realfast

Parking lot management system - technical assessment

Nature of the game

We aim to understand your development approach and the level of expertise you bring to building software. While a real-world problem at scale would be ideal, it's not practical due to time constraints. Instead, we present a problem statement for you to solve as if it were a real-world scenario. This assessment consists of two progressive levels that build upon each other. Failure to follow the instructions will result in automated rejection, and exceeding the time limit will impact our evaluation.

Rules of the game

1. Time limit:

- You have two full days to implement a complete solution covering both levels.

2. Coding style:

- Emphasize object-oriented or functional design skills; create elegant, high-quality code.
- Demonstrate your decision-making process when faced with undefined workflow or boundary conditions.
- Show progressive enhancement from Level 1 to Level 2 through clear Git commit history.

3. Language and environment:

- Solve the problem in an object-oriented or functional language without external libraries (except for testing).
- Ensure your solution builds and runs on Linux; use Docker if necessary.
- Utilize Git for version control; submit a zip or tarball with Git metadata for review.
- Avoid checking in binaries, class files, jars, libraries, or build output.

4. Testing:

- Write comprehensive unit tests/specs; consider test-driven development for object-oriented solutions.
- Ensure tests cover both Level 1 and Level 2 functionality.

5. Project structure:

- Organize your solution inside the `parking_lot` directory.
- Adopt the structure, organization, and conventions of mature open-source projects.
- Include a README.md with clear instructions for both levels.

6. Executable scripts:

- Update Unix executable scripts `bin/setup` and `bin/parking_lot` in the bin directory for automated testing.
- `bin/setup`: Install dependencies, compile code, and run unit tests.
- `bin/parking_lot`: Run the program, accepting input from a file and printing output to STDOUT.

7. Input and output formatting:

- Adhere to syntax and formatting guidelines for both input and output.
- Use the provided automated functional test suite (`bin/run_functional_tests`) for validation.
- Refer to `functional_spec/README.md` for setup instructions for functional tests.

8. Confidentiality:

- Do not make your solution or this problem statement publicly available on platforms like GitHub, Bitbucket, blogs, or forums.

Problem statement

Level 1: Single parking lot management

You own a parking lot that can hold up to 'n' cars at any given time. Each slot is numbered starting from 1, increasing with distance from the entry point. Create an automated ticketing system allowing customers to use the parking lot without human intervention.

Core requirements:

- When a car enters, issue a ticket with the car's registration number and color; allocate the nearest available slot.
- Upon exit, mark the slot as available.
- Provide the ability to find:
    - Registration numbers of cars of a particular color.
    - Slot number for a given registration number.
    - Slot numbers for all cars of a specific color.

Interaction methods:

1. Interactive command prompt-based shell.
2. Accept commands from a file.

Level 1 commands:

- `create_parking_lot <capacity>`
- `park <registration_number> <color>`
- `leave <slot_number>`
- `status`
- `registration_numbers_for_cars_with_colour <color>`
- `slot_numbers_for_cars_with_colour <color>`
- `slot_number_for_registration_number <registration_number>`
- `exit`

Example (Single lot):

To install dependencies, compile, and run tests:

```
None
$ bin/setup
```

To run the program and launch the shell:

```
None
$ bin/parking_lot
```

Assuming a parking lot with 6 slots, run the following commands in sequence, producing output as described below each command. Note: `exit` terminates the process and returns control to the shell.

```
None
$ create_parking_lot 6
Created a parking lot with 6 slots
$ park KA-01-HH-1234 White
Allocated slot number: 1
$ park KA-01-HH-9999 White
Allocated slot number: 2
$ status
```

```
Slot No.    Registration No    Colour
1           KA-01-HH-1234     White
2           KA-01-HH-9999     White
$ registration_numbers_for_cars_with_colour White
KA-01-HH-1234, KA-01-HH-9999
$ exit
```

Level 2: Multi-lot management with dispatcher

Thanks to your successful Level 1 implementation, the business has expanded! You now manage multiple parking lots and need to extend your system to handle two new scenarios:

2.1 Legal regulations across multiple lots

The original legal regulations around finding cars still need to be adhered to, but now they must work across all parking lots simultaneously. The three query commands from Level 1 now need to be modified to search across all available parking lots in one go.

For example, if you issue `slot_numbers_for_cars_with_colour White`, it should return all white cars irrespective of which parking lot they're parked in.

2.2 Customer distribution via dispatcher

Implement a dispatcher system to distribute customers between parking lots using configurable rules. Parking lots now have sequential numbers starting from 1, increasing with distance from the dispatcher.

Dispatcher workflow:

1. Customer comes to dispatcher
2. Dispatcher picks a parking lot based on the selected rule:
    - **Even Distribution**: Dispatch to the emptiest parking lot (lowest occupancy percentage). If multiple lots have equal occupancy, pick the closest one to the dispatcher.
    - **Fill First**: Dispatch to the closest parking lot that isn't full.
3. Customer reaches the assigned parking lot and parks (same as Level 1)
4. Customer leaves the parking lot (same as Level 1)

Additional level 2 commands:

- `dispatch_rule <even_distribution|fill_first>`

Important notes:

- Your solution must handle both single parking lot operations (Level 1) and multi-lot operations (Level 2) seamlessly.

- When multiple parking lots exist, the system should automatically operate in multi-lot mode.
- All Level 1 commands must continue to work in Level 2, but with extended functionality to search across all lots.

Example ssage

Level 2 example (Multi-lot):

```
None
$ bin/parking_lot
$ create_parking_lot 5
Created a parking lot with 5 slots
$ create_parking_lot 3
Created a parking lot with 3 slots
$ create_parking_lot 6
Created a parking lot with 6 slots
$ dispatch_rule even_distribution
Dispatcher is now using the Even Distribution rule
$ park KA-01-HH-1234 White
Allocated slot number: 1 in parking lot: 1
$ park KA-01-HH-9999 Red
Allocated slot number: 1 in parking lot: 2
$ park KA-01-BB-0001 Black
Allocated slot number: 1 in parking lot: 3
$ registration_numbers_for_cars_with_colour White
KA-01-HH-1234
$ status
Parking Lot: 1
Slot No.    Registration No    Colour
1           KA-01-HH-1234      White

Parking Lot: 2
Slot No.    Registration No    Colour
1           KA-01-HH-9999      Red

Parking Lot: 3
Slot No.    Registration No    Colour
1           KA-01-BB-0001      Black
```

File input example:

```
None
$ bin/parking_lot file_inputs.txt
```

file_inputs.txt:

```
None
create_parking_lot 6
park KA-01-HH-1234 White
park KA-01-HH-9999 White
leave 1
status
registration_numbers_for_cars_with_colour White
slot_numbers_for_cars_with_colour White
slot_number_for_registration_number KA-01-HH-9999
```

Submission requirements

1. **Complete implementation**: Your solution must handle both Level 1 and Level 2 requirements.
2. **Progressive enhancement via Git**: Your Git commit history should clearly demonstrate the evolution from Level 1 to Level 2. We expect to see clean, logical commit progression that tells the story of your development process.
3. **Comprehensive resting**: Include unit tests for both levels, with test commits clearly separated.
4. **Clear documentation**: Explain your design decisions and how to run both levels.
5. **Working scripts**: Ensure `bin/setup` and `bin/parking_lot` work correctly.