Project Milestone 7

CS 4500 Software Development

Due: Friday, March 26, 11:59pm

Submission: Create a release on GitHub tagged `p7`. Add a `p7-exec.zip` to the release, containing:

1. The `testManager` executable for the testing task
2. Any packages/modules necessary for the above executable to function
3. At least 4 tests in `tests/`

Here is a sketch of the `p7-exec.zip` layout:

```
|-- testManager
|-- tests
|   |-- 1-in.json
|   |-- 1-out.json
|   ...
...
```

Submit a ZIP with the following on Handins:

1. For the design task, the file `adversary.md` in `Snarl/Planning/`
2. For the programming task, any new or updated *relevant* source files under `Snarl/src`. Place the observer and player interfaces under `Common` and the implementations in separate directories.
3. For the testing task, a folder `Snarl/tests/Manager` with the test harness source code and your test suite for

Here is an example sketch of the Handins ZIP layout:

```
Snarl
|-- planning
|   `-- adversary.md
|-- tests
|   `-- Manager
|        |-- testManager.<ext>
|        |-- 1-in.json           <-- these files can be under an additional "tests"
|        ...                         directory, just like in the exec zip
`-- src
    |-- Common
    |   |-- Player.<ext>
    |   `-- Observer.<ext>
    |-- Game
    |   |-- GameManager.<ext>    <-- let's say I modified the game manager for this
    |   ...                          task
    |-- Observer
    |   |-- LocalObserver.<ext>
    |   ...
    `-- Player
        |-- LocalPlayer.<ext>
        ...
```

Design Task: Adversaries

Design an interface for SNARL adversaries. An adversary is similar to a Player, in that it interacts with the Game Manager on every turn. However there are a some differences:

- An adversary gets the full level information (comprised of rooms, hallways and objects) at the beginning of a level
- An adversary gets an update on all player locations, but only when it's about to make a turn

Note: the extent of what adversaries see might change if we determine they are too powerful.

Scope: We are looking for data definitions, signatures and purpose statements à la Fundies, or definitions and interface specifications approximating your chosen language (if it has such constructs). You are encouraged to use examples and/or diagrams to illustrate interaction between adversaries and the Game Manager

Programming Task: Player and Observer

Implement a local Player and a local Observer component

Player

Provide an interface for players (e.g., `interface` in Java, abstract class in Python) and an implementation of a local Player component adhering to that interface.

The Player component (some of you called it the User) should implement the following functionality:

1. Receive an update containing the player avatar's position in the level and the current state of their immediate surroundings. When an update is received, the update should be rendered to the user. This can be as simple as rendering the update to the console or using your GUI code to render a graphical representation. You can improve the interface later.
2. Provide a move to the Game Manager. A move can be empty, meaning the player stays put. The user should be interactively prompted to choose a destination. This can be as simple as meaningfully asking for the destination coordinates, using arrow keys, cardinal directions, etc. You can improve this later.

Observer

Provide an interface for observers (e.g., abstract class in Python, `interface` in Java) and implement a local Observer component. The functionality of this component is to receive updates from the Game Manager on every state change (after each actor moves) and to render them. The interface of this observer can be as simple as rendering the information to the console, or it can render a graphical representation in a GUI window.

Scope: We will look for good code design, readability, unit tests, and whether we can find the functionality we asked for above in the code.

Testing Task: Tracing the Game Manager

Implement a test harness, `testManager`, which exercises the game manager and rule checker over several turns of 1 to 4 players. The test harness we are implementing here is a *tracing* harness, meaning it collects a trace of interactions based on a simulated stream of inputs.

Reminder: Unless otherwise stated, JSON arrays are ordered.

Test Input

The test input for `testManager` has the following shape:

```
[ (name-list), (level), (natural), (point-list), (actor-move-list-list) ]
```

The following data definitions apply:

- A **`(name-list)`** is a list of 1 to 4 **`(string)`** and represents a list of player names to register.
- A **`(level)`** is as defined in Milestone 4. Assume that the input for the current testing task will always contain exactly one key and exactly one exit in the list of objects.
- A **`(natural)`** is a natural number represented as a JSON number (`0`, `1`, `2`, ...), determining the maximum number of turns to perform.
- A **`(point-list)`** is a list of initial player and adversary positions. Assume the positions are valid initial positions: they are pair-wise distinct, each in a room on a traversable tile. The list will be at least n elements long, where n is the length of the name list. The first n elements are player positions (in the same order as the name list), any subsequent elements are adversaries. You can populate the level with adversaries of your choosing. Adversaries will be stationary.

  This list simulates a random stream of valid initial positions.
- An **`(actor-move-list-list)`** is a list of **`(actor-move-list)`**. This list contains a list of moves for each registered player, in the same order as their names are listed in the **`(name-list)`**. As such, assume that the length of this list is always equal to the length of the list of names.

  Each list simulates an input stream of moves coming from the respective player.
- An **`(actor-move-list)`** is a list of **`(actor-move)`**
- An **`(actor-move)`** is the following JSON object:

  ```
  {
    "type": "move",
    "to": (maybe-point)
  }
  ```

- A **`(maybe-point)`** is one of the following:
  - **`null`** representing a skipped move
  - **`(point)`** as defined in Milestone 3, representing an *absolute* position within the level.

The expected test harness' behavior is as follows:

1. Register the n players named in the **`(name-list)`** with the Game Manager. The order of the list determines the order of the players.
2. Populate the given **`(level)`** with players and adversaries. The players' initial locations are given by the first n locations given in the **`(point-list)`**. Any elements after that are adversary positions.
3. Issue an initial update to each player in order.
4. Every time a player is supposed to move, take a move from their respective **`(actor-move-list)`** and have the Game Manager validate and perform the move. If the move is invalid, take the next move from the respective list. After a successful move, process the interactions.
5. After a player moves, issue an update to each player *still in the game*.
6. When any of the following occurs, stop and return the result.
   - The given number of turns was performed
   - One of the move input streams is exhausted
   - The level is over

Here is an example (eliding the level for brevity):

```
[ ["ferd", "dio"]
, (level)
, 5
, [ [3, 2], [5, 8], [14, 12] ]
, [ [ { "type": "move", "to": [4, 2] }
    , { "type": "move", "to": [5, 3] }
    ]
  , [ { "type": "move", "to": [3, 6] }
    , { "type": "move", "to": [4, 7] }
    , { "type": "move", "to": [6, 7] }
    ]
  ]
]
```

This input means:

1. Register `"ferd"` and `"dio"`
2. Place them at $(3, 2)$ and $(5, 8)$ in the given level
3. Place a stationary adversary at $(14, 12)$
4. Perform at most 5 turns.

5. In fact, only 2 turns will be performed:

    - Turn 1: `"ferd"` moves from $(3, 2)$ to $(4, 2)$ (1 down); `"dio"` tries to move from $(5, 8)$ to $(3, 6)$ (2 up, 2 left) which is invalid, so the next attempt from $(5, 8)$ to $(4, 7)$ (1 up, 1 left) will succeed
    - Turn 2: `"ferd"` moves from $(4, 2)$ to $(5, 3)$ (1 down, 1 right); `"dio"` moves from $(4, 7)$ to $(6, 7)$ (2 down)

Test Output

```
[ (state), (manager-trace) ]
```

Here, **`(state)`** is as defined in Milestone 5 and is the state that existed at the end of the sequence determined by the input.

A **`(manager-trace)`** is a JSON array of **`(manager-trace-entry)`**.

A **`(manager-trace-entry)`** is one of the following JSON arrays:

- **`[ (name), (player-update) ]`**, representing a player update (only updates for players still in the game should be included)
- **`[ (name), (actor-move), (result) ]`**, representing a response to a given move request of a given player

A **`(result)`** is one of:

- `"OK"`, meaning "the move was valid, nothing happened"
- `"Key"`, meaning "the move was valid, player collected the key"
- `"Exit"`, meaning "the move was valid, player exited"
- `"Eject"`, meaning "the move was valid, player was ejected"
- `"Invalid"`, meaning "the move was invalid"

A **`(player-update)`** is the following JSON object:

```
{
  "type": "player-update",
  "layout": (tile-layout),
  "position": (point),
  "objects": (object-list),
  "actors": (actor-position-list)
}
```

The interpretation of the fields is as follows.

- `"layout"` is a 5×5 array of tiles visible to the player, as outlined in Milestone 3 (layout) and Milestone 4 (visible area). For simplicity, represent "void" as "wall" (`0`).
- `"position"` is the player's *absolute* position
- `"objects"` are the objects in the player's field of view (order is *not* significant)

- `"actors"` are the actors in the player's field of view (order is *not* significant)

Scope: Your executable must run on the Khoury Linux VMs, accept JSON with the format given above on standard input and return results to standard output. We will be running our own tests and your team's tests through your testing harness. Ideally, your code from previous milestones shouldn't need to be modified extensively.