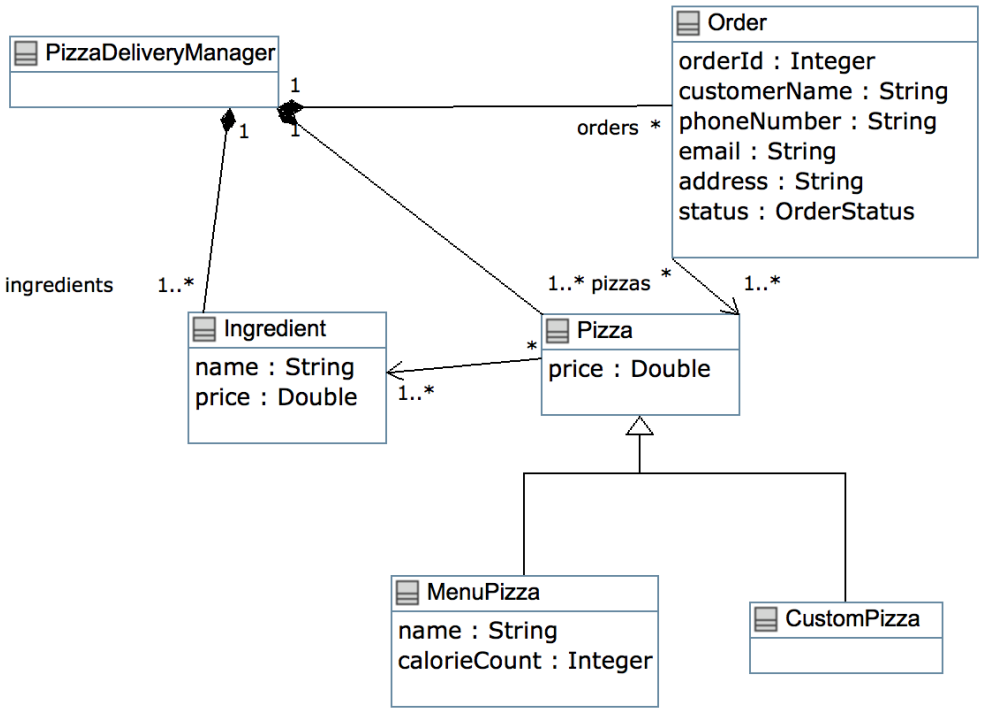
ECSE439/539-A1 Group3: Sherry Kuai (260614908) & Jeremy Mallette (260660818) 10/10/2018

Assignment 1 Report

I. Class Diagram



(*Enum type OrderStatus{InProcess, Delivered} is not shown in the class diagram.)

II. How to Run

Inside the project directory, go to **.**/src/ca/mcgill/ecse439/pds/application, run PizzaDeliverySystem.java as Java Application (May need JavaSE-1.7 for compatibility), then at the home portal, you may choose your use-case scenario (Customer / Management).

III. Discussion & Analysis

The modelling of the system is built upon a class diagram we drew based on the problem description. Through noun analysis, the classes that need to be defined are very straightforward. The system itself is defined to be a class that is composed of the rest classes. Here we use singleton design pattern, because only one instance of the class can exist. To simplify the model, we choose not to have a separate class for customers, addresses, etc. though it may be ideal to have for registered customers in the real-life situation. We include all the customer information as attributes in the Order class. For the contact information, at least one of the phone number and the email should be provided according to the description. The ideal way is to create PhoneNumber class and Email class, and to have compositions with multiplicity of 0..1 between the Order class and each of them. Again, to simplify the model, we define them as String type, and add constraints before the constructor of Order class and setters of these attribute to reinforce the requirement. It would be worthwhile in the final product to do something like regex validation for phones numbers and emails, but again for the minimum prototype described in the problem description, we chose to be less strict. We also created an enum type OrderStatus to indicate the status of an order, so that we can remove the order when its status changes to “Delivered”. The associations between Order and Pizza and between Pizza and Ingredient are one-way, such that a Pizza instance will not know which Order instance it is part of, and an Ingredient instance will not have knowledge of which Pizza instance it is in.

The implementation of the system uses the MVC design pattern. For the model, we use the auto-generated java code from the umple file. The view is implemented with Java Swing. As for the controller, we implemented functions that allow adding/creating, removing, and updating Ingredient, Pizza and Order objects. The restriction of having at least one contact information is enforced here as well. In the PizzaDeliveryController.updateOrder(…) method, we check whether both of the phone number and the email are null or empty. In addition, in all update methods, the objects to be updated are checked for existence in the system, in order to reduce the chance of having bugs. All these checks were carried out used `Before` tags in the Umple description for simplicity. In the process of building the persistence layer, since we define PizzaDeliveryManager as a singleton, we had an issue of loading and copying the model into current instance of it. The issue was with the unmodifiable list generated by the Umple file: when we try to use an iterator to copy each element to the current instance, a `ConcurrentModificationException` was raised. After several tests and attempts, we realized that the iterator of list will remove the element and, causing the size of list to change. The problem was solved by copying all lists we need into newly defined array lists, and using for loops to iterate over them.

Though there is convenience in having auto-generated boilerplate code, it also imposes certain constraints. Where one could theoretically design the system to perfectly suit the system, when using tools to generate code, one is limited somewhat by the assumptions made by that tool. Several times there were both small and large design decisions that could've been manually optimized, such as the above. Another example is how Umple would not accept multiple references to the same pizza object, which complicated the process of tracking the number of pizzas. To compensate for this, we chose to add an integer array attribute that would share indexes with the list of pizza types and store the quantity requested of each. This certainly is not the most elegant solution, but the because the validation issue was nestled in the auto-generated model, more direct solutions could break the software workflow. Overall it is difficult to solve issues like this because any time the code is re-generated, the changes would be overwritten. All this being said, the advantage of eliminating redundant code is a huge advantage. There is obviously a compromise to be made between using auto-generated code and manually written code, but ultimately it is the preference of the developer. As tools such as Umple progress to offer more features and fine-tuning adjustments, the ease-of-use and advantage of such tools will increase to a phenomenal degree. Some macro's similar to Java's `@Override` could allow developers to optionally and non-invasively fine-tune some Umple standards. In our case, tuning just a few of these parameters / assumptions, could have saved hours debugging and finding inelegant solutions. It is difficult because though some would argue that the model just wasn't made properly, it can be frustrating to see small, but critical design decisions that you are not able to control. As an overall statement on UI design, Java Swing is notably unwieldly, but well documented. Though as a personal opinion, I think Java encourages overuse of OOP (too much of any good thing can become bad). Adding layers and layers of encapsulation and abstraction in not always necessary when searching for a simple tool/attribute. A good example of this is when using the tables in Java Swing --implementing a child class for cell renderers, models, and editors, just to achieve very basic functionality is tiring and pointless for something so small-scale. Once again, I'd like to reiterate that OOP paradigms are not a bag at all, but there are scenarios when it becomes cumbersome with very little reward.

The assignment was a challenge for us, but we have learned a few things about software design and modelling from it. It requires us to think of every detail before actually implementing the system. During the modelling process, some details like multiplicities and associations between classes were not thought through very carefully and from our experience almost every bug can be fixed afterwards. However, when we proceeded further into implementation, the seemingly insignificant parts started to cause a lot of troubles. We had to regenerate the model and constantly evolve our system. Also, as we did not start very early, we did not have a well-planned schedule. Hence, time is wasted on waiting for each other to finish our own part. For future projects, a schedule should be made in advance in order to build programs efficiently.