SW Craftsmanship Dojo
Developer EVOLUTION

Smelly Mars Rover

The orange belt graduation requires you to refactor a smelly Mars Rover library, strictly adhering to the TDD refactoring techniques we mastered into these two belts. Refactor it with a proper local and remote CI pipeline with related quality gates to prevent "smelly code" from reaching production.

Kata Objective:

Have a fully refactored Mars Rover library, with a declarative codebase, where all the tests are green, the coverage is 100%, and local and remote pipelines are green.

Backlog:

Read the codebase, and annotate the plan you've in mind to clean the code of all these smells in the backlog at the commencement of your test. Use the four stages of the Refactoring Priority Premise to organize your backlog. You can pull in the match with the Mikado technique if you're up for the fight.

Rules:

🛰 NASA is landing a robotic rover on a rectangular plateau on Mars. This plateau must be explored systematically so that the rover's on-board cameras can survey the terrain and send images back to Earth.

A rover's position is represented by:

- Two integers (X, Y) indicating its coordinates, and
- A letter indicating its current heading:

1) N (North)
2) E (East)
3) S (South)
4) W (West)

The plateau is divided into a grid, and the rover moves across it based on simple navigation commands. For example, a position of 0 0 N means the rover is at the bottom-left corner, facing North.

Examples:

NASA can send the following commands to the rover:

- L ➔ Turn 90 degrees left without moving from the current spot.
- R ➔ Turn 90 degrees right without moving from the current spot.
- M ➔ Move forward one grid point in the direction it is facing.

Important:

- Moving North from (x, y) goes to (x, y+1).
- Moving East from (x, y) goes to (x+1, y), and so on.

The input consists of:

- First parameter: The rover's starting position: two integers and a letter (e.g., 1 2 N).
- Second parameter: A string of movement instructions (e.g., LMLMLMLMM).

The rover processes all instructions in sequence.

Output: After executing all the commands, the rover reports its final coordinates and heading.

Starting Position	Instructions	Expected Output
1 2 N		1 2 N
1 2 N	L	1 2 W
1 2 W	L	1 2 S
1 2 S	L	1 2 E
1 2 E	L	1 2 N
1 2 N	R	1 2 E
1 2 E	R	1 2 S
1 2 S	R	1 2 W
1 2 W	R	1 2 N
1 2 N	M	1 3 N
1 2 E	M	2 2 E
1 2 S	M	1 1 S
1 2 W	M	0 2 W
1 2 N	LMLMLMLMM	1 3 N
3 3 E	MMRMMRMRRM	5 1 E

Scoring system

1. *CodeBase:* must be a public GitHub repository
2. *First push:* an empty repo generated by your cookie-cutter's yellow belt version
3. *Second push: the graduation codebase of your choice into the repo, cloning it from the cohort repo* graduation *directory. In the graduation folder README.md, add your full name and the link to your graduation public repository.*
4. *Time Box:* 3 hours (6 🍅) to code and refactor
5. *Solo mode:* no pair, no mob programming
6. *Helper:* transparent use of Gen-AI code assistant
7. *Backlog:* a BACKLOG.md file with the high-level refactoring plan
8. *Notes:* a NOTES.md file with your pomodoro flow
9. *Tech Debt:* a TECH_DEBT.md file with the catalog (checklist) of all the smells to fix
10. *Coverage:* 100% to be monitored in watch mode – Note: test must be green all the time
11. *Refactoring pillars [white & yellow belts] following the Refactoring Priority Premise:*
    - Code is readable, discovering the domain vocabulary
    - No hidden complexity, obvious code
    - Code complexity under 4 (four) cyclomatic (IDE, local, remote)
    - No linter or check style errors
    - No code smells
    - No DRY
    - Bonus: mitigate primitive obsession
    - Extra Bonus: explicit Mikado checklist
12. *Local Quality Gate:*
    - Git Hooks
    - Linter & Checkstyle
    - Complexity PlugIn
    - Conventional Commits: prevent to commit if not compliant with the rules
13. Remote Pipeline GitHub Actions):
    - Build pipeline – Continuous Build
    - Test Pipeline – Remote Quality gate (local replica)
    - CD pipeline: not necessary.
14. *Demo time:* run the tests. They should all pass. The code must be declarative (no procedural) and clean.

KATA - Setup phase:

You can add the automation you believe could be a helper to remove manual setup. For this reason, you can use any initialization, API test, or docker containerization script/s that lets you start coding with a "boilerplate" in one click (*cookie-cutter*). The *cookie-cutter* can initialize your development environment and the project. The cookie-cutter can smoke-test the final repository to assert the testing framework is working correctly. The cookie-cutter can inject all the CI/CD steps, like the quality gates, the Docker containerization, and the API test.

The *cookie-cutter* **CAN'T** inject code snippets relevant to solving your kata.

Feel free to use the right cookie-cutter, without over-engineering it!

```
                       .-.
                      |_:_|
                     /(_Y_)\
.                   ( \/M\/ )
 '.               _.'-/'-'\-'._
   ':           _/.--'[[[[]'--.\_
     ':        /_'  : |::"| :  '.\
       ':     //   ./ |oUU| \.'  :\
         ':  _:'..' \_|___|_/ :   :|
           ':.  .'  |_[___]_|  :.':\
            [::\ |  :  | |  :   ; : \
             '-'   \/'.| |.' \  .;.' |
             |\_    \  '-'   :       |
             |  \    \ .:    :   |   |
             |   \    | '.   :    \  |
             /       \   :. .;       |
            /     |   |  :__/     :  \\
           |  |   |    \:   | \   |   ||
          /    \  : :  |:   /  |__|   /|  ‘The Force is strong with this one’
          |     : : :_/_|  /'._\  '--|_\                         Darth Vader
          /___.-/_|-'   \  \
                         ‘_’
```