IN2013, Week 3 – Object Oriented Design: adding non-problem domains, design with off-the shelf components

Model Answers

Dr Peter T. Popov

11th October 2018

Question 1 (design class diagram). Consider BAPERS analysis class diagram provided in the Appendix. Refine the class diagram to produce a design class model adding details:

- to the class attributes and operations (methods).
- Revisit the associations and apply aggregation/composition as necessary.
- Make sure that constructors are defined for all classes.

Answer: *The refined class diagram is shown below.*

First of all, we now specify the types of the class attributes, the return values of their methods and the type of parameters used in the operations (now called methods).

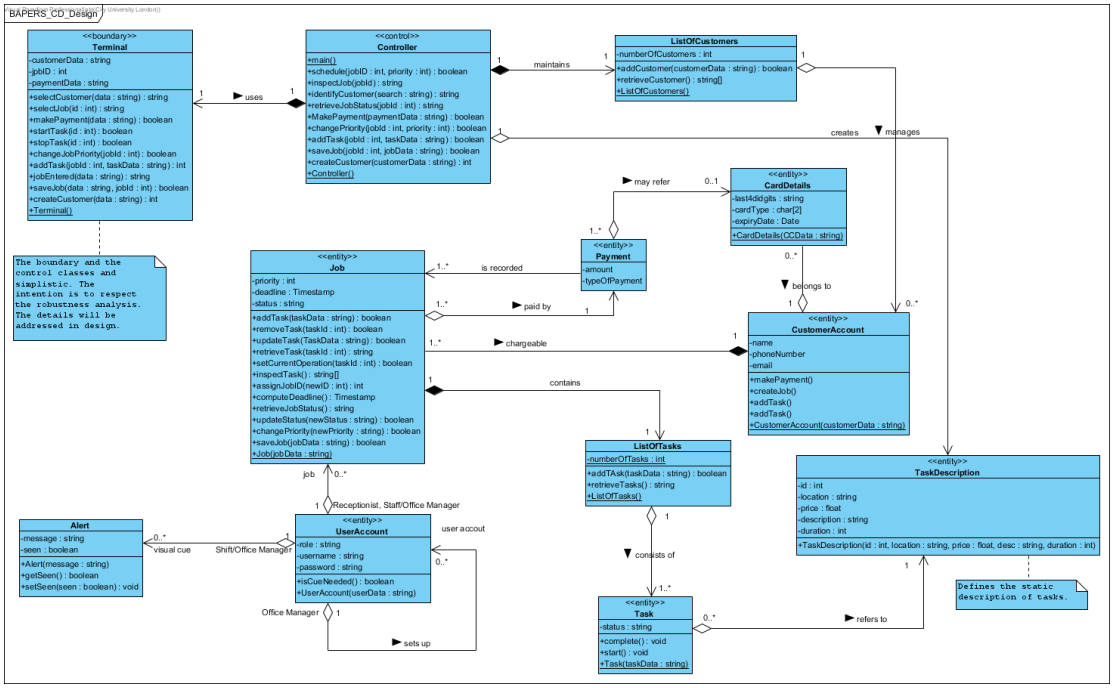
I added a new class ListOfCustomers, which is a mere array, that maintains the lists of Customers. I could have extended the same 'trick' everywhere I use aggregation in the revised diagram, but have not done it for the sake of simplicity.

The problem domain classes have been refined to specify fully the attributes and methods. Constructors have also been added as operation with the same name and class scope.

A new operation main() has been added with class scope which will be the way to get started an execution is many OO languages.

The dependencies that existed in the analysis class diagram have been replaced with aggregations. Now the Controller will be started first, in turn it will instantiate the Terminal (we assumed implicitly that this instantiation will be done by the main() method).

We revisit the associations used in the analysis class diagram: all associations become either aggregations or compositions. We also add explicit navigability, so that it becomes clear which instance should contain references to the other instance for two classes associated in the class diagram.



Additional assignment: ***Non*** **problem domain** (GUI, DB).

- Add a set of boundary classes and develop them into fully fledged classes using a set of 'visual controls' (refer to what you did in Java module building user interfaces). Integrate the boundary classes with the problem domain classes they are related to, e.g. Job class will be associated with a boundary class JobForm, aggregation would be appropriate here. You may decide to replace the single control class with a series of control classes to split the logic of the applications between smaller control classes.
- For the DB domain consider using an interface, say DBConnectivity, with a minimal set of operations (e.g. connect(), write(), read(), closeConnection() and an implementation class, which provides a realization of this class similarly to the example discussed in class). Model the fact that most/all entity classes are persistent and need to store/retrieve data from the DB via the interface DBConnectivity.

Answer: *Instead of refining the entire class diagram below I give a fragment of integrating GUI and a minimalistic design model of DB connectivity, which allows for integration with a relational database.*

The model is simplified. Instead of using a design model that is tightly linked to a particular GUI framework, e.g. Swing, the model answer is given at a level of details which still requires further refinement before it can be implemented.

The boundary classes (called 'Forms') are expected to be derived form a generic Form class, i.e. inheritance is used here.

The forms will have data entry fields. I used attributes of type TextArea for this purpose (Name, phone, Email for the customer form). Also the forms are meant to 'submit' data for processing. This is modeled by a set of buttons. Modelling in detail the events of clicking the buttons and how they get processed is possible is not

provided. This level of detail will be only available once we commit to a particular implementation "framework": using Java Swing library will lead to one set of "visual" controls, adopting a web-based framework such as Strut or Java Faces – will lead to a quite different design of the GUI.

In this model answer the implementation details are ignored and instead I assume that the click events are 'delegated' to a set of operations of the form itself. E.g. clicking a button 'save' would lead to activating the method saveCustomer() of the form, etc.

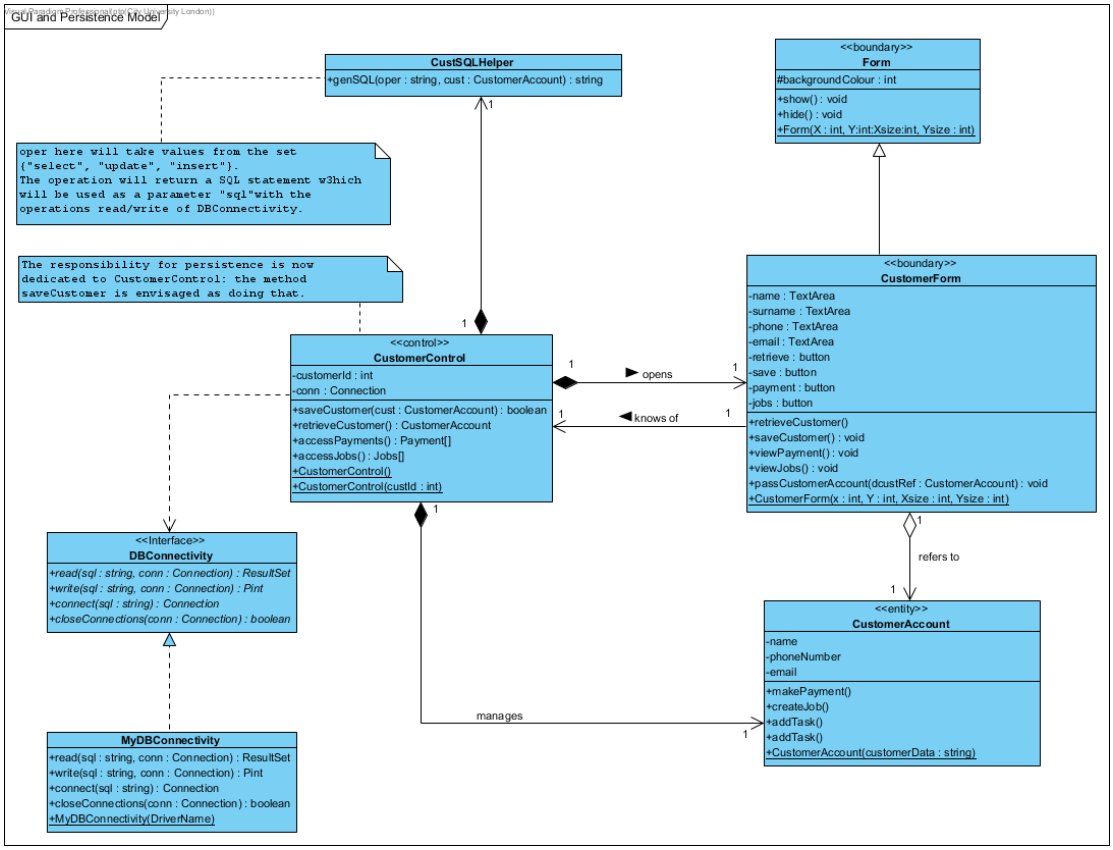
Note also that I added a couple of buttons to customer data – payment and jobs buttons, which are there so that the user can switch from dealing with customer to dealing with payments and jobs. This aspect is not very well thought out – there might be much clearer solutions (e.g. using multi-tab forms), but I hope you get the idea.

Finally, the forms are expected to visualize data, e.g. CustomerForm is expected to visualize data related to a customer. I store the data in an object of type CustomerAccount. The CustomerForm, therefore, should be associated with the CustomerAccount class. Here I use aggregation between CustomerForm and CustomerAccount. How does the CustomerForm become aware of the CustomerAccount? This is explained below.

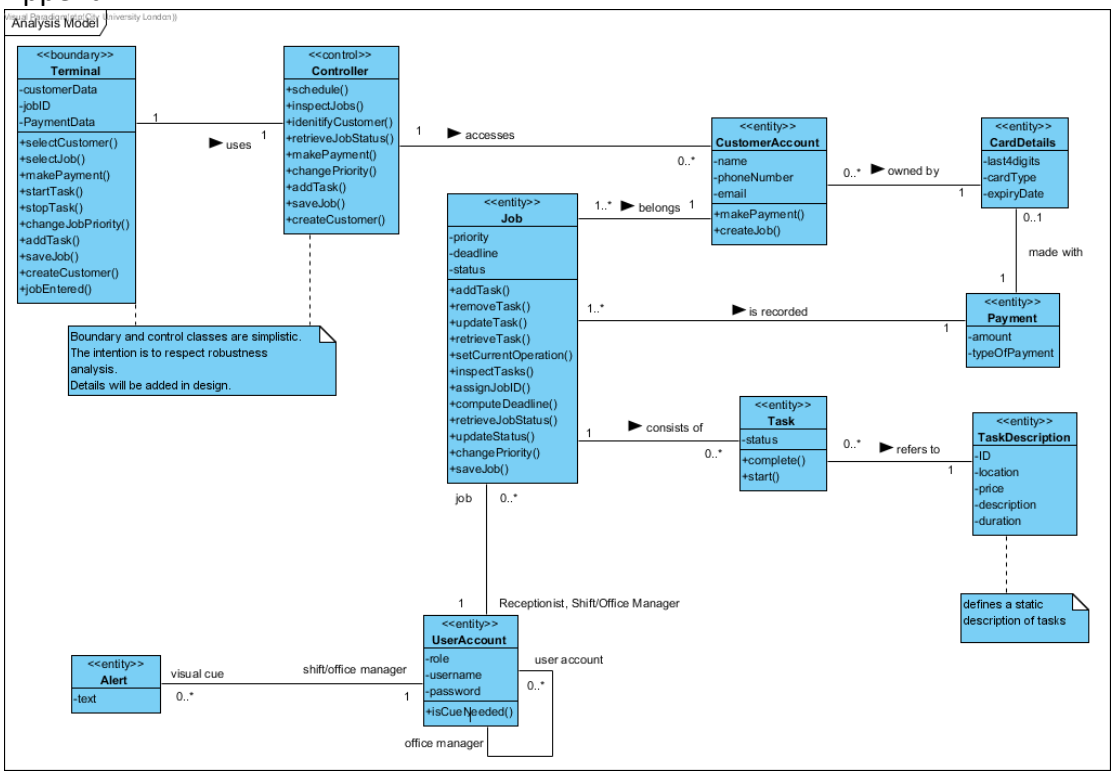
The Controller (used in analysis) must be refined in design. Typically, a number of control classes will be used. This trick should be familiar to you. I saw many are comfortable with the concept of controllers. In this model answer I show a single example of a smaller control class CustomerContol, but other control classes may be added to the diagram if the example is to be developed fully.
First let look at the interaction between the CustomerControl on the one hand and the CustomerForm and CustomerAccount on the other hand. Both the instances of CustomerForm and CustomerAccount are created by the CustomerControl class – captured by a composition. Once CustomerControl instantiates the CustomerAccount it is expected to pass a reference to it to the CustomerForm (the method passCustomerAccount() is used. Its only parameter is a ***reference*** *to an instance of CustomerAccount. Thus, CustomerForm now becomes indeed aware of the CustomerAccount instance and can access it).*

Finally, the fragment shows how persistence of entity classes can be achieved. Here I use the same minimalistic way of accessing database connectivity – using an interface, DBConnectivity which defines a minimal set of abstract operations necessary for interacting with a relational DB, and an implementation class, which hides the implementation details but provides an implementation of the interface (i.e. a solution compliant with the contract defined by the interface).
Note that the CustomerControl has a dependency with DBConnectivity interface. This dependency indicate that the instances of CustomerControl will be able to access the interface! This is the level of detail that I expect to be used in modeling persistence in design.
Further detail can be added by creating a dedicated helper class to create sql statement (SELECT … WHERE …, UPDATE … or INSERT …) which are implied in the

attribute sql:string of the methods defined in the interface DBConnectivity and the implementation class MyDBConnectivity.



Appendix



Last updated: 17th October 2018