BearGo

Identifying GRASPS in project

AGM Islam
Grad Student, Computer Science
agm_islam1@baylor.edu
Baylor University

Maisha Binte Rashid
Grad Student, Computer Science
maisha_rashid1@baylor.edu
Baylor University

Razwan Ahmed Tanvir
Grad Student, Computer Science
razwan_tanvir1@baylor.edu
Baylor University

Swapnil Saha
Grad Student, Computer Science
swapnil_saha1@baylor.edu
Baylor University

Tonni Das
Grad Student, Computer Science
tonni_jui1@baylor.edu
Baylor University

Sep 24, 2022

Contents

1 Project vision

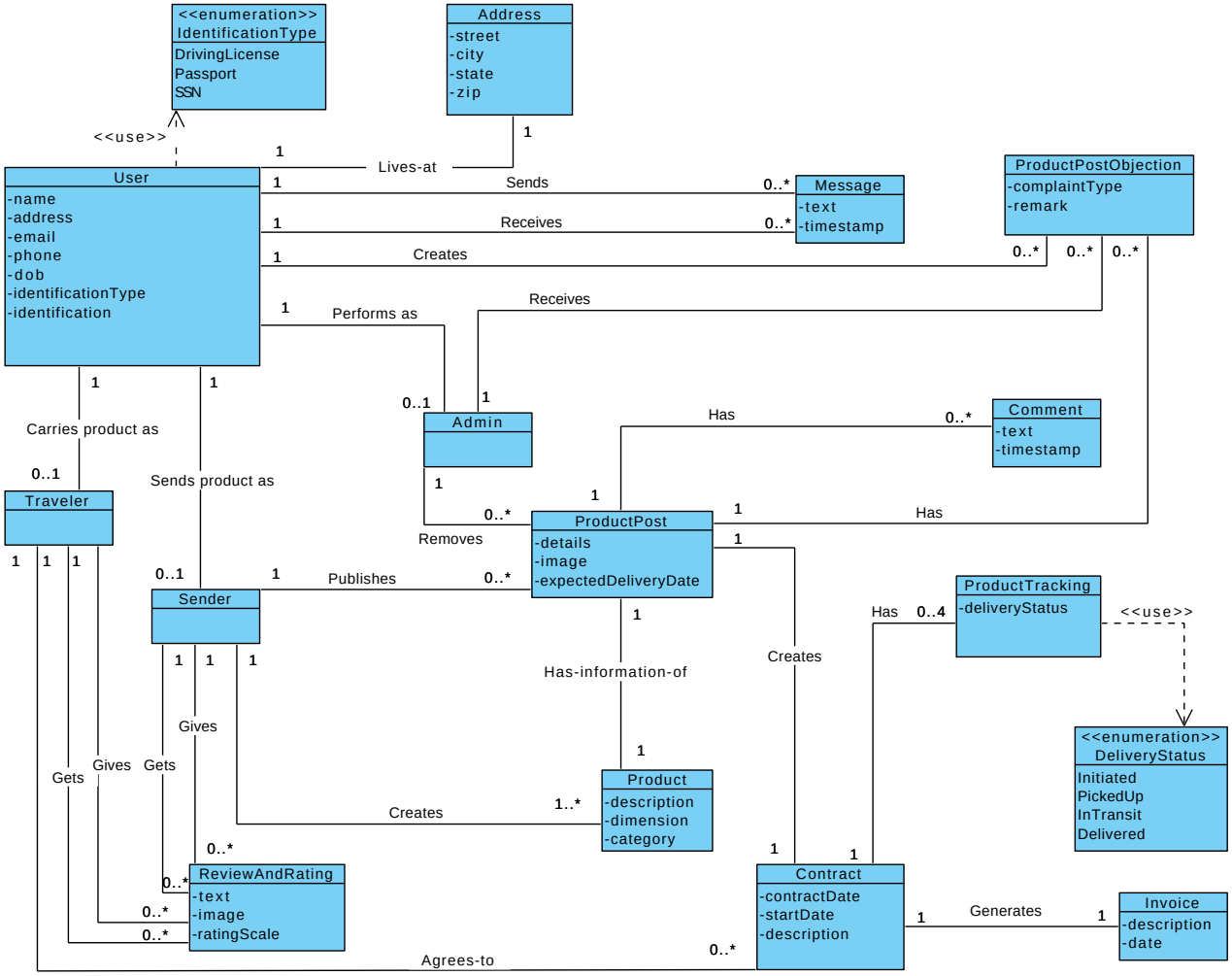
The vision of the project is to create a platform for people to send items to their known acquaintances in a convenient way.

Functional Requirements

1. Product Tracking: Traveler will update the status of different stages of the product delivery. The sender will finally confirm if the product is delivered.
2. Contracts of User: Admin will be able to see any user's activity such as what deals s/he has made so far and what are the statuses of those deals.
3. Report Review Confirming: Admin can choose to personally talk to reporters of a ProductPost or just notify them that the post was removed.
4. Create ProductPost: Product information posting (routing info, pick up location & travel time) in the application feed to find a traveler who can deliver the product.
5. Comment of ProductPost: Comment section under sender's ProductPost in which users can write comments if they have any queries regarding the productPost or other relevant details.
6. Create BlogPost: Registered users can upload their travel stories as blogs in Travel Blogging option.
7. Search ProductPost: Users can search other senders' productPosts based on different filters such as - date and location.
8. Ban User: A user can report against another user and the admin is responsible to ban a user from our platform.
9. Update ProductPost: Senders can also update the productPost. Senders can change the date, route, and pickup location. But once an invoice has been made or the delivery has passed then the sender can not update the productPost anymore.
10. Contract Creation: Sender will create a contract and after agreeing with a suitable traveler, he will confirm the contract. The system will generate an invoice.
11. Moderate ProductPost: Admin will decide to remove/keep the ProductPost if the ProductPost is reported by the users.
12. Send Notification: Real-time web push notification and email notification. Notification will be triggered on different events like: agreements signed, products delivered, etc.
13. Review and Rating: When the contract of a ProductPost will expire, both traveler and sender will see an option there to post a review and rating(1 to 5) to each other.
14. Peer-to-peer Chat: Any traveler/sender can contact to any particular sender/traveler to discuss the product details or any other travel information.
15. Social Media Share: A sender can share his/her ProductPost on social media.

2 GRASP in Domain Model

2.1 Project domain model



2.2 GRASP pattern: Information Expert

1. *User* - To assign the responsibility of knowing how many productPosts a user has, how many contracts a user has, how many reviews and ratings the user has, we look for a class that has the information to determine these. *User* class knows all these along with all the *Sender*, *Traveler*, *Admin*, and *ReviewsAndRating* instances. *User* is an information expert for these responsibilities.

2. *User* - To know about the streets, states, cities, and zip codes of a user, it is necessary to know about all the *Address* instances. *User* is an information expert for this responsibility.

3. *ProductPost* - To know about the user's opinion about a *ProductPost*, it is necessary to know about all the *Comment* instances. *ProductPost* is an information expert for this responsibility.

4. *Contract* - *Contract* is the information expert on answering *Contract* description (which users are in this agreement, what is the product, what is the status of the delivery currently, etc.). It is necessary to know about the *Product* instance of a *ProductPost* and *User* instances and *ProductTracking* instances. A *Contract* instance contains these. A *Contract* is an information expert for this responsibility for knowing what is the product of this contract and who are the sender and traveler of this contract.

5. *ProductTracking* - To fulfill the responsibility of knowing the status of any product delivery, a *ProductTracking* needs to know *DeliveryStatus*. The *ProductTracking* is the information expert on answering its *DeliveryStatus*.

To fulfill the responsibility of knowing and answering the user's total number of contracts and their states, three responsibilities will be assigned to three design classes, *Contract*, *ProductPost*, *ProductTracking*. The fulfillment of this responsibility requires information that is spread across these classes of objects. So, they are the "partial experts" who will collaborate on the task.

2.3 GRASP pattern: Creator

1. *ProductPost* is a creator of *Product*.
   Whenever a sender creates a productPost, the system will create a *Product* instance by extracting the information from ProductPost. Therefore, *ProductPost* class is responsible for creating a *Product* instance, hence the creator.

2. *ProductPost* is a creator of *Contract*.
   Whenever a sender creates a productPost system will create an initial *Contract* instance for a particular productPost using the information from ProductPost. So, the *ProductPost* class is responsible for creating a contract.

3. *User* is a creator of *ProductPost* and *ProductPostObjection*.
   A user creates these class instances, hence *User* is a creator.

4. *Contract* is a creator of *Invoice*.
   A *Contract* instance contract generates an *Invoice* instance using the data from the class.

2.4 GRASP pattern: Low Coupling

1. Coupling exists between *ProductPost* and *Product*. For creating an instance of *ProductPost*, we will have attributes that refer to *Product* as productPost will contain the details about the dimension, description, and category of the product. *ProductPost* also will have attribute of *Contract*. For creating a *Comment* instance, a parameter with productPostID is required and an attribute referring to *Comment* instances is in *ProductPost*. This information is collaborative and simple to reuse, so, we are considering them low-coupling.

2. Low Coupling exists between *ProductPostObjection* and *ProductPost*. *ProductPostObjection* will have attribute of *ProductPost*.

3. Coupling exists between *User* and *Address*. There will be an attribute referring to multiple instances of *Address* as a user can have multiple addresses. This dependency is easier to follow and thus refers to low coupling according to our consideration.

4. Coupling exits between *Contract* and *ProductTracking*. *Contract* will keep information about some attributes from *ProductTracking* and *ProductPost* which is why low coupling exists between them.

2.5 GRASP pattern: High Cohesion

1. *ProductPost* has low cohesion as it has the responsibility to create both product and contract. However, our system suggests that a product and contract must be created when a user creates a ProductPost. Hence we can not remove the product and contract creation responsibility from ProductPost.

2. *User* has high cohesion because a user has moderate responsibility and it collaborates with other classes to fulfill a task. *User* class only collaborates and delegates.

2.6 GRASP pattern: Controller

Here are some of the controller classes we initially identified.

1. *ProductTrackingController*: *ProductTracking* controller handles updating the status of the product delivery. It routes requests to *Contract* entity and interacts with the view.
2. *ContractController*: If an *User* instance has an identificationType of "Admin", it can get any other registered user's contract details with productTracking status. So, *ContractController* controller handles this task. In addition, it handles updating *Contract* instance attributes.
3. *ProductPostController*: This controller routes *User* instances' request to create a *ProductPost* instance with small and specific operations. Furthermore, it handles the situation where any *User* instance requires searching for a specific *ProductPost* instance with possible filtering operations.
4. *ReportProductPostController*: It routes an *User* instances' request to store a complain against a *ProductPost* instance.
5. *ChatController*: This controller class handles peer-to-peer chat between two *User* instances.

2.7 GRASP pattern: Polymorphism and Dynamic binding

In our system model, we have a *User* class that will have a *getReport*() method. But the sub-classes *Admin*, *Traveler*, and *Sender* will implement it differently. For example, for *Sender*, it will give the created *ProductPost* related information. For *Traveler*, it will give the delivered *ProductPost* related information. If we have $User\ u = new\ Traveler()$ and call $u.getReport()$, we will use the *getReport*() implementation of the *Traveler* class. In this way, we can avoid writing conditional statements.

2.8 GRASP pattern: Pure Fabrication

For responsibility assignment, the Expert suggests that *User* instances are persistent and responsible for many user-related tasks. However, we will have to place a lot of work related to DB operations to the object *User* which is incohesive. *User* class is then coupled to the DB interface and it does not bring reusability. We can create a new class (e.g. SpringBoot Repository) responsible for DB persistence to ensure high cohesion and reusability. This way the responsibilities are handled by an artificial class (e.g. UserRepository) with a very specific set of related tasks (DB operations) to support reuse and high cohesion.