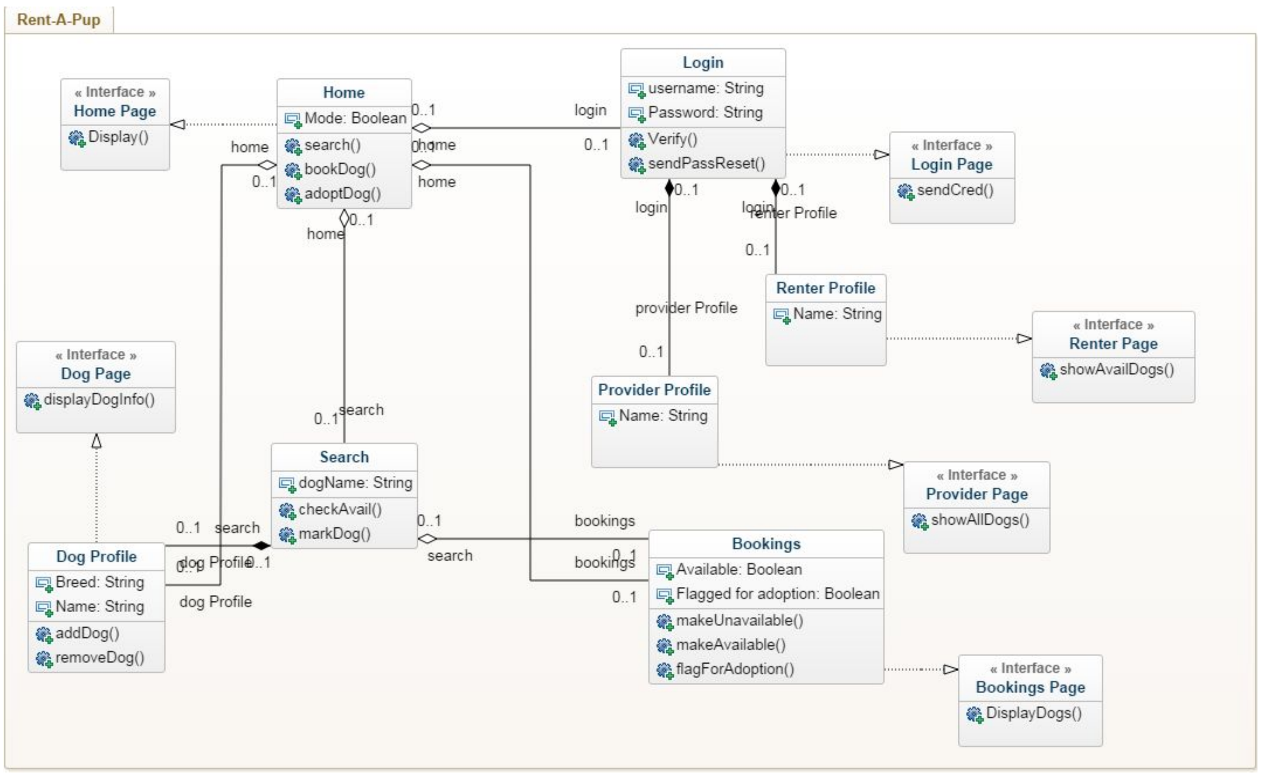
Project Part 3 : Refactoring
Team : Calvin, Elias, Elijah, Callie

Project Part 2 Class Diagram :



Project Part 3 Class Diagram :

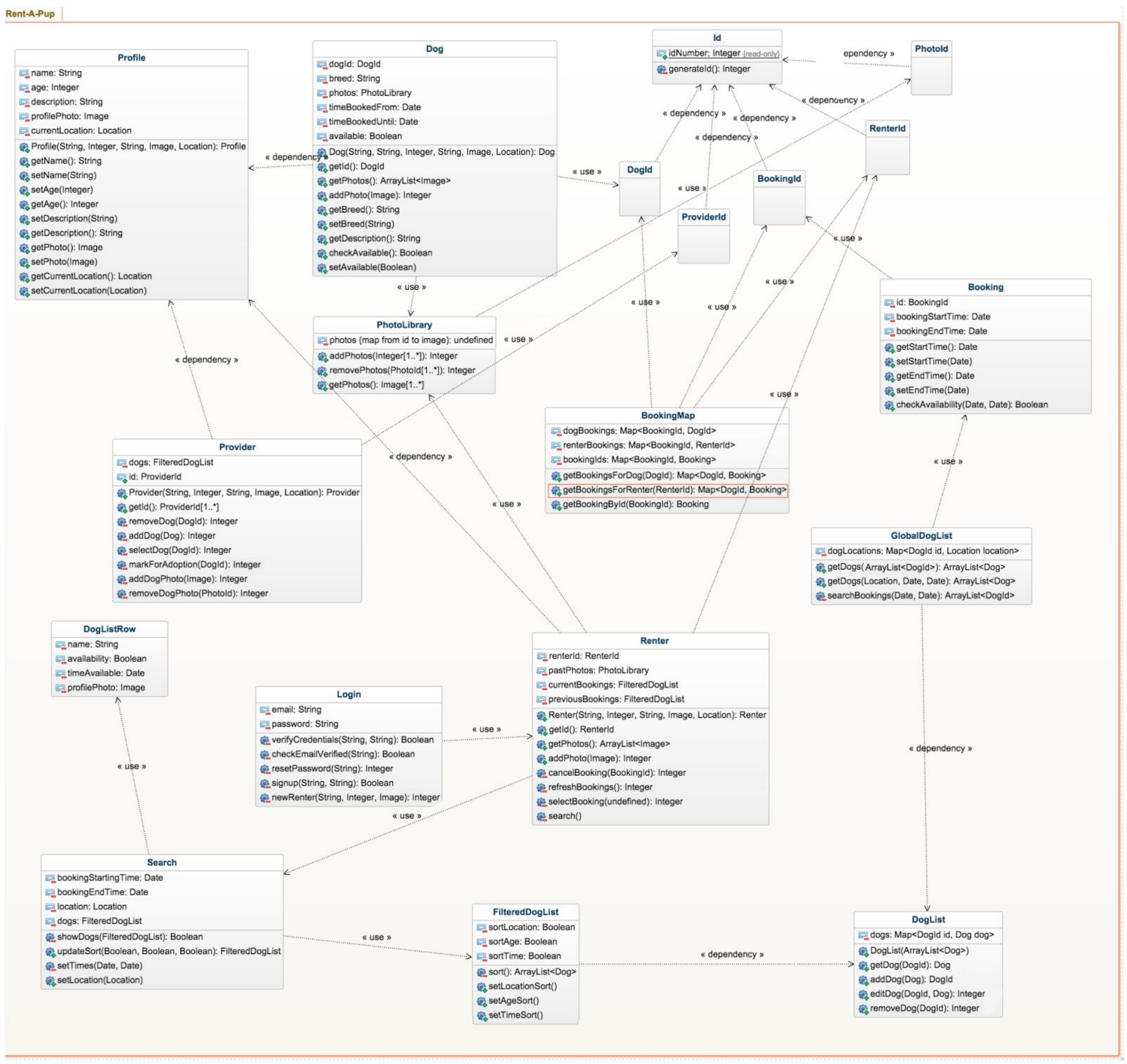

Rent-A-Pup
Profile
name: String
age: Integer
description: String
profilePhoto: Image
currentLocation: Location
Profile(String, Integer, String, Image, Location): Profile
getName(): String
setName(String)
setAge(Integer)
getAge(): Integer
setDescription(String)
getDescription(): String
getPhoto(): Image
setPhoto(Image)
getCurrentLocation(): Location
setCurrentLocation(Location)
Dog
dogId: DogId
breed: String
photos: PhotoLibrary
timeBookedFrom: Date
timeBookedUntil: Date
available: Boolean
Dog(String, String, Integer, String, Image, Location): Dog
getId(): DogId
getPhotos(): ArrayList<Image>
addPhoto(Image): Integer
getBreed(): String
setBreed(String)
getDescription(): String
checkAvailable(): Boolean
setAvailable(Boolean)
Id
idNumber: Integer (read-only)
generateId(): Integer
PhotoId
RenterId
DogId
BookingId
ProviderId
« dependency »
« use »
PhotoLibrary
photos (map from id to image): undefined
addPhotos(Integer[1..*]): Integer
removePhotos(PhotoId[1..*]): Integer
getPhotos(): Image[1..*]
Booking
id: BookingId
bookingStartTime: Date
bookingEndTime: Date
getStartTime(): Date
setStartTime(Date)
getEndTime(): Date
setEndTime(Date)
checkAvailability(Date, Date): Boolean
BookingMap
dogBookings: Map<BookingId, DogId>
renterBookings: Map<BookingId, RenterId>
bookingIds: Map<BookingId, Booking>
getBookingsForDog(DogId): Map<DogId, Booking>
getBookingsForRenter(RenterId): Map<DogId, Booking>
getBookingById(BookingId): Booking
Provider
dogs: FilteredDogList
id: ProviderId
Provider(String, Integer, String, Image, Location): Provider
getId(): ProviderId[1..*]
removeDog(DogId): Integer
addDog(Dog): Integer
selectDog(DogId): Integer
markForAdoption(DogId): Integer
addDogPhoto(Image): Integer
removeDogPhoto(PhotoId): Integer
GlobalDogList
dogLocations: Map<DogId id, Location location>
getDogs(ArrayList<DogId>): ArrayList<Dog>
getDogs(Location, Date, Date): ArrayList<Dog>
searchBookings(Date, Date): ArrayList<DogId>
DogListRow
name: String
availability: Boolean
timeAvailable: Date
profilePhoto: Image
Login
email: String
password: String
verifyCredentials(String, String): Boolean
checkEmailVerified(String): Boolean
resetPassword(String): Integer
signup(String, String): Boolean
newRenter(String, Integer, Image): Integer
Renter
renterId: RenterId
pastPhotos: PhotoLibrary
currentBookings: FilteredDogList
previousBookings: FilteredDogList
Renter(String, Integer, String, Image, Location): Renter
getId(): RenterId
getPhotos(): ArrayList<Image>
addPhoto(Image): Integer
cancelBooking(BookingId): Integer
refreshBookings(): Integer
selectBooking(undefined): Integer
search()
Search
bookingStartingTime: Date
bookingEndTime: Date
location: Location
dogs: FilteredDogList
showDogs(FilteredDogList): Boolean
updateSort(Boolean, Boolean, Boolean): FilteredDogList
setTimes(Date, Date)
setLocation(Location)
FilteredDogList
sortLocation: Boolean
sortAge: Boolean
sortTime: Boolean
sort(): ArrayList<Dog>
setLocationSort()
setAgeSort()
setTimeSort()
DogList
dogs: Map<DogId id, Dog dog>
DogList(ArrayList<Dog>)
getDog(DogId): Dog
addDog(Dog): DogId
editDog(DogId, Dog): Integer
removeDog(DogId): Integer

*diagram is missing single dotted <<use>> arrow from Provider to PhotoLibrary

Refactoring and Applied Design Patterns:

We refactored the functions in our class diagram to have the correct return values. This included the functions in Search, Login, Profile, Renter, Dog, Provider, and Booking. Search now has more functionality to specify how the database is being queried. We added setLocation(), setTimes() and updateSort() which returns a FilteredDogList because when the renter searches for dogs in the area within a specific time frame, updateSort() queries the petData collection in our database to find all documents that match the searched area and that do not have appointments scheduled at the searched time frame. In the Renter class we added more functions so the renter can now cancel a booking with cancelBooking() which returns an integer that similar to selectBooking(), will depending on the integer value, print out success, failure or an error message. An error message would be returned if for example a connection to the database cannot be made. The Login class now features checkEmailVerified() which will verify the renter is using a valid email address. Verify() in our previous class diagram was changed to verifyCredentials() which now returns a String. If the renter logs in with the correct login information, this function will return Success or Failure or an error message if for example a connection to the database was lost.

We added the classes DogListRow, FilteredDogList, GlobalDogList, DogList, PhotoLibrary, Id, BookingMap, and we changed the Profile structure. We made DogList, Id, and Profile to be superclasses to provide abstraction and minimize repetition in the code. We changed Provider Profile and Renter Profile to be combined into one Profile class that is now a superclass of the Renter, Provider and Dog classes. This creates a more organized structure because renters, dogs, and providers all have descriptions, names, photos, and locations that is then inherited from Profile. Id is a superclass to DogId(), ProviderId(), BookingId(), RenterId(), and PhotoId() which allows all different types of id's to inherit the generateId() function and the idNumber which is a read-only integer so it cannot be altered. The generateId() function also creates an ID that is unique to each dog*****. This way, other classes such as Booking, PhotoLibrary, Renter, BookingMap, Provider, and Dog can all use the uniquely generated ids to reference photos, bookings, dogs, etc. The PhotoLibrary class was added to again provide abstraction as all profiles (Dog, Renter and Provider) all use this class to change, add or remove their photo. DogList is a superclass of GlobalDogList and FilteredDogList because DogList provides the higher level functionality of editing, adding, and removing dogs while FilteredDogList and GlobalDogList provides more lower level functionality of sorting through the list of dogs and searching through bookings to return a list of dogIds. We added

BookingMap to return a list of all bookings made for renters or for dogs or to list bookings by BookingId while the class Booking is used internally when searching for bookings.

The design patterns that we applied to our new class diagram include template pattern, observer pattern, and command patter. When we added the Profile superclass, we utilized the template pattern because we localized common behavior to increase code reuse. If perhaps we wanted to expand this Rent-A-Pup service to include cats or other pets, we could simple add a new subclass of Profile that contains only cat or pet specific functionality. In creating the superclass Id, we again utilized template pattern because we abstracted the code to generate id's for renterId, dogId, providerId, bookingId and photoId. We also used it with our third superclass DogList in again abstracting code for the subclasses GlobalDogList and FilteredDogList. Another design pattern that we utilized was the observer design pattern because when a renter searches for dogs based on location and time frame, a FilteredDogList is returned which needs to remain updated such that the possibility of a double booking is minimized. Another design pattern that we used was the command design pattern because when a renter makes a booking and then needs to cancel their booking, a request is submitted through the cancelBooking() functionality that then must undo what the selectBooking() function did.

We are currently working on implementing the flyweight design pattern as well as the proxy design pattern, because there will be many different ways to "resort" a page that the user is on, and having to reload everything each time could get very costly, these should help minimize that cost. We are looking into implementing the Memento design pattern for deleting dogs from the database, because everyone makes mistakes and we want to minimize the potential impact of those mistakes when they do happen.