Kandidatnummer: 1100,
1081,
1040,
1001

PG3302

Software Design

28.11.2022

Mappeeksamen in gruppe (2-5 studenter)

Høyskolen Kristiania

Høst 2022

Introduction

How does the application work

TextAnalyzer is an application described by it’s own name.
The applications will help you analyze your text with technical details.
You can add your own text-files or use the pre-made one to runs some tests.
The main feature of this application is its use of multithreading.
As you select your file to be analyzed, you will also get the option to enter wished amount of threads to run (Recommended is 1 to 8). When the executed task is done you will get a report showing stats from the document and the option to either save or discard. When selected to save it will be stored in a local database using SQLite.
Later this report can be retrieve for inspection. If you decide to write your own text, you make it directly in the application with file name and content. After creation you can then analyze it if desired.

How to start the application

In the zip folder you will have a shortcut made to the executable file for the application or you can compile the project one time to make your own and get it following this URL from root. “TextAnalyzer/bin/Debug/net6.0/TextAnalyzer.exe”.

Development process

Idea phase

Everyone in the exam group had gathered for a meeting going over potential applications ideas we could develop for this exam. It was mainly two ideas we found worthy and interesting enough for the exam:

- A text analyzer application focusing on multithreading.
- A rouge-like game with auto generated dungeons.

After a session of voting, both candidates had two vote each, we had decided what application to do with a coin flip and the winner came out to be the text analyzer.

Design phase

We started to discuss how the structural design and how the user interface would be. As seen soon we made two diagrams for explaining how the application would work on a technical level ((Sandnes, n.d., 26)Class diagrams) and user experience and data flow ((Sandnes, n.d., 9)sequence diagrams). Designing this over a course of a day using tools like plantUML on the computer and whiteboard for drawing quick ideas we landed on these lists:

Core features

- Single and multithreaded analysis.
- Read from existing files.
- Storage with SQLite database.
- Save and retrieve rapports from database.

Potential added features

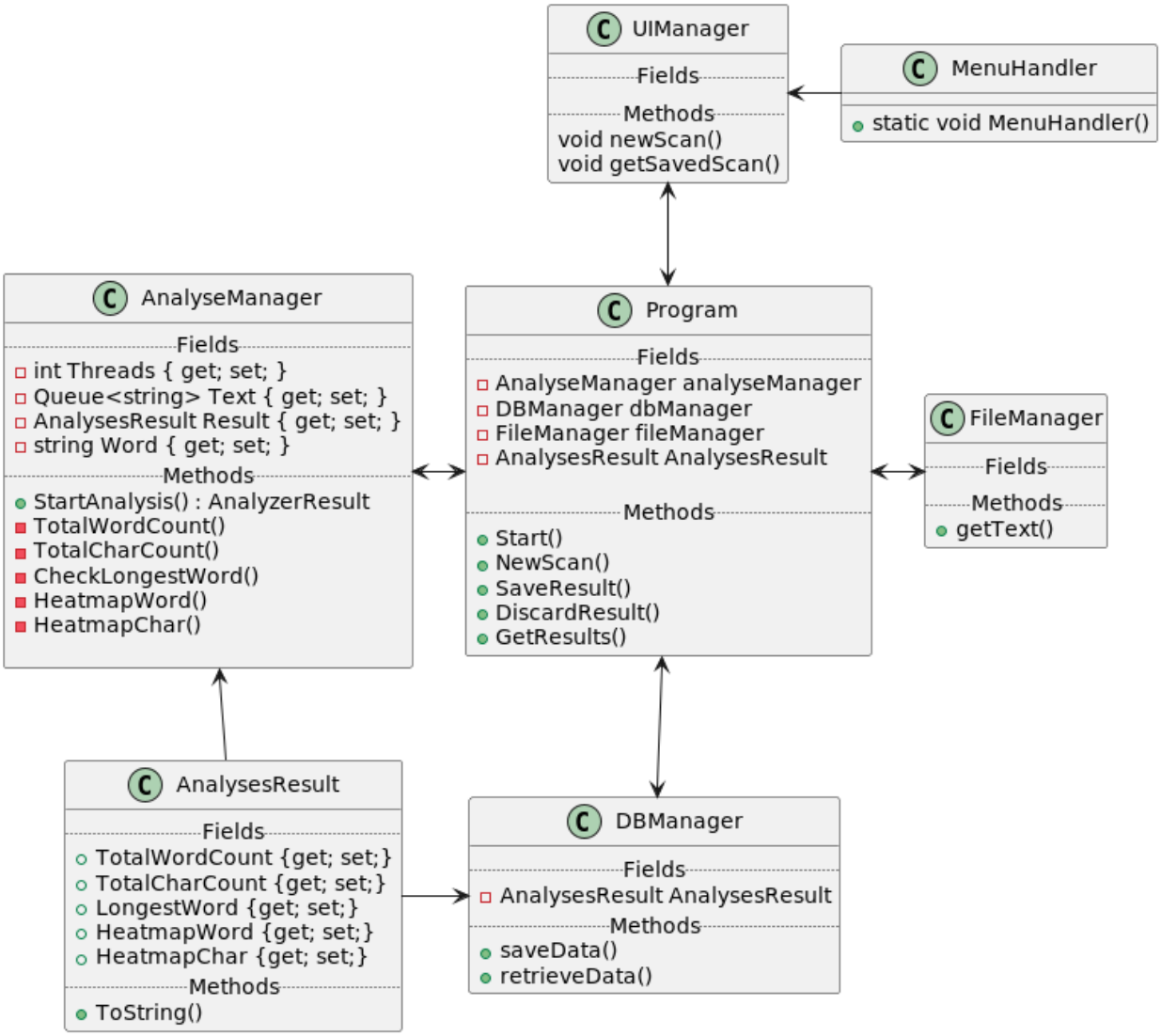
- Create your own files with content.
- Logger that stores log to a file.

Diagrams

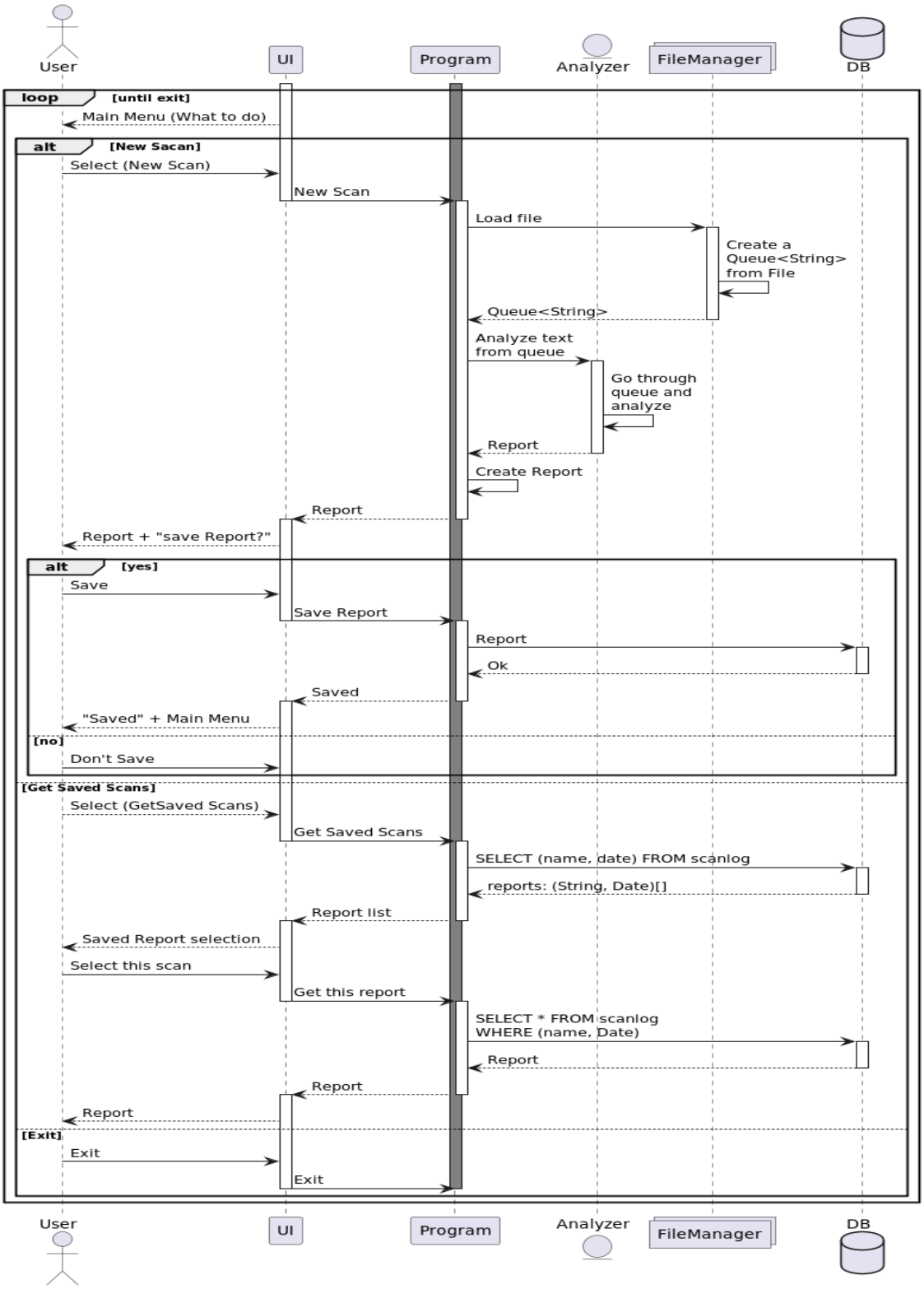
As things gets developed the original plannings would soon become outdated and experience with certain technology made us need more than planned. One thing the group is feeling proud about is the close similarity from the first planned diagrams compared to the end result.

Original diagrams

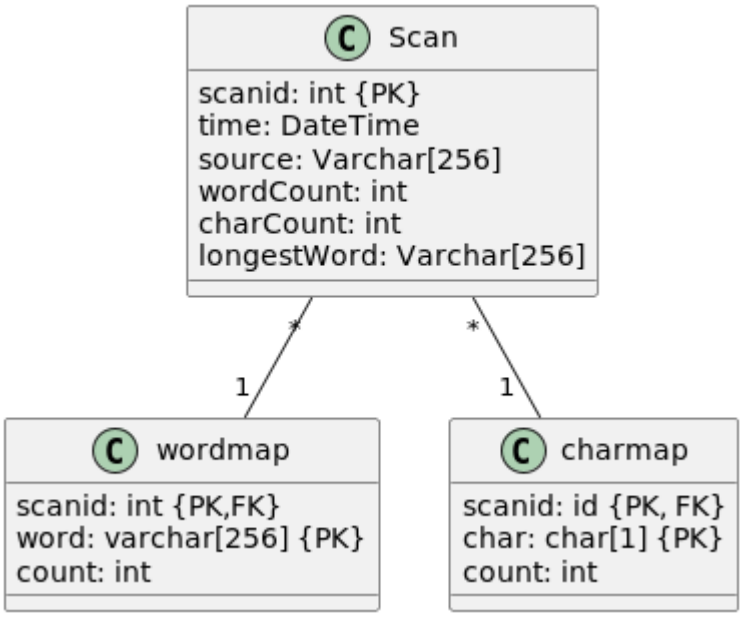
Class diagram



Sequence diagram

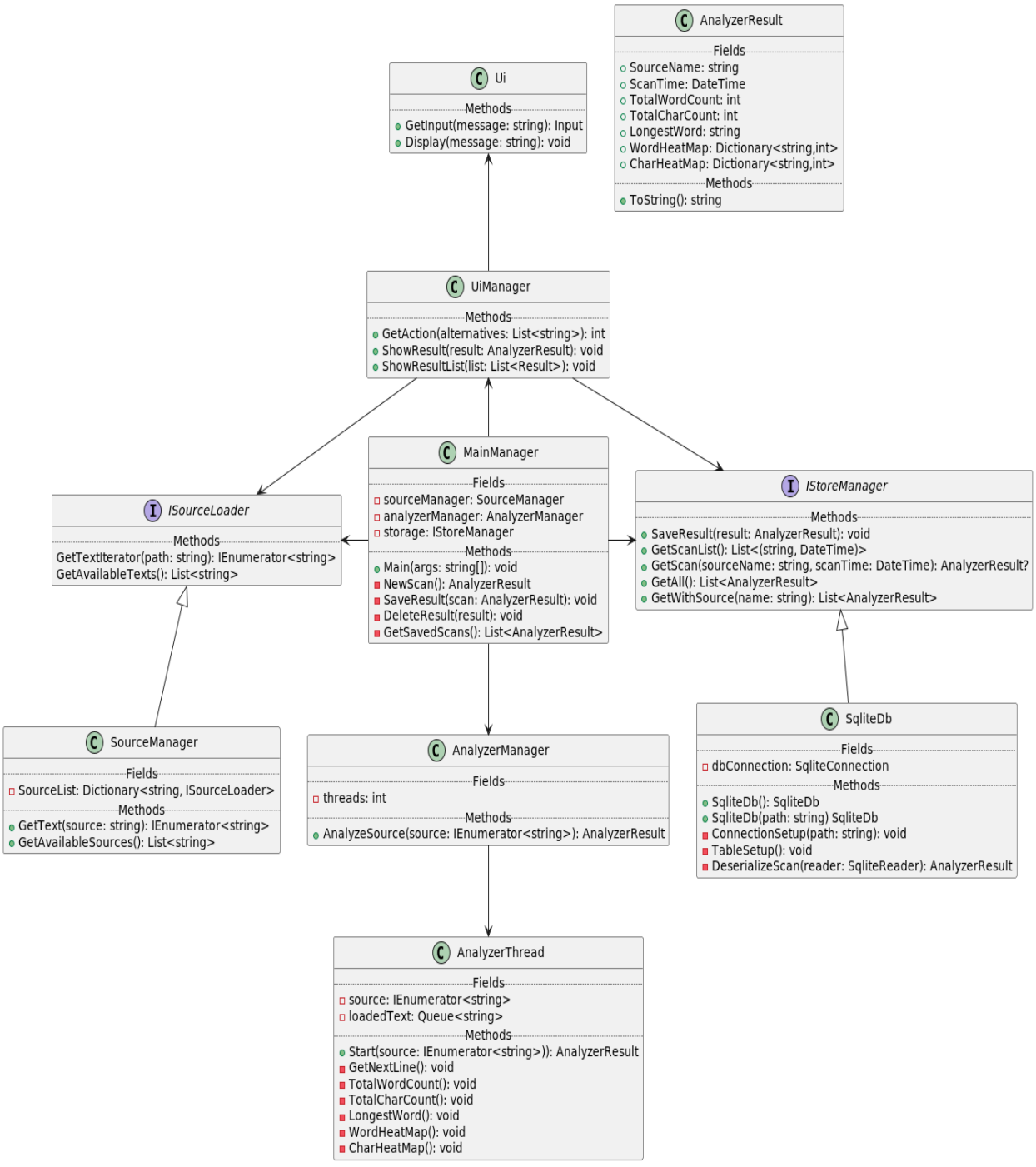


ER diagram

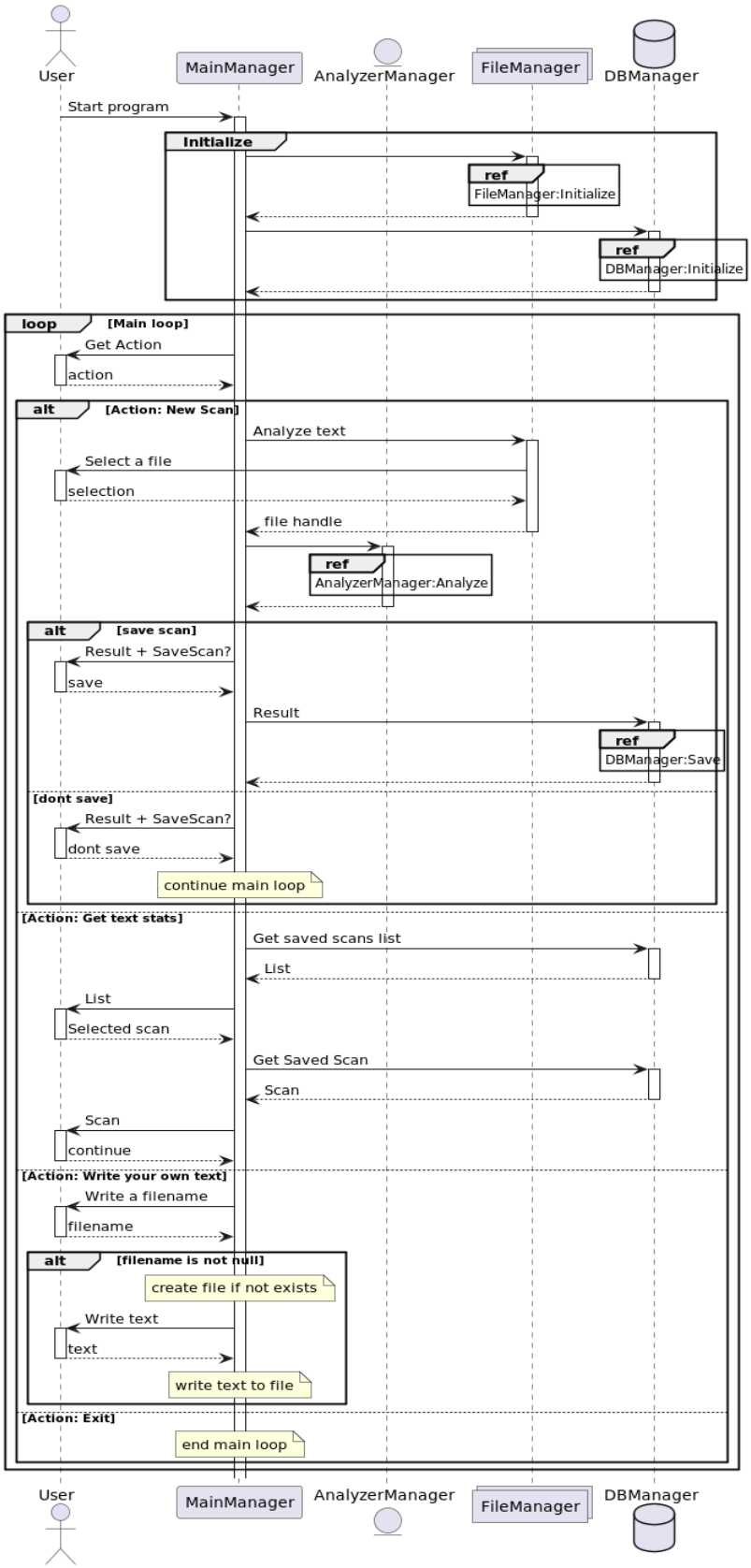

Scan
scanid: int {PK}
time: DateTime
source: Varchar[256]
wordCount: int
charCount: int
longestWord: Varchar[256]
*
1
*
1
wordmap
scanid: int {PK,FK}
word: varchar[256] {PK}
count: int
charmap
scanid: id {PK, FK}
char: char[1] {PK}
count: int

Finished project diagrams

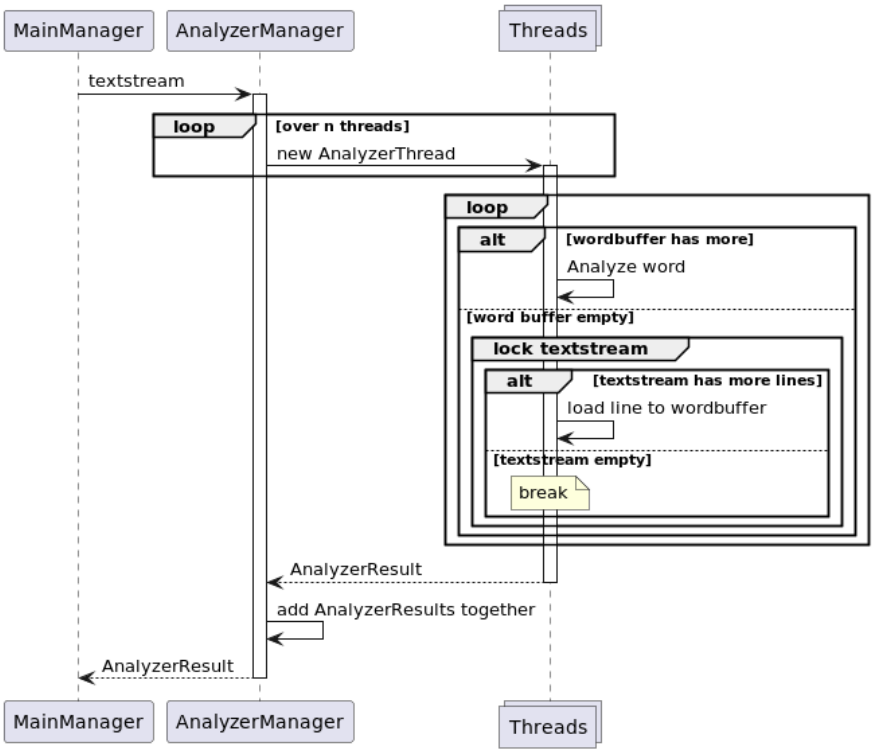
Class diagram



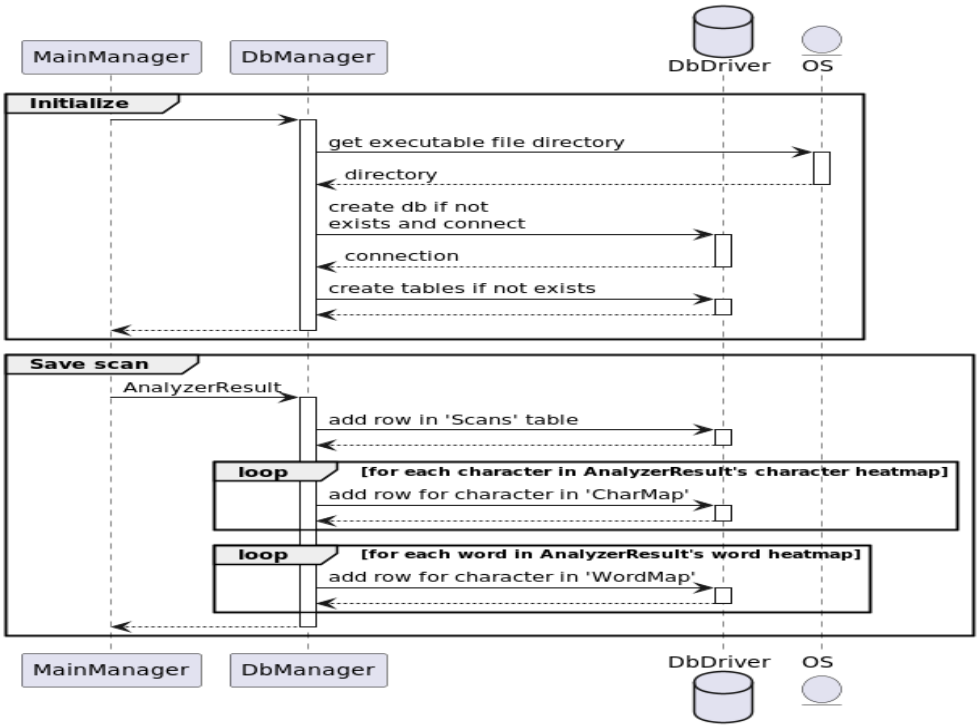
Sequence diagram



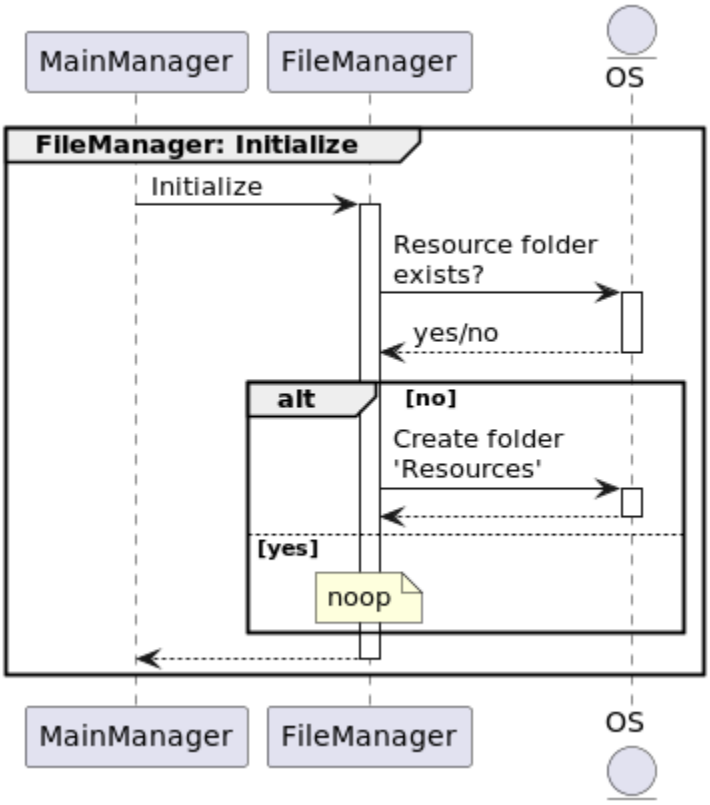
AnalyzerManager



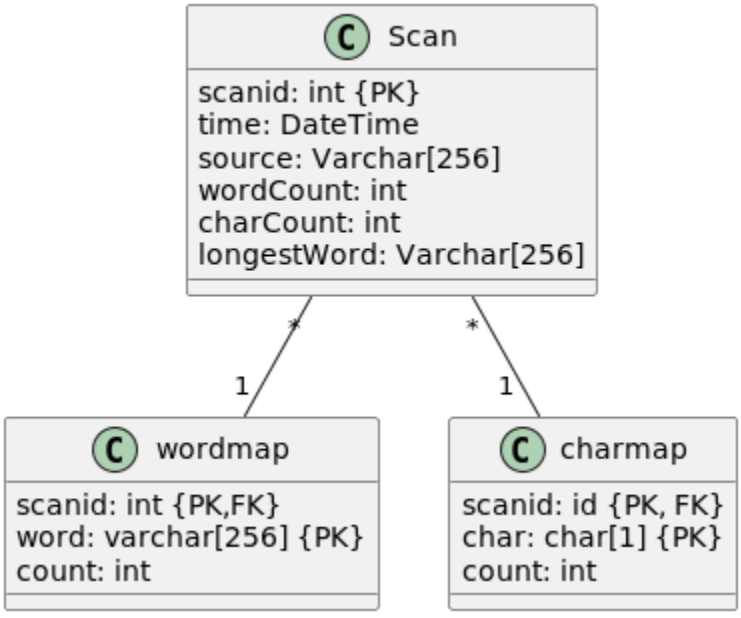
DBManager



FileManager



ER diagram



Syllabus

SOLID

“In software engineering, SOLID is a mnemonic acronym for five design principles intended to make object-oriented designs more understandable, flexible, and maintainable.”(“SOLID” 2022).

With this your take on SOLID is to lower complexity in the program, allowing for easier readability and future modification of the code. Writing the code with the intention that each class is responsible for only one function. Specialized classes are easier to read and modify as they do not affect other functions as severely when refactored.

Single-responsibility principle: Every entities in a program are responsible for one one role. That logic goes through all layers in a structure from the single method/function to a single module of a program.

This is the one principle from SOLID we have followed the most during development. The application are divided into four pieces doing one particular task. Example the Filereader manager handles everything with files, hence no other part of the application touches that aspect of the functionality.

Open-closed principle: "*software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification*" (“Open–Closed Principle” 2022). Meaning of this quotation could be summed up like this: Existing code can be extended with new functionality, without editing existing code.

A hard principle to follow, but is generally solved using interfaces and abstract classes for “contracts” with their children to obligate certain functionality. The place where this is implemented is in the interface class IdbManager where future databases could be configured within the application. Should it every need to be expanded new functionality in that area, writing the new features In the interface would make all children follow it, while protecting older code.

Liskov substitution principle: "*a principle in object-oriented programming stating that an object (such as a class) may be replaced by a sub-object (such as a class that extends the first class) without breaking the program.*" ("Liskov Substitution Principle" 2022).

This in general are followed and implemented by the modern languages like Java and C# by it self. Since modern OOP languages can handle subclasses replacing parent classes without effecting the execution of the program. Meaning this project have not focused on this principle.

Interface segregation principle: "*no code should be forced to depend on methods it does not use.*" ("Interface Segregation Principle" 2022). Interfaces that gets to big increase the chance of classes getting forced to implement code that they should not have. Meaning we need to split down interfaces when they include to much or/and do more than one thing.

During development we have only had the need to have one interface. It only implements the necessary database functionality to make the application run. Concluding with we have had eye on this principle, but not been a mayor focus.

Dependency inversion principle: "*a specific methodology for loosely coupling software modules.*" ("Dependency Inversion Principle" 2022). In traditional layer design lower level entities are consumed by higher level ones. This makes a hard dependency in the chain making it hard to alter it. With inversion pattern the high level entities consumes an interface the lower level implements. This breaks the direct dependency from from the lower levels and makes for more abstraction in the code.

Viewing on the project we have some potential to improve upon this principle. We could made interfaces for all the managers, making the application more open for modular based structure to allow for exchange between components. That would be the perfect way if the application where to follow the principle slavish. For the scale and practicality we only applied this logic to the database where it would be the most common place to change a module.

Multithreading

Multithreading, it is a way to process more data at once using multiple threads of execution. You can group them up to three places; CPU, operating system and application.

In this exam the relevant level is on application level. As mention earlier we made a text analysis program where the main bullet point is multithreading. How we implemented is very low level use of multithreading, but met many challenges on the way like race condition and C# specific syntax challenges, like how to initiate, wait and collect date for each thread.

Race condition was probably the easiest to fix using C# integrated statement named “lock” (BillWagner n.d.). Its function is to lock a certain object or variable and make it mutable only for the calling thread.

Initiation of threads was harder then first though. The way we solved that was to have a special Class that the thread could initiate from the main class. Using an array to keep track of the threads.

Making the program “pause” while the thread is running has a short solution but very interesting issue. If you don’t stop the main thread, the threads for analyses wont get enough time to work. Making the whole program slower then actually using it single-threaded.
Using the Thread.Join method it would pause the calling thread (the program in this case) til the running one is done with its operation.

For retrieving the results from the thread we needed to have a parallel array for outside reference. This way we could easily accessing the result from each thread and merge the result after the thread have closed without using advance return methods from the threads.

Testing

For testing many aspect of the program we used two main approaches for testing.
For the more technical aspect part as threading and analyses, file reading and database we used something called Unit-testing. There is some variation of this, where one is to write test first, then make it pass. Or opposite where one write the code then tests it.

The second part is called System testing or also very known as QA (quality assurance).
When it comes to our UI, it was tested solely on this approach. This way of testing is that a person is manually going through different parts of the program and checks the application executes task the intended way.

Implemented design patterns

Facade

"*Facade is a structural design pattern that provides a simplified interface to a library, a framework, or any other complex set of classes."* ("Facade" n.d.).
As a structural design pattern this achieves a somewhat a super-class for the user to sew together the needed functionalists from other modules of different kinds.

In the application there is one class named "MainManager.cs" that is facade. It calls all necessary public methods from Analyzer, File reader and Database folders. This achieves a one way for the UI to interact with the application and making for a good layering design.

Singleton

"*Singleton is a creational design pattern that lets you ensure that a class has only one instance, while providing a global access point to this instance."* ("Singleton" n.d.).
As a creational patterns that gives access to the same object wherever you are in the application.

This is probably an overused design pattern and very misused as well, but in the case for a logger this is perfect. In your application we added one extra feature, logging to file.
Since you only want one logger and make sure you don't log wrongly, a singleton class is good suited for this. The class "Logger.cs" handles different kind of log levels and saves it to a file while using the program.

DTO (Data Transfer Object)

"*In the field of programming a data transfer object is an object that carries data between processes.*" ("Data Transfer Object" 2021).

A design patter probably many programmer makes and uses without knowing is the DTO.

A class that is made solely for transporting data between different points within the application.

This one class is everywhere in our code; UI, Analysis, Facade and Database. Without this one class named "AnalyzerThread.cs" we would have experienced hard time sending around highly specialized data.

Version control – git and GitHub

Since everyone in the group have been learned git and GitHub, that became your natural version control of choice. First part of the project before we had only one branch for making the skeleton of a project. After the bare minimum infrastructure was in place and we had a fairly detailed class-diagram to work with. We branched out with different features and slowly merged it into the main branch. Before testing took place we got to set up GitHub-action that prevent any merge where unit-test does not pass.

Technical debt and refactoring

In the last group meeting when we talked about technical debt and refactoring.

The closest thing we feel goes under technical dept was a sudo-facade class that the UI used in the beginning. That took some time to later on the road to fix and integrate to the main facade. This was more done unintentionally but costed us some hours to correct and we categorize it as technical debt.

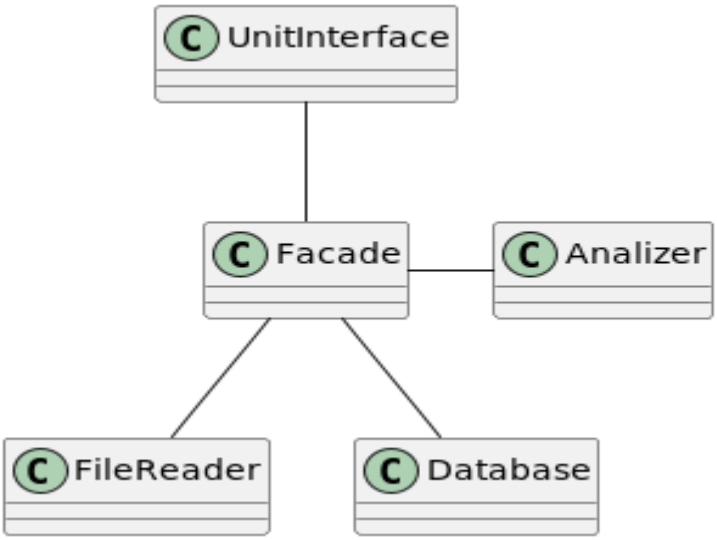
For refactoring we have been very consistent to use branches from GitHub which made us have little unnecessary refactoring. The main one goes back to the sudo-facade for the UI.

SQL injections

SQL injections are a huge thread if not taken account for. So to take actions against this we need to sterilize the parameter we want to use in our SQL queries.
In C# there is a built in sterilizing library named SqlCommand. Using this in our database while coding have made sure that we can safely put int parameter in the queries. Hence making the application safe for SQL injections.

Layering

UI: In this layer we are communication with the console command and have made a dedicated set of classes to work out this layer.
Business: We have the facade that connect the analyzer classes and to the other classes that bind us to the lowest layer.
Persistence: We have two group of classes here. File reader and Database. This bind the application to the database we want to save and retrieve reports and the actual text file we analyses.
Database: This layer is filled with SQLite as structural database and text-files as non static storage for analyzing files.



C# specific features and other details

During development we have actively been using the properties feature. Giving a much cleaner code with almost no setter and getter. In the DTO class we implemented operator overloading for easy merging from multiple threads. When it comes to best practice and naming convention we have followed the new dotnet6 for namespace ad fields. For fields we have had a focus on using public and private where it makes sense and named accordingly.

For pair programming we mostly been working together in group or alone. When we all have been together, the speed of needed information sharing is increased and help for explanation have given good result. During this project there have been no instance of standard pair programming learned from class but rather group programming.

Sources

BillWagner. n.d. “Lock Statement - C# Reference.” Accessed November 26, 2022. https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/lock.

“Data Transfer Object.” 2021. In *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Data_transfer_object&oldid=1015254260.

“Dependency Inversion Principle.” 2022. In *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Dependency_inversion_principle&oldid=1123295811.

“Facade.” n.d. Accessed November 26, 2022. https://refactoring.guru/design-patterns/facade.

“Interface Segregation Principle.” 2022. In *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Interface_segregation_principle&oldid=1076335422.

“Liskov Substitution Principle.” 2022. In *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Liskov_substitution_principle&oldid=1115990919.

“Open–Closed Principle.” 2022. In *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Open%E2%80%93closed_principle&oldid=1121006594.

Sandnes, Tomas. n.d. “PG3302_01_Introduction_UML-Class-Diagram.”

———. n.d. “PG3302_03_Layering_SeqDiag_Git_CSharp_TechDebt.”

“Singleton.” n.d. Accessed November 26, 2022. https://refactoring.guru/design-patterns/singleton.

“SOLID.” 2022. In *Wikipedia*. https://en.wikipedia.org/w/index.php?title=SOLID&oldid=1120488044.