B.M.S COLLEGE OF ENGINEERING

(An Autonomous Institution Affiliated to VTU, Belagavi)

Post Box No.: 1908, Bull Temple Road, Bengaluru – 560 019

DEPARTMENT OF MACHINE LEARNING

Academic Year: 2023-2024 (Session: Nov 2023- Feb 2024)

OBJECT ORIENTED PROGRAMMING (23AM5PCOOP)

ALTERNATIVE ASSESSMENT TOOL (AAT)

"SOLID PRINCIPLES: EMPLOYEE MANAGEMENT SYSTEM"

Submitted by

Student Name: Team Member 1: Team Member 2:	 Lohitha T Rajeshwari DM
USN: Team Member 1: Team Member 2:	 1BM21AI060 1BM22AI407
Semester & Section:	5B
Student Signature: Team Member 1: Team Member 2:	

Valuation Report (to be filled by the faculty)

Faculty In-charge:	**Dr. Sowmya Lakshmi B.S**		
Faculty Signature with date:		Score:	

Table of Contents

INTRODUCTION

SOLID is an acronym for a set of design principles created for developing software using objectoriented languages.The SOLID principles are designed to encourage the creation of code that is simpler, more dependable, and easier to enhance. Each letter in SOLID stands for one design principle. It's often emphasized that adhering to SOLID principles leads to various benefits, such as increased code quality, reduced complexity, enhanced reusability, and better testability. By following these principles, developers can build software systems that are more adaptable to change, have fewer bugs, and are easier to maintain in the long run. The introduction may provide context by highlighting common challenges in software development, such as code that is difficult to maintain, rigid to change, or prone to bugs. It may also mention the evolution of software engineering practices and the need for structured approaches to handle increasing complexity in software projects.The introduction may address the target audience, which typically includes software developers, architects, and engineers interested in improving their software design skills. It may also be relevant for students and educators studying or teaching software engineering principles.

SOLID is an acronym for a set of design principles created for developing software using objectoriented languages. The SOLID principles are designed to encourage the creation of code that is simpler, more dependable, and easier to enhance. Each letter in SOLID stands for one design principle. It's often emphasized that adhering to SOLID principles leads to various benefits, such as increased code quality, reduced complexity, enhanced reusability, and better testability. By following these principles, developers can build software systems that are more adaptable to change, have fewer bugs, and are easier to maintain in the long run. The introduction may provide context by highlighting common challenges in software development, such as code that is difficult to maintain, rigid to change, or prone to bugs. It may also mention the evolution of software engineering practices and the need for structured approaches to handle increasing complexity in software projects.The introduction may address the target audience, which typically includes software developers, architects, and engineers interested in improving their software design skills. It may also be relevant for students and educators studying or teaching software engineering principles.

SOLID as an acronym representing five fundamental principles in object-oriented programming.:

- Single Responsibility Principle (SRP)
- Open/Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP).

Problem Statement:(Employee Management System)

The purpose of this report is to analyze the current implementation of the **Employee Management System,** focusing on its design and adherence to software engineering principles. In particular, we will evaluate the codebase in terms of the SOLID principles, aiming to identify areas for improvement and propose solutions for enhanced maintainability and extensibility.

In the initial implementation, the code combines the behavior of managers and developers within a single Employee class. This violates the principle of separation of concerns and can lead to code that is difficult to maintain and extend. Each type of employee should have its own class to encapsulate its unique behavior and responsibilities. By adhering to SOLID principles, we aim to refactor the codebase to achieve better organization, flexibility, and scalability.

```
class Employee:
    def __init__(self, name):
        self.name = name  # Initialize employee name.

    def work(self):
        pass  # Placeholder method for performing work.

    def report(self):
        pass  # Placeholder method for generating a report.

class Company:
    def __init__(self):
        self.employees = []  # Initialize a list to store employees.

    def hire_employee(self, employee):
        self.employees.append(employee)  # Add employee to the list of employees.

    def show_report(self):
        for employee in self.employees:
            employee.report()  # Display report for each employee.

class Manager(Employee):
    def work(self):
        print(f"{self.name} is managing.")  # Display a message indicating manager is managing.

    def report(self):
        print(f"{self.name} is reporting the status.")  # Display a status report message.

class Developer(Employee):
    def work(self):
        print(f"{self.name} is developing.")  # Display a message indicating developer is developing.

    def report(self):
        print(f"{self.name} is reporting the progress.")  # Display a progress report message.
```

Now, let's apply each SOLID principle:

Single Responsibility Principle (SRP):

The Single Responsibility Principle requires that each class should have a singular, clearly defined purpose.The same applies to other programming entities. That is, it is necessary to decompose software entities so that each entity is responsible for the one job assigned to it.

Problem Statement: In the initial implementation, the Employee class handles both the behavior related to work (e.g., managing, developing) and reporting (e.g., reporting status, progress). This violates the SRP because a class should have only one reason to change. If there are multiple reasons to change, it makes the class harder to understand, maintain, and extend.

```python
# Define classes
class Workable:
    def work(self):
        pass  # Placeholder method for performing work.

class Reportable:
    def report(self):
        pass  # Placeholder method for generating a report.

class Employee:
    def __init__(self, name):
        self.name = name  # Initialize employee name.

# Manager class inherits from Employee, Workable, and Reportable
# Violates SRP by combining responsibilities of managing, working, and reporting in one class.
class Manager(Employee, Workable, Reportable):
    def work(self):
        print(f"{self.name} is managing.")  # Display a message indicating manager is managing.

    def report(self):
        print(f"{self.name} is reporting the status.")  # Display a status report message.

# Developer class inherits from Employee, Workable, and Reportable
# Violates SRP by combining responsibilities of developing, working, and reporting in one class.
class Developer(Employee, Workable, Reportable):
    def work(self):
        print(f"{self.name} is developing.")  # Display a message indicating developer is developing.

    def report(self):
        print(f"{self.name} is reporting the progress.")  # Display a progress report message.

# Example usage
manager = Manager("John")
manager.work()  # Output: John is managing.
manager.report()  # Output: John is reporting the status.

developer = Developer("Alice")
developer.work()  # Output: Alice is developing.
developer.report()  # Output: Alice is reporting the progress.
```

We refactor the code by segregating the responsibilities into separate classes. We create Workable and Reportable interfaces that define the behavior related to work and reporting, respectively. Then, each employee type implements the necessary interfaces. This way, each class has a single responsibility: either handling work behavior or reporting.

OUTPUT:

```
John is managing.
John is reporting the status.
Alice is developing.
Alice is reporting the progress.
```

OPEN-CLOSED PRINCIPLE:

Software entities (classes, modules, functions, etc.) must be open for extension but closed for modification.Changing existing code is bad because it has already been tested and works. If we change the code, then we have to do regression testing. Therefore, when adding functionality, you should not change existing entities, but add new ones using composition or inheritance. Even with this approach, you may have to slightly edit the old code in order to prevent bugs or hacky code. But changing the old code should be avoided as much as possible.

Problem Statement: The initial implementation of the Employee hierarchy is not open for extension and closed for modification. If we need to introduce a new type of employee, such as an Intern, we would need to modify existing classes, which violates the OCP. Modifying existing classes can lead to unintended consequences and increases the risk of introducing bugs

```python
class Employee:
    def __init__(self, name):
        self.name = name  # Initialize employee name.

    def work(self):
        pass  # Placeholder method for performing work.

    def report(self):
        pass  # Placeholder method for generating a report.

# Manager class inherits from Employee
class Manager(Employee):
    def work(self):
        print(f"{self.name} is managing.")  # Display a message indicating manager is managing.

    def report(self):
        print(f"{self.name} is reporting the status.")  # Display a status report message.

# Developer class inherits from Employee
class Developer(Employee):
    def work(self):
        print(f"{self.name} is developing.")  # Display a message indicating developer is developing.

    def report(self):
        print(f"{self.name} is reporting the progress.")  # Display a progress report message.

# Intern class inherits from Employee
# Violates OCP as adding a new type of employee requires modification of the existing code.
class Intern(Employee):
    def work(self):
        print(f"{self.name} is interning.")  # Display a message indicating intern is interning.

    def report(self):
        print(f"{self.name} is reporting.")  # Display a generic reporting message.

intern = Intern("Bob")
intern.work()
intern.report()
```

We refactor the code by designing classes to be open for extension and closed for modification. Instead of modifying existing classes, we create new subclasses to extend the behavior. For example, we introduce an Intern subclass without modifying the existing Employee, Manager, or Developer classes. This way, we can add new employee types without altering the existing codebase.

OUTPUT:

```
Bob is interning.
Bob is reporting.
```

LISKOV SUBSTITUTION PRINCIPLE:

The main idea behind the Liskov Substitution Principle is that for any class, the client should be able to use any subclass of the base class without noticing the difference between them.And therefore without any change in the execution behavior of the program. This means that the inherited class should complement, not replace, the behavior of the parent, and that the client is completely isolated and unaware of changes in the class hierarchy.

Subtype Requirement: Let φ(x) be a property provable about objects x of type T. Then φ(y) should be true for objects y of type S where S is a subtype of T.
Problem Statement: In the initial implementation, if we pass an instance of a subclass (e.g., Manager, Developer) to a function that expects an instance of the superclass Employee, there might be unexpected behavior or errors. This violates the LSP because subclasses should be substitutable for their base class without affecting the correctness of the program.

```
class Employee:
    def __init__(self, name):
        self.name = name  # Initialize employee name.
    def work(self):
        pass  # Placeholder method for performing work.
    def report(self):
        pass  # Placeholder method for generating a report.

class Manager(Employee):
    def work(self):
        print(f"{self.name} is managing.")  # Display a message indicating manager is managing.
    def report(self):
        print(f"{self.name} is reporting the status.")  # Display a status report message.

class Developer(Employee):
    def work(self):
        print(f"{self.name} is developing.")  # Display a message indicating developer is developing.
    def report(self):
        print(f"{self.name} is reporting the progress.")  # Display a progress report message.
# Function to perform work
# Violates Liskov Substitution Principle (LSP) as it directly depends on the specific implementations of work() method.
def do_work(employee):
    employee.work()

# Function to generate report
# Violates Liskov Substitution Principle (LSP) as it directly depends on the specific implementations of report() method.
def generate_report(employee):
    employee.report()
manager = Manager("John")
do_work(manager)
developer = Developer("Alice")
do_work(developer)
generate_report(manager)
generate_report(developer)
```

We refactor the code to ensure that all subclasses adhere to the contract defined by the superclass. Each subclass should override methods in a way that maintains the behavior specified by the superclass. By following the LSP, we ensure that any subclass can be used interchangeably with its superclass without introducing errors or unexpected behavior.

OUTPUT:

```
John is managing.
Alice is developing.
John is reporting the status.
Alice is reporting the progress.
```

Interface Segregation Principle (ISP):

Clients should not depend on interfaces they do not use. You shouldn't force a client to implement an interface that it does not use.Create thin interfaces: many client-specific interfaces are better than one general-purpose interface. This principle eliminates the disadvantages of implementing large interfaces.

Problem Statement: In the initial implementation, the Employee class enforces methods like work() and report() on all subclasses, regardless of whether they need those methods. This violates the ISP because classes should not be forced to depend on interfaces they don't use. Having large, monolithic interfaces increases coupling and makes the code harder to maintain.

```python
class Workable:
    def work(self):
        pass  # Placeholder method for performing work.
class Reportable:
    def report(self):
        pass  # Placeholder method for generating a report.
class Employee:
    def __init__(self, name):
        self.name = name  # Initialize employee name.
class Manager(Employee, Workable, Reportable):
    def work(self):
        print(f"{self.name} is managing.")  # Display a message indicating manager is managing.
    def report(self):
        print(f"{self.name} is reporting the status.")  # Display a status report message.

class Developer(Employee, Workable, Reportable):
    def work(self):
        print(f"{self.name} is developing.")  # Display a message indicating developer is developing.
    def report(self):
        print(f"{self.name} is reporting the progress.")  # Display a progress report message.
class Intern(Employee, Workable):
    def work(self):
        print(f"{self.name} is interning.")  # Display a message indicating intern is interning.
manager = Manager("John")
manager.work()
manager.report()
developer = Developer("Alice")
developer.work()
developer.report()
intern = Intern("Bob")
intern.work()
# Output: Bob is interning.
# Violates Interface Segregation Principle (ISP) as Intern class does not implement the report method from the Reportable interface.
# Attempting to call the report method on an Intern instance will result in an AttributeError.
# intern.report()  # This line would raise an AttributeError.
```

We refactor the code by defining smaller, more focused interfaces that represent specific behaviors. Each subclass then implements only the interfaces it needs. For example, the Manager and Developer classes implement both the Workable and Reportable interfaces, while the Intern class only implements the Workable interface. This way, we reduce the

coupling between classes and ensure that each class depends only on the interfaces it requires.

OUTPUT:

```
John is managing.
John is reporting the status.
Alice is developing.
Alice is reporting the progress.
Bob is interning.
```

Dependency Inversion Principle (DIP):

Modules at higher levels should not depend on modules at low levels. Both upper and lower level classes must depend on the same abstractions. Abstractions should not depend on details. Details must depend on abstractions.As development progresses, there comes a point where our application predominantly consists of modules. At this stage, it becomes essential to enhance our code using dependency injection. The functionality of high-level components relies on low-level components. You can utilize inheritance or interfaces to achieve specific behaviors.

Problem Statement: In the initial implementation, the Company class directly depends on concrete implementations of Employee, which makes it tightly coupled to those implementations. This violates the DIP because high-level modules should not depend on low-level modules directly. Instead, both should depend on abstractions.

```
class Workable:
    def work(self):
        pass  # Placeholder method for performing work.
class Reportable:
    def report(self):
        pass  # Placeholder method for generating a report.
class Employee:
    def __init__(self, name):
        self.name = name  # Initialize employee name.
class Manager(Employee, Workable, Reportable):
    def work(self):
        print(f"{self.name} is managing.")  # Display a message indicating manager is managing.
    def report(self):
        print(f"{self.name} is reporting the status.")  # Display a status report message.
class Developer(Employee, Workable, Reportable):
    def work(self):
        print(f"{self.name} is developing.")  # Display a message indicating developer is developing.
    def report(self):
        print(f"{self.name} is reporting the progress.")  # Display a progress report message.
class Company:
    def __init__(self):
        self.employees = []  # Initialize a list to store employees.
    def hire_employee(self, employee):
        self.employees.append(employee)  # Add employee to the list of employees.
    # Violates Dependency Inversion Principle (DIP) by depending on concrete implementation (Reportable) instead of abstraction.
    # Company class directly depends on the Reportable interface, making it tightly coupled to the implementation details of employees.
    def show_report(self):
        for employee in self.employees:
            if isinstance(employee, Reportable):
                employee.report()  # Display report for each employee.
company = Company()
manager = Manager("John")
developer = Developer("Alice")
company.hire_employee(manager)
company.hire_employee(developer)
company.show_report()
```

We refactor the code to introduce abstractions (e.g., interfaces) that decouple the Company class from specific implementations of Employee. The Company class now depends on abstractions (e.g., the Reportable interface) rather than concrete implementations. This way, we can easily swap different implementations of Employee without modifying the Company class, promoting flexibility and maintainability.

OUTPUT:

```
John is reporting the status.
Alice is reporting the progress.
```

CONCLUSION:

The SOLID principles provide a comprehensive set of guidelines for writing maintainable, flexible, and scalable software. These principles include Single Responsibility Principle (SRP), Open/Closed Principle (OCP), Liskov Substitution Principle (LSP), Interface Segregation Principle (ISP), and Dependency Inversion Principle (DIP). By adhering to these principles, developers can ensure that their code is modular, easy to understand, and less prone to bugs. The SRP emphasizes that each class should have only one reason to change, promoting better organization and reducing the risk of unintended side effects when modifying code. The OCP encourages code to be open for extension but closed for modification, enabling easier addition of new features without altering existing code. The LSP states that subclasses should be substitutable for their base classes without affecting the program's correctness, enhancing code reuse and flexibility.

The ISP advocates for the creation of smaller, more focused interfaces, preventing classes from depending on methods they don't use and reducing coupling between components. Finally, the DIP suggests that high-level modules should not depend on low-level implementations, but rather on abstractions, promoting loose coupling and facilitating easier code maintenance and testing. By applying these principles judiciously, developers can create software that is robust, adaptable, and easier to maintain, leading to increased productivity and overall better software quality.