1. **Discuss the metrics used to assess the features of an object-oriented program.**

Object-oriented metrics are measurements used to evaluate the complexity, quality, and efficacy of software engineering methods, people, and products. These metrics are utilised to assess developers' productivity and skill level and the design, coding, and testing of object-oriented software systems (itskawal2000, 2021).

When assessing an object-oriented program, quality, maintainability, efficiency, and adherence to best practices. There are established notions of software metrics, and many metrics are available to measure product quality. As object-oriented analysis and design approaches become more prominent, it is important to start exploring object-oriented metrics for software quality (Rosenberg & Hyatt,1997).

Numerous metrics have been proposed to assess aspects of object-oriented software, including size, inheritance, cohesion, and coupling (Dagpinar & Jahnke, 2003).

By evaluating the critical research chosen on the systematic mapping, Saraiva (2014) claimed that all metrics were grouped based on the internal characteristics of the software connected to six points: Size, Inheritance, Coupling, Cohesion, Complexity, and Software Architecture Constraints. Based on the metric description found in the preliminary study that indicated it, the measurements were then linked to an internal software property.

Turan & Tanriover (2018) stated that it is tough to test complex code, and it is also challenging to keep up. Developers must follow the boundary values of metrics when writing code to ensure it is well-written, comprehensible, and maintainable. For instance, the following are some typical metrics for judging object-oriented programmes:

- The Maintainability Index (MI) is a valuable indicator for assessing how simple it is to maintain a programme. It considers several variables, including complexity, coupling, cohesiveness, and code duplication, to produce a numerical score. This rating aids in determining the program's capacity for alterations, bug fixes, and future improvements.

- The Cyclomatic Complexity (CC) metric is used to assess the structural complexity of code. It is calculated by counting the code pathways that make up a program's flow. The number of controls affects the possible CC value flows based on different inputs. Code with complex control flows is more challenging and requires additional testing for appropriate code coverage. A probable code issue is indicated by high cyclomatic complexity.

- The "Depth of Inheritance" is the number of class definitions that reach the base of the class hierarchy. Tracing the purposes or redefinitions of specific methods and fields on a deeper scale could be more challenging. The complexity of the code may increase as the number of types in an inheritance hierarchy increase. On the other hand, a high inheritance depth may also indicate substantial code reuse. It can be challenging to decide what an ideal depth of inheritance is.

- Class Coupling is a programme design evaluation criterion, which analyses the relationship to various classes through parameters, local variables, return types, method calls, generic or template instantiations, base classes, interface implementations, fields declared on external styles and attribute decorating. High cohesion and low coupling for the types and methods are characteristics of well-designed software. High coupling denotes a complicated design that is difficult to reuse and maintain due to its extensive interdependencies on other types. A class will have zero class coupling if it does not reference other classes. The class coupling will be increased by referencing different courses in our class, such as developing complex-type properties.

Thanks to the SOLID design principles, we can construct flexible, reusable, and maintainable software. In the acronym SOLID, each letter stands for a distinct tenet (Abba, 2022). The acronym's notes are defined as follows:

- S: Single responsibility principle.
- O: Open–closed principle.
- L: Liskov substitution principle.
- I: Interface segregation principle.
- D: Dependency inversion principle.

According to Abba (2022), the program's compliance with core object-oriented design concepts such as encapsulation, inheritance, polymorphism, and SOLID principles is

evaluated by compliance with design principles. These rules can be broken, resulting in difficult-to-understand, difficult-to-maintain, and difficult-to-extend code.

As Forrest (2023) stated, metrics are essential in gaining valuable insights into object-oriented programs' quality, maintainability, and efficiency. It is crucial to approach the selection, creation, and utilisation of software metrics systematically and thoroughly as they provide various benefits, including:

- Early detection of potential issues in the program.
- Monitoring the program's progress and identifying areas for improvement.
- Comparing different programs and identifying any modifications made.
- Defending the program's development and maintenance costs.

2. Develop a Python program with three abstract methods and one subclass, allowing users to perform banking operations.

I have developed the following program based on the ideas of David-Williams (2023):

```
from abc import ABC, abstractmethod


class Banking(ABC):
    # An abstract class representing a banking operation

    @abstractmethod
    def deposit(self, amount: float) -> None:
        # Deposits an amount of money into the account
        pass

    @abstractmethod
    def withdraw(self, amount: float) -> None:
        # Withdraws an amount of money from the account
        pass

    @abstractmethod
    def transfer(self, amount: float, to_account: str) -> None:
        # Transfers an amount of money to another account
        pass


class BankAccount(Banking):
    """A class representing a bank account."""

    def __init__(self, name: str, balance: float):
        # Initializes a bank account
        self.name = name
        self.balance = balance
```

```
    def deposit(self, amount: float) -> None:
        # Deposits an amount of money into the account
        self.balance += amount

    def withdraw(self, amount: float) -> None:
        # Withdraws an amount of money from the account
        if amount > self.balance:
            raise ValueError("Insufficient funds")
        self.balance -= amount

    def transfer(self, amount: float, to_account: str) -> None:
        # Transfers an amount of money to another account
        self.withdraw(amount)
        to_account.deposit(amount)


if __name__ == "__main__":
    # Create a bank account
    account = BankAccount("John Doe", 1000.00)

    # Deposit money into the account
    account.deposit(500.00)

    # Withdraw money from the account
    account.withdraw(200.00)

    # Transfer money to another account
    other_account = BankAccount("Jane Doe", 0.00)
    account.transfer(300.00, other_account)

    # Print the balance of the account
    print(account.balance)
```

With this program, users can easily create a bank account and perform various banking operations like depositing, withdrawing, and transferring money. The Banking class is an abstract class which cannot be directly instantiated but instead must be subclassed. The BankAccount class is one such subclass that implements the abstract methods deposit(), withdraw(), and transfer(). The main() function is responsible for creating a bank account, performing banking operations, and displaying the account balance. This program serves as a great foundation for developing more intricate banking applications.

3. Read the article by Knox et al. (2018) and answer the following questions:

3.1. What is Component-based modelling?

Knox et al. (2018) claim that component-based modelling, in which processes within an integrated model are represented by pluggable model components, is the industry standard approach to model integration. IEMs, or integrated environmental models, are the names of these integrated models. Component-based models are frequently implemented as software systems, standards, or environmental modelling frameworks when building and integrating models around a single coherent structure. EMFs offer abstractions for specifying the input, output, and domains on which models are applied.

3.2. Upon what do component-based modelling frameworks depend?

According to Knox et al. (2018), component-based modelling frameworks depend on a constant underpinning structure into which all components can be integrated. The framework defines the input and output parameters in some methods to run each model.

3.3. Within the context of the work presented in this paper, what is Pynsim?

A Python-based object-oriented modelling framework called Pynsim aims to improve the style of earlier modelling frameworks. It seeks to make model integration, agent-based modelling, and using a 'component-based' process more accessible. It provides agent-based modelling by enabling each network element (node, link, or institution) to execute code individually at run time and adding a component-based model integration through 'pluggable' modules (Knox et al. 2018).

3.4. How does Pynsim achieve its goal when using object-oriented Python programming?

The aim of abstracting the complexity of the software infrastructure is a recurring theme in recent advancements in model frameworks, with the presumption that many model creators have limited software development and architecture skills. In contrast, scripting, primarily using Python, has become a frequent modelling feature (Knox et al. 2018).

References:

itskawal2000 (2021). *Object Oriented Metrices in Software Engineering*. [online] Available at: https://www.geeksforgeeks.org/object-oriented-metrices-in-software-engineering/.

Rosenberg, L.H. and Hyatt, L.E. (1997). Software quality metrics for object-oriented environments. *Crosstalk journal*, *10*(4), pp.1-6.

Dagpinar, M. & Jahnke, J.H. (2003). Predicting maintainability with object-oriented metrics-an empirical comparison. In *10th Working Conference on Reverse Engineering, 2003. WCRE 2003. Proceedings.* (pp. 155-155). IEEE Computer Society.

Saraiva, J. de A.G. (2014). *Classifying metrics for assessing object-oriented software maintainability: a family of metrics' catalogs*. [online] repositorio.ufpe.br. Available at: https://repositorio.ufpe.br/handle/123456789/12152 [Accessed 16 Jun. 2023].

Turan, O. & TANRIÖVER, Ö.Ö. (2018). An Experimental Evaluation of the Effect of SOLID Principles to Microsoft VS Code Metrics. *AJIT-e: Bilişim Teknolojileri Online Dergisi*, *9*(34), pp.7-24.

Abba, I. V. (2022). *SOLID Definition – the SOLID Principles of Object-Oriented Design Explained*. [online] Available at: https://www.freecodecamp.org/news/solid-principles-single-responsibility-principle-explained/.

Forrest, G. (2011). *The Importance of Implementing Effective Metrics*. [online] isixsigma.com. Available at: https://www.isixsigma.com/metrics-methodology/importance-implementing-effective-metrics/.

David-Williams, S. (2023). *SOLID principles in data engineering - Part 1*. [online] Available at: https://stephendavidwilliams.com/solid-principles-in-data-engineering-part-1 [Accessed 19 Jun. 2023].

Knox, S., Meier, P., Yoon, J. and Harou, J.J. (2018). A python framework for multi-agent simulation of networked resource systems. *Environmental Modelling & Software*, [online] 103, pp.16–28. doi:https://doi.org/10.1016/j.envsoft.2018.01.019.