Building a Simple Agent Framework 1

We are designing our agents in terms of GAME. Ideally, we would like our code to reflect how we design the agent, so that we can easily translate our design into an implementation. Also, we can see that the GAME components are what change from one agent to another while the core loop stays the same. We would like to design a framework that allows us to reuse as much as possible while making it easy to change the GAME pieces without affecting the GAME rules (e.g., the agent loop).

At first, it will appear that we are adding complexity to the agent — and we are. However, this complexity is necessary to create a framework that is flexible and reusable. The goal is to create a framework that allows us to build agents quickly and easily without changing the core loop. We are going to look at each of the individual GAME component implementations and then how they fit into the overall framework at the end.

G - Goals Implementation

First, let's create a simple goal class that defines what our agent is trying to accomplish:

```
@dataclass(frozen=True)
class Goal:
    priority: int
    name: str
    description: str
```

Goals will describe what we are trying to achieve and how to achieve it. By encapsulating them into objects, we can move away from large "walls of text" that represent the instructions for our agent. Additionally, we can add priority to our goals, which will help us decide which goal to pursue first and how to sort or format them when combining them into a prompt.

We broadly use the term "goal" to encompass both "what" the agent is trying to achieve and "how" it should approach the task. This duality is crucial for guiding the agent's behavior effectively. An important type of goal can be examples that show the agent how to reason in certain situations. We can also build goals that define core rules that are common across all agents in our system or that give it special instructions on how to solve certain types of tasks.

Now, let's take a look at how we might create a goal related to file management for our agent:

```
from game.core import Goal

# Define a simple file management goal
file_management_goal = Goal(
    priority=1,
    name="file_management",
    description="""Manage files in the current directory by:
    1. Listing files when needed
    2. Reading file contents when needed
    3. Searching within files when information is required
    4. Providing helpful explanations about file contents"""
)
```

A - Actions Implementation with JSON Schemas

Actions define what the agent can do. Think of them as the agent's toolkit. Each action is a discrete capability that can be executed in the environment. The action system has two main parts: the Action class and the ActionRegistry.

The actions are the interface between our agent and its environment. These are descriptions of what the agent can do to affect the environment. We have previously built out actions using Python functions, but let's encapsulate the parts of an action into an object:

```
class Action:
    def __init__(self,
                 name: str,
                 function: Callable,
                 description: str,
                 parameters: Dict,
                 terminal: bool = False):
        self.name = name
        self.function = function
        self.description = description
        self.terminal = terminal
        self.parameters = parameters
```

```
    def execute(self, **args) -> Any:
        """Execute the action's function"""
        return self.function(**args)
```

At first, it may not appear that this is much different from the previous implementation. However, later, we will see that this makes it much easier to create different agents by simply swapping out the actions without having to modify the core loop.

When the agent provides a response, it is going to return JSON. However, we are going to want a way to lookup the actual object associated with the action indicated by the JSON. To do this, we will create an `ActionRegistry` that will allow us to register actions and look them up by name:

```
class ActionRegistry:
    def __init__(self):
        self.actions = {}

    def register(self, action: Action):
        self.actions[action.name] = action

    def get_action(self, name: str) -> [Action, None]:
        return self.actions.get(name, None)

    def get_actions(self) -> List[Action]:
        """Get all registered actions"""
        return list(self.actions.values())
```

Here is an example of how we might define some actions for a file management agent:

```
def list_files() -> list:
    """List all files in the current directory."""
    return os.listdir('.')

def read_file(file_name: str) -> str:
    """Read and return the contents of a file."""
    with open(file_name, 'r') as f:
        return f.read()

def search_in_file(file_name: str, search_term: str) -> list:
    """Search for a term in a file and return matching lines."""
```

```
    results = []
    with open(file_name, 'r') as f:
        for i, line in enumerate(f.readlines()):
            if search_term in line:
                results.append((i+1, line.strip()))
    return results

# Create and populate the action registry
registry = ActionRegistry()

registry.register(Action(
    name="list_files",
    function=list_files,
    description="List all files in the current directory",
    parameters={
        "type": "object",
        "properties": {},
        "required": []
    },
    terminal=False
))

registry.register(Action(
    name="read_file",
    function=read_file,
    description="Read the contents of a specific file",
    parameters={
        "type": "object",
        "properties": {
            "file_name": {
                "type": "string",
                "description": "Name of the file to read"
            }
        },
        "required": ["file_name"]
    },
    terminal=False
))

registry.register(Action(
    name="search_in_file",
    function=search_in_file,
    description="Search for a term in a specific file",
    parameters={
        "type": "object",
        "properties": {
            "file_name": {
```

```
                "type": "string",
                "description": "Name of the file to search in"
            },
            "search_term": {
                "type": "string",
                "description": "Term to search for"
            }
        },
        "required": ["file_name", "search_term"]
    },
    terminal=False
))
```

M - Memory Implementation

Almost every agent needs to remember what happens from one loop iteration to the next. This is where the Memory component comes in. It allows the agent to store and retrieve information about its interactions, which is critical for context and decision-making. We can create a simple class to represent the memory:

```
class Memory:
    def __init__(self):
        self.items = []  # Basic conversation histor

    def add_memory(self, memory: dict):
        """Add memory to working memory"""
        self.items.append(memory)

    def get_memories(self, limit: int = None) -> List[Dict]:
        """Get formatted conversation history for prompt"""
        return self.items[:limit]
```

Originally, we just used a simple list of messages. Is it worth wrapping the list in this additional class? Yes, because it allows us to add additional functionality later without changing the core loop. For example, we might want to store the memory in a database and dynamically change what memories the agent sees at each loop iteration based on some analysis of the state of the memory. With this simple interface, we can create subclasses that implement different memory strategies without changing the core loop.

One thing to note is that our memory always has to be represented as a list of messages in the prompt. Because of this, we provide a simple interface to the memory that returns the last N messages in the correct format. This allows us to keep the memory class agnostic to how it is used. We can change how we store the memory (e.g., in a database) without changing how we access it in the agent loop. Even if we store the memory in a complicated graph structure, we are still going to need to pass the memories to the LLM as a list and format them as messages.

E - Environment Implementation

In our original implementation, we hardcoded our "environment" interface as a series of if/else statements and function calls. We would like to have a more modular interface that allows us to execute actions without needing to know how they are implemented or have conditional logic in the loop. This is where the Environment component comes in. It serves as a bridge between the agent and the outside world, executing actions and returning results.

```
class Environment:
    def execute_action(self, action: Action, args: dict) -> dict:
        """Execute an action and return the result."""
        try:
            result = action.execute(**args)
            return self.format_result(result)
        except Exception as e:
            return {
                "tool_executed": False,
                "error": str(e),
                "traceback": traceback.format_exc()
            }

    def format_result(self, result: Any) -> dict:
        """Format the result with metadata."""
        return {
            "tool_executed": True,
            "result": result,
            "timestamp": time.strftime("%Y-%m-%dT%H:%M:%S%z")
        }
```