System Implementation of a Waiter Robot

Project Overview

This project implements a restaurant simulation where humanoid robots act as waiters. It applies object-oriented principles like encapsulation, inheritance, and polymorphism to model robot interactions with the kitchen and tables. Robots process various requests to deliver a seamless dining experience.

Running the application

To execute the restaurant simulation:

1. Ensure Python 3 is installed (the solution was tested on Python 3.12).
2. Navigate to the project root.
3. Run the sample script:

   ```
   python restaurant_sample.py
   ```

   The script demonstrates:

- Setting up the system: orchestrator, tables, robots, and menu
- Placing and processing food orders
- Serving food and handling payments
- Printing formatted bills

System Architecture

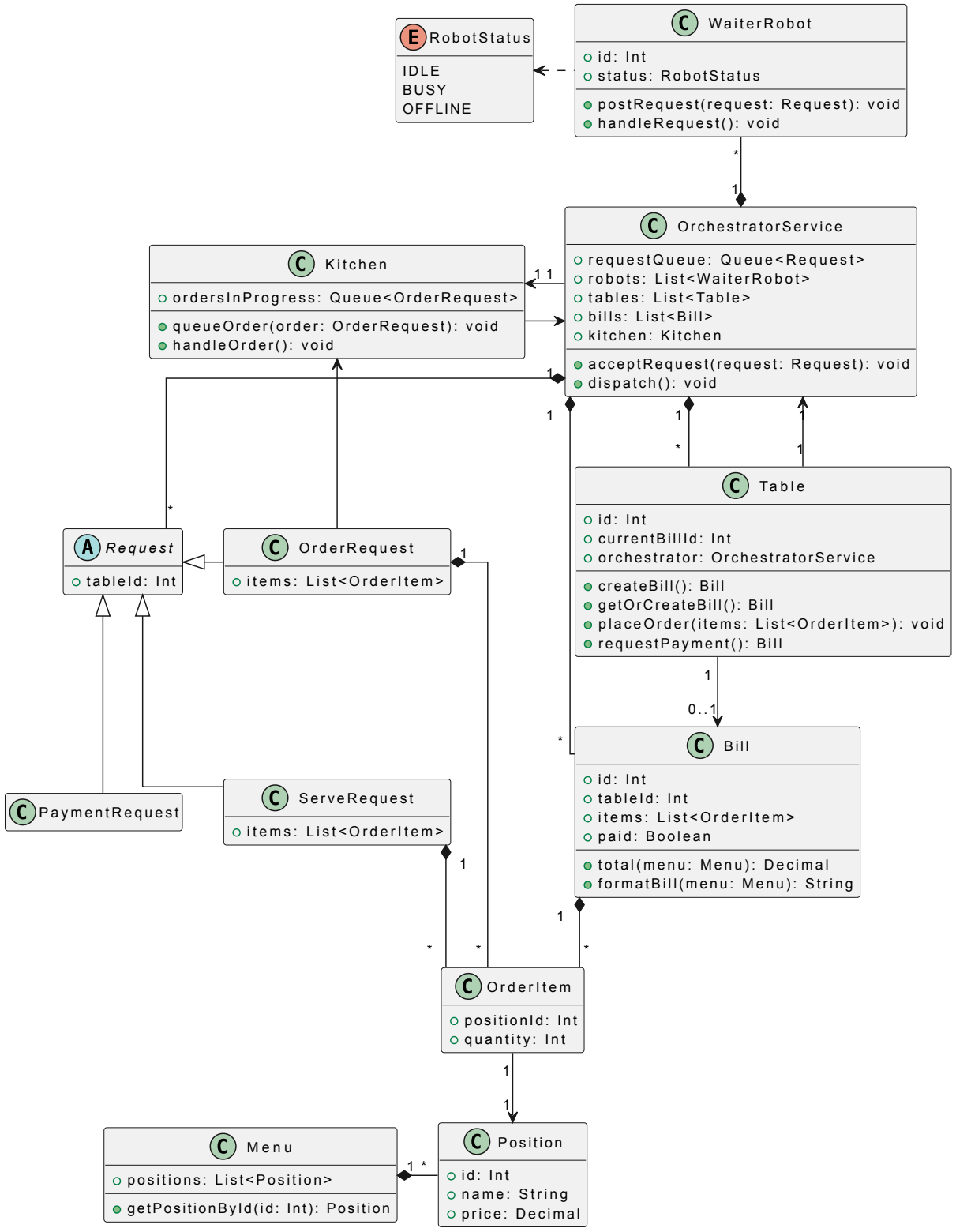

Figure 1. Updated class diagram

Orchestrator Service (`restaurant/orchestrator.py` **)**

The central controller that manages all interactions between components: dispatches requests, manages resources, coordinates the overall flow of operations, and contains references to other system components.
In a real-life setting the queue would be processed asynchronously and the tasks would be assigned as soon as possible, but in this implementation `dispatch` method must be called to process new requests. This was done to better visualize the data flow.

Tables (`restaurant/table.py`)

Tables allow customers to place orders and request payments. They track bills and interact with the orchestrator.

Waiter Robots (`restaurant/robot/waiter_robot.py`)

Robots serve food and process payments. They transition through different states (as defined in `restaurant/robot/robot_status.py`) and handle assigned requests via the `handle_request` method.

Kitchen (`restaurant/kitchen.py`)

Processes food orders and generates serve requests. Queue processing is triggered manually to simulate real-time cooking.

Menu (`restaurant/menu/menu.py`)

Stores available items and prices. Orders refer to items as described in `restaurant/menu/order_item.py` and `restaurant/menu/position.py`.

Requests (`restaurant/requests/request.py`)

All requests inherit from `Request` (`restaurant/requests/request.py`). Types include `OrderRequest`, `ServeRequest`, and `PaymentRequest`. The orchestrator uses these to route requests to the right component.

Bills (`restaurant/bill.py`)

Track ordered items, calculate totals, and manage payment statuses.

Testing Approach

The application employs a comprehensive unit testing strategy using Python's `unittest` framework that facilitates test automation and provides a variety of methods to ensure integrity of a software solution (Python Software Foundation, 2025c). Each class has a dedicated test suite that targets the behaviors of class objects including boundary testing with valid, edge-case, and invalid inputs to ensure robust error handling. Tests also verify that objects transition correctly between the possible states. While the focus is primarily on unit testing, some integration points between closely related components are also verified to ensure proper interaction. This approach ensures high code quality, facilitates refactoring, and provides documentation of expected component behavior.
The tests can be run with the following command:

```
python -m unittest discover -s test
```

Development Process and Reflection

The initial design provided a solid foundation, but during the implementation of the request processing logic it became apparent that a few additional helper methods are necessary, such as bill management in `Table` class. The application of the queues has proven reasonable allowing to process the requests in the order of their creation (Brookshear and Brylow, 2020, p. 441). The changes are reflected in the updated diagram on the Figure 1.
To simplify the logic for the simulation and to make the queue processing more transparent in the solution, the components that process and store the requests implement separate methods for accepting the requests and completing or routing them. However it's worth mentioning that in a real-life setting the requests would be processed asynchronously and independently by the different system components.
During the development process, the monetary values variables were updated to use `Decimal` instead of `double` to eliminate rounding errors typical for the floating-point

arithmetic (Python Software Foundation, 2025a, 2025b).

The application of the different testing approaches has proven to be highly beneficial for the development process and allowed for easily modifying and improving the components' behavior and performance without affecting their traits.

References

Atlassian (no date) *The different types of testing in software*, *Atlassian*. Available at: https://www.atlassian.com/continuous-delivery/software-testing/types-of-software-testing (Accessed: 7 April 2025).

Brookshear, J.G. and Brylow, D. (2020) *Computer science: an overview*. 13th edition, global edition. NY, NY: Pearson.

Python Software Foundation (2025a) *decimal — Decimal fixed-point and floating-point arithmetic*, *Python documentation*. Available at: https://docs.python.org/3.13/library/decimal.html (Accessed: 7 April 2025).

Python Software Foundation (2025b) *Floating-Point Arithmetic: Issues and Limitations*, *Python documentation*. Available at: https://docs.python.org/3.13/tutorial/floatingpoint.html (Accessed: 7 April 2025).

Python Software Foundation (2025c) *unittest — Unit testing framework*, *Python documentation*. Available at: https://docs.python.org/3.13/library/unittest.html (Accessed: 7 April 2025).