**Abstract**

The advent of online retail shopping has introduced many platforms on the internet, such as Amazon, Alibaba, and eBay. This has led to creation of inventory systems and retail system that requires complex database structures and complex queries to facilitate the massive platforms. In our project, we tried to capture some of the basics of the complexity that enables buyers and sellers to interact while having employees manage the system. We used MySQL DBMS hosted on Amazon Web Services platform along with PHP frontend to accomplish this task. Our platform allows users to register as buyers and allow registered users to upgrade to sellers. It allows users to search, and save items listed by sellers in their shopping cart as well as place orders from different sellers. Sellers are given additional ability to manage their inventory, update product listing, and ship orders. The platform also allows the moderators or employees to add new departments, and upgrade users. This was accomplished by combination of simple CRUD (create, read, update, delete) SQL statements, in addition some complex statements such as INNERJOIN and LIKE query parameters were also used to retrieve and manipulate data in our database.

**Introduction**

The goal of our database is to create an online environment where users can buy and sell products, and allow employees of the company to manage the platform. The database stores user information such as login information, addresses, names, user status, orders as well as information about products, such as product names, inventories, reviews, and type of product. The platform allows users to register, search for products. Registered users are provided with a shopping cart, where they can place any number of items from various sellers and check out the items. Registered users can provide and update their name and address, and their password information. Registered users can also apply to become a seller, and sellers can add products and list them on the platform. Seller can also update the listing. Both the buyer and the seller involved in transaction can view the order detail of current and past transactions. Sellers can fulfill the order placed by buyers by shipping the product and providing shipping tracking numbers. The moderator acting as employee are able to accept or deny users requesting an upgrade, and add new departments for sellers to list products under.

**Project Design**

**User types:**

Buyer

Each buyer has a shopping cart linked to their account. While browsing the market place, (1) buyers can add product(s) in their shopping cart. Buyers can then access the shopping cart to (2) place order for product(s) of their choosing. In the personal page, buyers can (3) apply to become a seller if the conditions are met, and (4) edit personal information and password information. (5) Buyers can write reviews on products and rate other sellers.

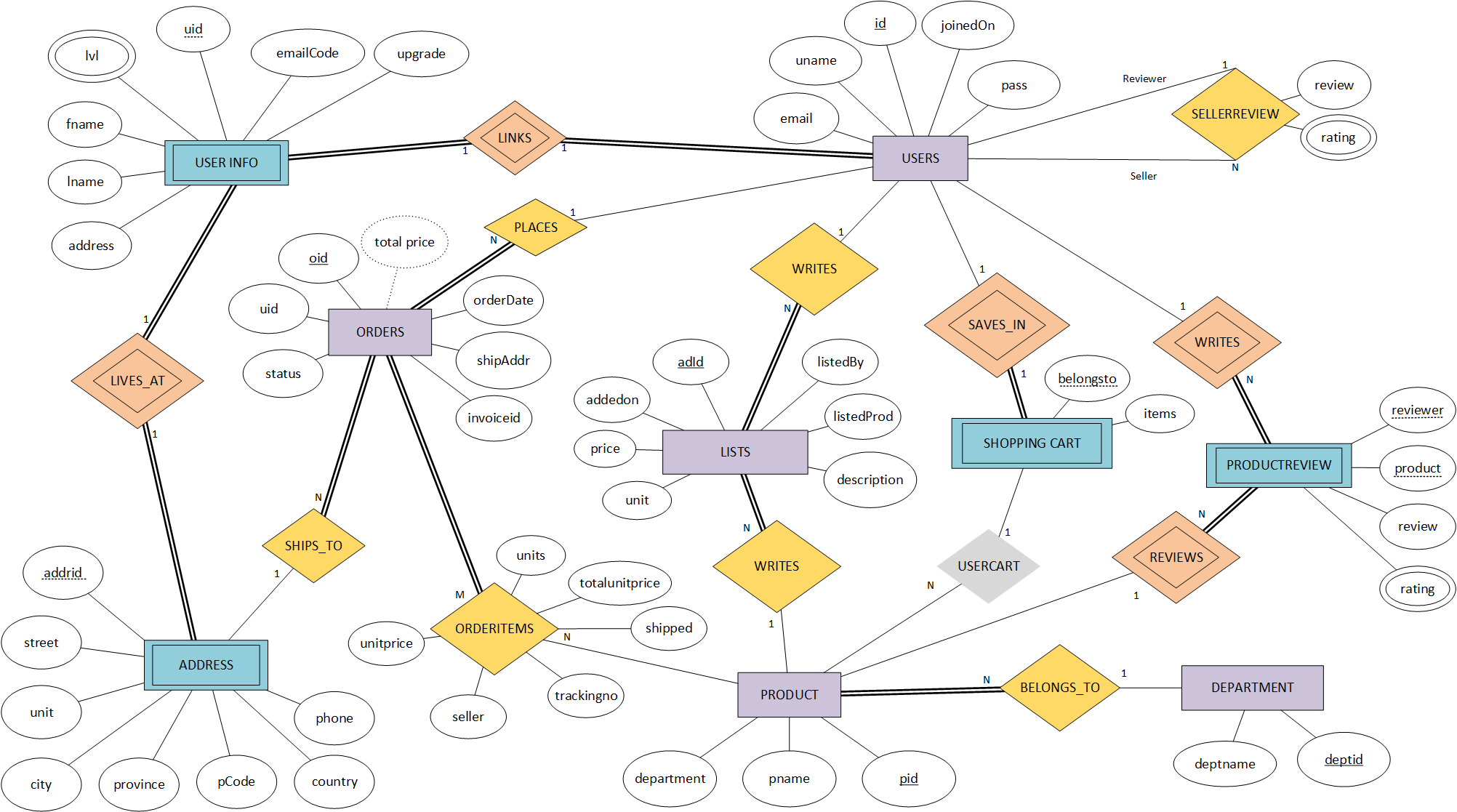
Seller

A seller is an upgraded buyer. (1) A seller create a listing which contains newly added product as well as relevant information such as product description, price, quantity available, and department it belongs to. The Seller can then edit or delete the listing(s). The system does security check so that a seller can edit or delete only their listing. (3) The seller can manage pending orders, where they can approve the orders for shipment to specific buyers, (4) and the seller can access the history of their fulfilled orders.

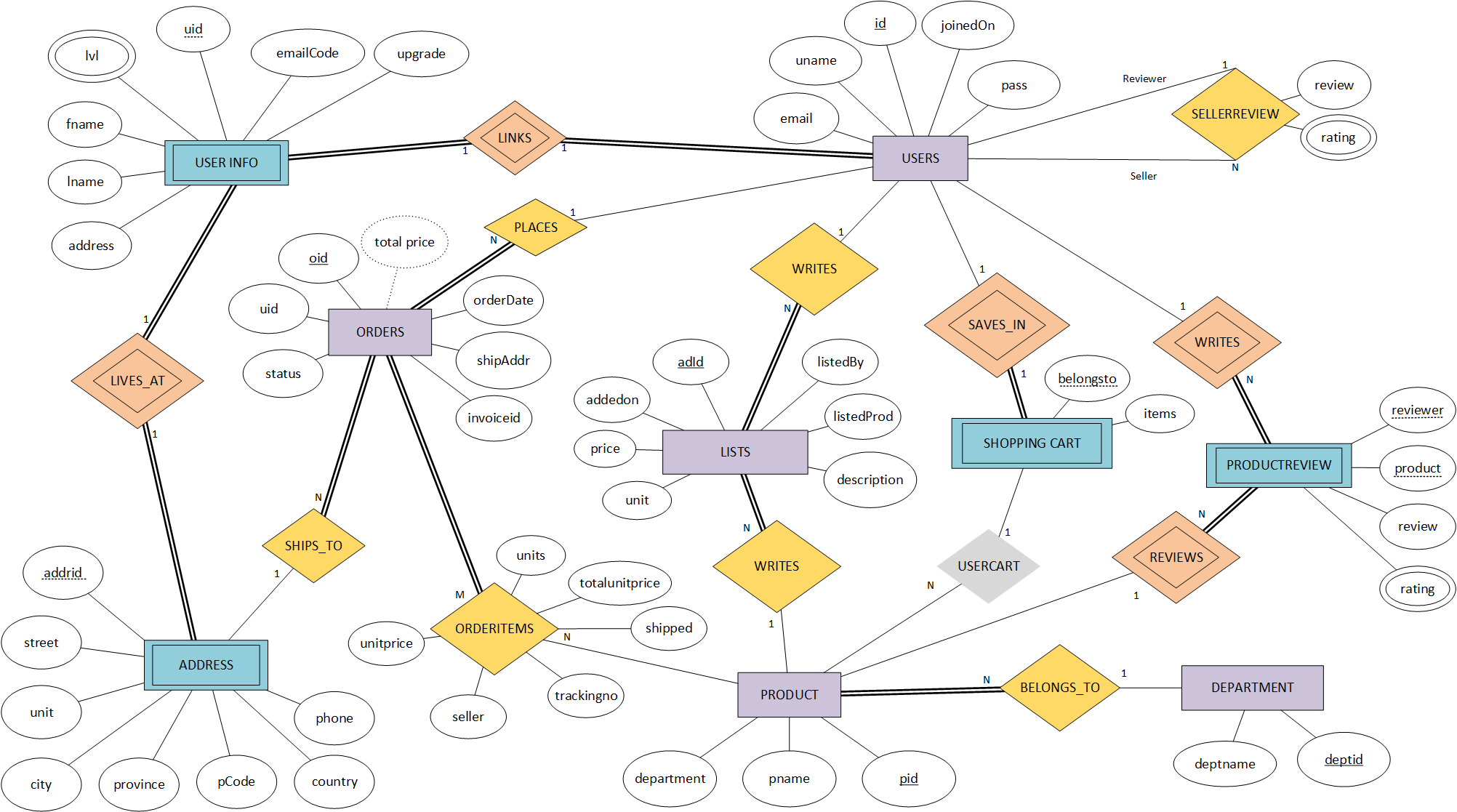
Moderator

Moderators are the overseers of the system. (1) They have the power to approve users who applied to become a seller, and potential employees who want to become moderators. (2) They can also add new departments, to accommodate the variety of products that may exists in the platform.

**ER**

The ADDRESS entity in the original ER diagram was an weak entity and contained 8 attributes – addressId, streetNo, streetName, streetType, unitNumber, postalCode, city, and province where addressId was supposed to be the partial key. In the new iteration of the ER diagram, the ADDRESS entity had following attributes updated - addressId was renamed to addrid which is now the primary key for the relation, streetNo, StreetName, streetType, were combined to into the street attribute to reduce complexity, unitNumber was renamed to unit, postalCode was renamed to pCode, and two additional attributes – country, phone were added to accommodate extra level of detail.

The DEPARTMENT entity in the original ER diagram contained 2 attributes – name and deptid, however in new iteration of the ER diagram, name was renamed to deptname to decrease confusion with other entities with similar attributes while doing join statement. The participation is also changed to reflect the correct relationship between DEPARTMENT and PRODUCT entities.

In the original ER diagram, LISTS was a relationship joining USERS to PRODUCT having 5 attributes - adId, description, unitsAvailable, price, and pId. In the new iteration of the ER diagram, LISTS was changed to its own entity type and now contains 7 attributes - adId, listedBy, listedProd, price, description, unit, and addedon, where unitsAvailable in original has been renamed to unit, and pId to listedProd, and two new attributes – listedBy and addedon were added. The decision to change LISTS from a relationship type to an entity type was brought on by the fact that a listing created by a seller required an unique identifier. The LISTS relationship represented an M:N cardinality between USERS and PRODUCT entities, but it was later changed to 1:N cardinality between USERS and LISTS and PRODUCT and LISTS with total participation on LISTS on both relationships.

The CONTAINS relationship in original ER diagram was changed to ORDERITEMS in the new iteration of ER diagram. In the original ER diagram, there were no attributes specified for the relationship. It has since changed to include six new attributes – unitprice, seller, trackingno, shipped, totalunitprice, and units.

The ORDERS entity in ER diagram contained 6 attributes – oid, uid, orderDate, shipAddr, status, and total price. In the new iterations, one new attribute was added – invoiceid.

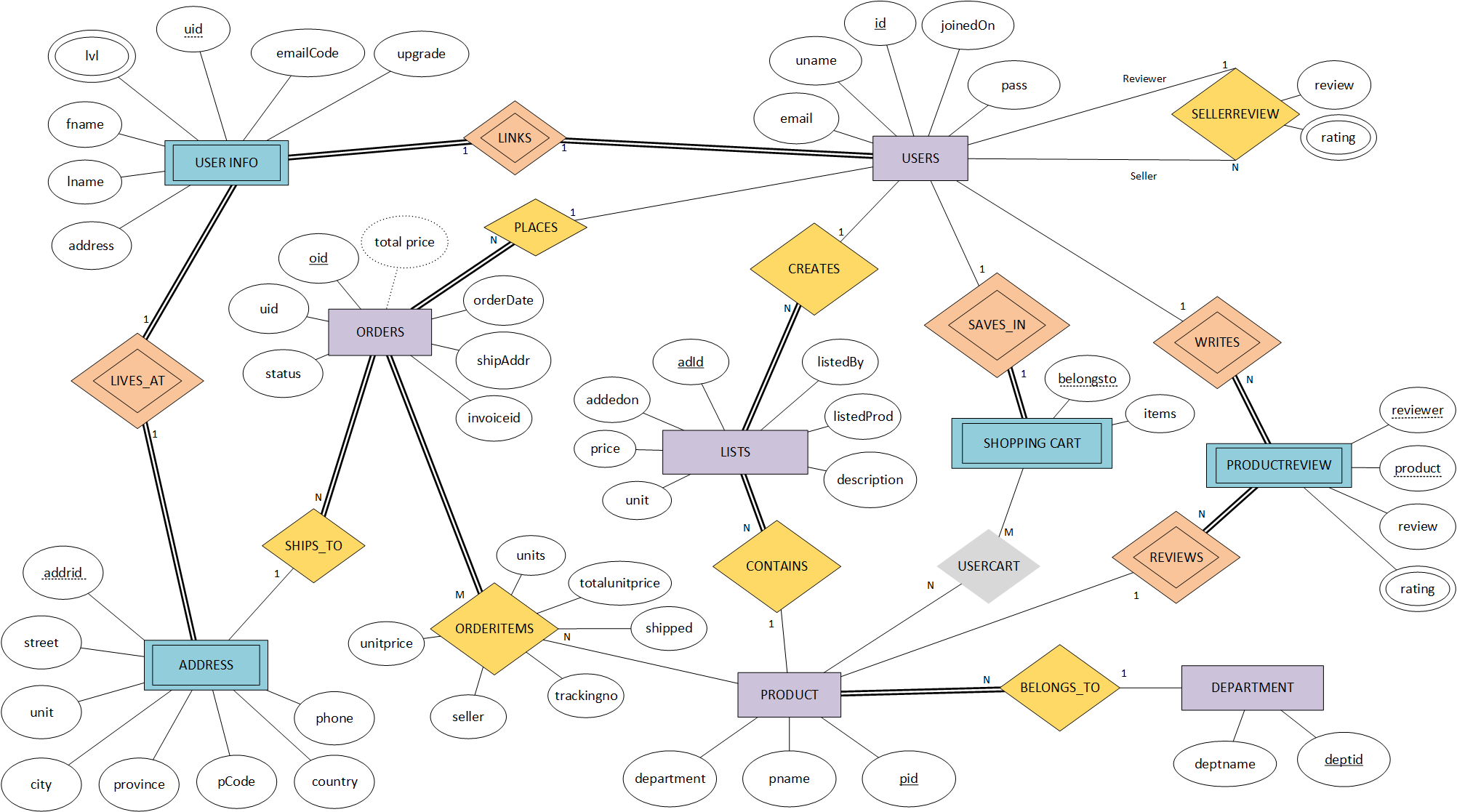
The PRODUCTREVIEW in the ER diagram contained 4 attributes – reviewer, product, review, and rating. In the new iteration, reviewer and product are shown as partial keys.

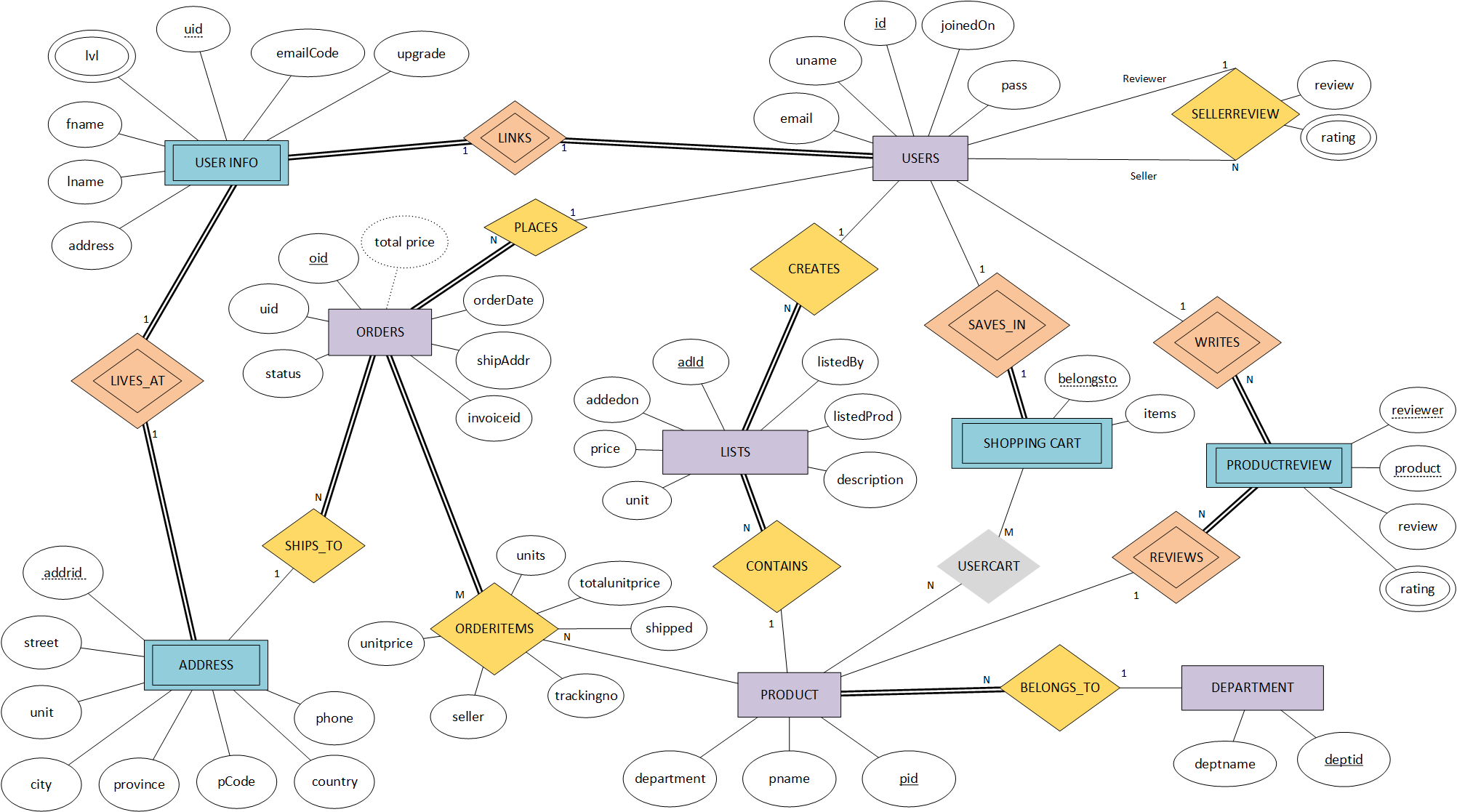
The SELLERREVIEW was an weak entity type in the original ER diagram and contained 4 attributes – reviewer, seller, review, and rating. In the new iteration, SELLERREVIEW was converted from weak entity to a relationship type to illustrate recursive relationship between USERS. It now contains 2 attributes – review and rating.

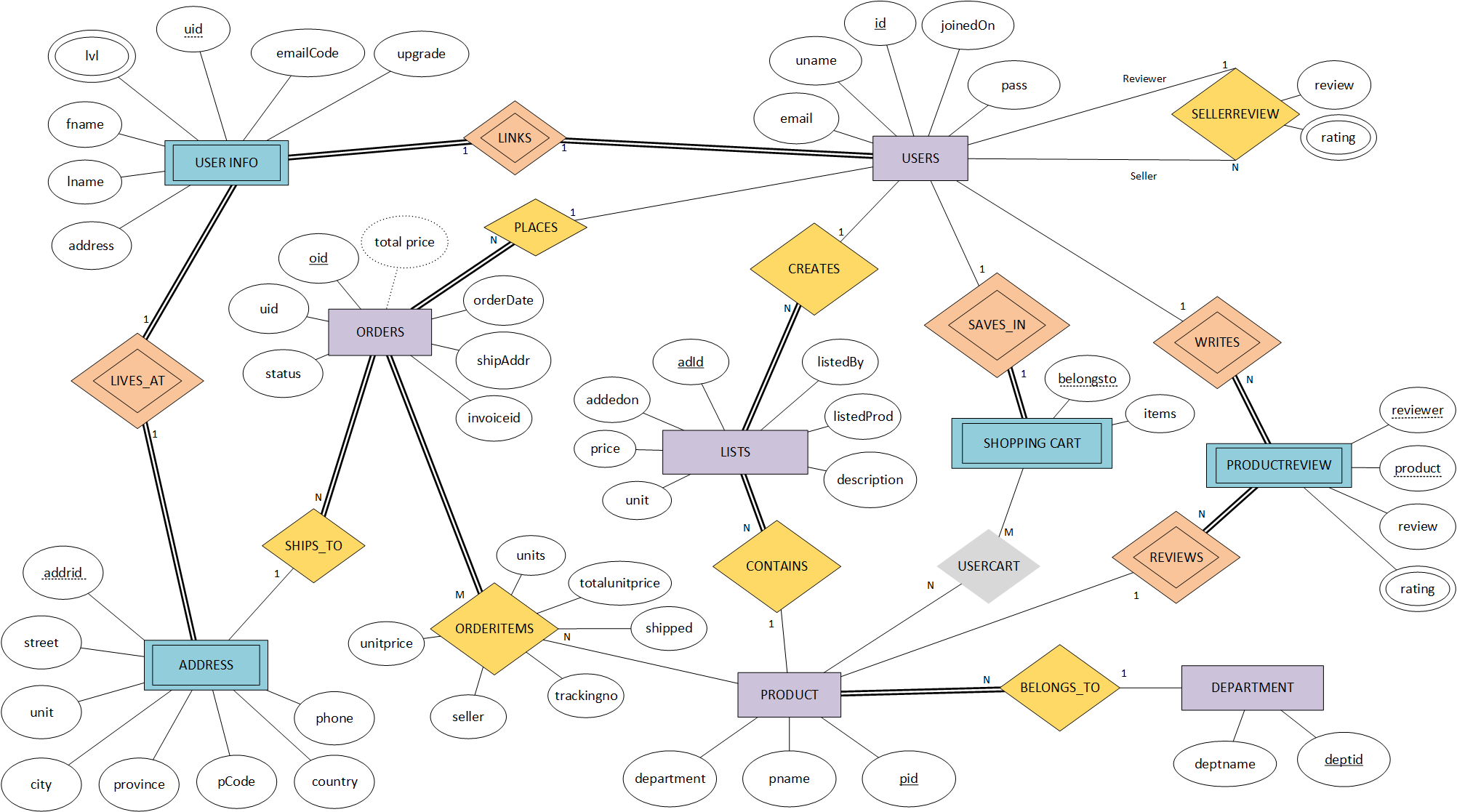
The PRODUCT entity in ER originally contained 3 attributes – pid, department, and name. In the new iteration, name attribute is renamed to pname to avoid confusion in join statement. PRODUCT now has a CONTAINS relationship with LISTS.

The SHOPPINGCART entity in ER originally contained no attributes. In the new iteration, two new attributes were added – items and belongsto. The CONTAINS relationship in the original ER diagram was a relationship between SHOPPING CART and PRODUCT entities with a 1:N cardinality. The CONTAINS relationship was renamed to USERCART with an M:N cardinality but was later removed, represented by the grey colour, because PHP allows serial array data to be stored as an attribute thus removing the need to create the USERCART relationship in the first place.

The USERINFO entity in the original ER diagram contained 7 attributes – level, uid, emailActLink, ccNumber, fname, lname, addressid. In the new iteration, ccNumber was removed, addressid was renamed to address, and a new attribute, upgrade, was added to specify if the user wants to become a seller, or moderator. In the new iteration USER INFO was changed from a regular to a weak entity as it cannot exist without a user. The LINKS relationship was modified to an identifying relationship to reflect this change.

The USER entity in the original ER diagram contained 4 attributes – email, passwordhash, uid, and username. In the new iteration, the joinedDate attribute was added to specify when the user joined the service. Username has since been renamed to uname. passwordhash has been renamed to pass, and uid has been renamed to id.

In the original ER diagram, PRODUCTREVIEW.ratings, and SELLERREVIEW.ratings, and USERINFO.level were multivalued attributes of their respective entities. However in the SQL environment, new tables were created for the multivalued attributes.

**Implementation**

The design of this database relied heavily on the ER diagram. The aforementioned ER diagram was used to create a relational schema diagram which was later used for external schema of the production database.

USERS table has five columns – id, uname, pass, email, and joinedOn. Where id is the primary key of USERS table.

USERS entity has identifying relationship LINKS with USERINFO entity. The relationship has 1:1 cardinality and has total participation on both side as each user can only have one set of personal information. Since USERINFO is a weak entity, we used foreign key approach, where USERINFO.uid is foreign key referencing the primary key USERS.id.

To represent where users live, USERINFO is linked to ADDRESS via 1:1 identifying relationship LIVES\_AT with total participation on both side because in our implementation, each user can only live and receive shipment in one address. We used foreign key approach, where USERINFO.address is a foreign key referencing the primary key ADDRESS.addrid.

USERINFO also contains column determining the status of the users (buyer, seller, moderator). This is done through checking USERINFO.lvl, where USERINFO.lvl is a foreign key referencing the primary key USERLVL.levelid.

USERINFO contains 7 columns – uid, fname, lname, lvl, address, emailCode, and upgrade. Where uid is a foreign key referencing primary key id in USER table, and lvl is a foreign key referencing primary key levelid in USERLVL, and address referencing partial key addrid in ADDRESS table.

The segregation of USERS, USERINFO, and ADDRESS entity is an unusual design, but it is a design that considers database security, restricting the availability of confidential user info, and the encapsulation of the address information allows other database entities to use the information at ease. The USERINFO is separated from USERS in the event of a query or database malfunction, and/or malicious SQL injection, it will now reveal user’s private information such as their names or address. The ADDRESS entity has been encapsulated because USERINFO entity and ORDERS entity refers to the information that is in the ADDRESS, this allows for an easier reference to address information.

Once a user has been upgraded to a seller status, they can add products and listing containing these products. The USERS entity has an m to n relationship LISTS with PRODUCTS entity. USERS entity has partial participation whereas PRODUCTS entity has total as users may or may not lists products, but all products are listed by users. Given the algorithm for m to n relationship, a new table called LISTS was created which contains LISTS.listedBy which is a foreign key referencing the primary key USERS.id, and LISTS.listedProd which is a foreign key referencing the primary key PRODUCT.pid.

We decided that sellers can only edit the listing once product has been added in the database. This decision was made since the ORDERITEMS entity pulls data from PRODUCT entity and if the seller was to remove the product, it will cause loss of integrity in the ORDERITEMS table. Instead of saving the state of orders in a separate table, which would be extremely redundant, we decided that user can update or delete the listings, but not the product itself.

In addition to creating new product and listing them for sales, users can place the orders to purchase seller’s products. The USERS entity has 1 to n relationship PLACES with ORDERS entities, where ORDERS has total participation because USERS may or may not have orders, but orders cannot exists without a user placing it. Given the algorithm, for 1:n cardinality, the entity on the n side – ORDERS, must contain foreign key that references primary key of entity on 1 side. In this case, ORDERS.uid is a foreign key referencing the primary key USERS.id.

Each ORDERS entities are linked to an ADDRESS entity by 1:n SHIPS\_TO relationships where the ORDERS entity has a total participation because an order cannot exists without a shipping address. However an address can exists without having any order shipped to it. Given the algorithm for 1:n cardinality relationship, the entity in n side must contain foreign key reference to primary key of entity on 1 side. This case, ORDERS.shipAddr is a foreign key which references primary key ADDRESS.addrid.

The ORDERS are related to PRODUCT via ORDERITEMS relationship, which is an m:n relationship with total participation on ORDERS side because an order must contain product, but product may not be contained in any orders. Given the algorithm for n:m relationship, a new table ORDERITEMS was created which contains foreign key ORDERITEMS.orderid which references the primary key ORDERS.oid, and foreign key ORDERITEMS.contains which the primary key PRODUCT.pid

The DEPARTMENT entity is linked to PRODUCT via BELONGS\_TO relationship, which has 1:n cardinality and total participation from the PRODUCT entity because a department may or may not contain a product, but a product cannot exists without belonging to a department. Given the algorithm for 1:n cardinality, the entity on the n side must contain a foreign key reference to a primary key on the 1 side. In this case, foreign key PRODUCT.department references the primary key DEPARTMENT.deptid.

The PRODUCT entity is linked to PRODUCTREVIEW entity via REVIEWS identifying relationship. It has 1:n cardinality 1:n with 1 on the PRODUCT side as a product can have many reviews, but a review is specific to the product. Given the algorithm for 1:n cardinality, the entity on the n side must contain a foreign key reference to a primary key on the 1 side entity. In this case, PRODUCTREVIEW is identified by FP PRODUCTREVIEW.prid, which references the primary key PRODUCT.pid.

USERS can add PRODUCTS in SHOPPING CART via SAVES\_IN relationship. The SAVES\_IN relationship has a 1:1 cardinality, and using the cardinality rule, SHOPPINGCART contains as foreign key SHOPPINGCART.scid which references the primary key USERS.id.

\*\*Furthermore, SHOPPINGCART can contain product via usercart relationship. It has 1:n cardinality with one cart being able to contain many products. Shopping cart saves set of orders linked to users,

PRODUCTREVIEWS is an weak entity linked to PRODUCT via REVIEWS, and USERS via writes. Both relationships have 1:n cardinalities, with PRODUCTREVIEW having 1 cardinality on both relationships. PRODUCTREVIEW entity is an weak entity, and therefore uses FK referencing primary keys in USERS and PRODUCT; productreviews.reviewer and productreviews.product respectively. PRODUCTREVIEW also has prid, a partial key used in conjunction with foreign keys to identify the review.

USERS entity is linked to seller review via writes relationships, which is recursive. The relationship is identifying and has 1:n cardinality. Hence USERREVIEWS contain as FK referencing PK of USERS entity, SELLERREVIEW.reviewer referencing USERS.id as well as SELLERREVIEW.seller which also references USERS.id.

**Implementation pt.2 (technical specification)**

The abundance of database system that are available on both commercial and open-source markets today, gave us many options to choose from. We considered many databases such as OracleDB, MongoDB, and NoSQL but given our needs and requirement for this project, MySQL database with running version 5.7 was the perfect choice. Along with MySQL being a very popular relational database, it has the ability to scale very easily and has many storage engines such as InnoDB, and MyISAM and its supports for variety of both primitive and complex data types.

Our project relies heavily on our database and variety of SQL statements to accommodate the front end operations and GUI. We will discuss some of the important SQL statements in following sections.

These SQL statement will follow chronological timeline of user’s experience on the platform.

Starting with sign up process, the first query checks if the user already exists in the system:

SELECT \* FROM `USERS` WHERE `uname` = `$uname`;

After this test is passed, and amongst other input sanitization, the user is created in our database. This effects three database tables using the following queries:

START TRANSACTION;

INSERT INTO `USERS` (“uname”, “pass”, “email”, “joinedOn”) VALUES (“$this->vuname”, “$this->hashpassword”, “$this->vemail”, $this->dtime);

INSERT INTO `ADDRESS` (“addrid”) VALUES ($newuserid);

INSERT INTO `USERINFO` (“uid”, “lvl”, “address”, “emailCode”) VALUES ($newuserid, 1, $newuserid, “$this->emailCode”);

COMMIT;