Pet Diaries

Project Part 3

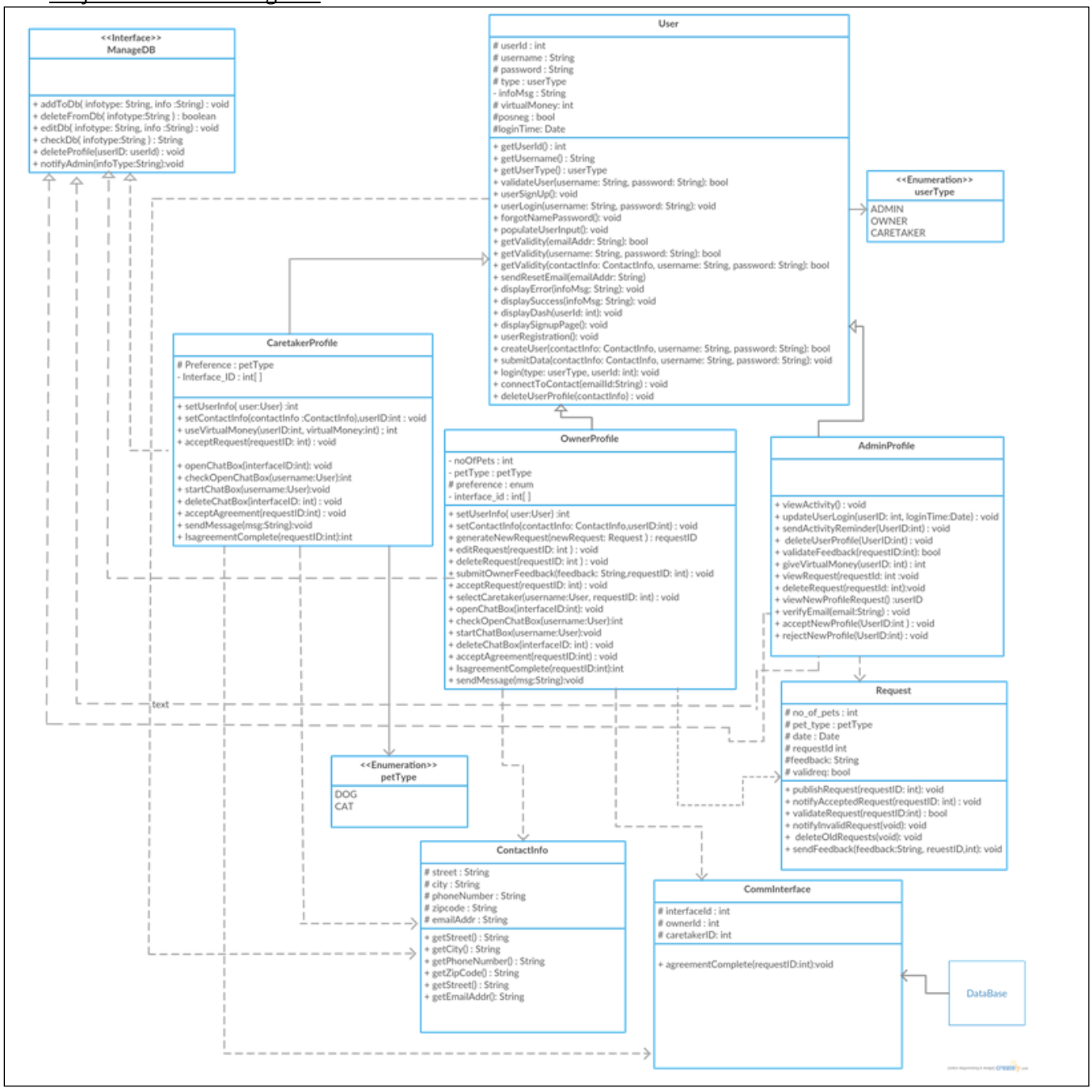
Team:

Ankita Manjrekar

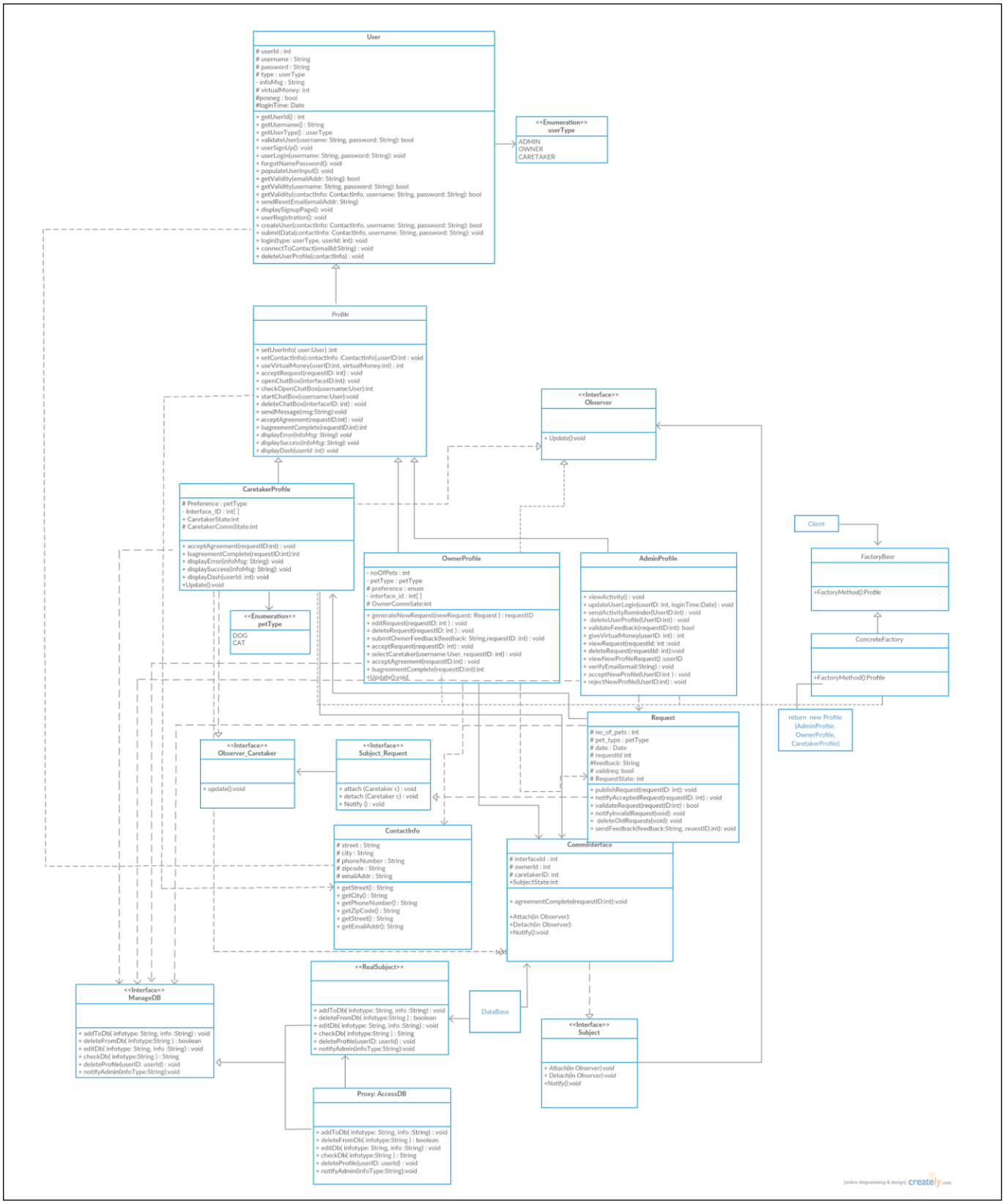
Anne Maria Vasu

Gayatri Mestry

I. Project Part 2 class diagram:

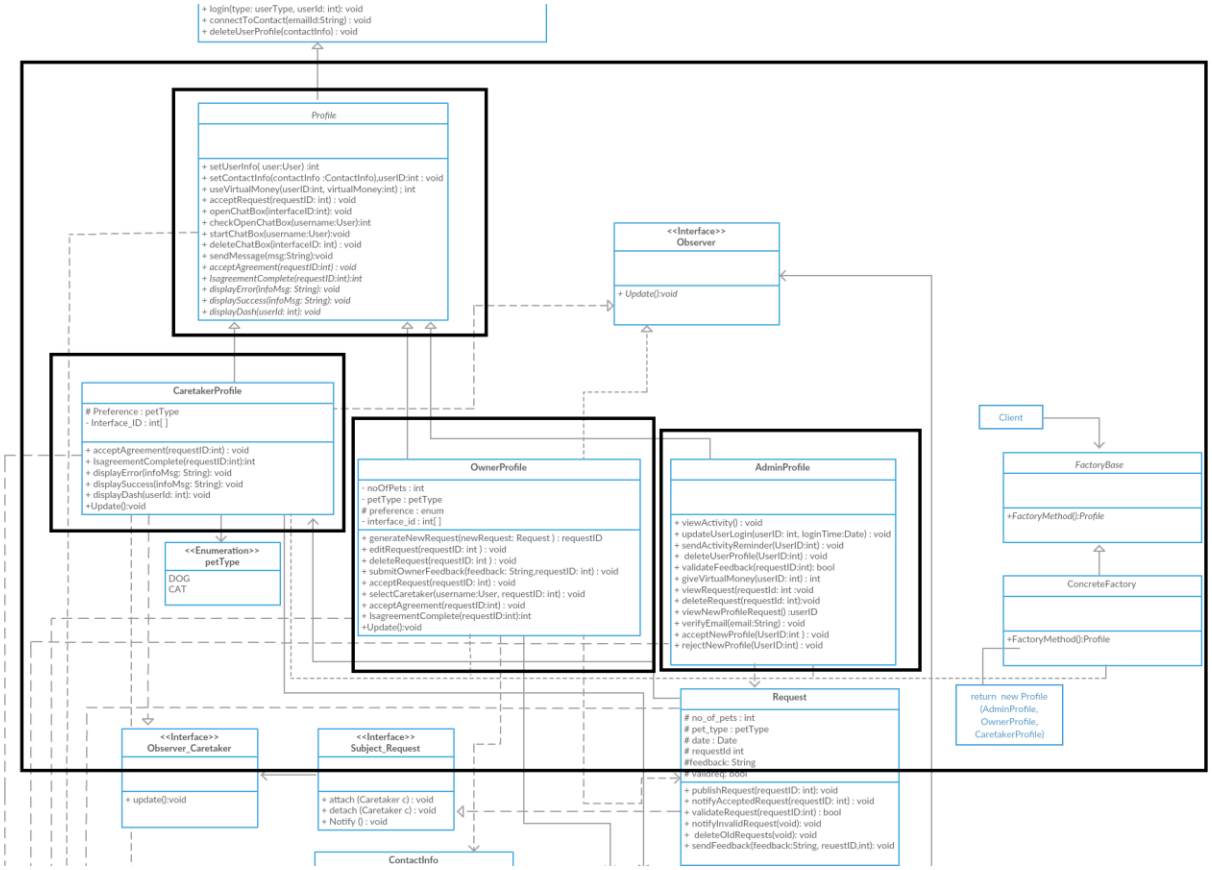


II. New class diagram implementing Design Patterns



III. Design patterns

1. Template Pattern

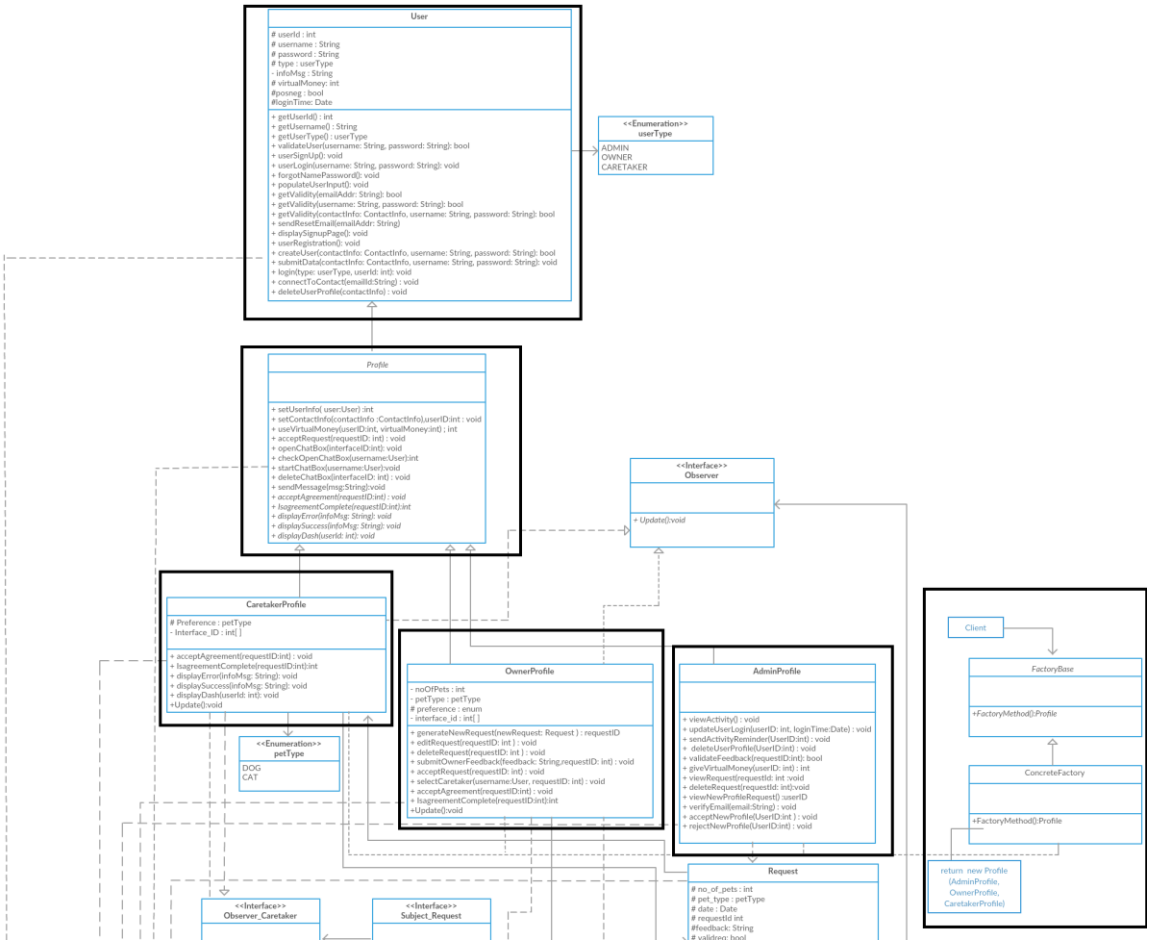


Portions of the new class diagram where the new design pattern has been implemented is shown above.

- All user profiles - AdminProfile, OwnerProfile or CaretakerProfile share a number of common algorithmic steps, and a few steps that are different.
- For example, all user profiles follow the same algorithm to set required contact information using a call to setContactInformation(). All user profiles use the same algorithm to open a new chat box interface using a call to openChatBox(id).
- However, steps done to display dashboard to different users can be different- displaying dashboard of an Owner might require execution of a different set of steps compared to displaying the dashboard of a caretaker.
- Template pattern has been implemented to avoid duplication of similar code in all classes, reusing what can be reused and changing code only in one place.

AdminProfile, OwnerProfile and CaretakerProfile extend from abstract class Profile. Profile has some concrete methods such as openChatBox, checkOpenChatBox etc., along with some abstract methods such as displayDash, displayError etc. which have follow separate steps and are defined as part of AdminProfile, OwnerProfile and CaretakerProfile classes.

2. Factory Pattern:



- There are three distinct users in Pet Diaries - Admin, Owner and Caretaker, each with a different profile. All three users have separate classes that represent them - Admin is represented via the AdminProfile, Owner via the OwnerProfile and Caretaker via the CaretakerProfile.
- The User class is a common base class that defines the **login** method. Abstract class inherits from User. Profile will use the **login** method to create objects of the AdminProfile, OwnerProfile and CaretakerProfile classes and hands over control to these objects - these objects display the required dashboard, allow user to select various options displayed in the dashboard etc.
- It is impossible for the Profile class to anticipate which type of profile class needs to be instantiated till the user hits the login button. Issues like tight coupling can be introduced if the Profile class uses an "if-else" block to determine what profile needs to be created.
- This can also lead to violation of open-closed design principle.

The factory pattern has been used for profile creation to determine the type of user profile to be created at runtime and to reduce number of code changes if more users are added in future. Portions of the new class diagram where the new design pattern has been implemented is shown below. As can be seen, Factory method defers instantiation to subclasses.

3. Observer Pattern:

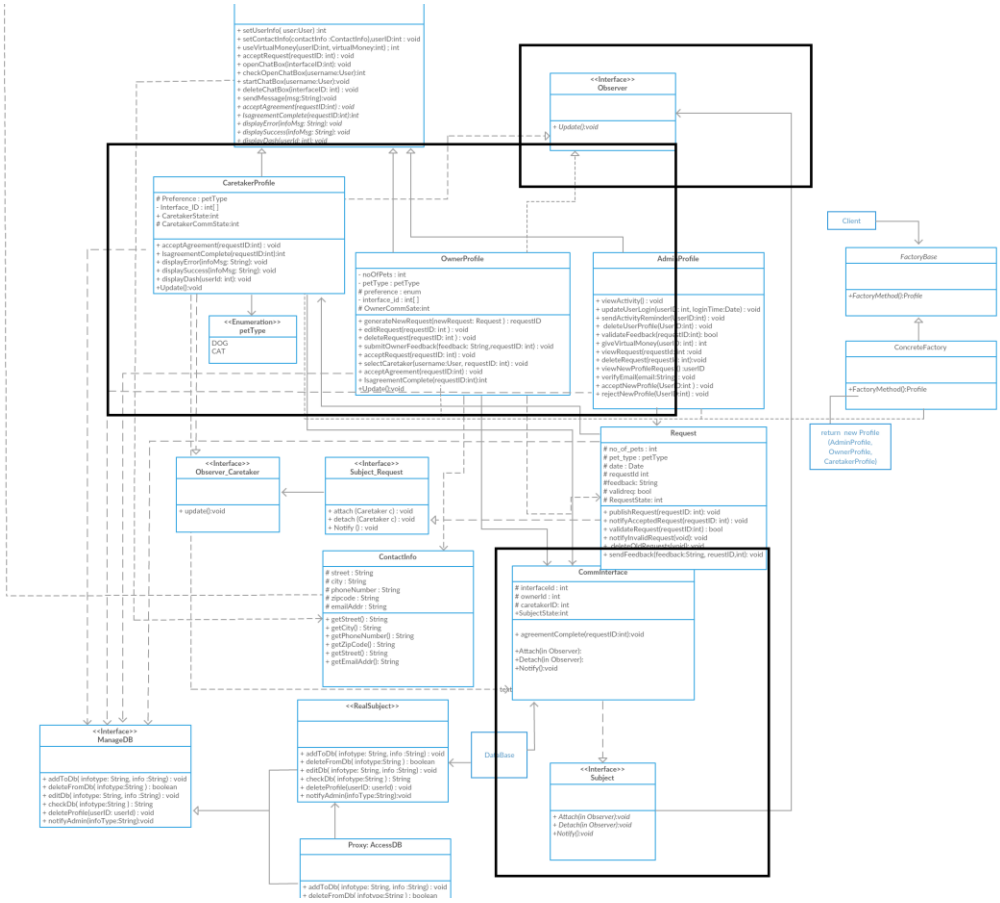
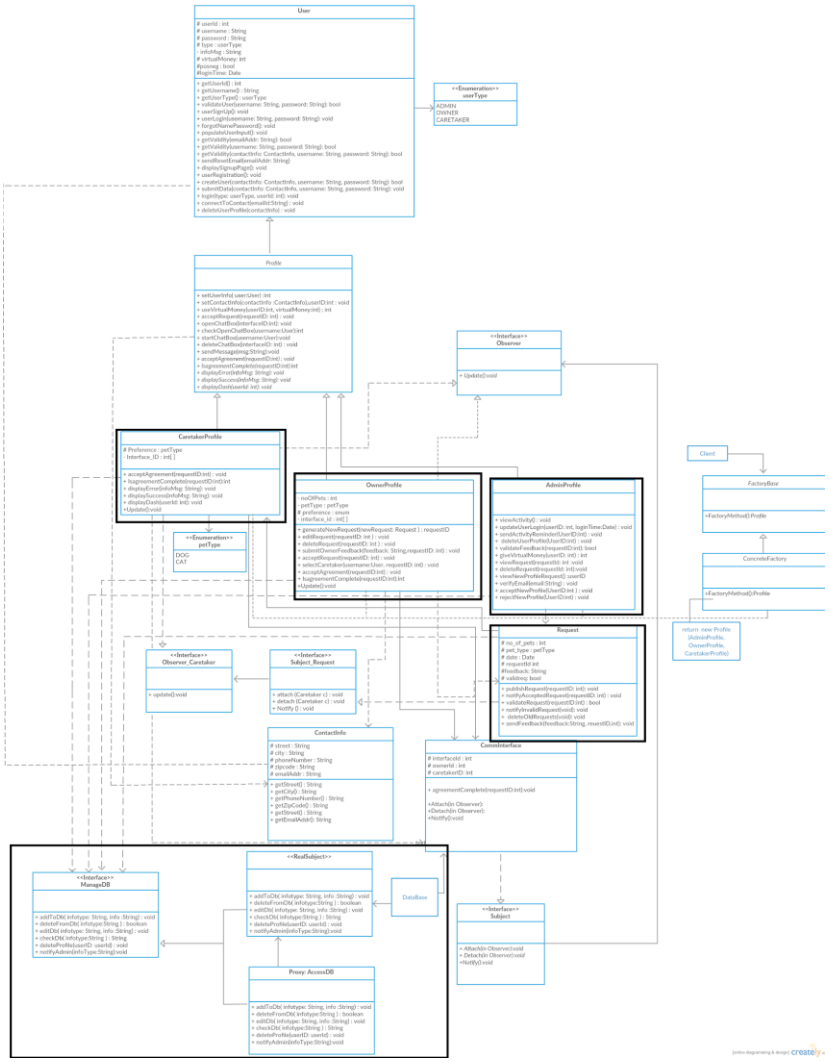


Fig: Observer Design Pattern for start communication use case

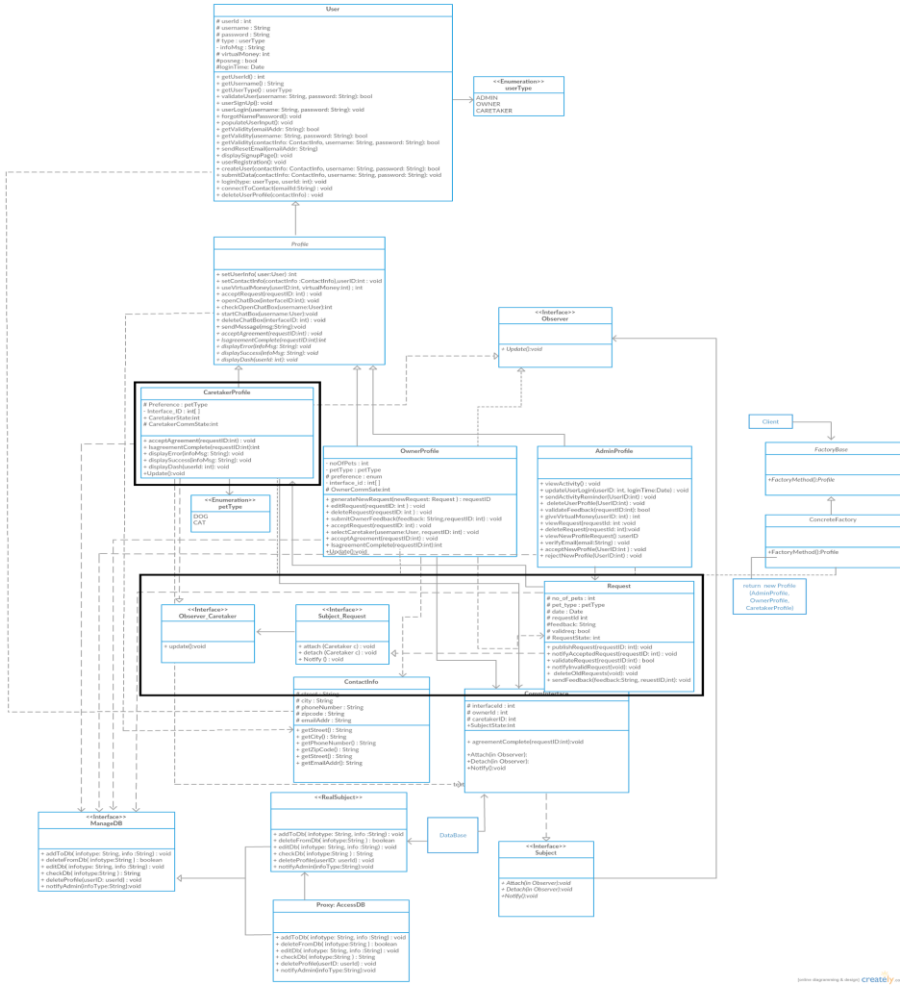
- Observer pattern is used to send notification to set of objects about any change in state. This functionality is useful in establishing agreement between owner and caretaker in start communication use case.
- Subject interface is used as subject and CommInterface is concrete class which implements Subject interface and registers CaretakerProfile objects and OwnerProfile objects as observers.
- Observer interface acts as observer which has update() method in it. CaretakerProfile and OwnerProfile are concrete classes which implement Observer interface.
- When acceptAgreement() method is called by either of Caretaker or Owner, the other actor gets notified by ConnInterface class which calls update() method in observer class.

4. Proxy Design Pattern :



- The proxy design pattern is being used since loading information from the database of Owner, Caretaker, Admin and Request information is expensive.
- The Proxy named AccessDB will act as a placeholder and decide when to access from the RealSubject that will actually go in and handle the DataBase.
- The components that work in the Proxy Design pattern for Pet_Diaries are:
  1. The Clients are the Admin, Owner, Caretaker and Request classes.
  2. The Subject is the ManageDB interface which holds all the functions that are to be carried out on the Database
  3. The proxy used is the AccessDB class which acts as a placeholder and communicates the information obtained from Database. It can help in catching the information obtained from database and sending it collectively to the ManageDB interface which communicates to the clients.
  4. The RealSubject is the one which accesses the Database and adds and extracts information from it.

5. Observer Design Pattern 2:



- Since the main purpose of the Observer pattern is to allow one object to inform others about a change of state, the observer pattern fits in the picture of a change or addition of Requests in the Pet Diaries project.
- The Owner posts request which after getting approved by the admin gets propagated to all the Caretakers.
- The request can be modified, deleted and validated and those changes will be seen by all the caretakers who watch the Request.
- The ConcreteSubject which is the Request class has the RequestState which checks the state of the Subject and upon function call of notifyAcceptedRequest() or other function call in the ConcreteSubject, those changes will be propagated to the observer through the Notify() method in the Subject.
- These changes as notified by the Subject will be communicated to the Observer which will use the Update() method in the Observer interface to communicate those changes to the ConcreteObserver which is the Caretaker class.

- The ConcreteObserver that is the CaretakerProfile class has its ObserverState stored as the CaretakerState.
- Thus all the changes in the request, that is addition of new requests, modification to them, deletion of requests will all be notified and updated to the Caretaker as and when they happen using the Observer design pattern.

IV. Refactoring:

- Refactoring will be implemented in Pet Diaries to improve design of software and make it easier to understand and extend practices.
- Following are some of the refactoring methods which will be used in application as software development proceeds.

1. Replace Temp with Query:
   Quite a few function calls, such as useVirtualMoney(), isAgreementComplete() are expected to hold values as a result of an expression. Instead all these expressions will be exported to functions, and these functions will be used instead of expressions.

2. Extract Method:
   Small functionalities such as checking number of open chat boxes and checking if agreement is completed are extracted into function checkOpenChatBox() and isAgreementComplete() respectively.

3. Replace Conditional polymorphism
   Refactoring is implemented in start user registration where classes CaretakerProfile, OwnerProfile and AdminProfile perform actions depending on the type of profile being created by inheriting methods from User class.

4. Separate Query from Modifier
   Queries are implemented using different methods rather than modifier methods. Queries are performed with methods such as *getZipCode()* , *getStreet()* and *getEmail()* etc. and modifier methods used are *setUserInfo(), setContactInfo()* etc.

V. Antipatterns:

1. Arrowheads
   There is chance of arrowheads (conditional blocks that look like arrows) turning up since a good amount of conditional checking will be implemented in functions belonging to classes such as Login, AdminProfile, OwnerProfile, Request etc. This will be avoided using adequate exceptions.

2. The Blob
   Pet Diaries is a reasonable sized application and it is highly possible that some class might end up doing a lot of work and monopolize processing. While class design has been optimized as much

as possible to prevent this, blob situations might be a reality as software development proceeds. To avoid this, continuous refactoring will be implemented during development phase. Transient associations will be removed and replaced with type specifiers to attributes and operational arguments.

3. Spaghetti
   A fairly large code base can have a significant amount of control flows, and this can lead to a spaghetti pattern. To prevent this abstract super classes such as Profile has been used. As development proceeds conditional statements will be simplified as well.

4. Old Baggage
   Pet-diaries application contains necessary classes which perform required functionality, design is open for extension to support additional functionality but does not contain any unused classes or dead code. It is done to improve maintainability of application in future.

5. Functional Fixation
   Use of driver classes is avoided so that there is no main top level routine which calls other routines. Calls to different functionalities are user action driven and are encapsulated in appropriate classes.