Pitfalls and SOLID Principles Applied

1. Pitfalls Found and Corrected

During the review of the original code, several common pitfalls were identified and corrected. These issues mainly revolved around code architecture, exception handling, and thread safety.

Pitfall 1: Lack of Single Responsibility (SRP)

In the original code, certain classes, such as `SaveManager`, were handling multiple responsibilities. For example, `SaveManager` not only handled the logic of saving data, but also performed file operations directly. This violated the Single Responsibility Principle (SRP), making the code harder to maintain and extend.

Correction: The `SaveManager` class was refactored to delegate file operations to a separate class `FileSaver`. This allows `SaveManager` to focus solely on the logic of saving data, while `FileSaver` handles the details of how the data is saved.

Pitfall 2: Dependency on Concrete Classes (DIP)

The original implementation of `SaveManager` depended directly on low-level classes like `FileWriter`, which violates the Dependency Inversion Principle (DIP). This meant that any change in the way data was saved would require modifications to `SaveManager`, reducing its flexibility.

Correction: An interface `DataSaver` was introduced to invert the dependency. Now, `SaveManager` depends on the abstraction `DataSaver`, allowing different saving strategies (e.g., file saving, database saving) to be used interchangeably.

Pitfall 3: Lack of Error Handling

Several methods in the original code lacked proper error handling, which could lead to crashes or undefined behavior in the event of invalid input or system errors.

Correction: Improved error handling was added to key methods. For example, methods in `Calculator` now check for invalid inputs (e.g., negative salary values) and throw appropriate exceptions, ensuring safer and more predictable execution.

Pitfall 4: Thread Safety

The original implementation of `SaveManager` was not thread-safe, which could cause data corruption or unpredictable behavior when accessed by multiple threads simultaneously.

Correction: The code was modified to include synchronized blocks to ensure that only one thread can execute file-saving operations at a time, making the code thread-safe.

2. SOLID Principles Applied

The code was refactored to follow the SOLID principles, resulting in a more modular, maintainable, and flexible system.

Single Responsibility Principle (SRP)

Classes were refactored to have a single responsibility. For example, `SaveManager` is now solely responsible for orchestrating the saving of data, while the actual saving mechanism is handled by a separate class that implements the `DataSaver` interface.

Open/Closed Principle (OCP)

The code was designed to be open for extension but closed for modification. For example, new saving strategies (like saving to a database) can be introduced by implementing the `DataSaver` interface without modifying the existing `SaveManager` class.

Liskov Substitution Principle (LSP)

While no direct violations of the Liskov Substitution Principle were found, care was taken to ensure that any subclass or implementation of an interface can be used interchangeably without breaking the functionality of the system.

Interface Segregation Principle (ISP)

Interfaces were kept focused and small. For example, the `DataSaver` interface contains only one method, `save(String data)`, ensuring that classes implementing this interface are not forced to implement methods they don't need.

Dependency Inversion Principle (DIP)

The high-level `SaveManager` class now depends on the abstraction `DataSaver`, rather than concrete classes like `FileWriter`. This allows for greater flexibility and easier maintenance.