Random Complaint Generator: Phase 2

Overview

You will have been given another team's code for phase 1 and will need to extend it. You will first want to examine the code base and identify any problems and fix them (e.g. was the unit testing complete, were there any bugs, etc…) You may not simply replace the code base with your own code. General architectural decisions must remain the same.

The extension will be for the program to generate a paragraph-long complaint to a restaurant (as opposed to two sentences). To do so, you will need to have at least 7 distinct types of sentences, so will need to add 5 more *GrammarRule* objects and their requisite *PhraseList* objects. There should be some re-use of some of the *PhraseList* objects in the *GrammarRule* objects. For example, two different *GrammarRule* objects may use the *FeelingPhrase* from the example in Phase 1. However, the complaint generator should never re-use the same item from one of the *PhraseList*s in a single complaint. This will require that either the *PhraseList* and/or *GrammarRule* must be extended to keep track of which items have been used each time a complaint is generated.

Your main program should randomly generate a sequence that the different types of sentences will be displayed, then randomly generate each type of sentence for the complaint and print them to the screen. Similar to phase one, the user should be able to sign their name (or not), and be able to generate new complaints as many times as they want.

Programmer Roles

Each student should take one of the three Classes/GUI (i.e. *PhraseList*, *GrammarRule and GUI)* and be the lead programmer. In addition, each student should take the lead in one of the following roles (but teams are expected to work together to review each other's work and make sure no major mistakes are being made):

- **Architect:** Work out how the classes and GUI will work together for the final program. What methods will each class need? What data will each class need? Where will the random numbers be generated? Etc…
- **Tester:** Create unit tests for each class, and an overall test for the project. Make sure to test for boundary cases and malformed input.
- **Document/Repo:** Ensure all code is documented and committed properly, including test code.

CuisineToGo Program: Phase 2

Overview

You will have been given another team's code for phase 1 and will need to extend it. You will first want to examine the code base and identify any problems and fix them (e.g. was the unit testing complete, were there any bugs, etc…) You may not simply replace the code base with your own code. General architectural decisions must remain the same.

Your application must be extended from Phase 1 as follows. Support for multiple restaurants should be added, and the user should be able to add new restaurants (with unique locations and new menu items) through the GUI. The application should now be able to generate reports on the average time to fill an order (units of time between an order being placed and delivery being finished), which menu items are most popular, and order amounts by restaurant (previously only computed in aggregate). Finally, convert addresses from an abstract grid to real locations with real addresses.

Displaying your driver's movements in real time is too complicated for the current scope of the class. Additionally, computing their progress around the map may require more work than is reasonable. As such, retain the notion of a 'unit' of time as 1 minute. Therefore, if you compute the distance between two locations (say, Turkuaz Cafe, East 3rd Street, Bloomington, IN and School of Informatics, Computing, and Engineering, 919 E 10th St, Bloomington, IN 47408) as 4 ½ minutes, that means that they are 5 'units' of time apart (round up). You may move your driver 1/5 of the distance between these two locations every time the Step button is pressed. **The drive does not have to stay on a road while moving in this way – your program provides an abstract representation of location.**

There are various ways to get data from Google Maps which you can use in your program. We encourage you to start thinking about this early. Documentation on the Google Maps API (which uses JavaScript, not Java) can be found here:
https://developers.google.com/maps/documentation/
Additionally, some groups in the past have chosen to use GMapsFX:
http://rterp.github.io/GMapsFX/

Programmer Roles

Each student should be the lead programmer on one of the following: *Inventory* and *Order* classes, GUI, and reports. In addition, each student should take the lead in one of the following roles (but teams are expected to work together to review each other's work and make sure no major mistakes are being made):

- **Architect:** Work out how the classes and GUI will work together for the final program. What methods will each class need? What data will each class need? Etc…
- **Tester:** Create unit tests for each class, and an overall test for the project. Make sure to test for boundary cases and malformed input.
- **Document/Repo:** Ensure all code is documented and committed properly, including test code.

Thneed Inc: Phase 2

Overview

You will have been given another team's code for phase 1 and will need to extend it. You will first want to examine the code base and identify any problems and fix them (e.g. was the unit testing complete, were there any bugs, etc…) You may not simply replace the code base with your own code. General architectural decisions must remain the same.

The extension will be for the program to keep track of inventory (current and projected), so the system can automatically indicate to a customer how long until their order is filled. The times in the application should use the actual system time. The following changes are anticipated:

- A new *Inventory* class which is updated whenever an order is filled or new inventory arrives at the warehouse. It should also keep track of scheduled inventory replacements so an estimated date can be provided for an item on backorder.
- A new field in the *Order* class, which indicates projected fill date.
- An addition to the GUI which displays the current inventory (separate out by size and colors), and allows the user to indicate when new Thneeds will be added to the warehouse inventory. As new inventory arrives (when the system time reaches the arrival date) and as orders are filled, the display should be adjusted accordingly.
- The program should also include a feature to generate reports that indicate how quickly orders are filled, which items are most/least popular, which items get backordered, and which customers order the most product. The reports should essentially help the business plan their inventory and marketing better in the future based on the past.

Programmer Roles

Each student should be the lead programmer on one of the following: *Inventory* and *Order* classes, GUI, and reports. In addition, each student should take the lead in one of the following roles (but teams are expected to work together to review each other's work and make sure no major mistakes are being made):

- **Architect:** Work out how the classes and GUI will work together for the final program. What methods will each class need? What data will each class need? Etc…
- **Tester:** Create unit tests for each class, and an overall test for the project. Make sure to test for boundary cases and malformed input.
- **Document/Repo:** Ensure all code is documented and committed properly, including test code.

Minesweeper: Phase 2

Overview

You will have been given another team's code for phase 1 and will need to extend it. You will first want to examine the code base and identify any problems and fix them (e.g. was the unit testing complete, were there any bugs, etc…) You may not simply replace the code base with your own code. General architectural decisions must remain the same.

The extension will be for the program to allow for the player to indicate the size of the game board (you may enforce reasonable limits), the difficulty of the game (which results in fewer or more mines), and a more sophisticated GUI. The GUI will have *CellButton* and *BoardDisplay* classes. The *CellButton* class will allow the user to reveal a cell, or mark it as a mine. It will display different colors for not-revealed, revealed and marked as mine. If it is revealed, it will either be blank or contain a number for the number of adjacent mines.

You should track the user's interactions with the game in the form of a history, and have a button to produce a separate window with reports on how frequently games of each difficulty are won/lost, as well as the record (best time) to win each difficulty level. Provide the ability to reset these values if desired. You will need to store data in a file between runs.

The *BoardDisplay* class will layout the *CellButton* objects, and include features allowing the user to enter the board size, select a difficulty level, reset the game, mark a cell as a mine, reveal a cell, and indicate how many mines there are in the board. It will also notify the user when they have won or lost the game. The user should have the option to re-start the game.

Programmer Roles

Each student should take one of the three Classes/GUI (i.e. *insert here)* and be the lead programmer. In addition, each student should take the lead in one of the following roles (but teams are expected to work together to review each other's work and make sure no major mistakes are being made):

- **Architect:** Work out how the classes and GUI will work together for the final program. What methods will each class need? What data will each class need? Etc…
- **Tester:** Create unit tests for each class, and an overall test for the project. Make sure to test for boundary cases and malformed input.
- **Document/Repo:** Ensure all code is documented and committed properly, including test code.

Earthquake Data: Phase 2

Overview

You will have been given another team's code for phase 1 and will need to extend it. You will first want to examine the code base and identify any problems and fix them (e.g. was the unit testing complete, were there any bugs, etc…) You may not simply replace the code base with your own code. General architectural decisions must remain the same.

The extension will be for the program to add a sophisticated GUI. The GUI will have an area for filtering the data that is currently displayed (E.g., by date, location, depth, mag, magType, place, status), an area for the data to be displayed as points on a map, and an area for the details of the currently selected point on the map to be displayed in text form. In addition, there should be two buttons to generate text file reports: one for generating the current selected point/event, and one for generating a file that contains all of the current points that are displayed on the map. Have the user specify the filename in a popup window (and warn them if they might be writing over an existing file).

There are various ways to get data from Google Maps which you can use in your program. We encourage you to start thinking about this early. Documentation on the Google Maps API (which uses JavaScript, not Java) can be found here:
https://developers.google.com/maps/documentation/
Additionally, some groups in the past have chosen to use GMapsFX:
http://rterp.github.io/GMapsFX/

Programmer Roles

Each student should be the lead programmer on one of the following: Overall GUI layout, map functionality, generating reports. In addition, each student should take the lead in one of the following roles (but teams are expected to work together to review each other's work and make sure no major mistakes are being made):

- **Architect:** Work out how the classes and GUI will work together for the final program. What methods will each class need? What data will each class need? Etc…
- **Tester:** Create unit tests for each class, and an overall test for the project. Make sure to test for boundary cases and malformed input.
- **Document/Repo:** Ensure all code is documented and committed properly, including test code.