Lab 4: OORMS iteration 2 – the kitchen and cancelling items

EEE320 - fall 2022

Introduction

The aim of this lab is to practice designing with class and sequence diagrams, translating your designs to code, and unit testing. To achieve this aim, you will extend an implementation of the Object-Oriented Restaurant Management System (OORMS) based on new and extended use cases.

Submission

Part 1 of this lab 4 must be submitted via the Moodle lab 4 part 1 assignment, not later than 08:00 hrs on Friday, 14 October 2022. Submit a single zip file containing all your diagrams from the part 1 task. Any format is acceptable, including photos of hand-drawn versions, as long as they're legible.

Part 2 of this lab must be submitted via the Moodle lab 4 part 2 assignment, not later than 08:00 hrs on Friday, 28 October 2022. The submission must consist of a lab report **in PDF form** including all diagrams described below, and a copy of your source files, packed together in a single zip file. **Do not** submit a zip archive of your entire project: just the `.py` source files.

Your lab report must be well formatted, using the lab report template provided on Moodle. Follow the instructions in the lab report template. You do not need to include copies of your code in the lab report itself.

Setup

Set up a PyCharm project using the OORMS iteration 2 start code provided with this handout.

You should be able to run the `oorms` module by right clicking the filename and choosing *Run 'oorms'*. This should display two windows: one restaurant view identical to the one from lab 3 (the server view), and one blank window (the kitchen view). You should also be able to run the tests in the `tests` module. This should show 8 tests passed.

Preparation

Review the use cases, user interface notes, and demonstration video provided below. The new behaviour you are required to implement is documented in the alternates and exceptions for the **Take a table's orders** use case, in the **Cook orders** use case, and in the user interface notes.

The OORMS iteration 2 start code **is not identical** to the version of OORMS specified for iteration 1, 'though it works quite similarly. Don't assume you understand how the code works: read it! Also, see the implementation hints section below.

Part 1

Part 1 is preliminary design for your system, using UML. In developing your design, you may make any changes to the system you deem necessary, including modifying classes, methods and attributes, removing classes, and adding new classes.

You must provide:

Warmup

1. a sequence diagram showing what happens when the server presses the *Place Orders* button, assuming there are new items to order (this behaviour has already been implemented; show how the existing code works using a sequence diagram);

Design

2. a sequence diagram showing what happens when the server presses the "item cancel" button (red 'X') for an item that has been placed (ordered from the kitchen) but that has not yet begun cooking;

3. a sequence diagram showing what happens when the cook presses the "Start cooking" button beside an item that is in the "PLACED" state;

4. a sequence diagram showing what happens with the cook presses the "Mark as served" button beside an item that is in the "READY" state; and

5. a class diagram representing the key aspects of your design changes. You are **not** required to show all classes, methods, attributes, and relations in the system — just the ones relating to the sequence diagrams identified above.

Complete these diagrams however you wish. Hand sketches on paper are often best for a first version, if you have a way to share them with your lab partner. Use the diagrams as tools for understanding the provided code and thinking about your design. The diagrams need not be beautiful but they must be legible and in correct UML notation. They should be at the same level of detail as the sequence diagrams from lab 3 and need not show details of how the user interface redraws itself, i.e., the internals of the `create_xxx_ui()` methods.

Label each diagram clearly with an appropriate title and your group members' names. **Keep copies for yourself** as you'll need them for part 2.

Part 2

Starting from your design diagrams, modify the OORMS iteration 2 start code to completely implement the use cases described below. You will almost certainly change your mind about some of your design decisions from Part 1 as you go. This is fine.

Test your code as you go using the Python `unittest` framework. There are some tests provided for you in the `tests` module. Use the same approach as the provided test cases: i.e., test that methods called on the controllers have the correct effect on the model and on the user interface. We encourage you to use a test-driven development (TDD) approach. If not that, then at least use an approach where you write a small amount of code and test it before moving on. **I will be marking your test cases** as well as your implementation code.

Create clean copies of the five diagrams listed in part 1, representing the final versions of your design as expressed in your code. Generate these designs using PlantUML or another diagramming tool of your choice. These five diagrams must be embedded in your lab report (not as separate files) with appropriate accompanying text.

Use Cases

Take a table's orders

Preconditions: The server is logged into the system and the restaurant view is visible. The server is ready to take the orders from a table.

1. The server selects the appropriate table from the tables in the restaurant.
2. The system presents a view of the table.
3. The server selects a seat at the table.
4. The system presents the current order for that seat. For a seat that has not yet ordered, this would be empty. The system also presents the available menu items.
5. The server selects an available menu item.
6. The system adds the selected item to the seat's order. Order items are initially in the "REQUESTED" state. *Steps 5 and 6 may be repeated any number of times.*
7. The server requests that the order be placed. Any items in the "REQUESTED" state are moved to the "PLACED" state.

8. The system returns to the view of the table, as presented in step 2. *Steps 3 through 7 may be repeated any number of times.*

9. The server indicates that they are done with the current table.

10. The system returns to the restaurant view.

Alternates and exceptions:

- At step 7 the server may request that the current order modification be cancelled. Any items in the "PLACED" state are unaffected, but any items in the "REQUESTED" state are removed from the order.

- During steps 4–6, the server may cancel any item that the kitchen has not yet started cooking. However, once the kitchen has started cooking an item, it may not be cancelled. See the *Cook orders* use case and the user interface notes.

Cook orders

Preconditions: The kitchen's order system is active.

1. As orders are placed by the serving staff (the item moves from "REQUESTED" to "PLACED"), items to cook appear on the order system display.

2. When a cook begins cooking an item, they indicate this on the order system display. This moves the item to the "COOKING" state.

3. When an item has been cooked and is ready to be served, the cook indicates this on the order system. This moves the item to the "READY" state.

4. When an item has been collected by the server and has left the kitchen, the cook indicates this on the order system. This moves the item to the "SERVED" state and causes it to disappear from the kitchen display.

Alternates and exceptions:

- An item whose order has been placed but which has not yet begun cooking may be cancelled by the serving staff. This will cause it to disappear from the kitchen's order list. Items which have begun cooking may not be cancelled by the serving staff.

User interface notes

Seats that have current orders with items are coloured green; seats that have no current orders are coloured grey.

In the order view, items on the order that have been placed ("PLACED", "COOKING", "READY" or "SERVED") have a solid green circle beside them. Items that have been requested but for which the order has not yet been placed have a hollow green circle beside them.

On the serving staff's order display, any item that can be cancelled ("REQUESTED" or "PLACED") will have a cancel button displayed. This is a small red square with an 'X'. Pressing this button cancels the item, removing it from the order. Both the server view and the kitchen view are immediately updated. Items that have started cooking must not have a cancel button.

On the order system display, each item will have a button allowing the cook to indicate it has moved to the next phase. Items in the:

- "PLACED" state will have a button reading "Start cooking"
- "COOKING" state will have a button reading "Mark as ready"
- "READY" state will have a button reading "Mark as served"
- "SERVED" state will no longer appear on the interface.

Demonstration video

The video provided with this handout has been developed by our user interface team to demonstrate the required UI design. Most of the UI has been provided for you in the `oorms.py` file; see the **TODO** notes in the code.

Implementation hints

The `ServerView` and `KitchenView` are implemented as subclasses of an abstract `RestaurantView` class (inherits from `abc.ABC`). The `RestaurantView` class contains all the code that is common to the two concrete views. See the `oorms.py` file lines 10–37, 40–43, and 129–132.

When anything in the state of the currently active orders changes, we need to update both the `ServerView` and the `KitchenView`. However, we don't want either view to know about the other, or the controllers to know about views they don't manage. We solve this by giving the `Restaurant` a list of views. Views are registered with the `Restaurant` by calling its `add_view` method. Then, when a change is made to the restaurant's state that requires the windows to re-draw, we call the `Restaurant`'s `notify_views` method. This iterates over all the registered views, calling an `update` method on each of them.

In the controller classes, if a controller does something that doesn't affect the restaurant but does requires its managed view to refresh itself, it calls `view.update()`. However, if it makes a change to the restaurant state that some other view might also need to react to, it calls `restaurant.notify_views()`.

See the following lines:

- `model.py` lines 10–17 (the code in `Restaurant` for managing views)
- `oorms.py` line 24 (adding the current view to the `Restaurant`s list of views)
- `oorms.py` lines 37–38 (the `update` method for views)
- `controller.py` lines 16, 30, 34, and 49 (calls to the view's `update` method)
- `controller.py` lines 54 and 59 (calls to the restaurant's `notify_views` method)

In the provided code, `OrderItem` has a boolean `ordered` attribute. You should replace this with an appropriately-named attribute that holds the order's current state ("REQUESTED", "PLACED", "COOKING", "READY", or "SERVED") and modify the other code in `OrderItem` as necessary. A Python enumeration would be a one good option. See the `tests.UI` enumeration and its use in `tests.ServerViewMock` for an example.