SED505-Design Patterns

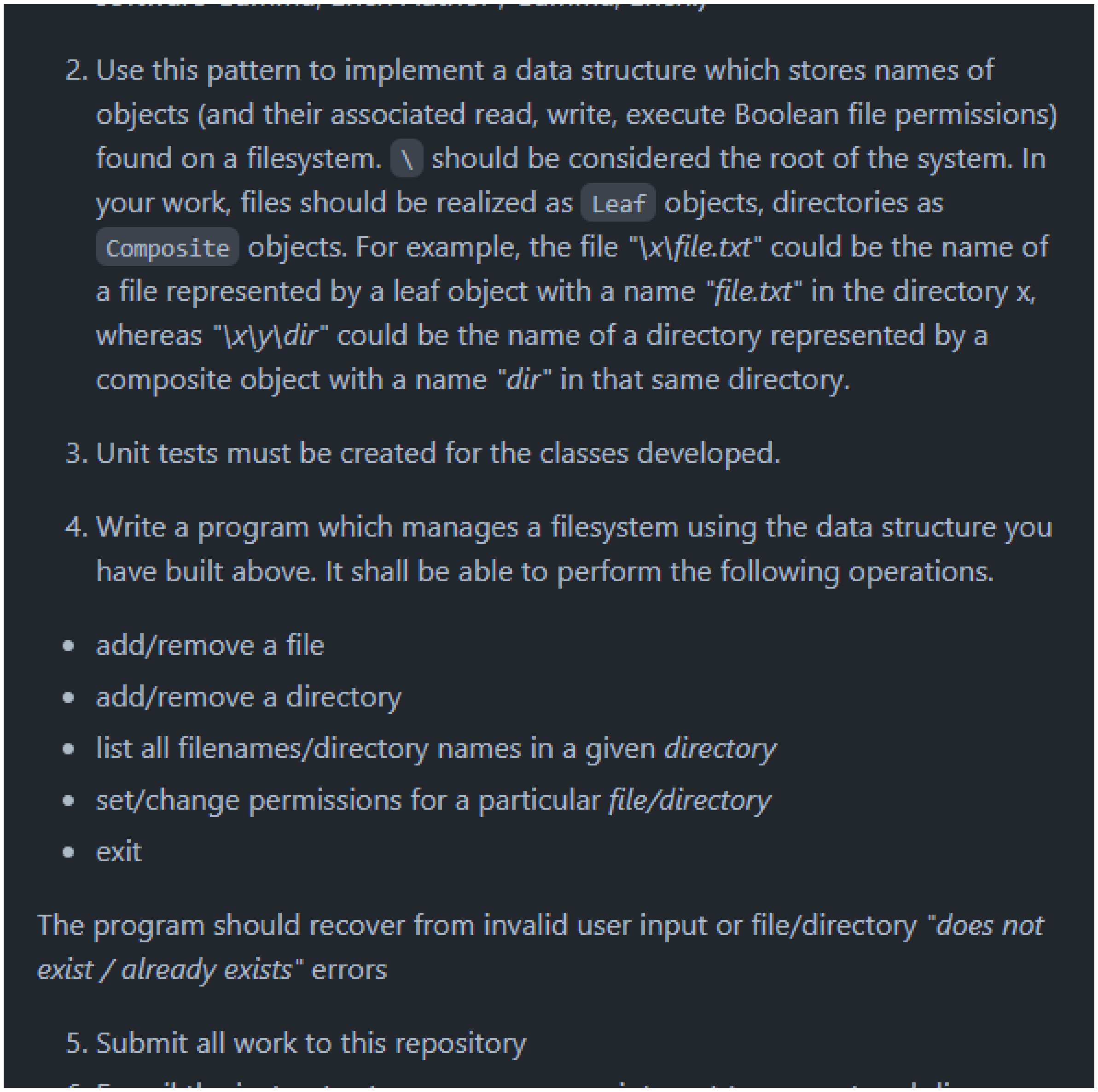
Assignment 2- Questions

(Question#1) What type of a design pattern (creational, structural, behavioural, custom) did you employ in this assignment? Why?

(Answer) We used structural design pattern, the adapter pattern, since it is used to allows incompatible interfaces of different classes to work together by creating an adapter class that bridges the gap between them it was perfect since we were trying to connect our legacy code (boat.java, car.java, airplane.java) to our contemporary code (TravelSimulator.cpp). By applying this method, we were able to use the existing code without modifying their source code or creating new subclasses.

(Question #2) Describe a software design pattern that you have used before this course, even if you were not aware of it.

(Answer) In System Development and Design we made a data structure which stores names of Objects. The question was as follows.



2. Use this pattern to implement a data structure which stores names of objects (and their associated read, write, execute Boolean file permissions) found on a filesystem. `\` should be considered the root of the system. In your work, files should be realized as `Leaf` objects, directories as `Composite` objects. For example, the file "*\x\file.txt*" could be the name of a file represented by a leaf object with a name "*file.txt*" in the directory x, whereas "*\x\y\dir*" could be the name of a directory represented by a composite object with a name "*dir*" in that same directory.

3. Unit tests must be created for the classes developed.

4. Write a program which manages a filesystem using the data structure you have built above. It shall be able to perform the following operations.

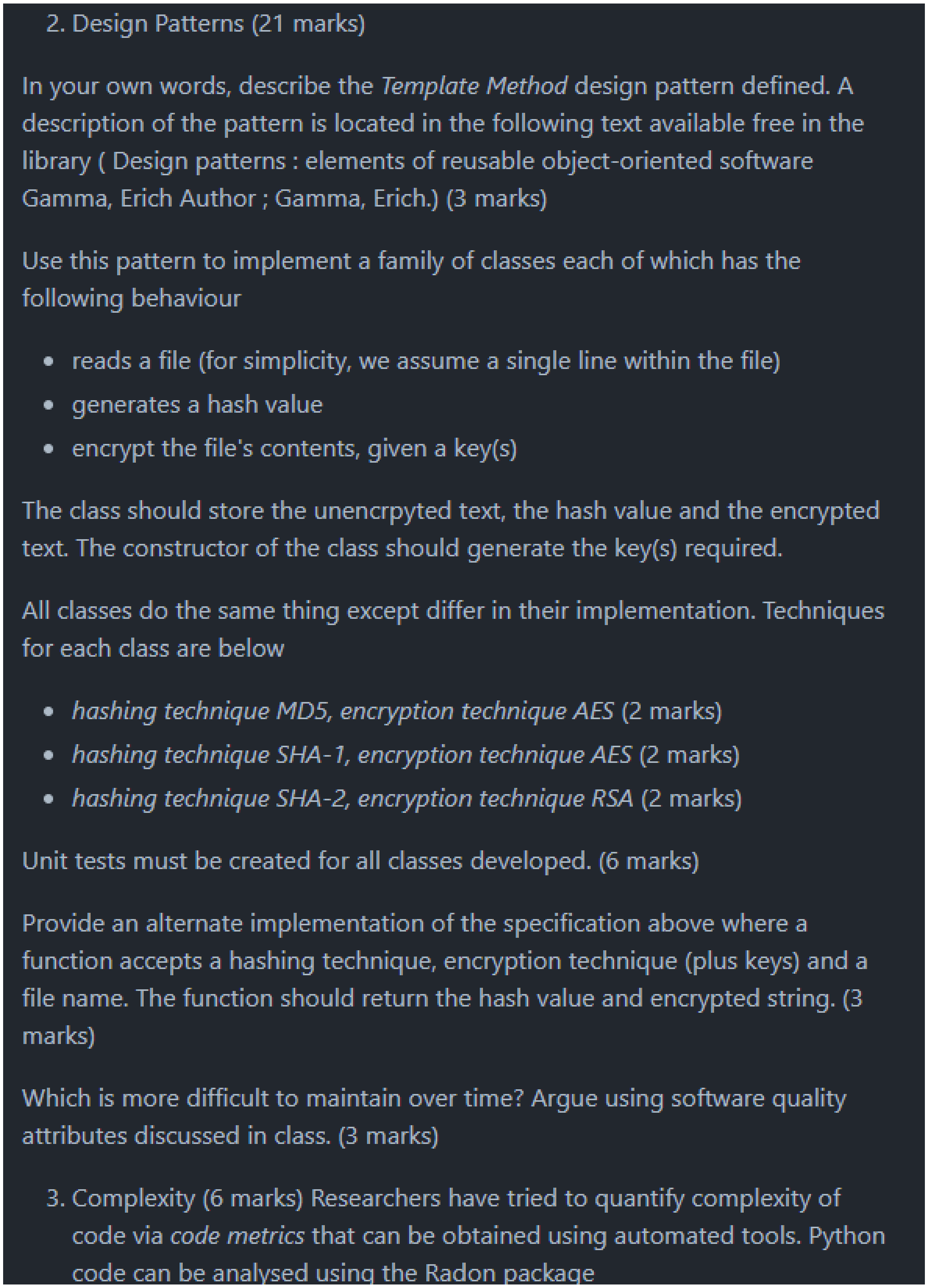
- add/remove a file
- add/remove a directory
- list all filenames/directory names in a given *directory*
- set/change permissions for a particular *file/directory*
- exit

The program should recover from invalid user input or file/directory "*does not exist / already exists*" errors

5. Submit all work to this repository

This was done by using composite design method.

Another design pattern we used in this course was the template design pattern. The question is as follows



2. Design Patterns (21 marks)

In your own words, describe the *Template Method* design pattern defined. A description of the pattern is located in the following text available free in the library (Design patterns : elements of reusable object-oriented software Gamma, Erich Author ; Gamma, Erich.) (3 marks)

Use this pattern to implement a family of classes each of which has the following behaviour

- reads a file (for simplicity, we assume a single line within the file)
- generates a hash value
- encrypt the file's contents, given a key(s)

The class should store the unencrpyted text, the hash value and the encrypted text. The constructor of the class should generate the key(s) required.

All classes do the same thing except differ in their implementation. Techniques for each class are below

- *hashing technique MD5, encryption technique AES* (2 marks)
- *hashing technique SHA-1, encryption technique AES* (2 marks)
- *hashing technique SHA-2, encryption technique RSA* (2 marks)

Unit tests must be created for all classes developed. (6 marks)

Provide an alternate implementation of the specification above where a function accepts a hashing technique, encryption technique (plus keys) and a file name. The function should return the hash value and encrypted string. (3 marks)

Which is more difficult to maintain over time? Argue using software quality attributes discussed in class. (3 marks)

3. Complexity (6 marks) Researchers have tried to quantify complexity of code via *code metrics* that can be obtained using automated tools. Python code can be analysed using the Radon package

(Questions #3) Do you feel standard design patterns are of great assistance, or great hindrance, in software design? Give advantages and disadvantages.

(Answer)

Design patterns, as conceptualized in software engineering, serve as reusable solutions to recurrent problems within specific contexts. Their utility, however, is contingent upon the nature of the software design challenge at hand.

Advantages of Standard Design Patterns:

1. Structured Approach: Design patterns provide a structured approach to problem-solving, ensuring that developers don't have to "reinvent the wheel." They offer a tried and tested methodology that has been refined over time.

2. Consistency and Readability: Implementing standard patterns can lead to more consistent code across projects. This consistency can make the codebase more readable and maintainable, especially for new team members who are familiar with the patterns.

3. SOLID Principles Adherence: Many design patterns inherently adhere to the SOLID principles, ensuring that the software design remains robust, scalable, and modular.

4. Efficiency: For complex problems, design patterns can expedite the development process. Instead of conceptualizing a solution from scratch, developers can leverage existing patterns, adapting them to the specific requirements of their project.

Disadvantages of Standard Design Patterns:

1. Overengineering: There's a risk of overengineering when design patterns are applied indiscriminately. Not every problem requires a pattern-based solution, and sometimes, a simple, straightforward approach is more effective.

2. Learning Curve: For novice developers, there can be a steep learning curve associated with understanding and implementing design patterns effectively.

3. Potential Mask for Inadequacy: Relying too heavily on design patterns might mask a developer's lack of in-depth coding knowledge or understanding of the underlying problem.

4. Rigidity: Strict adherence to a design pattern can sometimes introduce rigidity, making it challenging to adapt the software to evolving requirements.

In conclusion, while design patterns offer valuable blueprints for addressing common challenges, it's essential to approach them judiciously. Often, the most effective solutions are tailor-made, drawing inspiration from standard patterns but adapted to the unique nuances of the specific software being developed. Custom patterns, crafted with the specific software in mind, can offer a blend of standardization and flexibility, ensuring that the software is both robust and adaptable.

(Question #4) What are some major drawbacks of the Singleton design pattern?

(Answer)

The Singleton design pattern ensures that a particular class has only one instance throughout the lifecycle of an application and provides a global point of access to that instance. While it can be useful in certain scenarios, it also comes with several drawbacks:

1. Global State: Singleton often represents a global state, which can lead to hidden dependencies between classes. This can make the system harder to understand and modify.

2. Testing Challenges: Singletons can pose challenges for unit testing. Since they maintain state across invocations, tests can become interdependent, where the outcome of one test affects the outcome of others. Moreover, mocking Singletons can be difficult.

3. Scalability Issues: In multi-threaded environments, ensuring that a Singleton instance is created only once can require locking mechanisms. These locks can become a bottleneck, affecting performance.

4. Inflexibility: The Singleton pattern can make it challenging to extend the class in the future. For instance, if a need arises to have multiple configurations of the class, the Singleton pattern would need to be refactored.

5. Violation of Single Responsibility Principle: The Singleton pattern combines the responsibility of managing the instance creation and the actual business logic of the class, violating the Single Responsibility Principle.

6. Difficulties in Subclassing: Extending a Singleton class can be tricky. If the constructor is made private to ensure a single instance, the subclass might not be able to instantiate its parent class.

7. Lifetime Management: It's not straightforward to manage the lifecycle of a Singleton. Deciding when to destroy and reclaim the resources of a Singleton can be ambiguous, especially in languages without garbage collection.

8. Portability Issues: The use of Singletons can lead to portability issues, especially when moving from a single-threaded to a multi-threaded or distributed environment.

9. Code Smell: Over-reliance on Singletons can be considered a "code smell." It might indicate that the design is over-centralized or that modules are overly coupled.

10. Memory Usage: Since the Singleton instance is created and remains in memory for the application's entire lifecycle, it can lead to increased memory usage, especially if the Singleton holds large amounts of data.

In conclusion, while the Singleton pattern has its use cases, it's essential to be aware of its drawbacks and apply it judiciously. It's often beneficial to consider alternatives, such as dependency injection or service locators, which can offer similar benefits without many of the associated challenges.