# **Refactoring a Java Program with GitHub Copilot**

## **Introduction**

Welcome to this Java refactoring exercise, where you’ll learn how to transform a minimal or messy Java application into a clean, maintainable solution—using GitHub Copilot. We’ll adopt a **prompt-first** approach:

1. We provide or write a prompt.
2. Copilot generates code suggestions.
3. We then insert or modify the suggested code.

By the end, you’ll have a well-structured Java application that demonstrates best practices such as logging, configuration management, and handling errors gracefully.

## **Objective**

1. **Leverage GitHub Copilot in Java development**
   * Inline suggestions (using your IDE’s specific shortcut).
   * Copilot Chat (for conversational code generation, if available).
2. **Practice the Prompt-First Methodology**
   * Write or use a prompt first.
   * Accept or refine the generated code.
3. **Refactor a Java project to**
   * Separate configuration data from logic.
   * Use logging and handle errors precisely.
   * Include optional testing with JUnit.

| **Note**: Below prompts are provided solely as examples of effective prompt construction. As participants, we must engage in an iterative process to refine these examples and develop the optimal prompt for the task at hand. |
| --- |

## **Step 1: Project Setup**

**Objective:**Generate an entire Maven project workspace for refactoring using GitHub Copilot Chat only.

**Initialize the Project Workspace:**In Copilot Chat, instruct:

| @workspace: Initialize a new Maven project named "java-copilot-refactor" with a standard structure. Create the following folders automatically: - src/main/java/config (for configuration) - src/main/java/aggregator (for business logic) - src/test/java/aggregator (for unit tests) - Generate a pom.xml with a basic Maven setup for Java 17. |
| --- |

| **Note**: Above prompt would help you to generate the required commands. Once commands are created, hover on the commands and click on **Insert into Terminal** option |
| --- |

This command instructs Copilot to create the entire project scaffold automatically, ensuring a consistent structure without any manual file creation.

**Verify the Workspace Structure:**Next, confirm the structure by asking:

| @workspace: Provide a detailed overview of the directory structure for the "java-copilot-refactor" project. |
| --- |

* This step helps you ensure that all necessary folders—such as configuration, business logic, and test directories—are correctly created.

**Challenges for Step 1:**

* Evaluate whether additional folders (e.g., a dedicated scripts folder for automation or a docs folder for documentation) are needed. Use further /fix prompts to add these directories until the workspace is perfectly organized.
* Consider simulating a multi-module structure and refine your prompt so that your workspace scales to larger projects.

**Step 2: Generate the Initial Data Aggregator Script**

**Objective:**Create the core Java class that performs data aggregation using GitHub Copilot Chat only.

**Generate the Data Aggregator Class:**Open the file automatically via Copilot Chat:

| #file /new: Create a new file named DataAggregator.java in src/main/java/aggregator. |
| --- |

| /start: Generate a Java class named DataAggregator that reads a CSV file na#file /new: Create a new file named DataAggregator.java in src/main/java/aggregator. |
| --- |

| **Note**: Above prompt would help you to generate the required commands. Once commands are created, hover on the commands and click on **Insert into Terminal** option |
| --- |

The prompt specifies the class name, the source file, and the required functionality. This ensures Copilot provides a complete initial implementation.

**Refine the Generated Code:**Once the basic code is generated, refine it by specifying:

| /fix: Enhance DataAggregator.java to use try-with-resources for managing file reading and to handle exceptions more robustly by catching IOException and NumberFormatException separately. |
| --- |

* This command instructs Copilot to update the code to follow modern Java resource management and error handling practices.

**Challenges for Step 2:**

* Extend the functionality to return the total sales as a value instead of just printing it, so you can test it more effectively.
* Experiment with prompts to integrate streaming or iterator patterns for large CSV files.

**Step 3: Implement Configuration Management**

**Objective:**Separate configuration data from business logic by creating a dedicated configuration class.

**Detailed Process and Prompts:**

**Generate the Configuration Class:**Create the configuration file via Copilot Chat:

| #file /new: Create a new file named Config.java in src/main/java/config. /start: Generate a Java class named Config that contains a public static final field CSV\_FILE\_PATH. The value should be loaded from the environment variable "CSV\_FILE\_PATH" if set, otherwise default to "sales\_data.csv". |
| --- |

* This prompt ensures that configuration details are separated from the core logic, making the application more flexible and maintainable.

**Integrate Configuration into Data Aggregator:**Update DataAggregator.java by instructing:

| /fix: Update DataAggregator.java to import Config and replace any hard-coded file paths with Config.CSV\_FILE\_PATH. |
| --- |

**Challenges for Step 3:**

* Consider enhancing Config.java to load additional settings (e.g., logging level, column index) from a properties file. Refine your prompt until all necessary configuration values are externalized.
* Generate a sample .env.example or app.properties file via Copilot Chat for documentation purposes.

**Step 4: Enhance Logging and Error Handling**

**Objective:**Replace manual print statements with a proper logging framework and improve error handling.

**Detailed Process and Prompts:**

**Add Logging Dependencies:**Generate dependency entries for SLF4J and Logback:

| /start: Generate Maven dependency entries for SLF4J API (version 2.0.7) and Logback Classic (version 1.4.5) to be added to the pom.xml. |
| --- |

* **Refactor Logging in Data Aggregator:**Update DataAggregator.java to use logging instead of System.out:

| /start: Refactor DataAggregator.java to replace all System.out.println calls with SLF4J logging calls. Use logger.info for normal messages and logger.error for exceptions. |
| --- |

* This prompt is detailed, specifying the log levels to be used, so the generated code follows best practices.

**Deepen Error Handling:**Refine exception handling by instructing:

| /fix: Enhance DataAggregator.java by wrapping file operations in try-with-resources and separating catch blocks for IOException and NumberFormatException, logging each error with a detailed message. |
| --- |

**Challenges for Step 4:**

* Generate prompts to create a logback configuration file that outputs logs to both the console and a file.
* Ask Copilot Chat to suggest additional log statements to provide more context (e.g., logging the start and end of file processing).

**Step 5: Implement Automated Testing with JUnit**

**Objective:**Develop a suite of automated tests for your data aggregation functionality using JUnit and Mockito, all through GitHub Copilot Chat.

**Detailed Process and Prompts:**

**Add Testing Dependencies:**Generate the necessary Maven dependency:

| /start: Generate the Maven dependency entry for JUnit 5 (JUnit Jupiter version 5.9.2) with test scope and add it to the pom.xml. |
| --- |

* **Generate a Test Class:**Create the test file using Copilot Chat:

| #file /new: Create a new file named DataAggregatorTest.java in src/test/java/aggregator. |
| --- |

| /start: Generate a JUnit 5 test class for DataAggregator that tests the calculateTotalSales method using a temporary CSV file. The tests should verify that the correct total is calculated and handle exceptions properly. |
| --- |

* **Integrate Mocks:**If external calls are involved, refine the test:

| /fix: Enhance DataAggregatorTest.java by generating Mockito-based mocks for file I/O operations to simulate reading from a CSV without accessing the actual file system. |
| --- |

**Challenges for Step 5:**

* Develop tests that simulate edge cases, such as an empty CSV file or malformed data.
* Experiment with prompts to capture and verify log output during tests, ensuring that error messages are correctly logged.

**Step 6: CI/CD Integration with GitHub Actions**

**Objective:**Automate your build, test, and deployment process using GitHub Actions, all through Copilot Chat.

**Detailed Process and Prompts:**

**Generate the Workflow File:**Instruct Copilot Chat to generate a workflow file:

| /start: Generate a GitHub Actions workflow file named ci.yaml in the .github/workflows/ directory. The workflow should: - Set up Java environments (Java 11, 17, and 20) - Install Maven dependencies - Run the test suite using Maven - Report the test results |
| --- |

* **Enhance the Workflow File:**Refine the workflow by adding caching and triggers:

| /fix: Enhance the ci.yaml workflow to include caching for Maven dependencies and configure it to trigger on push events to the main branch. |
| --- |

**Challenges for Step 6:**

* Experiment with integrating additional code quality tools (e.g., SonarQube) into your workflow. Refine your prompts until the workflow is robust and fully automated.
* Simulate a CI/CD run and adjust the workflow prompts to resolve any issues reported by GitHub Actions.

**Step 7: Stage, Commit, and Push Changes Using GitHub CLI**

**Objective:**Generate all necessary Git commands to stage, commit, and push your refactored code—completely automated via Copilot Chat.

**Detailed Process and Prompts:**

**Generate Git Commands:**Instruct Copilot Chat:

| /start: Generate the Git commands to stage all changes, commit them with the message "Modernize Java ETL Pipeline: Upgrade to Java 11 features, implement robust error handling, integrate SLF4J logging, and add comprehensive JUnit tests", and push the changes to the branch feature/modernize-java-etl. |
| --- |

* **Verify Commit Status:**Confirm by asking:

| @workspace: List all Git branches to confirm that the branch "feature/modernize-java-etl" is updated with the latest commits. |
| --- |

**Challenges for Step 7:**

* Refine your commit messages iteratively to include specific details about each improvement.
* Experiment with prompts to clean up commit history, such as squashing commits, ensuring a clean and professional Git log.

**Step 8: Create and Refine the Pull Request**

**Objective:**Generate a pull request using the GitHub CLI and then refine the PR summary entirely via Copilot Chat, ensuring the description clearly communicates all improvements.

**Detailed Process and Prompts:**

**Generate the Pull Request:**In Copilot Chat, enter:

| /start: Generate the GitHub CLI command to create a pull request from the branch feature/modernize-java-etl to the main branch. The PR title should be "Modernize Java ETL Pipeline" and the description must include details on: - Upgrading to modern Java features (HttpClient, try-with-resources, Java Streams) - Replacing System.out.println with SLF4J logging - Implementing robust error handling with custom exceptions - Utilizing environment-based configuration - Adding comprehensive JUnit tests |
| --- |

* **Refine the PR Summary:**Once the PR is created, further refine the description:

| /fix: Review the current pull request description and generate an enhanced, detailed summary that explains the migration to modern Java practices, the integration of SLF4J logging, improvements in error handling and resource management, the shift to environment-based configuration, and the addition of comprehensive automated tests. |
| --- |

* **Update the PR:**Finally, update the PR description with:

| @git: Update the pull request description with the enhanced summary. |
| --- |

**Challenges for Step 8:**

* Refine the summary iteratively to include specific performance improvements or code reduction metrics.
* Experiment with different formats and details until the PR summary clearly and concisely communicates every aspect of your modernization work.

**Additional Advanced Challenges**

* **Implement Update Functionality:**Use Copilot Chat to generate a new API route that provides a PUT endpoint for updating product details, then integrate and test it within the ETL pipeline.
* **Integrate a Global Error Boundary:**Generate a custom React error boundary component that catches errors in the UI and displays a fallback interface. Refine your prompts until the component robustly handles exceptions.
* **Refactor State Management with Context:**Migrate any shared state management to React Context for global accessibility. Generate a custom hook through Copilot Chat and refactor your components accordingly.
* **Automate Deployment with GitHub Actions:**Further refine your CI/CD pipeline by generating an advanced GitHub Actions workflow that supports parallel testing, caching, and automated rollback in case of failures.

## **Conclusion**

This comprehensive guide has demonstrated how to refactor a Java program using only GitHub Copilot Chat in VS Code. By crafting detailed, iterative prompts based on prompt engineering best practices and refining them at each step, you transformed basic code outputs into a robust, production-ready application. The process covered project setup, configuration management, enhanced logging and error handling, automated testing, CI/CD integration, and generating a clear pull request summary—all without any manual file or folder creation. The challenges provided at each step encourage further exploration of advanced techniques, ensuring that you continuously improve your development workflow.

**Happy coding—and enjoy your journey to mastering GitHub Copilot Chat for refactoring Java programs and streamlining your development process!**