Java Text Adventure Engine – Logbook

Project Description

The player interacts with a virtual world via text commands.
The world reacts to the commands by altering its own state, and outputting a readable response text about the current state to the player.

If no command is found for the input, the system tries to help the user, by displaying similar commands, where the hamming distance to the entered command is the lowest.

There also is a help command, which shows all valid commands.

The commands also have parameters, which are entered after the command name:

Example: "`use door`" invokes the use command with the door object.

Implementation

- "Reverse" development:
  - First, usage scenario was created.
  - Then, the representation of a game in JSON format was created (see game/test.json).
  - Next, a class model representation of the JSON model was created.
  - Finally, the classes were implemented in Java, resulting of the game's class hierarchy.

Scenario

You have to get out of your apartment in the morning.

- You wake up in your room
- The room is dark. The following objects are present, but invisible:
  - blanket, light, legs, desk, drawer, wardrobe, room_key, clothes, door
- Get your blanket off your body (`use blanket`) -> Enables Light
- Switch on light (`use light`) -> Makes everything visible, except drawer, room_key, clothes
- Get up (`use legs/feet`) -> Enables desk, wardrobe, door
- Look at your desk (`look at desk`) -> Makes drawer visible
- Open drawer (`use drawer`) -> Makes key visible
- Pick up key (`Take key`)
- Open wardrobe (use wardrobe) -> Makes clothes visible
- Put on clothes (take clothes) -> adds clothes to inventory
- `Use key with door` -> Makes the hallway visible
- `Go to hallway`
- `Go to kitchen`

- The kitchen contains a `counter, stove, sink, bread box, toast, toaster, apt_key, fridge.` Everything except for the `counter` and `fridge` is invisible.
- Get some toast
    - `Use bread box`
    - `take toast`
- Put it in the `toaster: use toast with toaster`
- Switch the `toaster` on: `use toaster`
- The `key` is added to your inventory, as it jumps out of the toaster with your toast.
- `Go to hallway`
- `Use key with apartment door`
- You have successfully left the apartment ☺

Commands

Contains a "text" that is displayed to the player on interaction. Can contain a "do" (Action) and an "if" (Condition) property. If every condition is fullfilled, the "text" is displayed and the "do" block is executed.

JSON format: {
```
      "text": <string>,
      "do": [<action>],
      "if": [<condition>],
      "item": <string> //only for the use with command
}
```

The following commands exist:

- `use (with):`
    - Performs the use actions, if it is enabled and visible.
    - If there are 2 objecs given, check the second object for a valid use_with statement (use key with door) and perform its actions, it the right items were used.
- `look (at):`
    - Performs the look actions if the object is visible.
- `take/pick (up):`
    - Performs the take actions if the object is enabled and visible.
- `go (to/towards)/walk (to/towards):`
    - Switches to another Location, if it is visible and enabled.
    - Is not bound to an object.
- `inventory/items:`
    - Displays items that the player currently has in his/her inventory.
    - Is not bound to an object.

Actions

Are contained in the do property of a command and are performed if every condition of the if property is fullfilled. Actions have a "type", a list of "targets" (object ids).

JSON format:
```
{
      "type": <visible/enabled/take/remove/state>,
      "targets": [<string>],
      "value": <bool for visible/enable, string for state>
}
```

The following action types exists:

- `visible`
  - Sets every target objects visible attribute to the provided "value".
- `enable`
  - Sets every target objects enabled attribute to the provied "value".
- `take`
  - Removes the targets from their location and places them into the players inventory.
  - If targets are not set, the containing object is used as target.
- `remove`
  - Removes the target objects from the players inventory. It is no longer available in the game.
- `state`
  - Sets the state of the target objects to the provided "value".
  - If targets are not set, the containing object is used as target.

Conditions

A condition (if) property can be contained inside a command. It contains a list of conditions, which are checked from top to bottom. If all conditions are fulfilled, the actions of the current command are performed. Each condition has a "type" and an "else" property. The else property contains the text, that is displayed, if the current condition fails.

JSON format:
```
{
      "type": <item/state>,
      "items": [<string, only for item type>],
      "state": <string, only for state type>,
      "targets": [<string, only for state type>]
}
```

The following condition types exists:

- `item`: Contains an attribute called "items", which contains item ids which need to be in the players inventory.
- `state`: Checks, if the state of the "targets" objects matches a given "state"

Interaction with the Game

Interaction between the user and the game is done by putting a command into the game engine, and the game engine then changes the game state and prints changes to the user in form of text.

Changes that cause text being printed to the user:

- When a command is executed → the command's text
- When a command condition fails → the condition's else text
- When the current location changes → the name of the new location
- When the user wants to use/take an item which is disabled → information message
- When the user wants to go to a location which is not adjacent to the current location → information message
- When the user tries to interact with an item that is invisible or does not exist → error message

Refactoring

- Creates packages for actions, commands, conditions and entities
- Add Win action, so the game can actually be won
- Interactions were planned as singleton, later on multiple instances of the same interaction type were needed with different parameters → singleton pattern was dropped
- Added list of ignored phrases to the Interaction classes, to be able to, for example, type "go to" instead of "go", and achieve the same result

Design patterns

Template Method

Multiple abstract classes in the system use the template method pattern, to be able to manage different behaviors of items in the same way:

- **Interaction**: Different implementations of the interaction class override the abstract method `applyInternal()`, which is called when a command is entered in the console. The concrete implementations then knows, what to do with the entered text.
- **Condition:** Different implementations of the condition class override the abstract method `check()`, which is called when the condition needs to be verified. The concrete implementations then knows, what to check to verify the condition.
- **Action:** Different implementations of the action class override the abstract method `apply()`, which is called when an action should be applied to the game state, after the conditions have been verified. The concrete implementations then know, what to actually change in the game state.

External dependencies

- Jackson JSON parser
- Google Guava Utils

Test driven development

- Create Unit tests for game parsing
- Decide that unsuccessful parsing just returns a null - game

- Create unit test loading with empty JSON object "{}" -> expecting null
  - Add dependency to Jackson JSON parse API
  - Implement GameLoader class which calls Jackson's JSON parser with the Game class
  - Catches JsonParseException and JsonMappingException, returns null
  - Throw IllegalArgumentException in Game's constructor, when start location or locations map is null
- Create unit test loading a game with empty locations and a simple start location
  - Create constructor parameters in game for start location, start text and locations, and add Jackson annotations
- Create unit test loading a game with 1 location
  - Update BaseEntity class: add annotations for Jackson to enable JSON property mapping
  - Realize a mapping of entity ID -> entity for all entities (locations and items) must be present in the Game class -> create such a mapping in the constructor, and build it after the game with all its entities has been loaded from JSON.
- Create unit test loading external JSON file, which contains all types of game entities ("validGame.json")
  - two locations adjacent to each other, first location is start location and contains all commans: look, use, use_with, take
  - Create mapping annotations for Jackson in all game classes, to be able to load them from JSON.
  - Try to load game from JSON
  - For every error, fix it! -> mostly jackson problems, could all be handled easily
  - **Game can already be loaded from JSON**
  - Update unit test: testing if object hierarchy and all properties get instantiated correctly.
- Create GameState, which manages a game in process
- Create interaction package which contains
  - Interaction Manager, which takes the Players input and delegates it to the target interactions (use, take, go …)
  - Interaction, which is the base type for each interaction possible
  - Concrete interaction classes, which try apply Commands to

Console Interface

The last step in creating our game engine is a way for the user to actually interact with the game.

A standalone java program was created, which loads the game from a JSON file that is supplied as command line parameter.

This program is a simple console interface, which reads lines from the standard input, feeds them to the game engine. The system output is provided to the game engine, for it to print response messages to the user.