Mid-term Project paper

Interactive Calculator with Command Pattern and Plugin Architecture

Nishaben Patel **UCID:np857**

1.Introduction:

In this project, I have implemented an interactive calculator application that supports basic arithmetic operations (addition, subtraction, multiplication, and division). The project is designed using the Command pattern and integrates a Read-Eval-Print-Loop (REPL) interface, allowing users to input commands dynamically.

Additionally, the program follows a plugin architecture that supports dynamic loading of commands without requiring modifications to the main application.

This paper outlines the architectural choices, coding practices, and the concepts of testing, REPL, and design patterns demonstrated in the project.

2. Program Architecture

- **Command Pattern:**

The Command pattern is fundamental to the architecture of this application. It is used to summarize each arithmetic operation as a command. By using the Command pattern. This approach not only simplifies the process of adding new commands (such as arithmetic operations) but also provides a way to reuse, undo, or store commands for future execution.

Each command (Add, Subtract, Multiply, and Divide) inherits from a base Command class, which defines an **abstract method execute ().**

Below is an overview of the command structure:

```
class Command (ABC):
    @abstractmethod
    def execute (self, *args):
            pass
class AddCommand(Command):
    def execute (self, *args):
        return args[0] + args[1]
```

The CommandHandler class is responsible for registering commands and invoking them based on user input. When the user types in a command (such as "add 2 3"), the REPL interface passes the command to the CommandHandler, which looks up the corresponding command and executes it.

- **REPL Interface:**

The REPL (Read-Eval-Print-Loop) is the core interactive loop that allows users to enter commands, evaluates those commands, and prints the result to the console.

The REPL runs continuously, reading user input, splitting it into command and arguments, and then passing the command to the CommandHandler for execution. The loop also handles errors, such as invalid commands or improper arguments, by displaying appropriate error

messages and prompting the user for new input.

Here is an example of the REPL loop in action:

```
while True: #REPL Read, Evaluate, Print, Loop
        user_input = input(">>> ").strip().lower()
        if user_input == 'exit':
            print("Goodbye!")
            raise SystemExit("Exiting...")
        elif user_input == 'menu':
            self.print_available_commands()
        else:
            parts = user_input.split()
            command_name = parts[0]
            # Check if command is valid
            if command_name not in self.command_handler.commands:
                print(f"No such command: {command_name}")
                continue
```

The REPL enhances user interaction by making the calculator behave like a simple command-line tool, where users can interact with the program without manually editing the code.

- **Plugin Architecture:**

One of the main goals of the project was to support dynamic command loading using a plugin architecture. Plugins allow the system to automatically load new commands without modifying the main code. This was implemented by scanning the plugins directory for modules, loading them dynamically, and registering their commands.

```
def load_plugins(self):
    # Dynamically load all plugins in the plugins directory
    plugins_package = 'calculator.plugins'
    for _, plugin_name, is_pkg in pkgutil.iter_modules([plugins_package.replace('.', '/')]):
        if is_pkg:
            plugin_module = importlib.import_module(f'{plugins_package}.{plugin_name}')
            for item_name in dir(plugin_module):
                item = getattr(plugin_module, item_name)
                try:
                    if issubclass(item, (Command)):
                        self.command_handler.register_command(plugin_name, item())
                except TypeError:
                    continue
```

The plugin loader searches for all available plugins in the designated directory and imports them at runtime. Each plugin registers its commands with the CommandHandler, making them available for use in the REPL. This architecture provides flexibility by allowing new features to be added without directly changing the core logic of the program.

- **Menu Command:**

To improve usability, a menu command was added to display all the available commands at any time. This command retrieves a list of registered commands from the CommandHandler and prints them to the user. The menu can be invoked either at the start of the application or by typing "menu" in the REPL.

This feature ensures that users are always aware of the available operations and how to access them.

```
def print_available_commands(self):
    available_commands = self.command_handler.get_registered_commands()
    print ("Available commands: " + ", ".join(available_commands))
```

output:

Type 'menu' to see all available commands.

>>> menu

Available commands: add, divide, exit, multiply, subtract

- **Testing and Code Coverage:**

Testing is a critical aspect of ensuring the functionality and reliability of the program. I use **pytest** to write unit tests for both the command execution and REPL functionality. The tests cover a variety of scenarios, including valid and invalid commands, proper argument handling, and REPL interaction.

For instance, the test for the **"add"** command ensures that the command returns the correct result when given valid inputs:

file: tests/ test_commands.py

```
def test_add_command(capfd):
    command = AddCommand()
    result = command.execute(5, 3)
    assert result == 8, " correctly add two numbers"
```

I also tested invalid commands to verify that the system handles them gracefully:

```
def test_execute_command_invalid():
    handler = CommandHandler()
    result = handler.execute_command('invalid_command', 1, 2)
    assert result == "No such command: 'invalid_command'"
```

To ensure code coverage it's show 97%, I used coverage reports to identify any untested parts of the code and wrote additional tests as needed. This ensures that the entire code is thoroughly verified and reduces the risk of undetected bugs.

pytest –cov

coverage report -m

Name	Stmts	Miss	Cover
calculator/__init__.py	48	0	100%
calculator/commands/__init__.py	16	1	94%
calculator/plugins/add/__init__.py	4	0	100%
calculator/plugins/divide/__init__.py	6	0	100%
calculator/plugins/exit/__init__.py	5	0	100%
calculator/plugins/menu/__init__.py	8	0	100%
calculator/plugins/multiply/__init__.py	4	0	100%
calculator/plugins/subtract/__init__.py	4	0	100%
tests/__init__.py	0	0	100%
tests/conftest.py	0	0	100%
tests/test_calculator.py	72	7	90%
tests/test_commands.py	74	0	100%
TOTAL	241	8	97%

3. Design Patterns

1. Command Pattern

The Command pattern plays a key role in organizing the different calculator operations as separate, reusable commands. This pattern simplifies the execution of commands and allows the system to grow by adding new commands easily.

2. Plugin Architecture

Loose connection between the main program and the commands is made possible by the plugin design. This means that new features may be introduced without changing the core code by just developing new plugins and putting them in the proper directory.

4. Code Practices (Readability and Maintainability)

One of the key goals in this project was to ensure the code is readable and maintainable. This was achieved by observing to good coding practices, such as:

- Using meaningful names for classes, methods, and variables.
- Breaking down the program into small, manageable modules.
- Including comments that explain complex logic where required.
- Writing unit tests to ensure all parts of the code are verified.

The program is structured to facilitate easy updates and scalability. By separating the commands from the main REPL logic and using plugins, the system can be extended without introducing new bugs.