Project 1

Yin Hu, Mahnoor Irfan, Nicholas Synovic (Team 3)

Table of Contents

Functional Requirements

GitHub Repository

The GitHub repository where our work is centralized can be found here:

https://github.com/NicholasSynovic/homework_comp473-mars-Rover-kata

Kata Requirements

We have successfully implemented all the features described in this kata post. The following requirements were met:

- Plateau: We created a Plateau class to handle the position and possible moving spaces of the Rovers. This was implemented using a n-dimensional array provided by `numpy`,
- Rover: We created a Rover class to be instantiated per Rover that handles the current location of a Rover, the orientation of the Rover, and converting user commands into actionable tasks (i.e. rotation and movement),
- IO: We created a specific IO class to handle the user input and output. This class is meant to keep track of the number of Rover's instantiated and to communicate to each of them their specific commands, and
- Orientation: We created an Orientation data structure based on circularly linked lists to simplify the orientation updating of each Rover.

We've also added several quality-of-life features including:

- A barebones terminal user interface (TUI) that displays the "Mars plateau" and the positions of the Rovers in it,
- More descriptive I/O that labels the different relevant attributes of the Rovers, and
- A command line interface that allows for users to parametrically create different amounts of Rovers and terrain topologies.

Implementation Details

The following is a description of how the application operates:

We start by getting the user input from the terminal regarding the number of columns and rows that the plateau (i.e. grid) is composed of. Additionally, we get the number of Rovers that the user wants to interact with. Bound checks are implemented so that non-negative and non-zero values must be provided. If the number of Rovers is greater than the area of the grid, we through an error.

We then instantiate the grid of the proper size, followed by generating a list of unique Rover objects. Each Rover is assigned a unique ID starting from 1 and incrementing thereafter by 1. As part of their instantiation, each Rover is placed on the grid in order starting from (0, 0) (i.e. bottom left) and moving along the x-axis. If the x-axis bounds are reached, our algorithm wraps back around to (0, 1) and repeats this process. No two Rover's may overlap and given that we enforce that there is at least one free space per Rover on the grid through our previous contract, our algorithm can confidently place Rovers on a grid with 0 overlap.

After this, we instantiate our IO class and enter a while loop. This loop starts by printing the current state of the board, getting the commands for each Rover, and then executing each of the commands in order. The user inputs commands are the same order in which the Rovers were created. Rover commands are executed similarly. The User sees the current Rover's column and row position, as well as its orientation in cardinal directions. Commands are pre-processed by the IO class to remove all non-relevant characters (e.g. anything not "R", "L", or "M").

Rover movement is handled per Rover class. No handler was created to handle this as Rovers execute their commands sequentially and in order after the user has instructed all of them. Due to this batch processing, some commands may not be executed. In the case that a Rover is unable to complete its commands, it gracefully stops executing at its last completed command.

Nonfunctional Requirements

Implementation Language

We decided to go with `Python 3.10` as our implementation language due to its ease of use and review, simplistic project structure, and trivial documentation practices. Additionally, we found that this application did not require a high-performance or throughput language due to its single-threaded nature and limited design scope.

Tooling

We are leveraging `Git` as our VCS and `GitHub` as our cloud VCS. Additionally, we are leveraging `GitHub CI` to handle automatic testing when we push and merge our code as well as packaging our code using poetry, our build manager.

Locally, we are leveraging `Poetry` to manage our dependencies. Our tests are written using the `pytest` framework to make writing and executing tests simple. Our code is

checked for security, formatting, and code smells using `bandit`, `black`, and `flake8` respectively. These checks are done per commit using `pre-commit`.

We leverage `GitHub CI` to automatically do both `pytest` testing and `pre-commit` checks per commit on our main branch. As we are leveraging our main branch as our "release" branch, we believe that leveraging our compute credits for release related changes is more cost effective in the long run.

Architecture Decision Record (ADR)

To document our architecture decisions, we leveraged `Talo`. Our records can be found in our GitHub repository `docs/adr` directory. We utilized the following sources in understanding ADRs and picking our tooling:

- M. Keeling, "Love Unrequited: The Story of Architecture, Agile, and How Architecture Decision Records Brought Them Together," *IEEE Software*, vol. 39, no. 4, pp. 90–93, Jul. 2022, doi: 10.1109/MS.2022.3166266.
- U. Zdun, R. Capilla, H. Tran, and O. Zimmermann, "Sustainable Architectural Design Decisions," *IEEE Softw.*, vol. 30, no. 6, pp. 46–53, Nov. 2013, doi: 10.1109/MS.2013.97.
- "Architectural Decision Records (ADRs)." Accessed: Sep. 23, 2024. [Online]. Available: https://adr.github.io/

Object Oriented Programming Design Patterns

Whenever possible, we adhered to DRY (Don't Repeat Yourself) practices across our classes and methods. Additionally, we utilized the Pub-Sub and Factory patterns for sending commands to our Rovers from the IO class and instantiating our Rover classes respectfully. Furthermore, implemented separation of concerns for each of the classes as such:

- The IO class needs to know the current state of the Plateau, but cannot directly modify the Plateau's state in any direct manner
- Rovers operate on valid commands, but command validation is handled by the IO, rather than the Rover
- Rovers do not know the size of the Plateau that they are on, but do know their current coordinates and orientation