Interface»
TokenizeBehaviour
+ tokenize(rawText:string) : list(string)
BlacklistFilter
- inclusions:list(string)
+ applyFilter(uniqueWord:string) : bool
WhitelistFilter
- inclusions:list(string)
+ applyFilter(uniqueWord:string): bool
WhitespaceSeperator
+ tokenize(rawText:string) : list(string)


orchestrates
orchestrates
Formatting
- extensionToFormatType : dict<string, FormatType>
+ generateOutput(filename:str, descriptorInformation:list(DescriptorInfo), formatType:FormatType)
+ getFormatTypeFromExtension(fileExtension:string) : FormatType
1
Uses format type
0..*
«Interface»
FormatType
- fileExtension : string
+ interpret(filename:str, descriptorInformation:list(DescriptorInfo))
CSVFormat
- fileExtension : string
+ interpret(filename:str, descriptorInformation:list(DescriptorInfo)) : string
TXTFormat
- fileExtension : string
+ interpret(filename:str, descriptorInformation:list(DescriptorInfo)) : string
FileAccess
+ performImport(fileMethod:FileMethod, path:string) : boolean, string
+ performExport(fileMethod:FileMethod, path:string, source:string) : boolean
1
has
1..2
«interface»
FileMethod
+ importFile(path:string) : bool, string
+ exportFile(path:string, source:string) : bool
behaviour
Local
+ importFile(path:string) : bool string
+ exportFile(path:string, source:string) : bool

Approach to implementation

The approach to implementation was calculated, while complying with common design patterns and principles, resulting in easy-to-read code.

Services

Overall, there were four towers, each of them was a service, managed by *WordStatsManager*. Dependency injection was managed in the *__main__* entry to the application, and only necessary services were provided. A demonstration of dependency injection was included in the application code, in which services and the manager are passed as an argument into the constructors for the classes.

```python
# Main entry point to the application to intansiate services
if __name__ == '__main__':
    __summarisingAccess:Summarising = Summarising()
    __tokenisingAccess = Tokenising()
    __formattingAccess = Formatting()
    __fileAccess = FileAccess()
    __wordStatsManagerAccess = WordStatsManager(__summarisingAccess, __tokenisingAccess, __formattingAccess, __fileAccess)
    __consoleApp:ConsoleApp = ConsoleApp(__wordStatsManagerAccess)

    main()
```

Furthermore, dependency injection guidelines are followed, by assigning the services to a private variable in the class. This is reflected in the UML class diagram as well.

```python
class WordStatsManager():
    def __init__(self, summarisingAccess, tokenisingAccess, formattingAccess, fileAccess:FileAccess):
        self.__summarising:Summarising = summarisingAccess
        self.__tokenising:Tokenising = tokenisingAccess
        self.__formatting:Formatting = formattingAccess
        self.__fileAcess:FileAccess = fileAccess
```

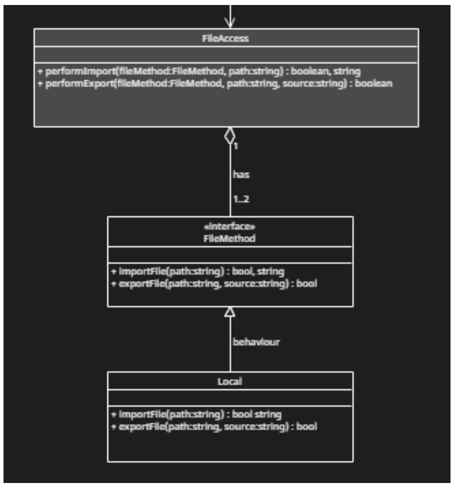
Dependency Inversion principle & strategy pattern

Dependency inversion was used much of the time, to rely on abstractions, rather than lower-level classes. Such examples include the following interfaces, which can be identified with the label <<interface>> in the UML class diagram:

Descriptor, *Filter*, *TokenizeBehaviour*, *FormatType*, and most evidently in *FileAccess*.

My general approach was to use the strategy pattern where applicable to closely follow HAS-A relationship, rather than IS-A. The strategy pattern is important for keeping behaviour differences abstracted to an interface within a HAS-A behaviour relationship.

In the *FileAccess* component, the abstraction relied upon is the *FileMethod* class. *FileAccess*, uses the *FileMethod* interface to comply with local import/export behaviour, allowing for future behavioural additions and upgrades. The *FileMehod* abstraction is defined and inherits from the ABC class in the provided code snippet.

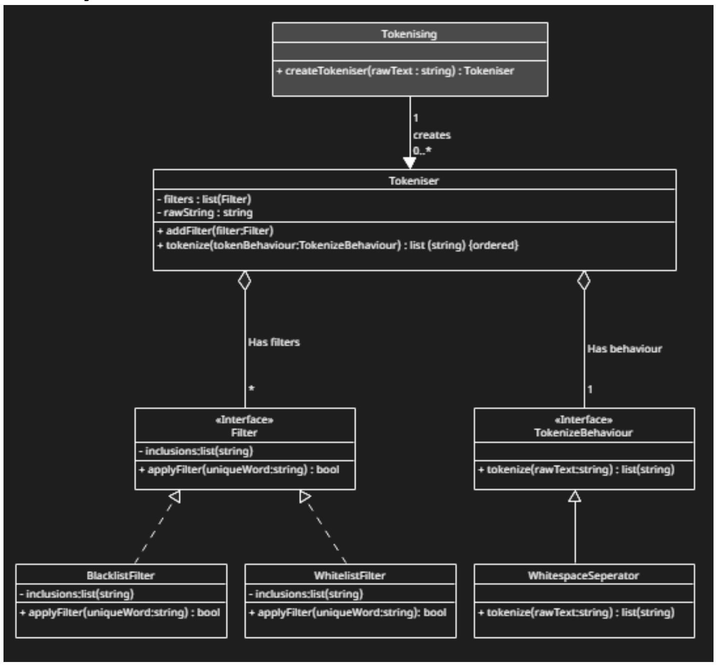

FileAccess
+ performImport(fileMethod:FileMethod, path:string) : boolean, string
+ performExport(fileMethod:FileMethod, path:string, source:string) : boolean
1
has
1..2
«interface»
FileMethod
+ importFile(path:string) : bool, string
+ exportFile(path:string, source:string) : bool
behaviour
Local
+ importFile(path:string) : bool string
+ exportFile(path:string, source:string) : bool

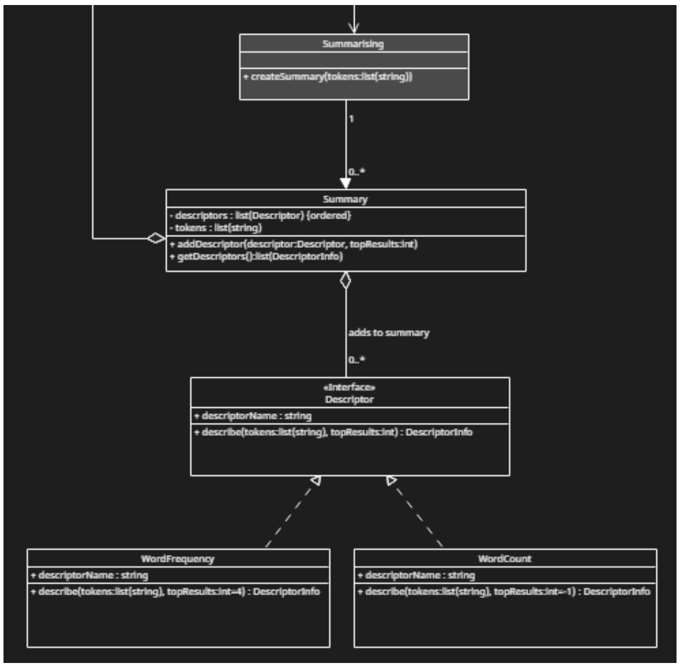
```python
# Encapsulated in FileAccess component
class FileMethod(ABC):
    '''Specifies import and export methods for a filetype'''
    def __init__(self):
        pass

    @abstractclassmethod
    def importFile(self, path:str):
        '''Import file to analyze'''
        pass

    @abstractclassmethod
    def exportFile(self, path:str, source:str):
        '''Export analysis file'''
        pass
```

Factory Method Patten


Tokenising
+ createTokeniser(rawText : string) : Tokeniser
1
creates
0..*
Tokeniser
- filters : list(Filter)
- rawString : string
+ addFilter(filter:Filter)
+ tokenize(tokenBehaviour:TokenizeBehaviour) : list (string) {ordered}
Has filters
*
Has behaviour
1
«Interface»
Filter
- inclusions:list(string)
+ applyFilter(uniqueWord:string) : bool
«Interface»
TokenizeBehaviour
+ tokenize(rawText:string) : list(string)
BlacklistFilter
- inclusions:list(string)
+ applyFilter(uniqueWord:string) : bool
WhitelistFilter
- inclusions:list(string)
+ applyFilter(uniqueWord:string): bool
WhitespaceSeperator
+ tokenize(rawText:string) : list(string)


Summarising
+ createSummary(tokens:list(string))
1
0..*
Summary
- descriptors : list(Descriptor) {ordered}
- tokens : list(string)
+ addDescriptor(descriptor:Descriptor, topResults:int)
+ getDescriptors():list(DescriptorInfo)
adds to summary
0..*
«Interface»
Descriptor
+ descriptorName : string
+ describe(tokens:list(string), topResults:int) : DescriptorInfo
WordFrequency
+ descriptorName : string
+ describe(tokens:list(string), topResults:int=4) : DescriptorInfo
WordCount
+ descriptorName : string
+ describe(tokens:list(string), topResults:int=-1) : DescriptorInfo

The factory method design pattern was also implemented into the codebase, as it is a versatile way to ensure future code upgradability by software engineers. Each factory incorporates the create blank, naming convention, which avoids confusing others who are inspecting or modifying the codebase. Given that the top-level classes do not keep a record of created classes they are only handled by association via employing the factory method. It is important to note, that the class one layer down lower, forms an aggregation relationship between itself and its abstract interfaces, but is not considered a factory.

```python
class Tokenising():
    '''Responsible for creating new Tokenisers, which are used to split words into tokens'''
    def __init__(self):
        pass

    def createTokeniser(self, rawText:str) -> Tokeniser:
        '''Creates a Tokeniser, responsible for splitting text into tokens strings that are necessary in calculating statistics'''
        tokeniser:Tokeniser = Tokeniser(rawText)
        return tokeniser
```

In the code snippet, the Tokenising class is one example of the factory method pattern being utilized. Each tokenizer instance is initialized via its constructor with raw text, which will later be processed and then 'Tokenized'. In the context of the application, tokens are defined as the individual words being analysed, represented by the string datatype.

By implementing the factory method, if the developers of the application decide to loop the program or update its capabilities, more than one *tokeniser* can be created, thereby supporting code-reuse.

Features & behaviours

Some features such as filters were included in the final version, as they are optional and can't easily be extended. Behaviours bound to an interface allow developers to make run-time decisions on what they want to happen in their program.

Although filters were not required assessment, it proved the upgradability of the feature and proved that it was a viable interface to implement into the program. The filters included were the *BlacklistFilter* and *WhitelistFilter*, though they are never executed in the *WordStatsManager* process() method.

Naming Conventions

camelCase - For method names and attributes
test_PascalCase - Used exclusively for unit tests.
addInstanceType – Any method with add in front of it adds a new instance in a private list
createVariable – The method name in factories to create instances, following the Factory method pattern

Naming edge case (descriptorInfo versus descriptorInformation)

- The descriptorInfo variable refers to a single instance of DescriptorInfo
- The descriptorInformation variable refers to a list containing descriptorInfo

Why: descriptorInfos did not look or feel appropriate in the code's context, furthermore the name is too similar which would lead to confusion. The trailing label: Information, can refer to more than one informational piece, therefore it was decided upon.

Unit Tests (TDD)

Unit tests were separated into individual modules, categorised by the component that they belong to. My workflow for unit tests followed the "Test Driven approach" with the red, green and refactor stages. First, I wrote the test to initially fail, then in the green phase, I wrote the bare minimum to satisfy the test condition in the application code. Finally, in the refactor phase, I ensured that the code was manageable, and that the condition is satisfied. The refactor phase also included applying fixtures to parameters for often re-used setup functions and instances.

I ensured that by the end of development all the tests were passed. However, for any tests that could not be completed without modifying the external environment of the software, I did not include, as this could cause issues if a developer breaks the code, it could pose a risk to the user's system or their data. A solution would be to use mock objects; however, this was not within the scope of the assessment.

Major Assumptions

- Programmers are not likely to abuse descriptorName current state as a public variable in Descriptor.
    - This was a major point in the assignment, in which I had to decide the trade-off of using an interface, instead of a concrete class. It could be argued that developers may develop a dependency on the descriptorName attribute if they follow bad coding practice.
- Unit tests which involve external factors are likely to have a negative effect on the system, and should be avoided.
    - The FileAccess component and WordStatsManager were of concern during unit testing, as it is almost impossible to test the file import/export functionality without mock tests which were not in the scope of the assessment, however it is a tool that may be beneficial to use in the future.
- It was assumed PyDoc is only necessary in the baseclass for interfaces. The intent of the abstractions is the same as the sub-classes, and therefore it did not seem as if it was needed.
- Assumed that managing dependency injection from __main__ is acceptable, as it is not in the global name-space, and it is also not defining the services within the scope of a class, thereby avoiding tight coupling between components.
- Though I knew the program needed to be upgradable, the selected component features focused on, assumed which areas developers would want to expand capabilities in.
    - Open/closed principle was adhered to as closely as possible to make it easier to modify to software.