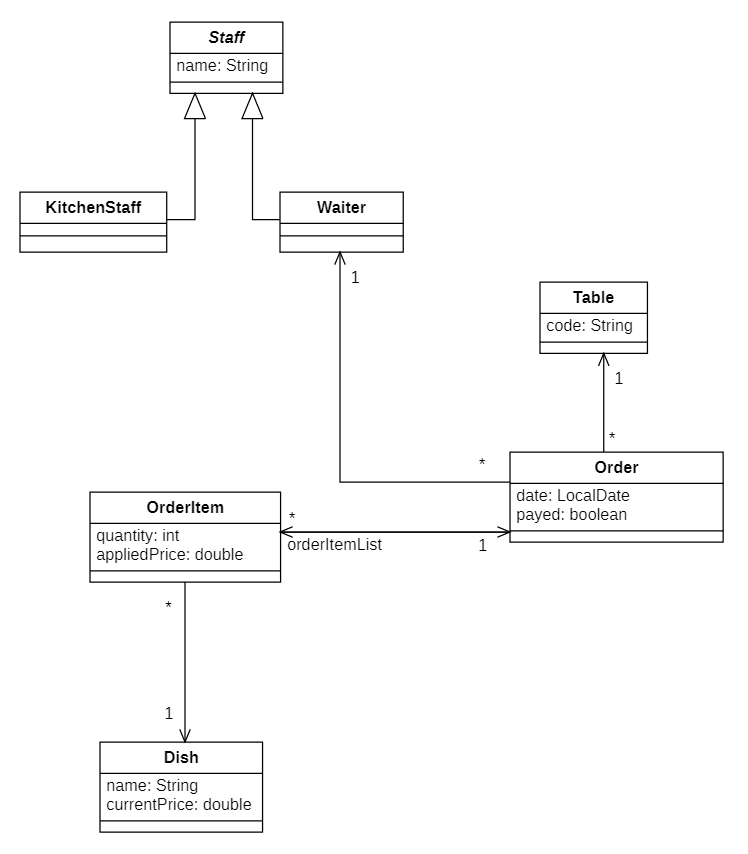
**Design patterns in RestaurantApp**

**Preparation**

* Download the zip file *surname\_firstname\_studentnr\_Restaurant2023* unzip, rename and copy the project to your github-folder. so e.g.:   
  Broos\_Jan\_r0670812\_Restaurant2023
* Before starting with the design patterns, we first make the entity diagram (in the package *model*).
* Some important notes to bear in mind:
* Note the associations and the directions of the associations (which may or may not form attributes in the entity concerned) and the multiplicities involved (1 or \* which affect the annotations to be added).
* As names of the associated attributes you use the default that IntelliJ proposes: e.g. *private Staff* or *private Table* Only when a class diagram contains a role attribute, you are obliged to use this (and not what IntelliJ proposes). This is the case where there is a \* and you have to use a plural. Be sure to respect this and all other names or the tests will not work.
* Because MySQL does not accept pre-defined words such as “Table”, “Order” and “date” in a query you have to change the table names/column names that JPA has to generate, e.g. @Entity(name=”restaurant\_table”), @Entity(name=”table-order”) and @Column(“orderdate”).
* After you have created all the required attributes in an entity, generate a no-arg constructor and all the getters and setters
* The entity "Staff" is abstract.



**Staff** is a member of the restaurant staff who may be **KitchenStaff** or **Waiter**.

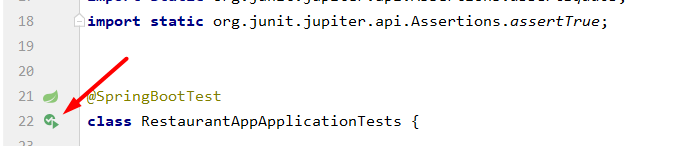
**Table** represents the tables located in the restaurant

**Dish** is an entity that keeps track of the name and current price of each dish offered in the restaurant

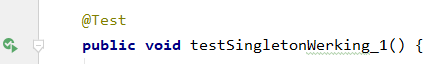
**OrderItem** keeps track of how much of one particular dish was ordered with what applied price for an **Order**

**Order** collects the objects of OrderItem that were registered by a waiter on a certain date for a certain Table

* In the start project, there are tests ready in the "test" folder. These tests are all in comments. After each implementation of a pattern, uncomment and execute the tests concerned. In IntelliJ, you can run tests by clicking in the margin of the test class file at the start of the class:



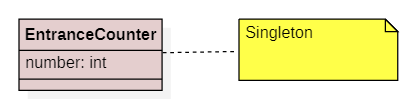
* You can also choose to perform only one test method by clicking in the margin at the level of that test method:

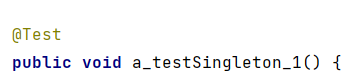


* **Please note** that no test will work in IntelliJ unless the corresponding methods being tested do not exist (or have a different signature). So to avoid having to code all the tested methods before you can run the tests, we recommend that you leave all tests in comments and systematically (if you have programmed anything) uncommented.

**Singleton pattern**

* Now we start with a concrete implementation of the Singleton pattern for the class EntranceCounter. The EntranceCounter represents the device that counts the customers coming in and then changes the number.   
  Create the class below with no-arg constructor and getter and setter. Please note that a class coloured grey is a "normal" class (POJO) and will not be stored in the database.



* Look in the presentation which code to add to ensure that only one object can be created from this class.
* Open the test file of your project and uncomment the testSingleton method and run the test file.
* If all your tests work,
* Open the MainController
* Uncomment the code inside the if branch *(request.getParameter("singletontest") != null)* and uncomment the test:  in the test-file
* run this test
* run your project
* In your browser, navigate to localhost:8080 and click on the "singleton test" button:

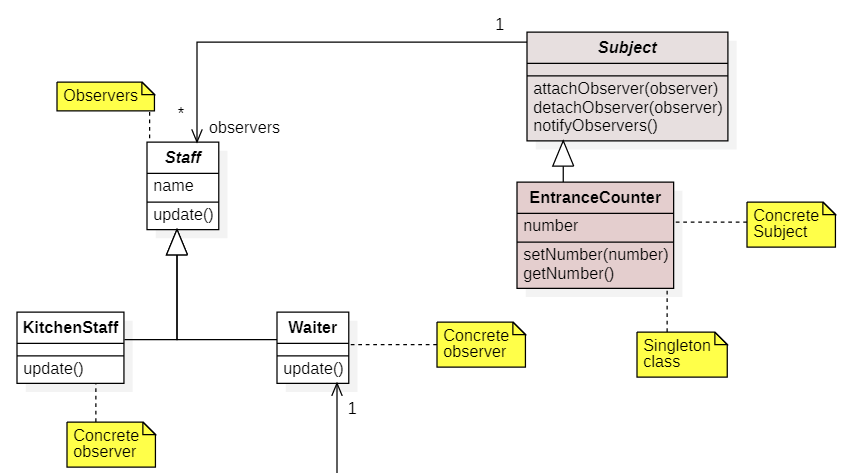
In your output window you will now see the proof of the singleton pattern's operation.

Afbeelding met tekst

Automatisch gegenereerde beschrijving

**Observer pattern**

* Now we start with a concrete implementation of the "Observer pattern" as seen in DesignPatternsGoF.pptx. We want to implement that both the kitchen staff and the waiters are informed as soon as customers have entered. Depending on whether they are waiter or kitchen staff, they will react differently to event.



* So create an abstract class Subject (is also a "normal" class (POJO), not an entity) and provide this class with the necessary methods to apply the observer pattern (according to the class diagram and the pattern we saw in the presentation). Also don't forget to add the necessary getter and setter if you are going to implement the association with *Staff* (observers).
* Adapt your existing class EntranceCounter according to the class diagram and add in the method setNumber(int number) the code so that not only the value of the attribute *number* is adapted but also all observers are notified.
* Now also adjust the classes Staff, Kitchen Staff and Waiter so that they also contain an update() method.
  + In the class Staff, the update() method is simply abstract;
  + In the KitchenStaff class, make sure that this update method generates the following output in the output window (with System.out.println)



"Serge" is the name of a kitchen staff member and "5" is the number of customers that have just arrived. For your information: this number is retrieved by calling up an EntranceCounter object and then using the getter for this object to retrieve the value of the number.

* + In the Waiter class, make sure that this update method generates the following output in the output window (with System.out.println)



In the same way to what you did for Kitchen Staff.

* Open the test file and uncomment the 2 testObserverPattern methods. Then run BOTH tests. You cannot run these tests independently of each other as they will not work.
* If all your tests work,
* Open the Main Controller
* Uncomment the code inside the if branch *(request.getParameter("observertest") != null)*
* run your project
* In your browser, navigate to localhost:8080 and click on the "observertest" button.

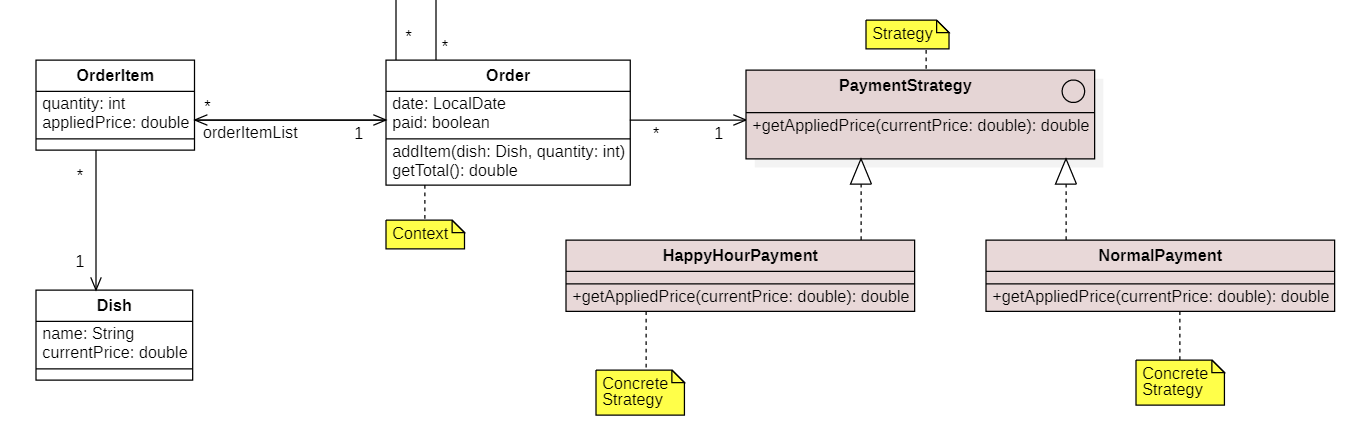
In your output window you will now see the proof of the observer pattern's operation.

Afbeelding met tekst

Automatisch gegenereerde beschrijving

**Strategy pattern**

* Now follows a concrete implementation of the Strategy pattern as seen in DesignPatternsGoF.pptx. This implementation should ensure that the applied price entered in "OrderedItem" (by Order) depends on the strategy applicable at any given time:
  + when the HappyHour payment is applicable, the appliedPrice (in OrderItem) = 0.8\*actualPrice (in Dish).
  + When the NormalPayment is applicable, the appliedPrice (in OrderedItem) = currentPrice (in Dish).



* In order to do so, add an interface PaymentStrategy and 2 concrete implementations HappyHourPayment and NormalPayment:
  + In PaymentStrategy you create an (abstract) method according to the class diagram
  + In HappyHourPayment you implement this method by returning a double = 0.8 \* currentPrice (in the parameter of this method)
  + In NormalPayment you implement this method by returning a double = parameter of the method.
* Also modify Order by adding the association. The attribute that should keep track of the association with PaymentStrategy is made "@Transient" because we are not going to store it in the database.
* Add the necessary getter and setter.
* Make sure that the paymentStrategy is set to NormalPayment by default (= in the Order constructor).
* In Order, add the *addItem* method. In this method, create a new OrderItem object and fill its attributes with the values contained in the parameters of this method and by using the setters. Caution:
  + the order of the OrderItem object, you must set it to "this order" (=this: you have an association in the 2 directions => the OrderItem object must know its OrderItem object and the OrderItem object knows its OrderItem objects).
  + the appliedPrice of the new OrderItem object is set to the value returned when you call the *getAppliedPrice* method of the PaymentStrategy attribute, and as a parameter you enter the currentPrice of the dish that was ordered.
  + You then add this new OrderItem object to the ArrayList of OrderItems of the Order
* In Order, also add the getTotal() method. This method goes through all the ordered items in the orderItemList and adds up the numbers multiplied by the appliedPrices. This total will then be returned.
* Open the test file and uncomment the 3 testStrategyPattern methods. Then run the tests.
* If all your tests work,
* Open the Main Controller
* Uncomment the code inside the if branch *(request.getParameter("strategytest") != null)*
* run your project
* Click on the "strategy test" button

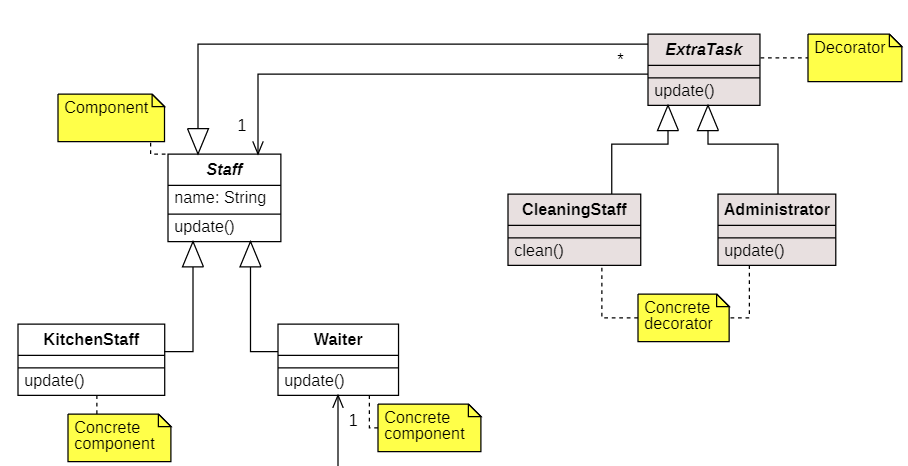
In your output window you will now see the proof of the strategy pattern's operation.

Afbeelding met tekst

Automatisch gegenereerde beschrijving

**Decorator pattern**

* What remains now is the concrete implementation of the Decorator pattern as seen in DesignPatternsGoF.pptx. This implementation must ensure that the staff can perform additional tasks depending on the situation, in addition to the tasks they already had:
  + Each staff member can also take on administration when someone comes in. This means that, in addition to his "normal" task as kitchen staff or hall staff, he must also be able to register the customers who come in in the customer file.
  + A staff member should be able to clean at certain times in addition to his regular task (update).



* Create the abstract class ExtraTask according to the class diagram (note the inheritance and the association). Also take care of the getter and setter of this attribute. In the update() method in this class ExtraTask you call the update() method of the associated staff attribute.
* Create the concrete classes CleaningStaff and Administrator.
  + In the class Administrator you create a method update() in which you FIRST execute the update() method of the superclass and then put the following output behind it (via System.out.println):



* + The number of customers in the output above is of course taken from your Input Counter again.
  + In the class CleaningStaff you create a method clean() in which you show the following output in the output window (with System.out.println)



Where "Manu" is the name of the staff member.

* Open the test file and uncomment the 4 testDecoratorPattern methods. Then run the tests.
* If all your tests work,
* Open the Main Controller
* Uncomment the code inside
* the if branch *(request.getParameter("decorator test") != null)*
* run your project
* Click on the "decorator test" button

In your output window you will now see the proof of the decorator pattern's operation.

Afbeelding met tekst, persoon, schermafbeelding

Automatisch gegenereerde beschrijving