



Departament d'Enginyeria  Informàtica i Matemàtiques  UNIVERSITAT ROVIRA I VIRGILI	Tècniques avançades de programació
	TAP
	Curso 24/25
	Segunda Convocatoria
	Pràctica: Minecraft Agent Framework

Pràctica TAP (Minecraft Agent Framework)

EQUIPO:

Mireya Marín Calvo
Nayra Li Luna Rodríguez

Index

Index.....	1
Minecraft Agent Framework.....	2
Clase AgentFramework.....	2
Clase Agent.....	2
Implementación de Agentes.....	3
Software Testing.....	4
MAIN.....	4
Conclusión.....	5

Minecraft Agent Framework

Implementamos fichero AgentFramework.py, un **framework de agentes** en el que varios **agentes** pueden ser registrados, ejecutados en **threads independientes** y detenidos de forma controlada. El uso de hilos es esencial para que los agentes puedan operar de manera simultánea, sin bloquearse entre ellos.

Clase AgentFramework

Gestiona el ciclo de vida de múltiples agentes. Permite registrar, ejecutar y detener agentes de manera centralizada.

Métodos:

- **register_agent**: Registra un agente en el framework, añadiéndolo a la lista de agentes.
- **unregister_agent**: Elimina un agente del framework.
- **run_agents**: Inicia todos los agentes en **hilos separados**.
 - o Crea un **hilo** para cada agente mediante `threading.Thread(target=agent.start)` y los inicia con `thread.start()`.
 - o Luego, espera a que todos los hilos finalicen con `thread.join()`, lo que asegura que el proceso principal no termine antes que los agentes.
- **stop_agents**: Detiene a todos los agentes llamando a su método `stop()`.

Clase Agent

Es el contrato que define la estructura básica que debe seguir cada agente. Los agentes pueden ejecutarse de forma independiente y ser detenidos de manera controlada.

Métodos:

- **start**: Inicia el agente y establece su estado de ejecución (`self.running = True`).
- **run**: El método que el agente ejecutará continuamente en un bucle mientras `self.running` sea `True`.
- **stop**: Detiene el agente estableciendo `self.running = False`.

Implementación de Agentes

1. **EmotionBot:** Interactúa con los jugadores a través del chat, detectando emociones positivas o negativas en sus mensajes. Responde de manera apropiada a las emociones detectadas (por ejemplo, "I'm happy for you!" o "I'm sorry to hear that.").
Programación Funcional: Utiliza funciones como `any()` para verificar si alguna palabra en el mensaje pertenece a las listas de emociones positivas o negativas, lo que es un enfoque funcional para procesar los mensajes.
2. **HouseBot:** Construye una casa en Minecraft a partir de un conjunto de bloques, y permite cambiar el material de las paredes entre madera y cristal. Usa métodos como `build_house` y `change_house` para gestionar la construcción.
Programación Reflectiva: `change_house` utiliza el `setattr()` para cambiar el material del bloque de las paredes a madera o cristal.
3. **InsultBot:** Envía insultos aleatorios cada cierto tiempo a los jugadores. Es un agente que genera comentarios de manera irónica, asegurándose de que no contengan caracteres incompatibles.
4. **OracleBot:** Responde a preguntas comunes de los jugadores, como "¿Cómo estás?" o "¿Cuál es el sentido de la vida?", ofreciendo respuestas predefinidas. Si no entiende una pregunta, responde con una frase de disculpa.
5. **TimeBot:** Mide y muestra el tiempo total que un jugador ha pasado en el servidor. Cada minuto, envía un mensaje al chat con el tiempo transcurrido.
6. **TNTBot:** Coloca TNT cerca de los jugadores y lo activa, generando una explosión. Interactúa con el entorno.
7. **Person (welcome):** Saluda al jugador, obteniendo su nombre. Este bot utiliza **programación reflectiva**, al captar el nombre del usuario como atributo de la clase. Además, en la función `greet()`, `self.ready_event.set()` utiliza un evento de sincronización.

Cada agente tiene un ciclo de vida que se controla en el `AgentFramework.py` mediante los métodos `start()` (inicia el agente), `run()` (ejecuta la tarea principal), y `stop()` (detiene al agente). Además, algunos agentes tienen funcionalidades específicas, como responder a eventos del chat o realizar tareas repetitivas.

Software Testing

En esta práctica, utilizamos **unittest** para probar que los bots funcionan correctamente. Cada bot tiene pruebas que simulan interacciones con el usuario para verificar sus respuestas.

Para hacer las pruebas más precisas, usamos **unittest.mock** y **MagicMock** para simular la interacción con Minecraft sin necesitar una instancia real del juego. Por ejemplo, en el caso del *OracleBot*, verificamos que respondiera correctamente a diferentes preguntas y a la instrucción "bye".

1. **Mock de Minecraft:**

`self.mc_mock = MagicMock()` crea un objeto simulado de Minecraft.

2. **Simulación de mensajes:**

`self.mc_mock.events.pollChatPosts = MagicMock(return_value=[event])` simula que `pollChatPosts()` devuelve un mensaje con el texto "hello".

3. **Verificación:**

`self.mc_mock.postToChat.assert_any_call("Hi there!")` verifica que el bot envíe el mensaje correcto ("Hi there!") cuando reciba "hello".

Usamos **MagicMock** para controlar lo que devuelven los métodos sin interactuar con Minecraft real, facilitando las pruebas del bot.

Integramos las pruebas con GitHub Actions, configurando un archivo **test.yml** para ejecutar las pruebas y generar un reporte de cobertura con *coverage.py*. También usamos un archivo *requirements.txt* para asegurar que todas las dependencias necesarias estén presentes. Esto ayuda a mantener el código probado y libre de errores.

MAIN

Para que se ejecuten los bots se ha implementado una clase main, que funciona como un menú. Al inicio se importan las librerías y los bots.

Se crea una clase **CommandHandler** que permite registrar, ejecutar, y mostrar comandos personalizados.

- **`_init_(self, mc)`**

Se usa como constructor. Crea la conexión con Minecraft y crea un diccionario (`self.commands`) donde guardar los comandos

- **`register_command(self, name, function, description="No description")`**

Añade al diccionario el nuevo comando. Si no se le pasa descripción se añade "No description" por defecto.

- **`execute_command(self, name, *args)`**

Se ejecuta el comando que tenga como nombre el pasado como argumento. Si no existe se lanza un mensaje de error.

- **`show_help(self)`**

Ejecuta una iteración imprimiendo todos los comandos con su descripción.

Clase main():

Se crea una instancia de CommandHandler.

Se crea un thread para que los bots se puedan ejecutar independientemente.

Para que sea el propio framework el que ejecuta los bots hacemos lo siguiente:

- Inicializar los agentes(bots) y los registrarlos en el framework para que puedan interactuar en el mundo de Minecraft.
- Llamar a **framework.run_agents()** para comenzar a ejecutar todos los agentes registrados y permite que escuchen los comandos.
- Uso de **ready_event** para esperar a que se acabe de ejecutar el bot **welcome** y los comandos de la clase **CommandHandler**. Una vez finalizados, ya se pueden ejecutar los bots.

Se le añaden los comandos de la siguiente manera:

```
handler.register_command("insultbot", lambda: InsultBot(mc).start_insulting(1), "Start InsultBot")
```

La función se pasa como una lambda para evitar crear código extra.

Se imprime por pantalla el mensaje:

```
Welcome to Minecraft! Type '--help' to see available commands.
```

Finalmente se realiza un bucle infinito donde se espera la respuesta del jugador. Con **mc.events.pollChatPosts()** se captura la respuesta y se trata para extraer el comando y sus argumentos. Luego se ejecuta el pedido.

Conclusión

En esta práctica, se ha desarrollado un framework de agentes que permite gestionar y ejecutar múltiples agentes de manera concurrente en un entorno de Minecraft. Utilizando hilos (threads), cada agente puede operar de forma independiente, lo que mejora la eficiencia y facilita la interacción simultánea con el entorno. Además, se han implementado diferentes tipos de agentes, algunos con programación funcional y reflexiva, demostrando la versatilidad y flexibilidad del sistema para adaptarse a diferentes tareas y comportamientos.

Enlace repositorio GitHub

Nuestro trabajo está disponible en el siguiente enlace: <https://github.com/MIREEYA/TAP.git>