Lab 5: OORMS iteration 3 – billing

EEE320 - fall 2022

Introduction

The aim of this lab is to practice writing use cases, developing simple user interfaces, designing with sequence diagrams, translating your designs to code, and unit testing. To achieve this aim, you will extend OORMS to include support for preparing, printing and logging bills.

Submission

You must complete and submit part 1 of the lab by 08:00 hrs on Friday, 4 November 2022, via the Moodle lab 5 part 1 assignment. Submissions should be in PDF form.

You later must complete and submit part 2 of the lab by 08:00 hrs on Friday, 25 November, via the Moodle lab 5 part 1 assignment.

The final submission must consist of a lab report **in PDF form** including

- the use case description,
- sequence diagrams, and
- class diagrams

described in Part 2 below, along with a copy of your source files, all packaged together in a single zip file. **Do not** send 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 lab 5 start code provided with this handout.

You should be able to run the `oorms` file by right clicking the filename and choosing *Run 'oorms'*. This will display two windows: the *Server View* window showing a map of the restaurant and the *Printer Tape* window which is initially blank.

You should also be able to run the unit tests in the `tests.py` file, and see that 8 tests passed.

The OORMS iteration 3 start code **is not identical** to the versions of OORMS specified for previous iterations. Don't assume you understand its structure or how the code works: read it!

Tasks

Preparation

Review the provided source code, the preparing bills section below, and the hints section before beginning part 1.

Part 1

Part 1 is preliminary analysis and design for your system. You must produce:

1. a complete use case description for **Preparing Bills**;
2. one or more user interface sketches, showing what your bill preparation user interface will look like and how the server would interact with it to prepare bills; and
3. sequence diagrams for each significant interaction with your bill preparation user interface.

I strongly recommend you complete your sketches and diagrams *by hand on paper*, but you may use any drawing tool of your choice (including PlantUML) if you wish. Use your diagrams as tools for thinking about your design. They need not be beautiful but they must be legible and in correct UML notation.

In developing your sequence diagrams you may make any changes to the system you deem necessary, including modifying classes, methods and attributes; removing classes; and adding new classes. Respect the GRASP principles in your design.

Label each diagram clearly with an appropriate title and your group members' names.

You will get feedback on your part 1 submission within a few days of submission.

Part 2

Based on your design diagrams, modify the OORMS iteration 3 start code to completely implement bill preparation, printing, and logging. 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.py` file. You must at least test your `model` classes; you may also want to test your `controller` classes. 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.

Create and include in your lab report with accompanying text:

- a revised copy of your use case description;
- sequence diagrams, drawn using PlantUML or another tool of your choice, showing all significant interactions with your bill preparation user interface and how they are handled; and
- a class diagram, drawn using PlantUML or another tool of your choice, showing any parts of the system that you created or significantly modified.

Your diagrams must represent the final versions of your design as expressed in your code.

Preparing bills

All bills for a table must be prepared together and printed at the same time.

The system must prevent the bills from being printed until all items ordered at a table are accounted for by the table's bills.

Any mapping of orders to bills is possible but it is *not* necessary to be able to move individual items from one order to another for billing purposes – e.g., we do not have to implement "put his tacos on my bill". (Remember, each seat has an order, and each order may contain multiple items.) We might have all orders for a table on one bill, or separate bills for each order, or some orders on separate bills and some combined on bills. For example, a table of eight might have three individual bills, one bill combining two orders, and one bill combining three orders.

There is no sales tax.

Once bills have been prepared they must be printed. This can be a single action in the UI (e.g. a *Print all bills* button). Each bill must list all the items on it, with price, and must include a total at the bottom. Exact formatting (header, footer, columns, etc) is up to you, but bills must be printed in a tidy format.

After bills have been printed the bills must be saved to a bill register (log) and all orders at the table must be empty, i.e., ready for a new group of people to sit down.

The printer is simulated here by the *Printer Tape* window, which simulates an infinitely long tape in a monospaced font a maximum of 40 characters wide.

Optional bonus: As an optional bonus, worth an additional 10%, you can make it possible to split the cost of individual items *evenly* between multiple bills. For example, a table of four might split a bottle of wine evenly across three bills. To receive the bonus you must provide supporting sequence diagram(s) and a correct implementation with your final lab report.

Hints

There is a *Create bills* button on the table user interface which is only visible when a table has at least one order with an item in it. Pressing the button results in a call to `TableController.make_bills()` which includes a comment about what is required there.

Your user interface can be whatever you'd like, but for simplicity it should probably be entirely button based (like the rest of the UI). There is a thorougly-documented `make_button` method in `ServerView` that you will find useful. *Anything* can potentially be a button – for example, all the menu items in `create_order_ui` are buttons.

Any time the state of your bill creation UI changes, you should re-draw it completely. This is generally easier than figuring out what parts of the interface need to be re-drawn, and since the re-draw is happening just once, at human speed, performance is not an issue. All the `create_XXXX_ui()` methods in the provided code take this approach.

You may assume that the orders for a table will always fit in a single column in the *ServerView* window — no need to implement scrolling or text wrapping. If necessary for your UI, you can increase the size of the window by altering the `SERVER_VIEW_WIDTH` and `SERVER_VIEW_HEIGHT` values in the `constants` module.

There are four `TODO:` notes in the code, indicating where you need to work. To find them, use *View > Tool Windows > TODO* or look for the blue markers in the editor's right margin.

Your design must include at least one new class in the `model` module to account for billing information. You may want more than one new class.

For printing bills, use the the `printer` attribute of `ServerWindow`. There is an example of output to the printer tape in the `TableController.make_bills()` method.

To format the bills, use formatted string literals (f-strings) and the format specification mini language. These allow you to print into fixed width fields, left- or right-aligned, which makes column alignment relatively easy. There are examples of f-string use in the provided code – search for `f'`.

As in previous versions of OORMS, event handler methods make use of the "extra arguments trick" documented in §54.7 of the Tkinter 8.5 reference by Shipman provided with this handout. You will almost certainly need to use this trick. See the examples in the provided code.