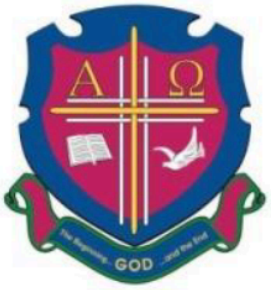

UGANDA CHRISTIAN UNIVERSITY

A Centre of Excellence in the Heart of Africa

FACULTY OF ENGINEERING DESIGN AND TECHNOLOGY

NAME: ISAIAH MUKISA

REG NO: S21B23/007

ACCESS NO: A94160

COURSE: BACHELOR OF SCIENCE IN COMPUTER SCIENCE (BSCS)

COURSE UNIT: SOFTWARE CONSTRUCTION

1. Single Responsibility Principle (SRP) Violation:
- The “Report” class violates SRP as it is responsible for both generating a report and writing specific reports for different roles.

2. Open/Closed Principle (OCP) Violation:
- The “Report” and “BonusCalculator” classes violate OCP because they need modification whenever a new role is added.

3. Liskov Substitution Principle (LSP) Violation:
- The “Manager” and “Developer” classes, both inheriting from “Employee”, may not adhere to LSP as they add specific methods (`calculate_manager_bonus`, `manage_team`, `calculate_developer_bonus`, `code_review`) not present in the base class.

4. Dependency Inversion Principle (DIP) Violation:
- The “Report” and “BonusCalculator” classes directly depend on the “Manager” and “Developer” classes there by violating DIP.

Refactoring Plan:

1. For Single Responsibility Principle Violation:
- Create a separate class for report generation.
- Each role should have its own class responsible for writing specific reports.

2. For Open/Closed Principle Violation:
- Introduce a strategy pattern for report generation and bonus calculation.
- Each role's report and bonus calculation logic should be encapsulated in its own class.

3. Liskov Substitution Principle Violation:
- Ensure that all subclasses adhere to the same interface by introducing abstract methods in the base class.

4.Interface Segregation Principle Violation:
- Break down the “employee” class into smaller interfaces for specific roles.

5. Dependency Inversion Principle Violation:
- Use dependency injection to decouple the “Report” and “BonusCalculator” classes from specific implementations.

1. **Here is a before snippet, before the classes were broken down further and refactored to follow the SOLID principles**

```python
class Report:
    def generate_report(self, employee):
        if employee.role == "Manager":
            self.write_manager_report(employee)
        elif employee.role == "Developer":
            self.write_developer_report(employee)

    def write_manager_report(self, manager):
        print(f"Manager Report: {manager.name}")

    def write_developer_report(self, developer):
        print(f"Developer Report: {developer.name}")

class BonusCalculator:
    def calculate_bonus(self, employee):
        if employee.role == "Manager":
            return employee.calculate_manager_bonus()
        elif employee.role == "Developer":
            return employee.calculate_developer_bonus()
```

Here the Report class and Bonus Calculator class were not following the SOLID principles

Here is an after of refactoring

```python
class BonusCalculator(ABC):
    @abstractmethod
    def calculate_bonus(self, employee):
        pass

# This is the implementation for writing a report for a Manager
class ManagerReportWriter(ReportWriter):
    def write_report(self, manager):
        print(f"Manager Report: {manager.name}")

# This is the implementation for writing a report for a Developer
class DeveloperReportWriter(ReportWriter):
    def write_report(self, developer):
        print(f"Developer Report: {developer.name}")

# This is the implementation for calculating bonuses for a Manager
class ManagerBonusCalculator(BonusCalculator):
    def calculate_bonus(self, manager):
        return 1000

# This is the implementation for calculating bonuses for a Developer
class DeveloperBonusCalculator(BonusCalculator):
    def calculate_bonus(self, developer):
        return 500
```

The above code shows how I have refactored and broken down the Bonus calculator and Report writing class into classes that follow the SOLID principles making it easier to read through the code

2. Here is another before snippet of the Manager and Developer classes before refacttoring

```
class Manager(Employee):
    def calculate_manager_bonus(self):
        return 1000

    def manage_team(self):
        print(f"{self.name} is managing the team.")

class Developer(Employee):
    def calculate_developer_bonus(self):
        return 500

    def code_review(self):
        print(f"{self.name} is conducting a code review.")
```

And here is the after snippet after refactoring most of the code

```
42  class ManagerRole(ABC):
43      @abstractmethod
44      def manage_team(self):
45          pass
46
47  # Interface defining the contract for developer roles
48  class DeveloperRole(ABC):
49      @abstractmethod
50      def code_review(self):
51          pass
52
53  # This is the implementation of a Manager with specific role-related actions
54  class Manager(Employee, ManagerRole):
55      def manage_team(self):
56          print(f"{self.name} is managing the team.")
57
58  # This is the implementation of a Developer with specific role-related actions
59  class Developer(Employee, DeveloperRole):
60      def code_review(self):
61          print(f"{self.name} is conducting a code review.")
```

In the above code the Manager and Developer classes have been further broken down to obey the SOLID principles, specifically **ISP**