**A code review of the provided code, identifying violations of each SOLID principle:**

**Single Responsibility Principle (SRP):**

Violation: The Report class is responsible for both generating reports and writing specific reports for managers and developers. This violates the SRP because the class has more than one reason to change.

Rationale: The Report class should be focused on a single responsibility, such as generating reports or formatting them. It should not handle the logic for different types of reports.

**Open/Closed Principle (OCP):**

Violation: The Report and BonusCalculator classes are not open for extension but closed for modification. Adding new types of employees or report formats would require modifying existing classes.

Rationale: The OCP states that classes should be open for extension but closed for modification. In this case, the classes should be designed in a way that allows for easy extension to support new functionality without modifying existing code.

**Liskov Substitution Principle (LSP):**

Violation: There are no explicit violations of the LSP in the provided code.

Rationale: The LSP states that objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program. This principle is more related to inheritance and polymorphism, and there are no evident issues with it in the provided code.

**Interface Segregation Principle (ISP):**

Violation: There are no explicit violations of the ISP in the provided code.

Rationale: The ISP states that clients should not be forced to depend on interfaces they don't use. In the provided code, there are no interfaces defined explicitly, but the principle can still be violated if clients are required to depend on unnecessary methods or functionalities.

**Dependency Inversion Principle (DIP):**

Violation: The Report and BonusCalculator classes directly depend on concrete implementations of Manager and Developer classes.

Rationale: The DIP states that high-level modules should not depend on low-level modules; both should depend on abstractions. In the provided code, the high-level modules (Report and BonusCalculator) directly depend on low-level modules (Manager and Developer). This tight coupling makes the code less flexible and harder to maintain.

**Here's a detailed refactoring plan addressing each SOLID principle violation in the provided codebase:**

**Single Responsibility Principle (SRP):**

Refactoring Task: Extract the responsibility of generating reports from the Report class into a separate ReportGenerator class. Ensure that the ReportGenerator class is solely responsible for generating reports based on employee roles.

Impact: This refactoring will improve the maintainability and readability of the code by separating concerns and adhering to the SRP.

Priority: High, as it addresses a fundamental violation of the SRP and improves the overall structure of the codebase.

**Open/Closed Principle (OCP):**

Refactoring Task: Introduce abstraction for report generation and bonus calculation to make the classes open for extension. Create interfaces or abstract classes for ReportGenerator and BonusCalculator to allow for easy extension with new functionalities.

Impact: Implementing the OCP will make the codebase more flexible and adaptable to future changes, reducing the risk of introducing bugs when extending functionality.

Priority: High, as it ensures that the codebase can accommodate future enhancements and new requirements without extensive modifications.

**Liskov Substitution Principle (LSP):**

No explicit violations of the LSP were identified in the provided codebase. Therefore, no specific refactoring tasks related to the LSP are required at this time.

**Interface Segregation Principle (ISP):**

No explicit violations of the ISP were identified in the provided codebase. However, if interfaces are introduced during the OCP refactoring, ensure that they adhere to the ISP by not imposing unnecessary dependencies on clients.

**Dependency Inversion Principle (DIP):**

Refactoring Task: Introduce dependency inversion by decoupling the Report and BonusCalculator classes from concrete implementations of Manager and Developer. Use dependency injection or inversion of control containers to inject dependencies into the Report and BonusCalculator classes.

Impact: Implementing the DIP will reduce the coupling between components, making the codebase more modular, testable, and maintainable.

Priority: High, as reducing dependencies between modules is crucial for improving code quality and facilitating future changes.