**Milestone 2**

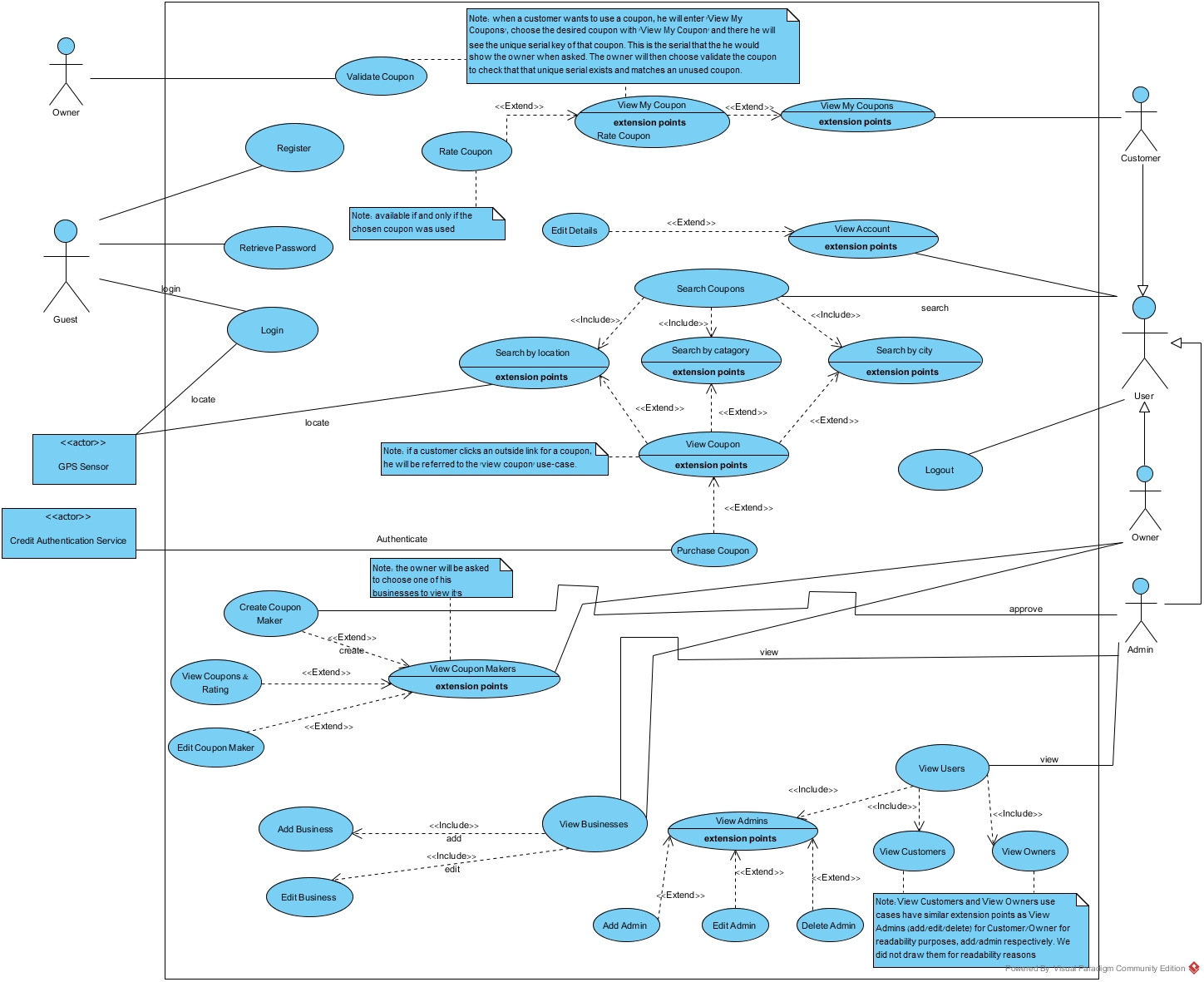
**Part A:**

**Task 1: Changes according to new requirements:**

* **Indexing Database:** we decided to add indexes to our database in relevant tables to improve queries runtime (for queries on fields that are often used).
* **Upgrading Server (fictive):** In order to provide the client a server that can support many users with better runtimes for loading pages and running queries, we changed the server’s host to high-end top of the line host.
* **Move logic to client-side:** today’s computers and smartphones are powerful, in-order to decrease the load from the server (thus increasing its capacity) we chose to move some of the logic to the client-side. In example, if the user submits a weak password (or an empty password) and presses ‘register’ the data would not be sent to the server, the code at the client-side will notify for a week password.
* **High Cohesion, Low Coupling:** For a single purpose (in example: edit business), its logic-handling and its view-handling would be separated into different objects (view object & control object).  
  Also, where needed, a class would offer public static methods so when another class needs to do a calculation that involves objects of the first class, he would not do the calculation himself, but use that function (this way, if the calculation formula is changed, it is transparent to the outside classes). For example: When a user registers he types a password, that password would be encrypted with a key and save in the database. Instead of having multiple functions using the same key to decrypt and encrypt, they would all use the same encrypt decrypt public & static functions offered by the relevant class (This way we could also hide the key for the encryption).
* **Easy new sensor implementation:** In order to add an option for a new senor, we would write an interface of a sensor, that has one method “samle()” that would return “Sensor Data”. For the “Sensor Data”: we would add this as an abstract class and for every type of data that could return from a sensor, we will implement a class for that data that would inherit from “Sensor Data” (for example: GPS sensor would implement the interface “Sensor” and therefore the method “sample()”, and a new class named “Location” would inherit from “Sensor Data”. This way, for a written code, if we want to change the sensor we could just change the line: “Sensor sens = New ‘SomeSensor()’” and everything else should stay the same.

**Note:** requirements document is attached at the end. Updates are marked in green.

**Task 2-A: Use-Case Diagram**



Note: After consulting with the staff, we decided that “Login” should not be involved in the sequence of each use-case that requires a logged-in user, this is for the reason that once a user is logged in, he does not need to login again to perform a task. Therefore, it is more correct and much clearer to have “Login” as a separate use-case, and have “isLoggedIn()” as a precondition for each required use-case.

Attached in file: ***UseCase.jpg***

Permissions Table:

This table will elaborate on specific cases where permissions are needed for an action and the use-case alone does not limit that action to the use of a specific user.

|  |  |  |  |
| --- | --- | --- | --- |
| **UC \ Actor** | **Admin** | **Owner** | **Customer** |
| **Add Business** | X |  |  |
| **Purchase Coupons** |  |  | X |
| **Edit Coupon Maker** | X |  |  |
| **Edit Business** |  | X |  |
| **“Delete” Business (move to N/A)** | X |  |  |

|  |  |
| --- | --- |
| Use Case #1 | |
| Use case name | Rate Coupon |
| Description | User will be asked to rate the experience of using a coupon. Rating options will range from 1 to 5 or to not rate at all. |
| Actors | Client |
| Pre-conditions | For some customer *cust* and coupon *coup*:  cust.isLoggedIn() & cust.ownsCoupon(coup.id) & coup.wasUsed() & coup.awaitsRating() |
| Post-conditions | For some customer *cust*, coupon *coup* and integer:  coup.awaitsRating() == False &  (coup.getRating()>0 & @before(coup.getNumOfVoters())+1=coup.getNumOfVoters() ) |  (coup.getRating()==0 & @before(coup.getNumOfVoters())+1=coup.getNumOfVoters() ) |
| Main success scenario | View My Coupons> View My Coupon > Rate Coupon   1. Customer decides to rate a coupon that he used sometime before. 2. Customer enters the system. 3. If customer is not logged in:    1. User is prompted to log in.    2. Reference: Login Use Case: Use Case #3 4. Customer enters “My Coupons” view. 5. Customer chooses the specific coupon that he would like to view. 6. Customer chooses the “Rate” option. 7. Customer chooses a rating (integer from 0 to 5). 8. The coupon is market as rated. 9. The rating of the CouponMaker of this coupon is updated. |
| Alternative / Extensions | 1. If system fails: system would restore the coupon and the coupon maker to their previous state (coupon will be market as unrated and coupon maker’s rating will be before the rating).   There are no alternatives or extensions for this use case. Of course the user can choose to never use the coupon or use it and never rate it, but there is no actual alternative. |

**Task 2-B: Use Case Scenarios**

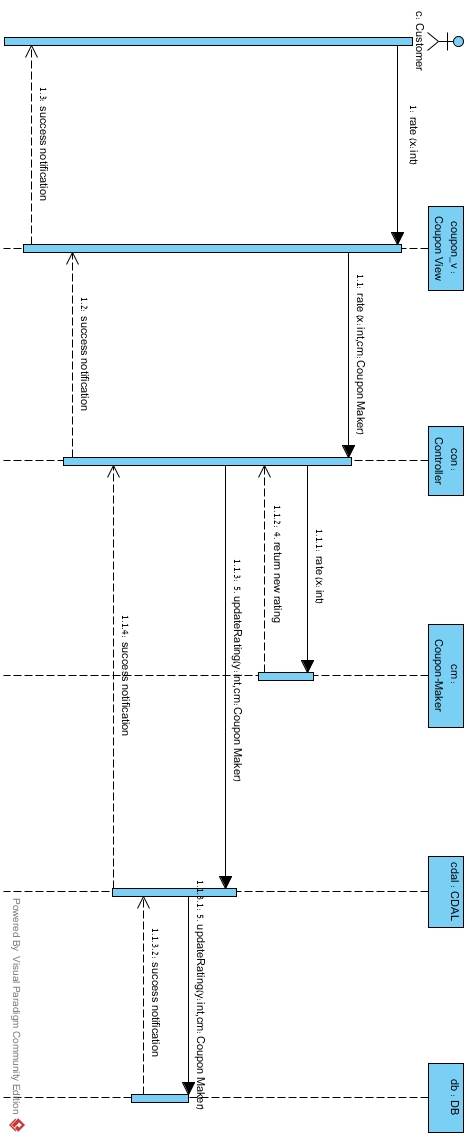
|  |  |
| --- | --- |
| Use Case #2 | |
| Use case name | View Account |
| Description | User will choose to view his or hers account details. |
| Actors | User |
| Pre-conditions | For some user *u*  u.isLoggedIn() |
| Post-conditions | True |
| Main success scenario | View Account   1. User will decide to view his account details 2. If user is logged out:    1. User is prompted to login    2. Reference: Login Use-Case (Use Case #3) 3. User enters the “Account” view. |
| Alternative / Extensions | 1. Edit Details 2. User enters “Edit Details” view. 3. User re-enters desired details. 4. UI will authenticate correctness (same as sign-up). 5. User details will be updated.   \*. If system fails: user details will be restored to the details before editing. |

|  |  |
| --- | --- |
| Use Case #3 | |
| Use case name | Login |
| Description | A guest (an unlogged-in user) will login with his user credentials. |
| Actors | Guest |
| Pre-conditions | True |
| Post-conditions | For some input *credentials* :  If credentials are correct : getUserByCred(*credentials)!=NULL*  else, if they are incorrect: getUserByCred(*credentials)==NULL* |
| Main success scenario | Login   1. User decides to login to the system. 2. User enters Login view. 3. User enters a valid existing username. 4. User enters a valid, existing password that is associated to the given username. 5. User presses “Login” 6. User is now logged in. |
| Alternative / Extensions | \*. If system fails: re-enter login view.  A. User entered incorrect details:   1. If user entered incorrect details:    1. System will prompt the user to re-enter his data    2. If user re-enters his credentials and presses “login”:       1. Return to 1. |

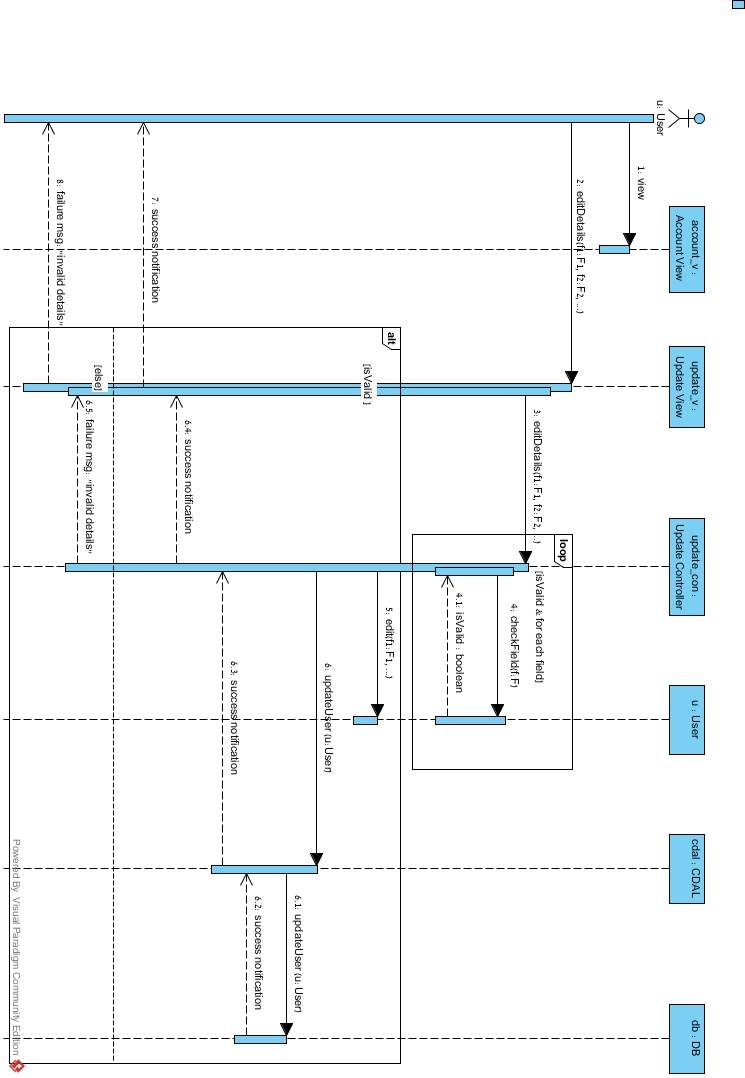
|  |  |
| --- | --- |
| Use Case #4 | |
| Use case name | View Businesses |
| Description | If user is an owner: he will be able to view his or hers businesses  If user is an admin: he will be able to view all existing businesses |
| Actors | Owner, Admin |
| Pre-conditions | For some user *user*:  user.isLoggedIn() && (user.isAdmin() || user.isOwner())  Note: if user is an owner and does not have business access is still granted but his view will be empty. |
| Post-conditions | True |
| Main success scenario | View Businesses   1. User decides to access businesses view 2. If user is not logged in    1. User is prompted to log-in.    2. Reference: Login Use-Case (Use Case #3) 3. The system shows a list of all relevant businesses (if the user is an owner, all of his/hers businesses, if the user is an admin, all existing businesses). |
| Alternatives/Extensions | 1. View Business    1. User select a specific business from the view and sees it’s details (such as rating, coupons that were purchased, ect.) 2. Edit Business (owner view) 3. System verifies that the user is an owner. 4. Owner re-enters data that he wishes to change (note that not all of the business data is changeable). 5. User presses “save”. 6. Edit Business (admin view) 7. System verifies that the user is an admin. 8. Admin chooses a new state for the business (for example: “inactive”). 9. User presses “save”.   \*. If system fails on b or c: restore data to its previous state before entering b or c (respectively).  Note: for legal purposes, a full delete option is not available (in order to track the business and owner related to a specific coupon). Instead, Admin would be able change the business status to ‘inactive’ in the edit function. |

**Task 3: Sequence Diagram**

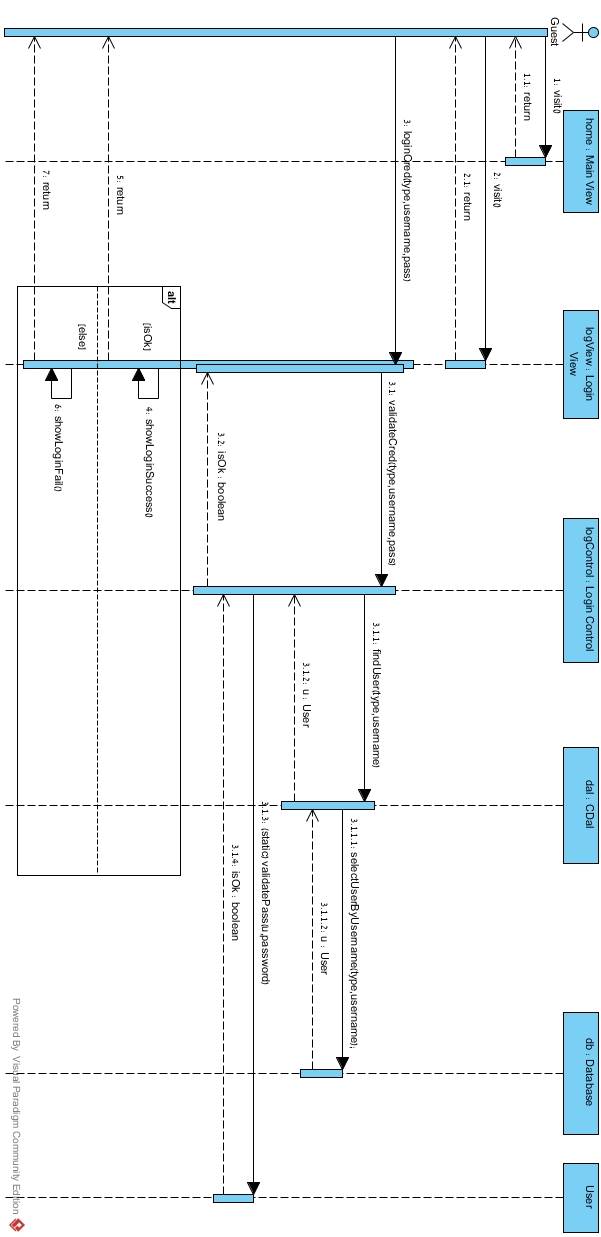
Sequence-UC1-Rate Coupon.jpg:



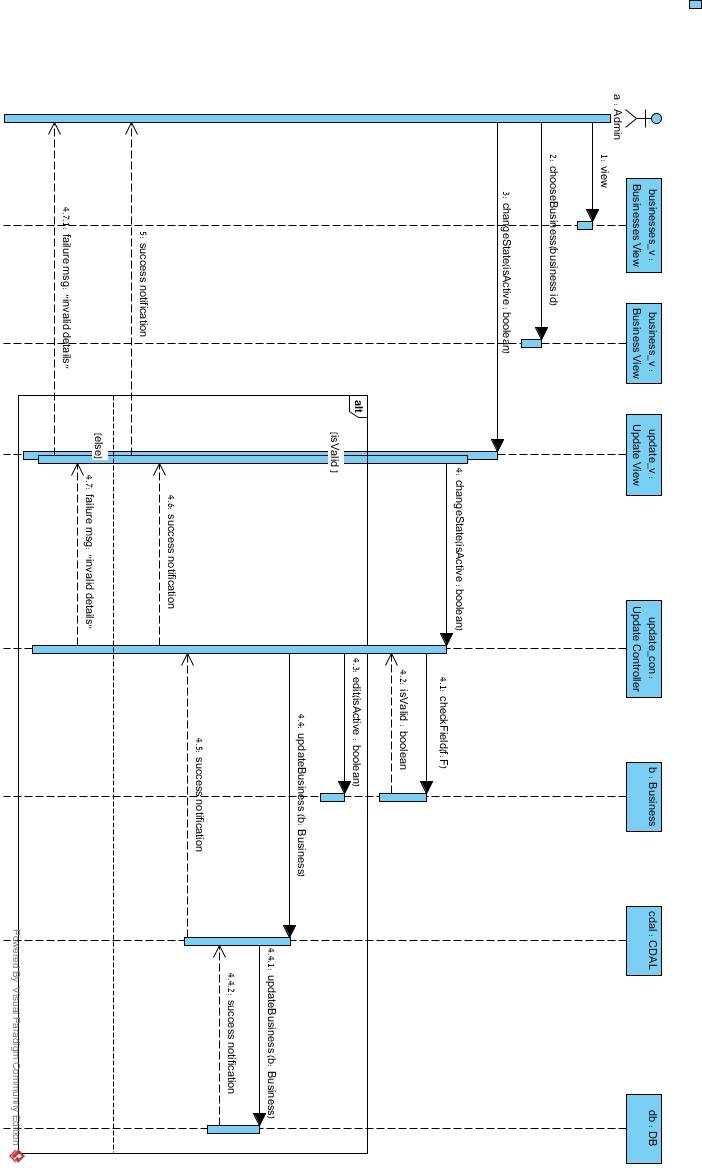
Sequence-UC2-View Account+Edit Details.jpg:



Sequence-UC3-Login.jpg:

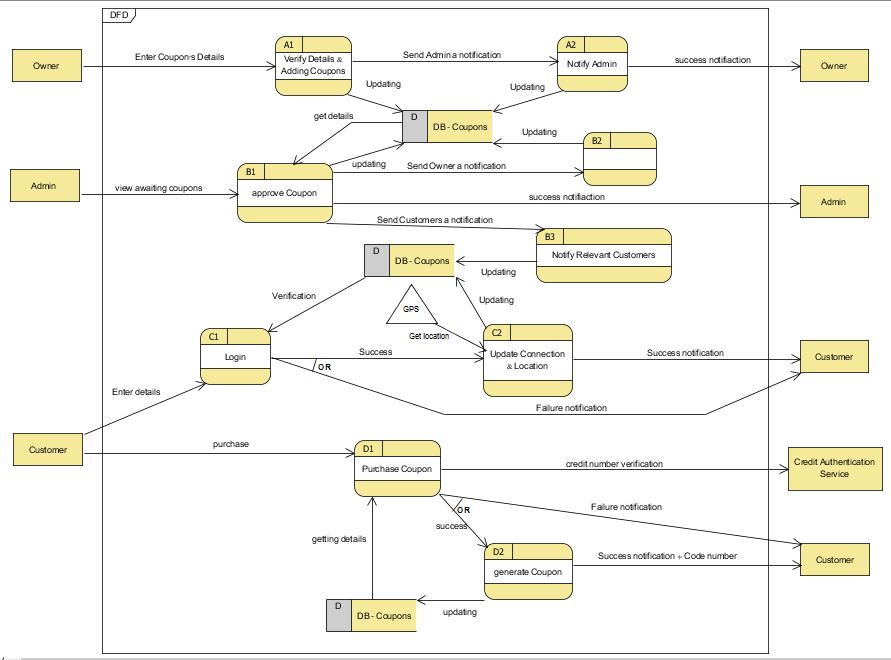


Sequence-UC4-View Business+Edit Details.jpg:



**Task 4: DFD**

DFD.jpg

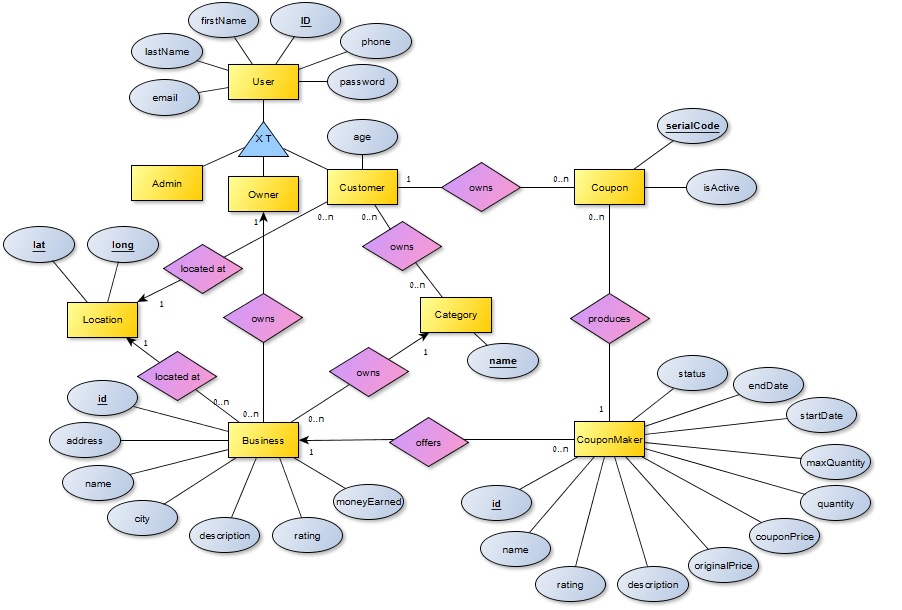


**Task 5: State Machine Diagram**

**Add s. machine**

**Task 6: Updating previous Diagrams**

ERD:



UML:

add here

Requirements Document:



**\***in case the images are too small: The original requirements.xls is in the repository

Notes:

**Git Repository:**

All of the project’s files (including the solution, pdf, word, diagrams and images are in our git repository)

https://github.com/oriba/Analysis-Design-2

Username: lolacoupons | Password: abcd1234

**Unit-Testing:**

1. Open:

NUnit: Coupons->NUnit-2.6.4->bin->nunit-x86.exe

1. Open tests:

File->open project->coupons->testUnit->bin->

debug->testUnit.dll

1. Run tests with the “Run” button.