**1. Design

- **Object-Oriented Programming (OOP)**:
  - The goal in OOP is to encapsulate data and behavior within classes and objects.
  - In Assignment 1, I created a `StyleChecker` class, which managed state using instance variables like `self.filename`, `self.tree`, and `self.imports`.
  - Methods such as `read_file()` and `analyze_imports()` relied on and modified these instance variables.
  - The overall analysis process was coordinated by the `run()` method, which organized the workflow step by step.
- **Functional Programming (FP)**:
  - FP emphasizes **pure functions** and **immutability**. Functions are designed to work independently without side effects.
  - In Assignment 2, I removed the class and replaced instance variables with **local variables** passed between functions.
  - The FP approach involved writing functions that took inputs and returned outputs without modifying any external state.
  - The code flow was managed by composing several pure functions, making each function easier to test and understand.

---

2. Immutability

- **OOP**:
  - In the OOP version, instance variables like `self.imports` and `self.classes` were updated throughout the lifetime of the `StyleChecker` object.
- **FP**:
  - In the FP version, data structures are **immutable**. Functions return new data instead of modifying existing variables.
  - For example, instead of updating an instance variable, `get_imports()` returns a list of imports directly.

---

3. Pure Functions

- **OOP**:

    - In the `StyleChecker` class, methods often depended on or changed the internal state, making them **impure functions**.
- **FP**:
    - In the FP version, each function is designed to be **pure**. Pure functions produce the same output for the same input and have no side effects.
    - For example, `parse_python_file(content)` returns the AST (Abstract Syntax Tree) without modifying any global state.

---

4. Function Composition

- **OOP**:
    - In the OOP approach, the `run()` method called several class methods in sequence to complete the analysis.
- **FP**:
    - In the FP approach, the analysis process was broken down into smaller, independent functions.
    - The main function, `generate_report()`, combines functions like `get_imports()`, `get_classes()`, and `get_functions()` to achieve the final result.
    - This encourages **modularity** and **reusability** because each function performs a specific task.

---

5. Naming and State Management

- **OOP**:
    - State was managed within the `StyleChecker` class, making the code more contextual but tightly coupled to the class structure.
- **FP**:
    - In FP, state was explicitly passed between functions as arguments, making the flow of data clearer and reducing dependencies between functions.

Summary of Changes

1. **Refactored Class Methods to Pure Functions**:
    - Methods in the `StyleChecker` class were converted to standalone pure functions.

2. **Eliminated Instance Variables**:
   - Replaced instance variables with local variables and function return values.
3. **Decoupled Analysis Stages**:
   - Each step of the analysis is now handled by an independent function, improving modularity.
4. **Immutable Data Flow**:
   - Data is not changed in place; instead, new data structures are returned by functions.