Annals of "Dunarea de Jos" University of Galati
Fascicle I. Economics and Applied Informatics
Years XXVI – nº2/2020
ISSN-L 1584-0409 ISSN-Online 2344-441X
www.eia.feaa.ugal.ro

DOI https://doi.org/10.35219/eai15840409125

Designing an Information System for Managing Restaurant Orders

Adrian LUPASC*, Madalin – Claudiu HUZUM**

ARTICLE INFO

Article history:
Accepted August 2020
Available online August 2020
JEL Classification
C99, I29, J15, L63, L86

Keywords:
UML, Use Case Diagram, Class Diagram

ABSTRACT

At first sight, managing order management activities in a restaurant may seem simple, but in reality, things are not like that at all. All decisions made can have a major impact on how the business will evolve over time. For an efficient and corresponding administration is required the existence of a management information system for order management received from restaurant customers, but also of all specific collateral data: locations, meals, waiters or menus. Computer systems for managing these types of activities have practically influenced the way a restaurant can be administered today, fact for which they are currently used on a fairly large scale. Thus, the purpose of this paper is to present how to design an application for order management at a restaurant level, application that can be useful to both restaurant employees and its customers. In this context, the paper presents two UML diagrams (Unified Modelling Language) which describe both the processing of the future computer system (UseCase Diagram) as well as its conceptual structure (Class Diagram) from the perspective of the data that this computer system will manage.

1. Introduction

Analysis and design of an information system represents fundamental stages in the development and maintenance of any computer system and must be developed and completed before writing the source code. The two stages were often ignored, but today it is unanimously recognized their importance because it was proven that of these depends writing and reusing the respective software in various other projects. For the analysis and design of modern information systems there is evolved modeling languages, and one of these is the Unified Modeling Language (UML).

2. UML language

UML is not simply an object-oriented modeling language, but is the standard universal language for software developers around the world. Language provides system architectures that work on the analysis and design of objects with a language suitable for specifying, viewing, building and documenting artifacts of software systems. Also, UML is a modeling language that provides a graphic expression of software structure and behavior. Thus, UML notations constitute an essential element of language for the actual realization of modeling, namely the part of the graphic representation on which any modeling language is based. Modeling in this language is done by combining UML notations within the diagrams offered, among the most important being the *use cases* diagram and the *classes* diagram. Thus, the analysis and design of a computer application involves the development of several categories of models (diagrams), the most important of which are:

- *UseCase diagram* – presents the functions (behaviour) of the future computer system and treats their problems and solutions in the way that the end user of the application perceives them;
- Class diagram – is based on static analysis of the problem (in terms of classes and associations) and describes the static properties of the entities specific to the problem modeled.

2.1. Use case Diagram

A *UseCase* diagram describes a collection of use cases (or scenaries) and actors and is used to describe the functionalities and behavior of the modeled information system, interacting with one or more actors. The *UseCase* diagram is a description in a structured language of a potential situation that an application may or may not solve and describes how an actor interacts with the organization or the system as a whole.

*, **Dunarea de Jos University of Galati, Romania. E-mail addresses: adrian.lupasc@ugal.ro (A. Lupasc - Corresponding author), huzum.madalin21@gmail.com (M. C. Huzum)

The *UseCase* diagram contains elements that can represent actors, association relationships, generalization relationships, and use cases. For this reason, it can be said that such a diagram can clearly describe the limits and behavior of the designed system.

To design the entire functionality of a large, complex computer system, can be created multiple *UseCase* diagrams to describe different functionalities of the application (create a UseCase diagram for each identified subsystem). Use cases may include other use cases as part of its behavior.

The specific elements of a usecase diagram are as follows:

- the actor - an actor is, in principle, a system user, but it can also be another computer system that interacts with the system being analysed;
- use cases - are represented in the form of an ellipse inside which the name of the scenario is written;
- associations - are used to indicate the relationship between an actor and a use case, in the sense that the actor participates in some way in that scenario (Lupaşc A., 2018).

Figure 1. shows the use case diagram of an information system for order management at a restaurant.

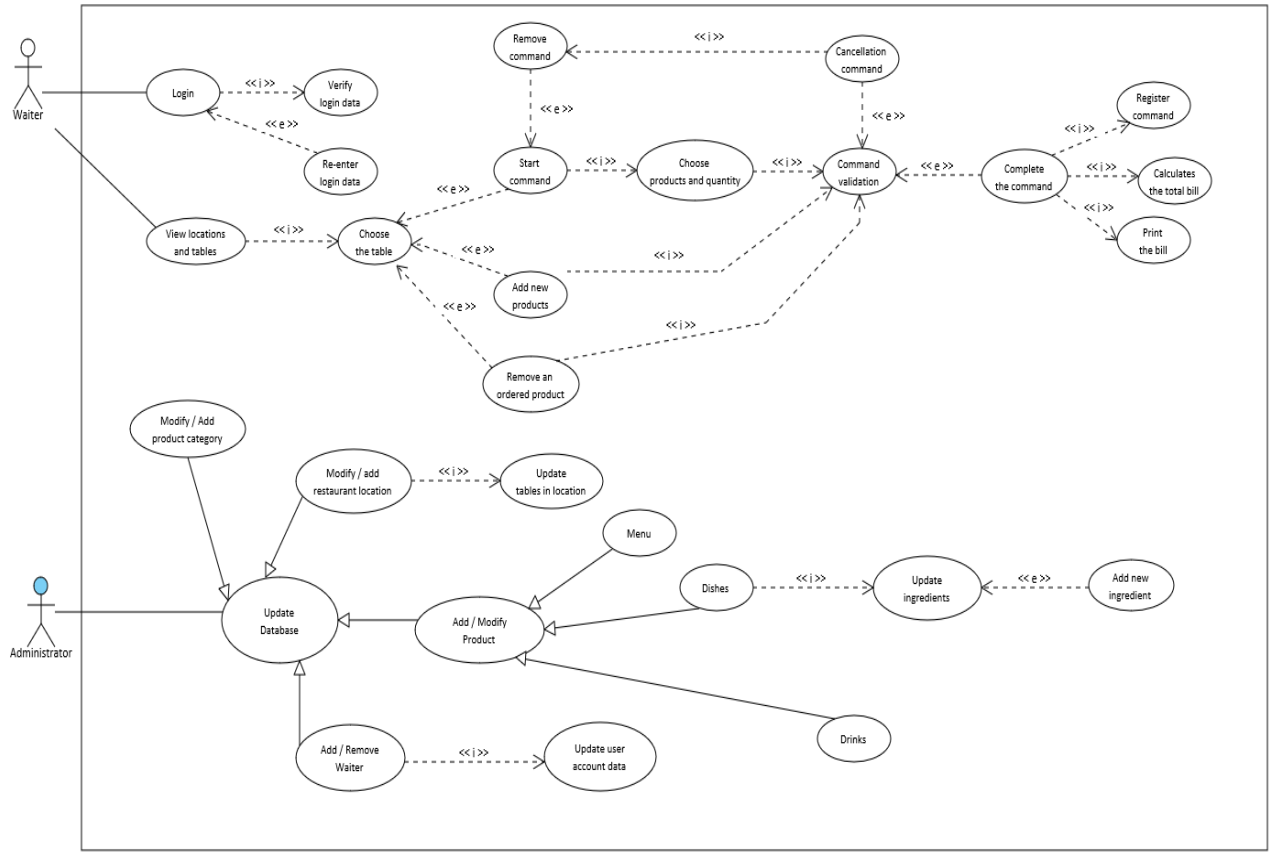

Figure 1. Use-case diagram for the proposed system

Source: create by authors

The Use-Case diagram described in *Figure 1* includes two actors: the waiter and the administrator. The waiter actor is linked to the *Login* use case, through which the user name and password of the waiter account are entered. *Login* scenario behavior includes the *Verify Login Data* scenario that examines authentication data and verifies that it matches an account. If they don't match any account, a warning message will be displayed that the username or password is wrong. In this case, the behavior of the *Login* scenario can be expanded with the *Re-enter Login Data* scenario that allows the user to enter other authentication data.

The waiter actor is also linked to the *View Locations and Tables* use case which allows the computer system user to view the restaurant's locations, along with their specific meals. This scenario case includes the behavior associated with the *Choose the table* scenario.

The behavior associated with the *Choose the table* scenario is related to the *Start Command* scenario by the inclusion relationship, when a table is selected to open a command, and the scenario by which that command is started. The *Remove Command* scenario is dependent on the *extend* type with the *Start Command* scenario because when a command starts, it can be immediately recorded as deleted if nothing has been ordered within it.

Using the *Choose Products and Quantity* use case, which is in an *include* dependency with the *Start Command* scenario, select the products ordered at that table. This use case also includes the *Command Validation* scenario, which re-checks product stocks and stores the order composition.

The *Command Validation* scenario is dependent on the *extend* type with the *Cancelation Command* and *Complete the Command* scenarios.

Using the *Cancelation Command* scenario, the user can cancel the client's order; this behavior includes the *Remove Command* scenario.

The *Remove an ordered product* scenario allows the application user to delete a product from the order composition even after it has been validated.

Between the *Complete the Command* use case and the *Register Command* scenarios, *Calculate the Total Bill* and *Print the Bill* there are dependencies of the *inlcude* type, because every time a waiter completes an order, it is registered in the database and the order bill is printed.

The administrator component of the computer system is specific to the head of the restaurant and gives him the right to manage the types of product categories, dishes, ingredients, drinks and drinks prepared by the bartender, locations, meals, etc. Thus, the actor *Administrator* is associated with the *Update Database* use case which represents a generalization of several use cases: *Modify / Add product category*, *Modify / Add restaurant location*, *Add / Modify Product*, *Add / Remove Waiter*. The *Add/Modify Product* scenario is also a generalization of the following use cases: *Menu*, *Dishes* and *Drinks*.

The *Dishes* scenario is dependent on the *include* with *Update Ingredients* scenario because when a dish is inserted or modified, its ingredients can be updated. The behavior of this scenario can also extend to that of the *Add New Ingredients* scenario. In this way, when an ingredient that does not exist is to be added to the composition of a dish, it can be added to the database so that it can be selected later.

The *Add / Remove Waiter* scenario is dependent on the *include* type with the *Update user account date* scenario because every time a waiter is added or changed, it is necessary to create or update its application authentication account.

The behavior associated with the *Modify / Add Restaurant Location* scenario is dependent on the *include* type with the *Update Tables in Location* scenario, because when a new location is added or an existing location changes, you can allow tables in that location to be changed.

2.2. Class Diagram

The class diagram is considered by most specialists in the field to be the pillar of all object modeling and shows what the system needs to do and how it will be built. In object-oriented modeling, classes, objects and relationships between them are the main elements for modeling. When object-oriented programming is used to make software systems, classes and relationships become the code itself. A class diagram shows how the different entities are related to each other, its main purpose being to describe the static structure of the modeled system. We say that the class chart is a static one because it does not describe what happens when the related classes interact with each other.

The class diagram associated with the designed information system is shown in *Figure 2*.

- ✓ *Location* class manages restaurant locations and includes private properties *rentalID*, *locationName* and *locationType*. The *Update Location()* method provides the possibility to update the restaurant's locations for the purpose of adding a new one or removing another;
- ✓ the *Table* class manages restaurant tables and includes private *tableID* and *numberOfPersons* properties. The *Update Number Persons()* method allows you to change the number of people assigned to a table while the *Add New Table()* method allows you to add a new table;
- ✓ there is an aggregation association between the *Location* and *Table* classes because the tables of a restaurant can only be found in one of its locations, and by removing a table, the location remains active;
- ✓ the *Waiter* class contains data on restaurant waiters who can handle orders. The properties that are present in this class are: *waiterID*, *waiterName* (the waiter's first and last name) and *availability* (this property indicates the waiter's availability). The availability of a waiter can be changed through the *Update Waiter Availability()* method;
- ✓ the *Customer* class manages the restaurant's customers and includes the properties: *customerID*, *customerName* (customer's first and last name), *phone* (customer's phone number) and *mail* (customer's e-mail address). The *Update Client Details()* method can change the specific data of each client stored in the database;
- ✓ *Account* class manages both customer and waiter accounts; this class includes *accountID*, *accountType* (which may have the values "O" for the waiter and "C" for the client), *username* (account username), *password* (account access password) and *creationDate*;
- ✓ *Waiter* and *Customer* classes are associated with the *Account* class to highlight that both waiters and customers managed in the app's database have app access accounts. All multiples of these links have a value of 1 (one) because an account has only one recipient, while the customer or waiter can have only one account for access to the information system;

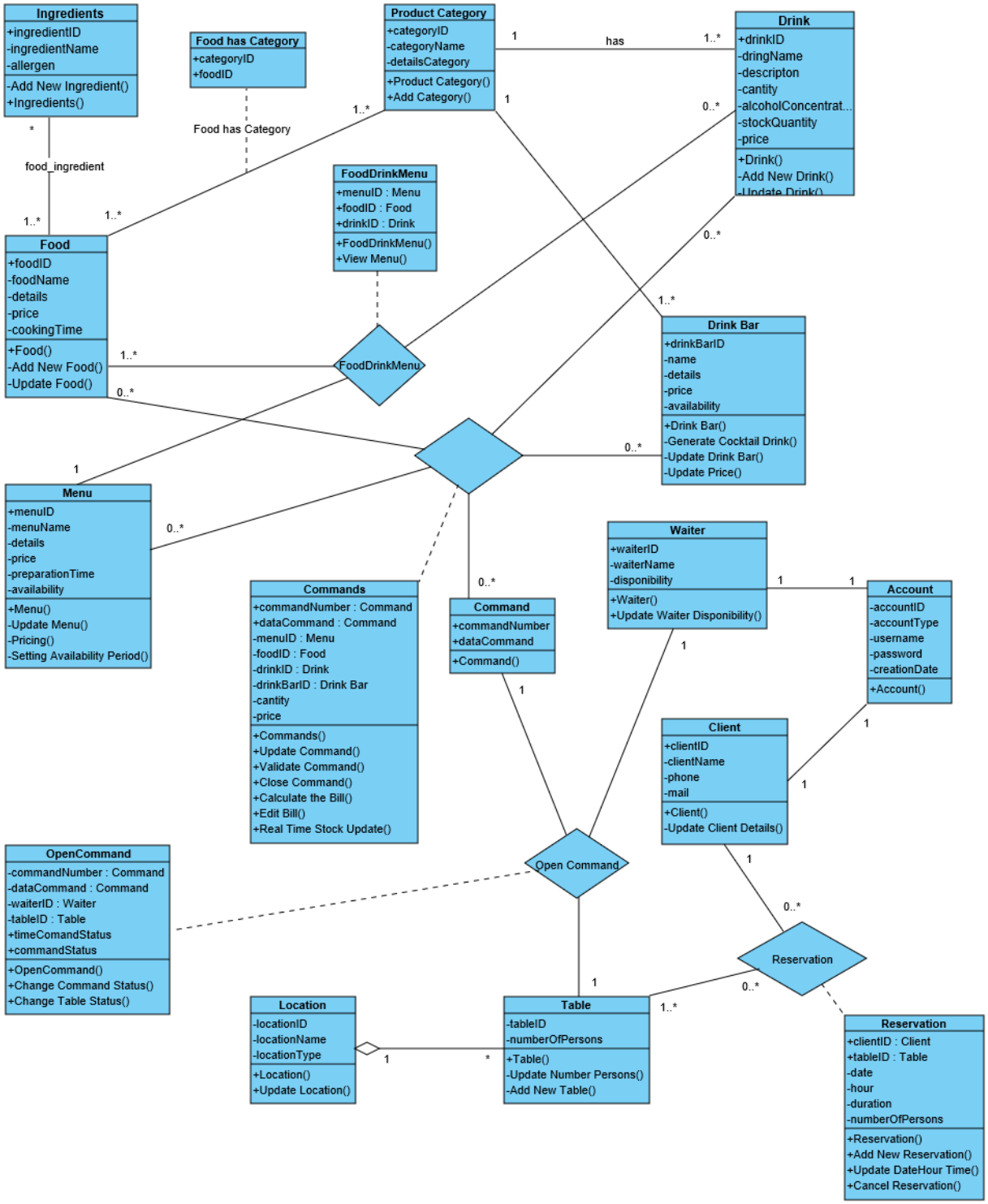

Figure 2. Class diagram for the proposed system

Source: create by authors

- ✓ the *Reservation* class association assigns objects to the *Customer* and *Table* classes and highlights the tables reserved by customers on a specific date. The *clientID* (*Customer* type) and *tableID* (*Table type*) properties identify the client and the table reserved by the client. The other properties store information about the date and time of the reservation, the time period for which the reservation is made, and the number of people booking. The methods specific to this class allow you to add a new appointment – *Add New Reservation()*, change the date or time of an appointment – *Update DateHour Time()* or cancel an appointment – *Cancel Reservation()*;
- ✓ *Ingredients* class manages the ingredients from which the restaurant dishes are formed and is characterized by the *ingredientID*, *ingredientName* (ingredient name) and *allergen* properties, with which we can set an ingredient as an allergen;

- ✓ *Product Category* class manages the categories of foods, drinks and bartender drinks and includes the properties *categoryID*, *categoryName* and *detailsCategory*;
- ✓ *Food* class manages restaurant dishes. The properties of the class are *foodID*, *foodName* (food name), *details* (food description), *price* (food price) and *cookingTime* (time allotted to food preparation);
- ✓ the association *food_ingredient* associate the objects of the *Food* and *Ingredients* classes to highlight the ingredients that make up each food. Multiplicity * indicates that any food consists of several ingredients while multiplicity *1..** indicates that any ingredient is found at least in one food;
- ✓ the association *Food_has_Category* associate the *Food* and *Product Category* classes with the role of highlighting the category in which each food present can be included in the restaurant menu;
- ✓ *Drink* class manages drinks marketed by the restaurant and includes the properties of *drinkID*, *drinkName* (name of drink), *description* (description of the drink), *quantity* (quantity of the drink), *alcoholConcentration* (alcoholic concentration of the drink), *stockQuantity* (quantity of drink that is in stock) and *price* (price of the drink);
- ✓ the *has* association links the objects of the *Drink* and *Product Category* classes to highlight the category of products to which each drink belongs;
- ✓ *Drink Bar* class aims to manage drinks prepared by the bartender and marketed by the restaurant. The class includes the properties of *drinkBarID*, *name* (name of drink), *details* (description of the drink), *price* (price of the drink) and *availability* (this property indicates the availability of the drink);
- ✓ *Menu* class includes data on restaurant-prepared menus. The properties that are present in this class are: *menuID*, *menuName* (menu name), *details* (menu description), *price* (menu price), *preparationTime* (time allocated to menu preparation) and *availability* (this property indicates the availability of the menu in the restaurant);
- ✓ the *FoodDrinkMenu* class association associates objects of the *Food*, *Drink* and *Menu* classes; these objects are the composition of each menu of the restaurant. The multiplications associated with the combinations highlight that a menu can only include food, but it cannot contain only drinks;
- ✓ the *Command* class manages the orders made and includes the *commandNumber* and *dataCommand* properties;
- ✓ the *Commands* class association manages orders made in the restaurant and links the *Command*, *Menu*, *Food*, *Drink* and *Drink Bar* classes. These items will then be stored in the database and in this way it will be possible to know at what price a product was ordered at a given time, being known that at the level of a restaurant, the price dynamics of the ordered products is quite high. Through the methods related to these classes, changes can be made such as: updating an order by accessing the *Update Command()* method, validating the order using the *Validated Command()* method, closing an open order using the *Close Command*() method, calculating and printing the bill using *Calculated the Bill()* and *Edit Bill()* methods. Also, ordering any product automatically leads to the call of the *Real Time Stock Update()* method through which a product's stock is updated in real time, even if an order is not validated and completed.
- ✓ the *Open Command* class association assigns objects to the *Command*, *Waiter* and *Table* classes. Objects related to this class retain data relating to the waiter and the table at which an order was opened. The *commandStatus* property stores the status of a command (free or busy). This status is changed by calling the *Change Command Status()* method. The *Change Table Status()* method is also designed to change the status of a table; it is called by means of the *Change Command Status()* method, so that a table becomes free only when the order status indicates its completion by printing the bill

3. Conclusions and final remarks

The activities specific to each economic entity are characterised by a large volume of data which cannot always be managed efficiently without the existence of a information system adapted to its specific information requirements. Thus, the marketing of products through orders within a restaurant is an important activity that can be managed efficiently if there is a specific information system.

In this context, the purpose of this work was to present a way of oriented object design of a information system (through the *UseCase diagram* and the *Class diagram*) to manage the specific activities of a restaurant. As is apparent from the results of the work, the implementation of a computer system based on the two UML diagrams presented, can lead to a number of advantages and benefits for all actors involved: waiters, administrator or clients. In this way, order management, adding a table in different locations, updating the contents of a menu, updating the product stock in real time are the easy functionalities of a potential IT system implemented for this purpose.

Also, implementation of an information system based on the two diagrams may also enable the development of a *Web* interface that allows customers to create accounts and make reservations for restaurant meals.

References

1. *Arlow, J., Neustadt, I., UML and the Unified Process – Practical object-oriented analysis and design, Addison-Wesley, 2002.*
2. *Booch G., Rumbaugh J., Jacobson J., The Unified Modeling Language User Guide, 2nd Edition, Addison-Wesley, 2005.*
3. *Georgescu C., Abordarea relațională și obiectuală în analiza sistemelor informatice, Editura Didactică și Pedagogică, R.A., București, 2002*
4. *Lupașc A., Designing an Application for the Public Transport Management, The Annals of "Dunarea de Jos" University - Fascicle I. Economics and Applied Informatics, Years XXIII – no3/2018, ISSN-L 1584-0409, ISSN-Online 2344-441X, pp.78-83, 2018.*
5. *Oprea, D., Dumitriu, F., Meșniță, G., Proiectarea sistemelor informaționale, Editura Universității „Alexandru Ioan Cuza", Iași, 2006*
6. *Parunak, H.V.D., Odell, J., Representing Social Structures in UML, In Wooldridge, M., Weiss, G., Ciancarini P. (Eds.): Agent–Oriented Software Engineering II, LNCS 2222, Springer, 2002.*
7. *Rumbaugh J., Jacobson J, Booch G., The Unified Modelling Language Reference Manual, Addison-Wesley, 1999.*