Contrato del Agente en el Framework

Tècniques Avançades de Programació

Alexandre Vicario Sabaté

Nota: esta práctica se realizaba, en un principio, en conjunto con Robert Alejandro Ionita Maglan, pero recientemente me notificó que, se le había informado por parte del profesorado, que esta asignatura no convalidaba para su proceso de cambio de carrera, así que decidió NO presentarse. El trabajo es íntegramente mío y lo presento en solitario a pesar de que fuera una práctica pensada para parejas.

Contrato del Agente en el Framework

1. Funcionamiento del Framework

El framework está compuesto por tres clases principales:
`AgentManager`: Carga y gestiona los agentes, además de procesar comandos.
`ChatListener`: Escucha los mensajes del chat de Minecraft y los envía al `AgentManager`.
`Agent`: Clase base que deben heredar todos los agentes para comunicarse con la API.

Cabe recalcar que toda la funcionalidad del Framework ya está completamente implementada y no requiere ningún tipo de modificación. El usuario debe limitarse a la creación de Agentes usando la API y la clase proporcionadas.
De todas maneras, se recomienda entender el funcionamiento del Framework para poder entender cómo diseñar un agente de la forma correcta.

1.1 `IoC` (Inversion of Control en el Framework)

El framework sigue el patrón de **Inversión de Control (IoC)**, que implica que el control sobre el flujo del programa es invertido respecto a los modelos tradicionales de programación.

En lugar de que los agentes gestionen su ciclo de vida y ejecución, el framework se encarga de invocar el código de los agentes en el momento adecuado

`> En este caso, el framework llamará a los métodos de los agentes cuando se reciba el comando vinculado a la acción del agente por el chat de Minecraft.`

Todas las clases descritas en este apartado usan, de una forma u otra, Inversion of Control, es decir, son responsables de controlar el flujo de vida de los agentes, e invocarlos cuando son requeridos.

1.2 `AgentManager` (Gestión de Agentes y Comandos)

¿Qué hace?

· Carga dinámicamente los agentes desde la carpeta `Agents/`.
· Registra los comandos de cada agente.
· Procesa los mensajes del chat y ejecuta comandos si son válidos.
· Permite recargar agentes sin reiniciar el framework.

¿Cómo Funciona?

> Carga de Agentes (`import_agents`)

- Escanea la carpeta `Agents/` en busca de archivos `.py` que comiencen con `AGNT_`.
- Importa el módulo y obtiene la clase del agente (si filename = classname).
- Instancia cada agente y obtiene sus comandos (`cmd_*`).
- Registra los comandos en un diccionario para ejecutar métodos según el input del usuario.

> Procesamiento de Mensajes (`process_message`)

- Identifica el comando enviado en el chat y lo busca en la lista de comandos registrados.
- Si es un comando válido, ejecuta el método correspondiente con los parámetros recibidos.
- Si no es válido, informa al usuario en el chat.

> Comandos Especiales del `AgentManager`

Dentro del `AgentManager`, hay varios comandos predefinidos que permiten al usuario interactuar con el sistema de una manera más flexible. Estos comandos no están vinculados a agentes específicos, sino que sirven para gestionar el estado del framework o para ayudar al usuario.

- **CMDLIST / COMMANDLIST**
  - **Función**: Muestra todos los comandos disponibles en el framework.
  - **Descripción**: Cuando se envía uno de estos comandos en el chat de Minecraft, el `AgentManager` responde con una lista de los comandos registrados. Esta lista incluye tanto los comandos asociados a los agentes como aquellos comandos especiales predefinidos del sistema.

- **HELP**
  - **Función**: Muestra una lista de los comandos disponibles con una breve descripción de cada uno.
  - **Descripción**: Cuando se ejecuta este comando, el `AgentManager` responde proporcionando una lista de los comandos más comunes, incluyendo `CMDLIST/COMMANDLIST`, `RELOAD`, `END/STOP`, y `HELP`. Este comando es útil para recordar cómo interactuar con el sistema y qué acciones están disponibles.

- **RELOAD**
  - **Función**: Recarga los agentes y sus comandos desde la carpeta `Agents/`.
  - **Descripción**: Este comando es útil cuando se desean actualizar o añadir nuevos agentes sin tener que reiniciar todo el framework. Cuando se ejecuta, el `AgentManager` limpia la lista de agentes y comandos registrados, vuelve a escanear la carpeta de agentes y recarga los agentes disponibles.

1.3 `ChatListener` (Escucha de Mensajes en el Chat)

¿Qué hace?

· Escucha continuamente los mensajes en el chat de Minecraft.
· Envía cada comando recibido al `AgentManager` para su procesamiento.
· Implementa un timeout opcional para detenerse si no hay actividad.

¿Cómo Funciona?

> Bucle de Escucha (`listen_for_chat_commands`)

- Se ejecuta en un loop infinito mientras `listening = True`.
- Captura los mensajes en el chat de Minecraft.
- Si se detecta `STOP` o `END`, detiene la ejecución.
- Si se activa el timeout y no hay actividad por `X` segundos, cierra el framework

> Comando Especial del `ChatListener`.

- **END / STOP**
  - **Función**: Cierra la escucha de comandos >> Cierra el Framework
  - **Descripción**: Cuando se ejecuta cualquiera de estos comandos (`END` o `STOP`), la escucha de comandos se detiene, y eso induce en un cierre del framework de manera controlada, cerrando conexiones y limpiando recursos. Es la forma de terminar la ejecución del framework cuando ya no se desea más interacción.

1.4 `Agent` (Base para los Agentes)

¿Qué hace?

· Permite que los agentes hereden y usen la API de `mcBotAPI`.
· Importa automáticamente los métodos `cmd_*` como comandos a `AgentManager`.
· Gestiona la ejecución de comandos y acciones.
· Es el encargado de la comunicación entre agentes y Framework, con toda la lógica ya implementada.

¿Cómo Funciona?

> Registro de Comandos (`import_`)

- Busca en la clase métodos que comiencen con `cmd_`.
- Los guarda en una lista con el formato:
  `("COMMANDNAME", self.cmd_commandName)`

  > El `AgentManager` usará esta lista para ejecutar las acciones de los comandos.

> Ejecución de Comandos (`execute_cmd`)

- Si el usuario pasa `HELP` tras un comando, muestra el mensaje de ayuda.
- Si se proporciona un parámetro, busca una acción y la ejecuta.
- Si no hay acción específica, intenta ejecutar la acción `DEFAULT` con los parámetros.
- Si no existe `DEFAULT`, informa al usuario de un error.
- Si los parámetros son inválidos, también informa al usuario de error.

2. Definición de Agentes

Los agentes serán los encargados de establecer comandos que ejecuten acciones usando la API.
Para una guía más visual y detallada, consultar la *Guía de creación de Agentes*.

2.1 Métodos de Comando (`cmd_*`)

Los métodos de comando deben seguir esta estructura:

- Todos los comandos deben comenzar su método con `cmd_`.
- La cabecera será **`cmd_*(self, *args)`**
- `*args` permite recibir los parámetros del comando.
- `help_message` debe explicar el uso del comando.
- `actions` define qué métodos se ejecutan según el parámetro recibido.
- Siempre se debe llamar a `self.execute_cmd(help_message, actions, *args)` al final del método.

2.2 Métodos Asociados a Actions

Los métodos vinculados en `actions` deben seguir la misma estructura:

- Su cabecera tendrá de parámetros `(self, *args)`, incluso si NO los requieren.
- Son invocados automáticamente por el **`AgentManager`** según el parámetro recibido.

2.3 Métodos Adicionales (Internos o Auxiliares)

Estos métodos no están vinculados a un comando ni a `actions`, sino que apoyan la ejecución de lógica interna.

- Reciben parámetros personalizados en lugar de `*args`, si es que los requieren.
- Son métodos auxiliares para los métodos de `actions`, no accesibles directamente vía comando.
- Pueden ser reutilizados en varias suobperaciones o comandos de la misma clase.

3. Integración con la API

3.1 Uso de la API (`mcBotAPI`)

Todos los agentes heredan automáticamente los métodos de `mcBotAPI`, a través de su herencia de **`Agent`**, por lo que no necesitan implementar funciones para interactuar con Minecraft.

- Se deben usar los métodos de `mcBotAPI` en lugar de escribir código redundante, en la medida de lo posible.
- Se accede a los métodos de la API con `self.api.method()`.
- Los métodos API se deben evitar en los métodos **`cmd_*`**

Para ver las funciones de la API de manera detallada, revisar la Guía de Creación de Agentes.

`-> Se han proporcionado tests unitarios que verifican la correcta ejecución de todos los métodos de la API.`

4. Instanciación del Framework

El framework se inicializa a través del archivo **FrameworkLauncher.py**, que se encarga de conectar con el servidor de Minecraft, crear una instancia del **AgentManager** y comenzar la escucha de comandos en el chat.

```python
# FrameworkLauncher.py
from MyAdventures.mcpi.minecraft import Minecraft
from Framework.agent_manager import AgentManager

print('Framework launched')

# Connect to the Minecraft server
mc = Minecraft.create()

# Create an instance of the AgentManager
agent_manager = AgentManager(mc)

# Import all agents and show their available commands
agent_manager.start(False)

# It can also be launched with automatic timeout -> agent_manager.start(True, timeoutInSeconds)

print('Framework stopped')
```

4.1 Inicialización

> Pasos de Inicialización:

· Se conecta al servidor de Minecraft con `Minecraft.create()`.
· Se crea una instancia de `AgentManager`, pasándole la conexión `mc`.
· Se inicia el framework llamando a `agent_manager.start(False)`.

4.2 Configuración de Timeout Automático

Por defecto, el framework permanece activo de forma indefinida hasta que se detenga manualmente en el chat con `STOP` o `END`.

Si se desea configurar un cierre automático tras un tiempo sin actividad, basta con modificar la línea:

```
agent_manager.start(False)
```

Por esta otra:

```
agent_manager.start(True, X)
```
donde X será el número de segundos para el timeout

> Si no se recibe ningún mensaje en el chat en X segundos, el framework se detendrá automáticamente.
> Si se envían comandos dentro del tiempo, el contador se reinicia