Lab Report 2(CA2.2) Structural Modelling

Student Name: Aleksejs Polikarpovs

Course: Contemporary Software Development

Module: PROC_IT801 Software Processes

Lecturer: Angela Sweeney

Submission Date: 27/10/2021

Contents

Aim

Aim is to produce appropriate class diagram based on XYZ online shop use-case diagram and use-case descriptions, which was produced earlier in CA 2.1.

Produced class diagram will give a clear overview of classes needed in development and their relationships.

Pre – requisites

To complete the report following items will be required:

- System requirements with a full description of how a system should operate
- Use-case diagram and use-case descriptions for XYZ online shop development
- Visual Paradigm modeling software

Methods

What is Class diagram

Class diagram is the main building block of object-oriented modeling. In the diagram we want to describe the different things (objects) that are in the system which we are developing. Those different things we are representing through classes.

Class is the blueprint for the object and every object created from a class is an instance of the class. Each class is made up of states (attributes) and behaviors (methods).

In the UML (Unified Modelling Language) class diagram, we represent class as a rectangular box, which is usually divided in three sections.

Class Name
Class attributes
Class methods

Table 1. Representing class in a Class Diagram

Class name. By giving a name to the class, we are basically giving an object type for the instance, which we create from that class.

Class attributes. In the object-oriented world it is also known as instance fields. They are used to represent characteristics (describe the object) for every instance created from that class. For example, every Person (instance type) has a first name, last name, date of birth etc.

Class methods. Methods representing behaviour of an object. For example, every person might say “Hello” to another person. Constructor is a special method which provides rules (necessary attributes) needed to create an object.

Four principles of Object-Oriented Programming

Each class in a class diagram should be created respecting four principles of Object-Oriented programming:

- **Inheritance** is the procedure in which one class inherits all the attributes and all behaviours from another class ("Is a" relationship).
- **Abstraction** hides unnecessary (unwanted) details from the system user.
- **Encapsulation** hides the values or the state of the object inside the class.
- **Polymorphism** provides ability to perform same action, but in different forms depending on the type of the object.

Abstract class

Using inheritance, we can create subclass for the superclass which we inherit from. All attributes and methods (abstract and non-abstract) of superclass will be available for the subclass. This way we are keeping our code DRY (Don't Repeat Yourself). Abstract methods should have an empty body, and must be implemented in a subclass, where method implementation is specified in a body.

Abstract classes are templates for other more specific classes. For example, Individual and Corporate customers, they are more specific classes for class Customer. Class Customer might be an abstract class, and Individual and Corporate are concrete classes.

Abstract class name and abstract methods in a Class diagram represented by *italic* font.

Access modifiers

Class attributes and methods have access modifiers, which are capable from where this attribute or method is available. In other words, it shows availability from outside of the class.

Symbol	Access modifier	Description
+	public	Available anywhere outside of the class
-	private	Available only inside the class
#	protected	Available within the class and subclasses, but not from outside

Table 2. Class access modifiers.

Access modifier symbol should be in front of the class member (attribute or method).

Each class member is followed by colon (:) where for attributes it specifies attribute type, and for methods it specifies return type.

Person
- name : String
+ Person (String name)
+ sayHello : String

Table 3. Class members with access modifiers

Relationships between classes

Objects don't operate in isolation and they have direct relationship associations between each other.

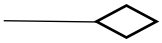
Association between objects is represented by a line drawn between two classes. Each association has multiplicity which shows how many instances may participate in the relationship from each side.

Value	Multiplicity
0 .. 1	Zero or one instance
0 .. *	Zero or more instances
1 .. 1 or 1	One instance. Default when multiplicity is not specified on diagram
1 .. *	One or more instances
3 .. 6	Given range (from 3 to 6 in given example)
7	Discrete number

Table 4. Multiplicity chart

In UML associations might be more descriptive:

- **Aggregation** represents "has a" relationship and it is more specific than the association. It represents a part-whole or part-of relationship, where subclass is independent from superclass.

  Aggregation is shown on diagram as: 

- **Composition** is a part of aggregation. Composition represents the whole-part of relationship, where subclass is dependent on superclass. If superclass is deleted subclass can't exist anymore.
  Composition is shown on the diagram as: 

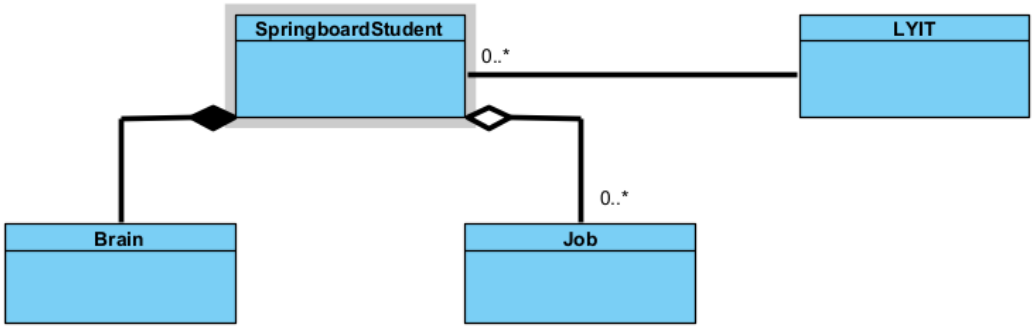


Figure 1. Relationship example

Creating a new Class diagram in Visual Paradigm

To create a class diagram, we should open the already created project with use-case diagram, which was produced previously as CA 2.1. After diagram is opened:

1. Tap on Diagram in top menu
2. Tap on New

3. Select Class Diagram

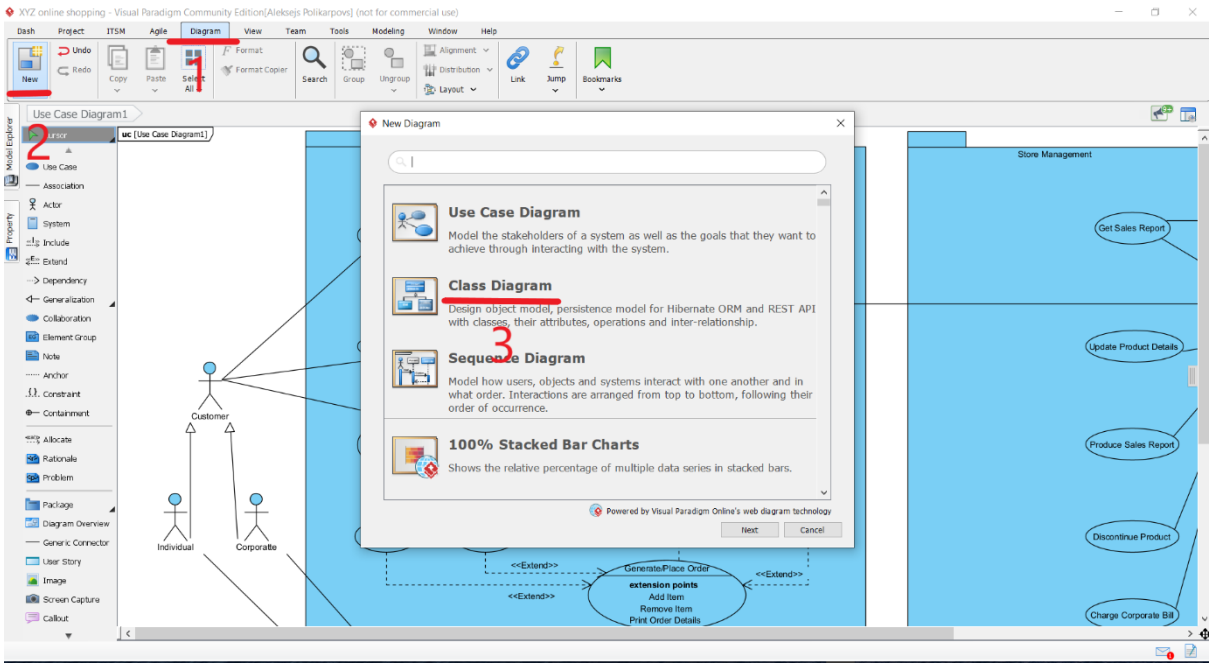


Figure 2. Create new Class Diagram

To switch between diagrams:

1. Tap on View.
2. Project Browser.
3. Choose diagram to switch.

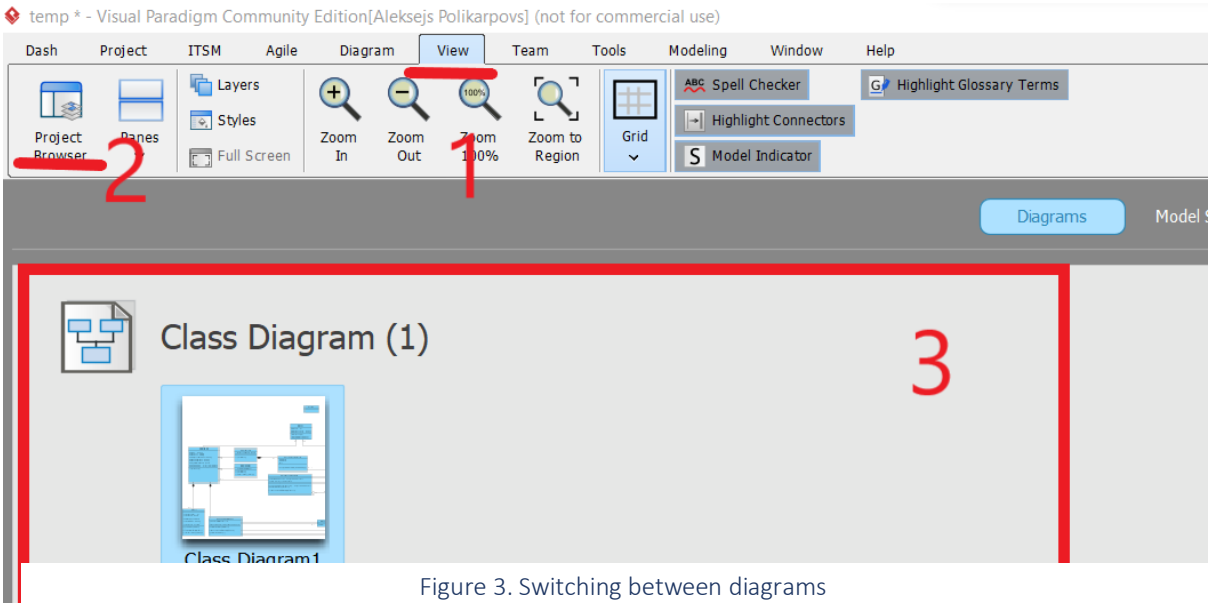


Figure 3. Switching between diagrams

Describing classes needed

In early stages, it is best to do Class diagram parallel with a Use-case diagram, because during continuous development we are getting better picture of the system.

Needed classes might be identified in the use-case description, directly in the system requirements itself or by developer in previous experience.

Noun identification technique is used to identify possible classes needed.

Table 5. Identifying classes for XYZ online shop

Customer	Keep. Class “Customer”.
Individual	Keep. Class “Individual” (type of Customer).
Corporate	Keep. Class “Corporate” (type of Customer).
Manager	Keep. Class “Manager” (type of ShopStaff). New abstract class ShopStaff is needed(generalization).
Administrator	Keep. Class “Administrator” (type of ShopStaff).
Credit Card Authority	Discard
Login into System	New class User is needed. Because during login we don’t know who is trying to login. It might be a customer or an administrator. Keep. Class “User” will be used to split Customer and ShopStaff.
Authenticate with Id and Password	Keep. Operation in class User
Register new Customer Details	Keep. Operation in class User
Browse Product Catalogue	Keep. Class “ProductCatalogue”.
Manage Shopping Basket	Keep. Class “ShoppingBasket”.
Add Item	Keep. Operation in class Customer and Shopping basket
Remove Item	Keep. Operation in class Customer and Shopping basket
Proceed To Checkout	Discard.
Generate Order	Keep. Class “Order”. Keep. Operation in class “ShoppingBasket”.
Print Order Details	Keep. Operation in class Order
Add to Corporate Bill	Keep. Operation ‘purchase’ in abstract class Customer
Pay by Card	Keep. Operation ‘purchase’ in abstract class Customer
Authorize Card Payment	Keep. Operation in class Individual
Get Sales Report	Keep. Operation in class Administrator and Manager
Update Product Details	Keep. Operation in class Administrator.
Produce Sales Report	Keep. Operation in class Administrator.
Discontinue Product	Keep. Operation in class Administrator.
Charge Corporate Bill	Keep. Operation in class Administrator.
Product	Keep. Class “Product”.
Book	Keep. Class “Book” (type of Product).
Software	Keep. Class “Software” (type of Product).
Recording	Keep. Class “Recording” (type of Product).
Sales Report	Keep. Class “SalesReport”.
Name	Keep. Attribute in class Customer.
Address	Keep. Attribute in class Customer.
Telephone number	Keep. Attribute in class Customer.
Email address	Keep. Attribute in class Customer.

Credit Card Number	Keep. Attribute in class Individual.
Product availability	Keep. Product class attribute
Product title	Keep. Product class attribute
Product copyright	Keep. Product class attribute
Product price	Keep. Product class attribute
Book author	Keep. Book class attribute
Recording artist	Keep. Recording class attribute
Recording running time	Keep. Recording class attribute
Software version	Keep. Software class attribute
Ordre number	Keep. Attribute in class Order
Order date	Keep. Attribute in class Order
Date shipped	Keep. Attribute in class Order
Order status	Keep. Attribute in class Order
Order total price	Keep. Attribute in class Order

After classes are identified we can add those classes in to XYZ class diagram by:

1. Tap on Class in left-hand side of class diagram menu.
2. Holding left mouse button drag it in to the diagram.
3. Change name by double click on class name and type class name
4. Add attributes and methods by right mouse button click on class rectangular, and select Add

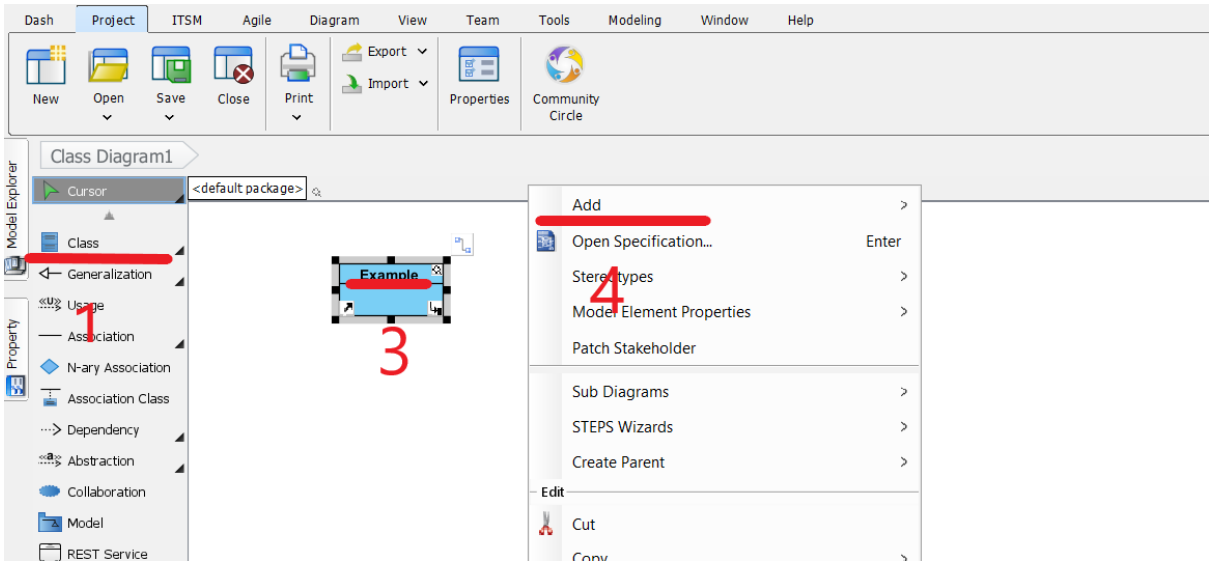


Figure 4. Adding class into class diagram

Notice that in Figure 5 we already specified inheritance between classes. In current situation classes User, Customer, ShopStaff and Product are abstract classes as they are templates for more specific classes.

Class Customer also has an abstract method “purchase”, which will have different implementation depending on type of subclass. In class Individual method “purchase” will include “authorizePayment” where Credit Card Authority will authorize current payment and “purchase” for Corporate will add order in order history, and corporate invoice will be issued on due date.

Class User is used to check (separate) if user who logins is a Customer or a ShopStaff and only new customer (Individual or Corporate) might be registered. ShopStaff (Manager or Administrator) accounts might be created only manually by a system development team.

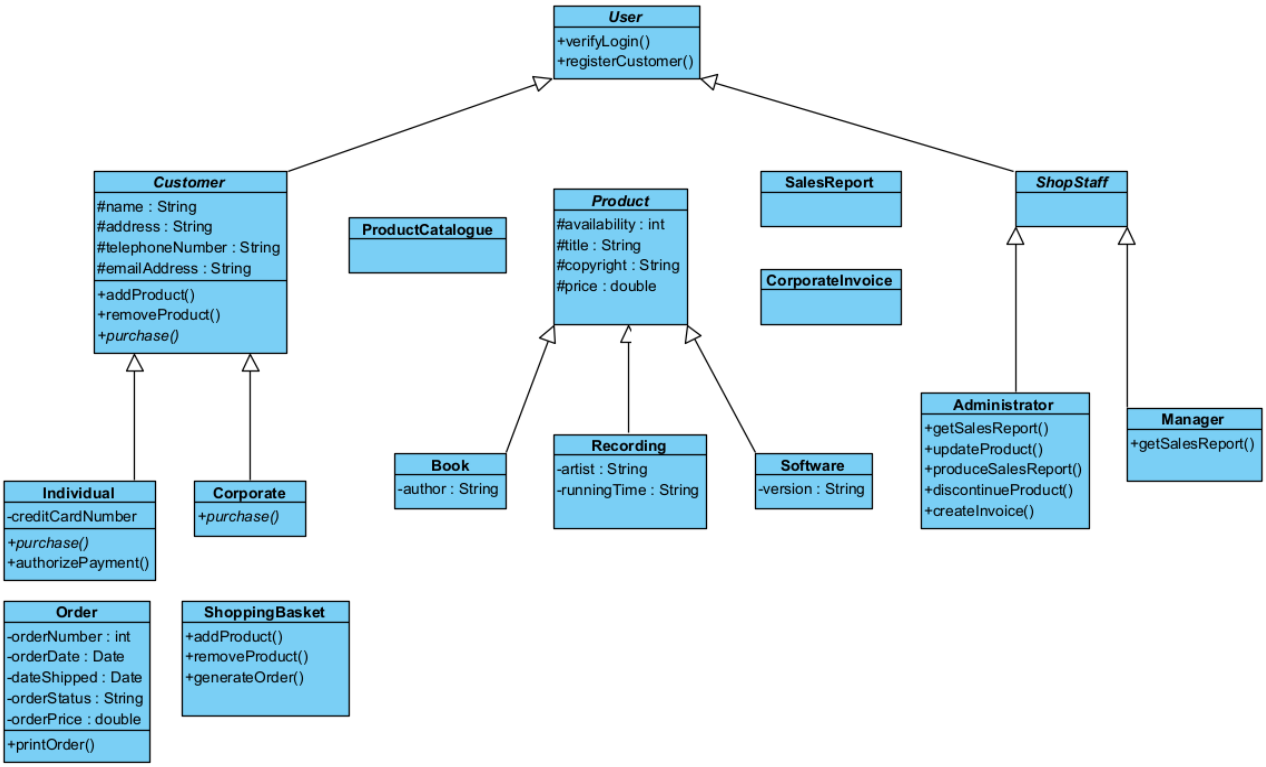


Figure 5. XYZ online shop classes

Specific attributes and operations for each class can come later, during development.

Adding associations and multiplicity

Like we noticed earlier, objects don't operate in isolation and they have direct relationship between each other.

To add association between classes:

1. Tap on Association in left-hand side menu.
2. Connect two classes with association line.
3. Right click on a line close to the class and select multiplicity.
4. Choose multiplicity value.

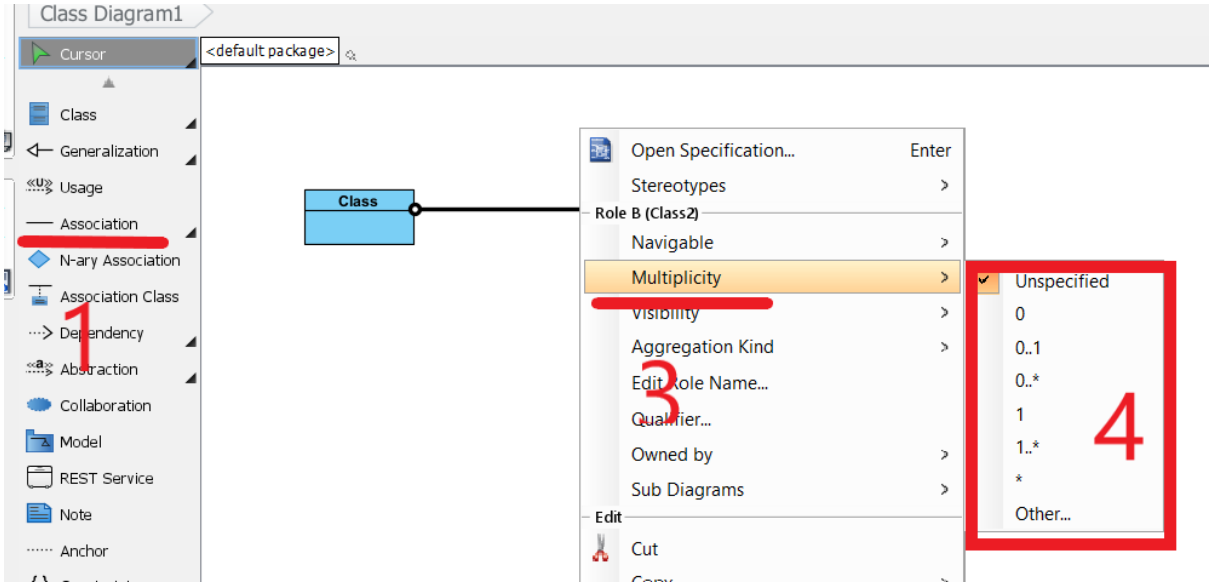


Figure 6. Adding association and multiplicity

Specify aggregation or composition:

1. Right click on the proper side of association line.
2. Select Aggregation Kind.
3. Choose Shared for aggregation, Composited for composition.

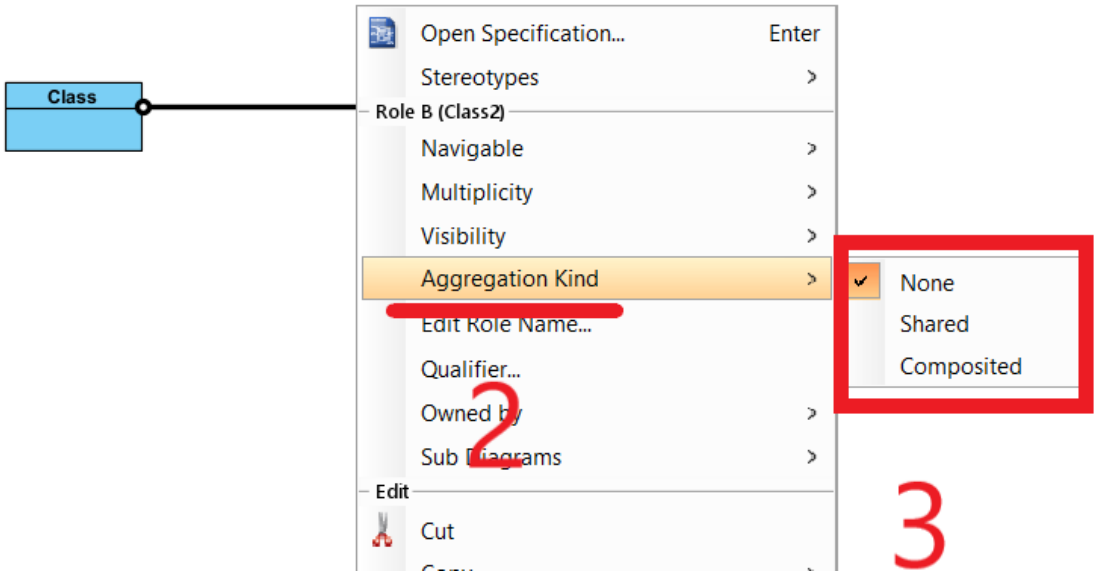


Figure 7. Selecting aggregation or composition

For example, customer might check for some Product in a Product Catalogue. Based on that, those two classes have a relationship between each other. This type of relationship is association. And if we are thinking, there is only one product catalogue available for all customers. Multiplicity for this association on the side of Product Catalogue will be one (default value if multiplicity is not specified

on diagram). From the other side, that catalogue might be checked by any number of customers, or might be not checked at all. Multiplicity in that case is 0 .. * (zero to many).

We have few composition examples in current scenario. For example, each customer has only one shopping basket. If customer decides to delete his account, shopping basket will be deleted as well. Multiplicity in this case might be not specified on diagram, as one customer has one shopping basket (default values).

Also, in the current scenario we have an example of aggregation. We have a product, which might be a book, recording product or software. That product might be in the product catalogue, where customer might purchase this product. If that product has low sales, administrator will discontinue this product from the catalogue. Even if this product is not in the catalogue, this product still exists. Multiplicity in this case is one catalogue and product is 0 .. * (zero or many).

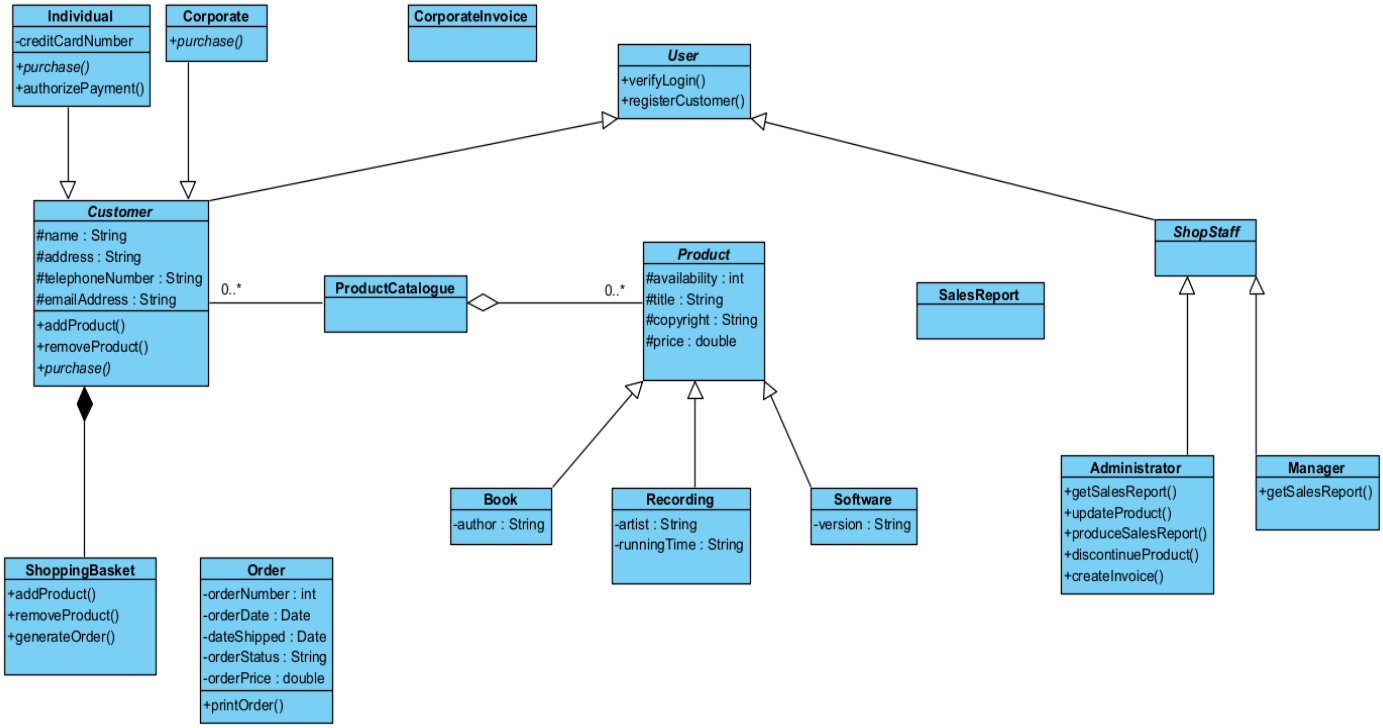


Figure 8. Association, composition, aggregation example

Now we need to finish all relationships between classes and think about missing attributes and operations which might be needed.

Let's take class User like an example. Once again, we need to go through requirements. To login into the system customer should enter his ID and password. Because that class is abstract and we want these attributes to be available for all inherited classes, we can declare inside User class two attributes with protected access modifier:

- # userID : int
- # userPassword : String

Class User already has methods to verifyLogin() and registerCustomer() if user doesn't have an account.

Current class is a base class for Customer and ShopStaff, and it doesn't have any other relationships.

After all associations and missing class members was added:

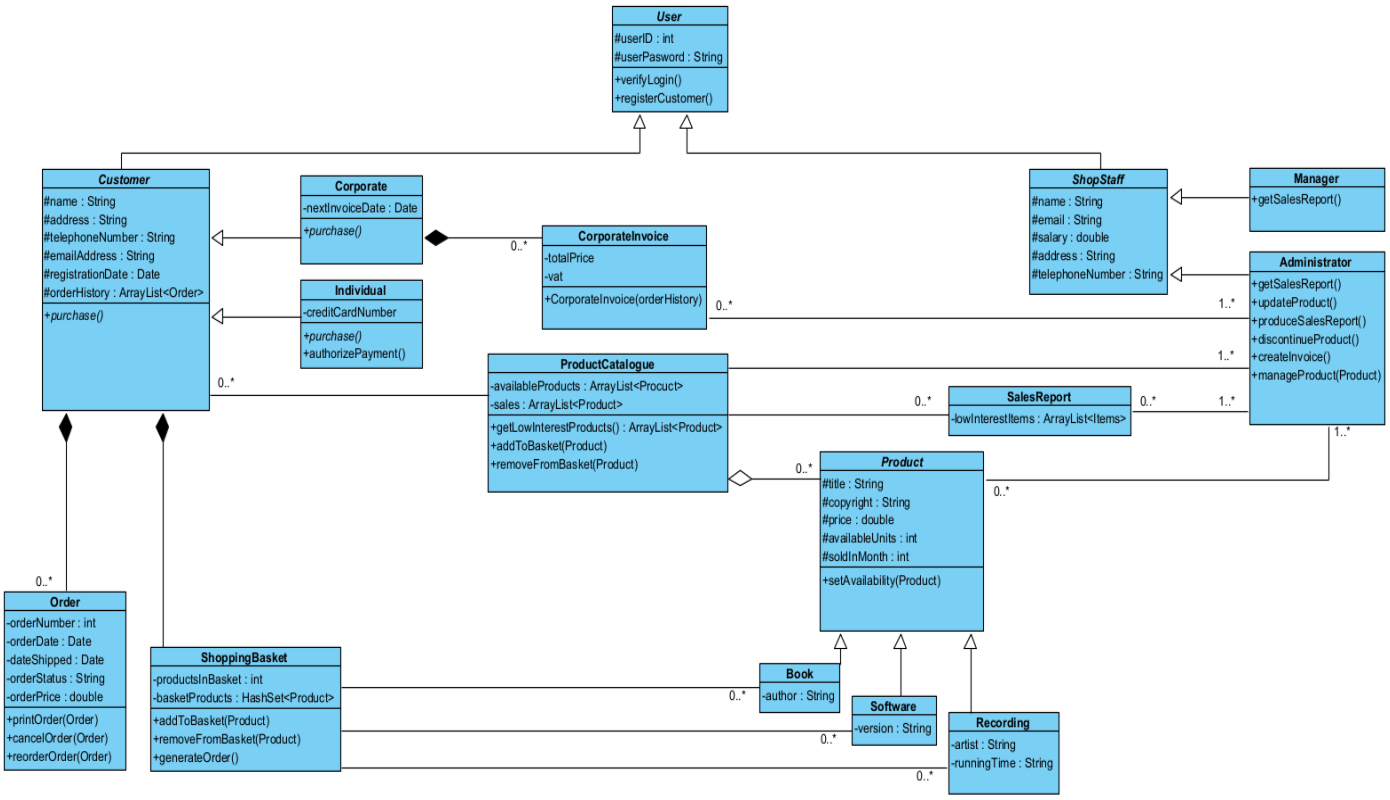


Figure 9. Class diagram with all associations.

Boundary object.

All objects which are in our diagram at the moment are entity (business) objects. It is easy to identify entity objects because they are modelling XYZ online purchase application.

Online Web Purchase Application will be viewed from an external world to the user to obtain the behaviour, data and information that is needed from the system.

If we check non-functional requirements in use case description, which was documented in previous CA 2.1 we can see that the system should be available on mobile devices as well. That means that Mobile Purchase Application will be as well another view from external world to the user. Those views are called boundary objects.

To add boundary objects into the diagram:

1. Select boundary class (add it if it is not on the diagram).
2. Right click on the mouse.
3. Select stereotypes.

4. Select boundary.

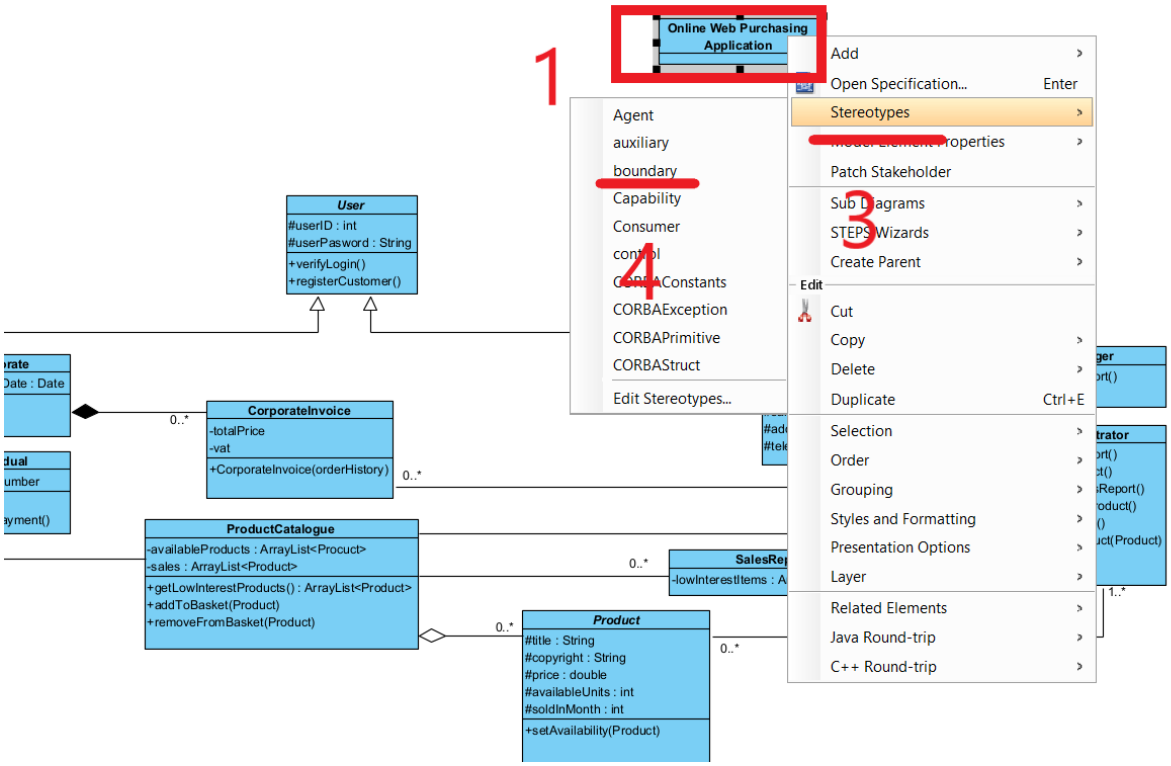


Figure 10. Adding boundary object

Online Web Purchasing application and Mobile Purchasing Application are front end systems (interface) from external environment to the business objects.

Controller object is the middle layer which provides rules for the user's interactions with the entity systems. Let's put Product Purchase Controller in our system:

1. Select controller class (add it if it is not on diagram).
2. Right click on the mouse.
3. Select stereotypes.
4. Select control.

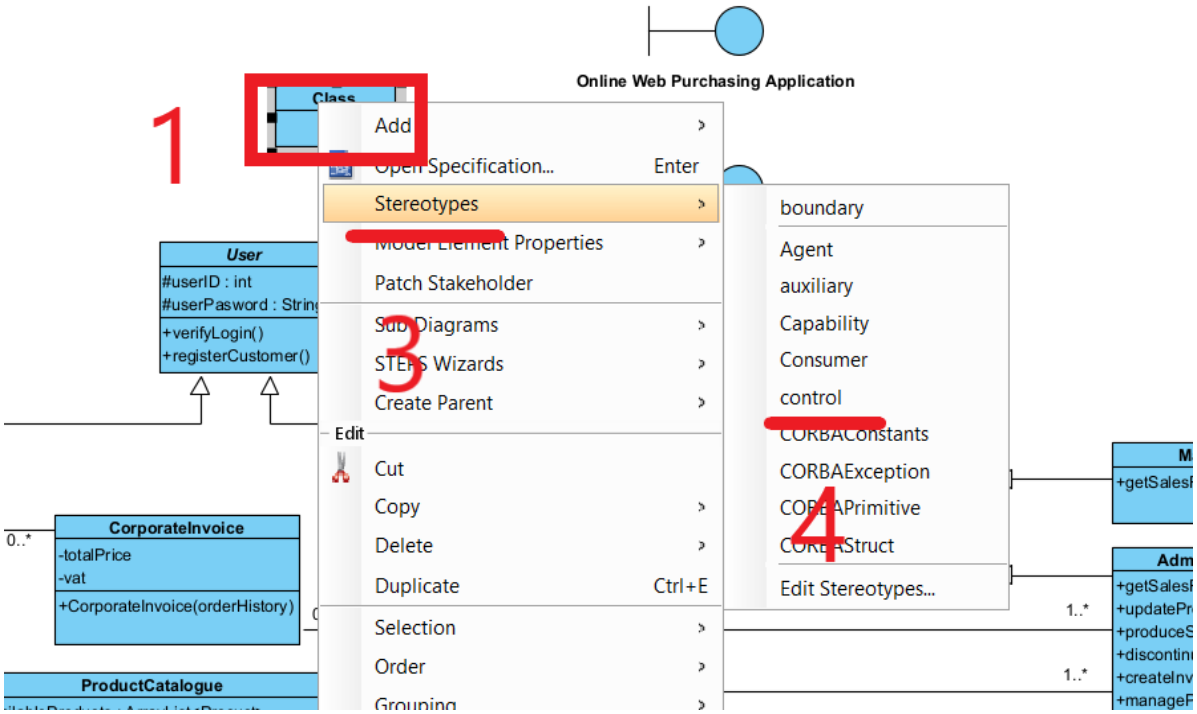


Figure 11. Adding control object

Now when we have boundary objects and control objects, we need to add Actors in our diagram and make associations between them.

To add Actor in Class diagram simply press on Model Explorer in left-hand side menu and drag actor in to the Class diagram.

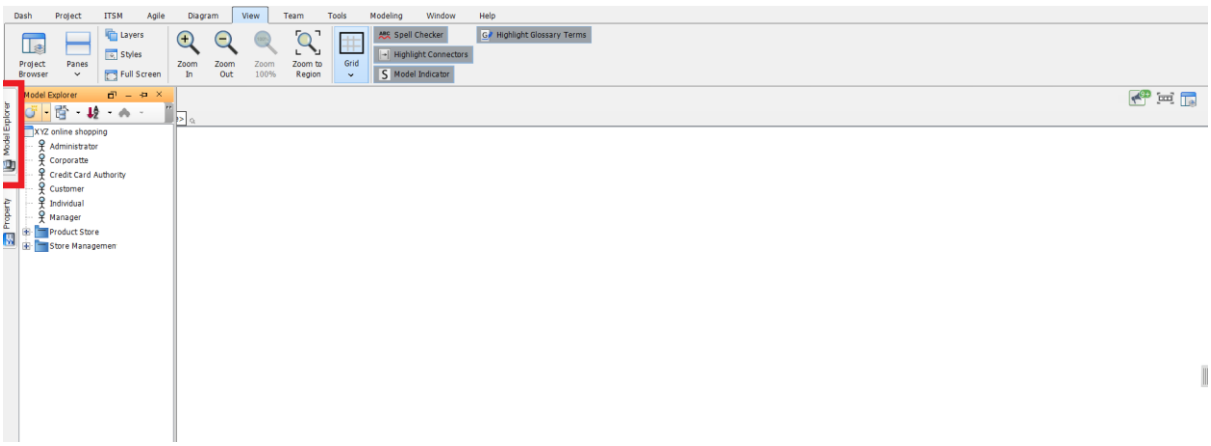


Figure 12. Adding actors in Class Diagram

If we think an application for mobile devices should be used only by Customers. And Administrator can do his work only from web version of application.

Now class diagram is ready.

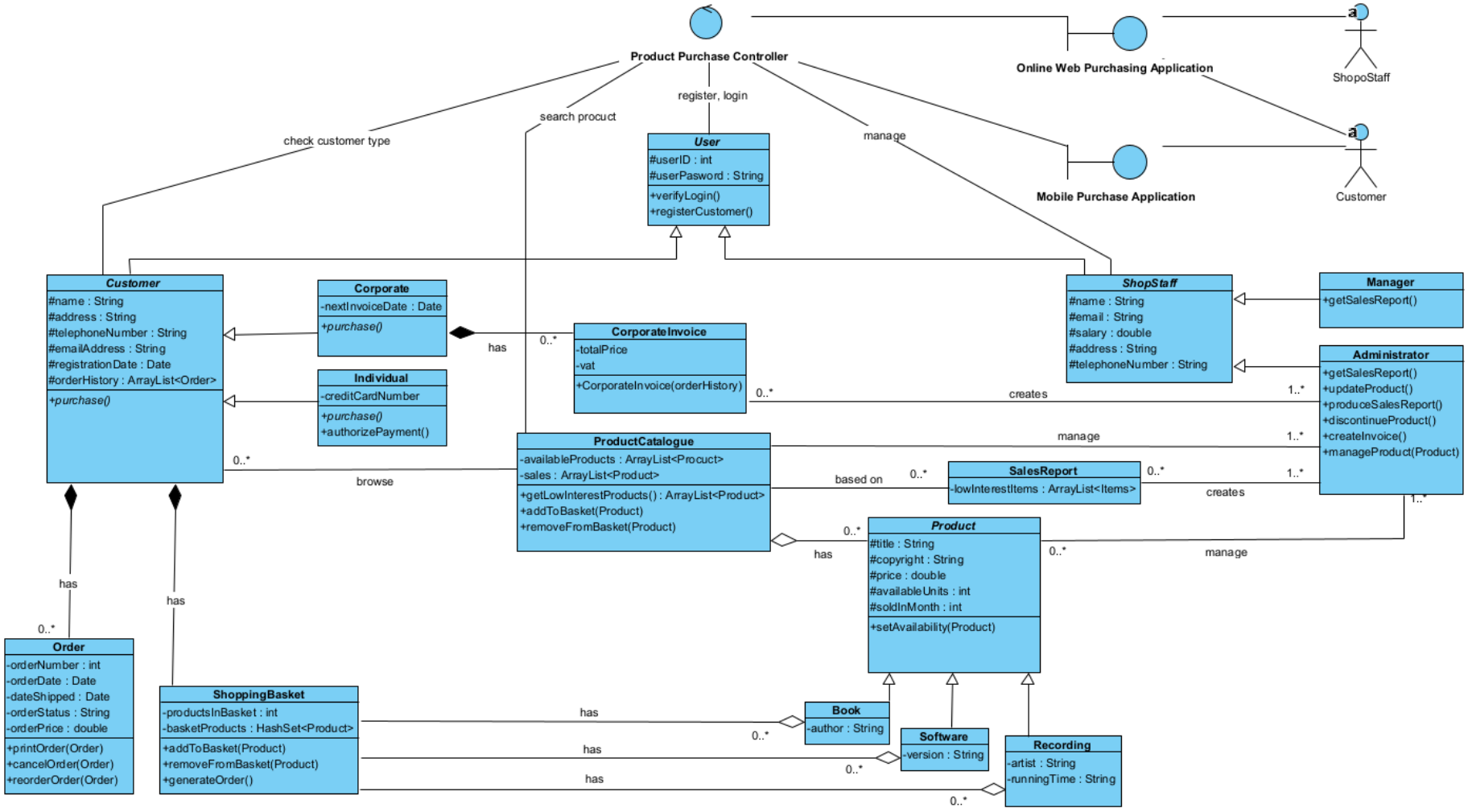


Figure 13. Final class diagram

Result/Discussion and Conclusions

During early development stages creating class diagram in parallel with use-case diagram, will benefit development with clearer visualisation of the system. This way there are more chances to build a structured system, with all necessary classes and class members. Class diagram also shows all relationships between classes and it is very important during development.

Sometimes system requirements might be not clear, and developer experience at finding specific classes, attributes and operations will play a big role. But it is very important to get feedback, and make changes if feedback is negative.

Even if the system is not complex, it is better to follow the UML approach to make sure that nothing is missed and that valuable system is delivered in the end.