Guía de Creación de Agentes

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.

Guía de Creación de Agentes

En este framework, los agentes son módulos que interactúan con el sistema mediante comandos de chat en Minecraft. Para crear un agente, se debe heredar de la clase base `Agent`, ubicada en `Framework/Utils/agent.py`. Esta clase base ya incluye toda la lógica necesaria para la interacción con el `AgentManager` y la ejecución de acciones sin necesidad de implementar métodos adicionales para la comunicación.

1. Formato del Agente

Cada agente debe ser implementado en un archivo cuyo nombre siga el formato `AGNT_className.py`, donde `className` es el nombre de la clase que define el agente.
Todos los agentes se ubicarán en la carpeta Agents.
Esta carpeta, además, contiene un agente template, el mismo del que se hablará en esta guía.

2. Estructura de un Agente

2.1 Métodos `cmd_*`

Cada agente debe tener métodos que definan los comandos que manejará. Los métodos deben comenzar con el prefijo `cmd_`, seguido del nombre del comando en minúsculas.
No debe haber nombres de comandos repetidos.
Ejemplos:

- `cmd_jump` → Se activará con el comando `JUMP`
- `cmd_walk` → Se activará con el comando `WALK`

2.2 Estructura métodos `cmd_*`

Cada comando tendrá asociados unos parámetros de acción. La estructura básica de un método **`cmd_*`** debe seguir la siguiente forma:

1. **Comando (`cmd_*`)**: Método que maneja el comando recibido a través del chat en Minecraft.
2. **Mensaje de Ayuda (`help_message`)**: Lista que contiene las instrucciones de uso del comando y sus parámetros.
3. **Acciones (`actions`)**: Diccionario que mapea los posibles parámetros del comando a métodos específicos que ejecutarán la acción correspondiente.
4. **Llamar** a **`execute_cmd`**

2.3 Definición de mensaje de ayuda

El mensaje de ayuda se mostrará en caso de que se ejecute COMANDO + HELP.
Debe proporcionar una descripción breve y concisa del funcionamiento del comando y los

parámetros que acepta, además de suboperaciones.
Para ver el formato, ver apartado 2.6.

2.4 Definición de `actions`

Actions formará una tupla `"COMANDO": método_asociado`.

- **`DEFAULT`**: Acción predeterminada que se ejecuta cuando no se pasan parámetros adicionales.
- **Comandos personalizados**: Si se pasa un parámetro personalizado, como `CUSTOM`, se ejecuta la acción asociada a ese parámetro

Para ver el formato, ver apartado 2.6.

2.5 Método `execute_cmd`

El método `execute_cmd` es utilizado para ejecutar la acción correspondiente a un comando. La lógica del comando pasará como parámetros el `help_message` (mensaje de ayuda) definido anteriormente al igual que el diccionario de `actions`, que mapea los parámetros posibles del comando a métodos.
La llamada se realizará de la siguiente forma SIEMPRE al final del método **`cmd_*`**:
`self.execute_cmd(help_message, actions, *args)`

2.6 Ejemplo `cmd_*` completo

El método **`cmd_*`** debe seguir esta estructura genérica:

```
def cmd_fake1(self, *args):
    """
    Command header
    """
    help_message = [
        "-> FAKE1 {PARAMETERS} Command description.",
        "* PARAMETER1: Parameter description.",
        "* PARAMETER2: Parameter description."
    ]

    actions = {
        "DEFAULT": self.default_mode, # executed when NO parameters
        "CUSTOM": self.custom_mode # executed if parameter = CUSTOM
    }

    self.execute_cmd(help_message, actions, *args)
```

3. Funcionamiento Agente

3.1 Comando sin parámetros

Si el comando no recibe parámetros adicionales, se ejecutará la acción predeterminada (**`DEFAULT`**) asociada al comando, si es que está definida.

3.2 Comando con parámetro

Si se pasa un parámetro, se ejecutará la acción correspondiente. Por ejemplo, si el comando `FAKE1` recibe el parámetro `CUSTOM`, se ejecutará la acción `custom_mode`.

ATENCIÓN: si no se pasa un parámetro y no hay salida **`DEFAULT`**, se mostrará un mensaje de error por el chat de Minecraft. Se mostrará otro si el parámetro pasado es erróneo.

4. Comunicación con el Framework

La clase `Agent` ya tiene implementada toda la lógica necesaria para comunicarse con el `AgentManager` y el sistema en general. No es necesario implementar métodos adicionales para registrar los comandos en el sistema o para gestionar la comunicación.

Cada vez que se ejecute el Framework, se hará una carga dinámica de todos los agentes que estén en la carpeta Agents. Para cada agente, `AgentManager` registrará sus comandos automáticamente y creará, también, una instancia. Como el agente se registrará automáticamente en el `AgentManager` al iniciar el Framework, podrá recibir comandos de chat en Minecraft, para ejecutar las acciones correspondientes, una vez el Framework haya terminado su inicialización (se mostrará un mensaje por el chat de Minecraft)

5. Recomendaciones para la Creación de Comandos

5.1 Formato de los Comandos

Los comandos deben ser simples y representar acciones específicas que el agente puede realizar. Se pueden definir suboperaciones dentro del diccionario `actions` solo si estas implican un cambio en la lógica de ejecución. Por ejemplo, un comando como `move` puede tener suboperaciones `forward` o `backward`, ya que estas modifican el comportamiento de la acción.

Sin embargo, si un comando requiere un valor adicional para su ejecución, este no debe registrarse como una suboperación en `actions`. En su lugar, debe vincularse a la acción `DEFAULT`. Por ejemplo, en el comando `walk 5`, donde `5` representa el número de bloques a caminar, `walk` ya define la lógica principal del comando, y el número de bloques debe tratarse como un parámetro interno en la ejecución de la acción `DEFAULT`, en lugar de una suboperación separada.

5.2 Manejo de Parámetros en Operaciones `DEFAULT`

Las operaciones definidas en `DEFAULT` pueden aceptar parámetros sin necesidad de registrarlos como suboperaciones en `actions`. El propio framework se encarga de diferenciar si los parámetros proporcionados corresponden a una suboperación registrada o si deben ser utilizados dentro de la ejecución normal del comando. Esto permite diseñar comandos más flexibles sin necesidad de registrar una gran cantidad de suboperaciones innecesarias.

5.3 Mantener la lógica de las operaciones fuera de `cmd_*`

El manejo de la lógica de las suboperaciones debe realizarse en métodos fuera del método **`cmd_*`**.
Luego estos métodos deben ser vinculados al comando correspondiente en el diccionario `actions` de **`cmd_*`**.

6. Interacción de los Agentes con la API

6.1 Acceso a los Métodos de la API

La clase `Agent`, de la cual heredan todos los agentes, ya hereda, a su vez, de la API `mcBotAPI`. Esto significa que cualquier agente que herede de `Agent`, es decir, TODOS los agentes, obtienen acceso a todos los métodos de la API sin necesidad de inicializarlos o registrarlos manualmente.

Para utilizar cualquier función de la API, basta con llamarla mediante `self.apimethod()`. Por ejemplo:

```
self.talk("Hola, Minecraft!")
```

Esta línea enviaría un mensaje al chat de Minecraft utilizando el método `talk` de la API.

6.2 Métodos de la API

A continuación, se describen los métodos disponibles en la API `mcBotAPI` y su funcionalidad:

talk(message)

Envía un mensaje al chat de Minecraft.

- **Parámetro:** `message` (str) - Mensaje a enviar.
- **Ejemplo de uso:** `self.talk("Hola, jugador!")`

move(x, y, z)

Mueve al jugador a una posición determinada.

- **Parámetros:** `x`, `y`, `z` (int) - Coordenadas a las que se moverá el jugador.
- **Ejemplo de uso:** `self.move(10, 64, 20)`

where()

Obtiene la posición actual del jugador.

- **Retorna:** Una tupla `(x, y, z)` con las coordenadas actuales del jugador.
- **Ejemplo de uso:**
  x, y, z = self.where()

  self.talk(f"Estás en: {x}, {y}, {z}")

get_player_orientation()

Obtiene la dirección a la que está mirando el jugador.

- **Retorna:** Un string con la dirección (`"NORTH"`, `"SOUTH"`, `"EAST"`, `"WEST"`).
- **Ejemplo de uso:**
  direction = self.get_player_orientation()

  self.talk(f"Estás mirando hacia {direction}")

arrange_positions()

Genera tes posiciones cercanas al jugador

- **Retorna:** Lista de coordenadas en formato `[(x1, z1, y1), (x2, z2, y2), (x3, z3, y3)]`.
- **Ejemplo de uso:**
  positions = self.arrange_positions()

  self.talk(f"Posiciones generadas: {positions}")

calculate_position(x, z, y, rotation, add_redstone, block_distance=5)

Calcula la posición final de un bloque basado en la orientación del jugador.

- **Parámetros:**
  - `x`, `y`, `z` (int) - Posición base.
  - `rotation` (str) - Dirección del jugador (`"NORTH"`, `"SOUTH"`, `"EAST"`, `"WEST"`).
  - `add_redstone` (bool) - Si se debe agregar una antorcha de redstone.
  - `block_distance` (int, opcional) - Distancia a la que se colocará el bloque (default: `5`).
- **Retorna:** Coordenadas `(x, z, y, xTorch, zTorch)`.

-> `xTorch, zTorch` corresponden a la posición de la antorcha si el booleano era True

- **Ejemplo de uso:**
  x, z, y, xTorch, zTorch = self.calculate_position(10, 20, 64, "NORTH", True)

place_block(x, z, y, xTorch, zTorch, block_id, add_redstone, support_block_id=None, activate_on_place=False)

Coloca un bloque en la posición deseada.

- **Parámetros:**
  - `x`, `y`, `z` (int) - Posición del bloque.
  - `xTorch`, `zTorch` (int) - Posición de la antorcha de redstone (opcional).
  - `block_id` (int) - ID del bloque a colocar.
  - `add_redstone` (bool) - Si se agregará una antorcha de redstone.
  - `support_block_id` (int, opcional) - Bloque de soporte debajo del principal.
  - `activate_on_place` (bool) - Si el bloque se activa al colocarlo.
- **Ejemplo de uso:**
  self.place_block(10, 20, 64, None, None, 1, False)

.