Architect Your Agent's Components – Student Notebook

In this notebook you will:

1. **Task 1 – Planner:** Fix a `Planner` so it returns the right kind of plan.
2. **Task 2 – Memory:** Fix `Memory` so its methods return the right kinds of values.
3. **Task 3 – Executor:** Fix `Executor` so it returns a useful result text for a step.
4. **Task 4 – Reflection:** Write the kind of *logical statements* you would draft before unit tests.
5. **Task 5 – Reusable helper:** Run a tiny helper that uses your components, then think about refactoring it.

```
from typing import List
```

Task 1 – Planner

Declarative rule

> Given a goal text, `create_plan(goal)` must return a **list of steps** (step descriptions). For this demo, a simple valid plan is a list that contains the goal text as its only step.

Example behaviour:

```
planner.create_plan("Onboard new engineer")
# one simple valid result:
["Onboard new engineer"]
```

Below is a `Planner` class.
Your job: fix the **return value** so it matches the rule above.

```
class Planner:
    """Planner component that turns a goal into a very simple plan."""

    def create_plan(self, goal: str) -> List[str]:
        """Return a list of step texts for the given goal."""
        steps = [goal]
        return "FIX_ME"  # TODO: return the steps list instead of "FIX_ME"
```

Task 2 – Memory

Declarative rule

> `add_memory` must accept a text value and return nothing (`None`).
> `get_context` must accept a query text and return a text string (it can be empty for now).

Example usage:

```
mem.add_memory("User likes dark mode")
context = mem.get_context("dark")
```

Below is a `Memory` class.

Your job: fix the **return values** so they match the rule above.

```python
class Memory:
    """Memory component that stores and retrieves text."""

    def add_memory(self, text: str) -> None:
        """Store the provided text for later use."""
        return "FIX_ME"  # TODO: this method should not return anything

    def get_context(self, query: str) -> str:
        """Return a text summary relevant to the query (can be empty for now)."""
        context = ""
        return 0  # TODO: return a text value instead of 0 (e.g. context or "")
```

Task 3 – Executor

Declarative rule

> An `Executor` must accept a step text and return a text description of the result.

Example behaviour:

```
executor.execute_step("Send welcome email")
# could return:
"[EXECUTOR] Completed: Send welcome email"
```

Below is an `Executor` class.

Your job: fix the **return value** so it matches the rule above.

```python
class Executor:
    """Executor component that runs a single step and returns a result text."""

    def execute_step(self, step: str) -> str:
        """Execute the given step and return a text description of the result."""
        result = f"[EXECUTOR] Completed: {step}"
        return 0  # TODO: return the result text instead of 0
```

Task 4 – Reflection: Thinking Like a Tester

Imagine you are writing unit tests **before** you write the full implementation.

Write one simple logical statement (in plain English) per component, for example:

- Planner: “If I give the planner a goal text, `create_plan` must return a list of step texts.”
- Memory: “If I add a memory text and ask for context with a query, `get_context` must return a text value (even if it is empty).”
- Executor: “If I give the executor a step text, `execute_step` must return a text value that describes the result and includes that step text.”

You can write your three statements in a separate notes file or directly in this markdown cell (in your own copy).

> In the answer key, you will see example unit test logic and one possible test function for each component.

Task 5 – Reusable Helper (Onboarding Example)

Now that you have three component interfaces (`Planner` , `Memory` , `Executor`), you can reuse them in a tiny helper.

Below is a small example that:

1. Creates a `Planner` and an `Executor` .
2. Builds a simple plan for an onboarding goal.
3. Executes each step and prints the result.

Later, you can copy this idea into a separate file and **refactor** it into a different helper, for example a *meal planner* using different data.

```
def run_onboarding_helper(role_type: str) -> None:
    """Run a tiny onboarding helper using Planner + Executor."""
    planner = Planner()
    executor = Executor()

    goal = f"Onboard new {role_type}"
    print(f"Goal: {goal}")

    steps = planner.create_plan(goal)
    print("Plan:", steps)

    print("\nExecuting steps:")
    for step in steps:
        result = executor.execute_step(step)
        print("-", result)


# Example: run the onboarding helper for an engineer.
run_onboarding_helper("engineer")
```

```
Goal: Onboard new engineer
Plan: FIX_ME

Executing steps:
- 0
- 0
- 0
- 0
- 0
- 0
```

Refactor idea – Meal planner (outside this notebook)

As a follow-up exercise in your project files:

1. Copy the `run_onboarding_helper` function into a new file, for example `meal_helper.py`.
2. Change the naming and data to match a meal-planning helper (e.g. `meal_type` instead of `role_type`).
3. Reuse the **same** `Planner`, `Memory`, and `Executor` classes.

Key idea:

> Define the components **once** (Planner, Memory, Executor) and reuse them with different data to create different helpers.