CARLETON UNIVERSITY

Animal Care System

CARLETON UNIVERSITY

System Design Document

Team Animal Farm

Azim Baghadiya - 101044100
Cameron Hawtin - 101047338
Nicolas Lalonde - 101031228
Nick Simone - 101034935

Submitted to:
Dr. Christine Laurendeau
COMP 3004 Object-Oriented Software Engineering
School of Computer Science
Carleton University

March-21-2019

Table of Contents

Tables and Figures

1. Introduction

1.1 Project Overview

Animal shelters provide a crucial service for homeless animals awaiting adoption into a loving home, as well as for the people who are seeking the comfort and companionship that a beloved pet can offer. Sometimes however, adoptions can go wrong, due to a personality mismatch between the animal and the client.

The Carleton University Animal Care System (cuACS) proposes to address this issue by providing a tool that automatically matches together, based on compatibility, shelter animals and the clients who wish to adopt them. The goal of the cuACS system is to generate an optimal set of matches, where a match consists of an animal available for adoption and a human client who is well suited to adopt it.

The cuACS system supports two categories of users: clients and shelter staff, and facilitates a few key features. These include the ability of staff to manage the shelter's animals and their detailed profile information, as well as to assess animal-client compatibility and compute an optimal set of animal-client matches. The system also allows clients to manage their profile information, including personal data and matching preferences.

The Animal-Client matching(ACM) algorithm uses the profiles of all clients and animals in order to compute the best possible set of matches between animals and clients. The ACM algorithm uses animal and client profiles to compute the best matches, and it outputs the match results. These results must consist of both a summary of matches, and the details of a selected match. The summary information indicates the names of the matched pairs of animals and clients. The detailed information for a given match specifies the exact rules that were used to compute that match, as well as the data supporting how and why that specific match was computed.

The cuACS user interface (UI) will be graphical in nature. The user features should be easily navigable, either as menu items and/or pop-up menus. The cuACS system will run on a single host, with all its data stored on that same host. Data will be stored in persistent storage (flat files) and will be loaded into the UI when cuACS launches. Modifications to data in the UI will be saved to persistent storage after every change.

Animal attributes include physical attributes, such as type of animal, breed, gender, age, and size. They also include non-physical attributes that will relate to temperament, personality traits, habits, etc. Specific values for these attributes will be included in every animal profile. A client profile will contain specific values for the attributes that are relevant to humans in the adoption process, as part of the client's personal data.

Each client profile must also contain the client's matching preferences, which are the values for the attributes that the client is looking for in an adopted animal.

All the attributes defined in the animal and client profiles will be used by the algorithm's rules to compute the best matches. This includes a client's own attributes, as well as their matching preferences. The algorithm must use its own unique set of rules for matching together animal and client profiles, based on the attributes and matching preferences contained in those profiles.

1.2 Document Overview

This document introduces the general system design for the Carleton University Animal Care System. It presents the different subsystems present in the system and their evolution through time, as well as the various strategies used to design these subsystems. The goal of this document is to leave the reader with a general idea of how the system works, and why it works in this specific way.

We first discuss the various subsystems and how they interact to form a cohesive system, as well as the rationale behind the division of labour between the subsystems. In this section, we discuss the design goals that drove our decisions for designing the system. We also discuss a preliminary system implementation and its drawbacks. We finally address how this preliminary system evolved into our current system design, which we explain in detail, and how our design goals drove this evolution. This section illustrates how our system design evolved through time.

We secondly discuss what strategies we used in designing our current system, explaining our strategy to deal with persistent storage as well as the various design patterns we opted to use. In this section, we provide an in-depth study of our persistent storage subsystem and the goals which resulted in the specific design decisions used in building it. We also discuss the various design patterns[GOF] used in our system, our rationale behind their use, and how they tie into our design goals. This section provides an explicit view of our reasoning in designing the current system.

In sum, this document provides the reader with an overview of the inner workings of our system and the decisions that led to its current incarnation. The reader is presented with the evolution of the system through time and what drove the changes. Then an overview of the various design strategies used is given. These sections provide a global overview of the functioning of the Carleton University Animal Care System and the reasons which justify its structure.

2. Subsystem Decomposition

2.1 Design Goals

The aim is to identify the goals and set standards for the quality of the design the system must have. The application domain and the non functional requirements are closely analyzed to come up with the criteria. These include: performance, dependability, maintenance, cost, and end-user-defined criteria.

The first criterion to examine is performance. Given that the current system is not an extensively large software application and only stores the attributes and preferences of a modest number of clients and animals, the memory and response time will not be largely affected by our choice of design. However, when working on a scalable software, it is essential to take into account the time that the software takes to acknowledge user requests, the space it occupies on the drive to run smoothly, and the number of tasks it can do in a fixed amount of time. Since we do not know the extent to which the application is likely to be scaled, it is important to consider performance in our design.

The next important criterion to consider is the system's dependability. This is an important criterion. The system should be robust. It should accept "bad" inputs without crashing (for instance, if the user enters a string in the age field which only accepts integers). The software system must provide constant availability. It should be available at all times for the clients and the staff to ensure their workflow is not disrupted. It should also provide fault tolerance. For example, leaving mandatory fields blank will disable the save button in editing animal or client profiles, so that profiles always have all the required information. And finally, the system should be secure. As the software is hosted locally and for a small non-profit organisation, attacks are not as likely to happen as in other situations, resulting in a smaller focus on security than would be warranted in other situations.

The third criterion to consider is maintainability. This is accomplished by using appropriate design patterns and other object oriented techniques to allow the ease of adding or changing functionality. Modularizing the code and providing appropriate comments will facilitate readability. Attempting to keep subsystems as loosely coupled as possible will keep the system flexible and facilitate adding or changing functionality.

The next criterion, cost, is not quite an important factor when it comes to the current state of our software, but is an important consideration when it comes to large scale applications. Utilizing the outstanding capabilities of Object Oriented Programming like Polymorphism and Inheritance will reduce duplication of code and allow objects with different internal structures to interact with the same external structure. This will help to minimize the costs of administration. Providing extensive live testing and verifying all the use cases will allow us to identify and fix bugs in the early stages. This will reduce the maintenance cost in the future software development stages.

Finally, the most important criteria are those defined by the end user. Verify that the software system provides all the functionality required by the client. Verify that all use cases trace back to the requirements in order to prevent redundant functionality. Lastly, provide a simple and well designed user interface that use elements like dropdowns and checkboxes when possible so that the software is easy to use for the end users.

Principal Design Criteria
Performance
Dependability
Maintainability
Cost
Functionality
Ease of Use

Table 2.1 Principal Design Criteria

2.2 Preliminary Subsystem Decomposition

2.2.1 Preliminary System Overview

This section deals with the decomposition of the code previously implemented for deliverable two, up to and including the add/view animals and add/view clients functionality. The code base is decomposed into three logical subsystems: Interface, Application Logic, and Storage, which follow in figure 2.1.

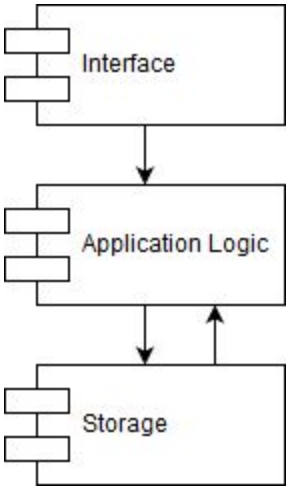

Figure 2.1 - Preliminary System Overview

2.2.2 Preliminary Subsystem Description

The subsystems are further decomposed into the classes that make them up, as shown in figure 2.2.

The Interface subsystem contains the objects that are used in the GUI. These are mainly boundary objects that the user interacts with. For example, if the user is a client, they can log in through the MainWindow and access the PostLoginClient view. From there they are able to access the ViewAnimals interface.

The Application Logic subsystem contains the main application logic, the classes for Animal and Human profiles, as well storing/sending any new profiles to the storage. Any requests from the view to the storage are handled through this subsystem. The CuacsAPI class is a Facade class for this subsystem. For example, if a client requests to view all animals in the system, the CuacsAPI class will provide a list of animals to the ViewAnimals interface class.

The Storage subsystem contains the code responsible for storing/accessing files containing profile information. The PersistentStorageAPI class is a Facade class for this subsystem. To continue with our example from above, if the ViewAnimals class requests the list of animals from the CuacsAPI class, the CuacsAPI class requests the list of animals from the PersistentStorageAPI class, and the list of animals in storage is returned to the CuacsAPI class, and then is displayed on the GUI using the ViewAnimals class.

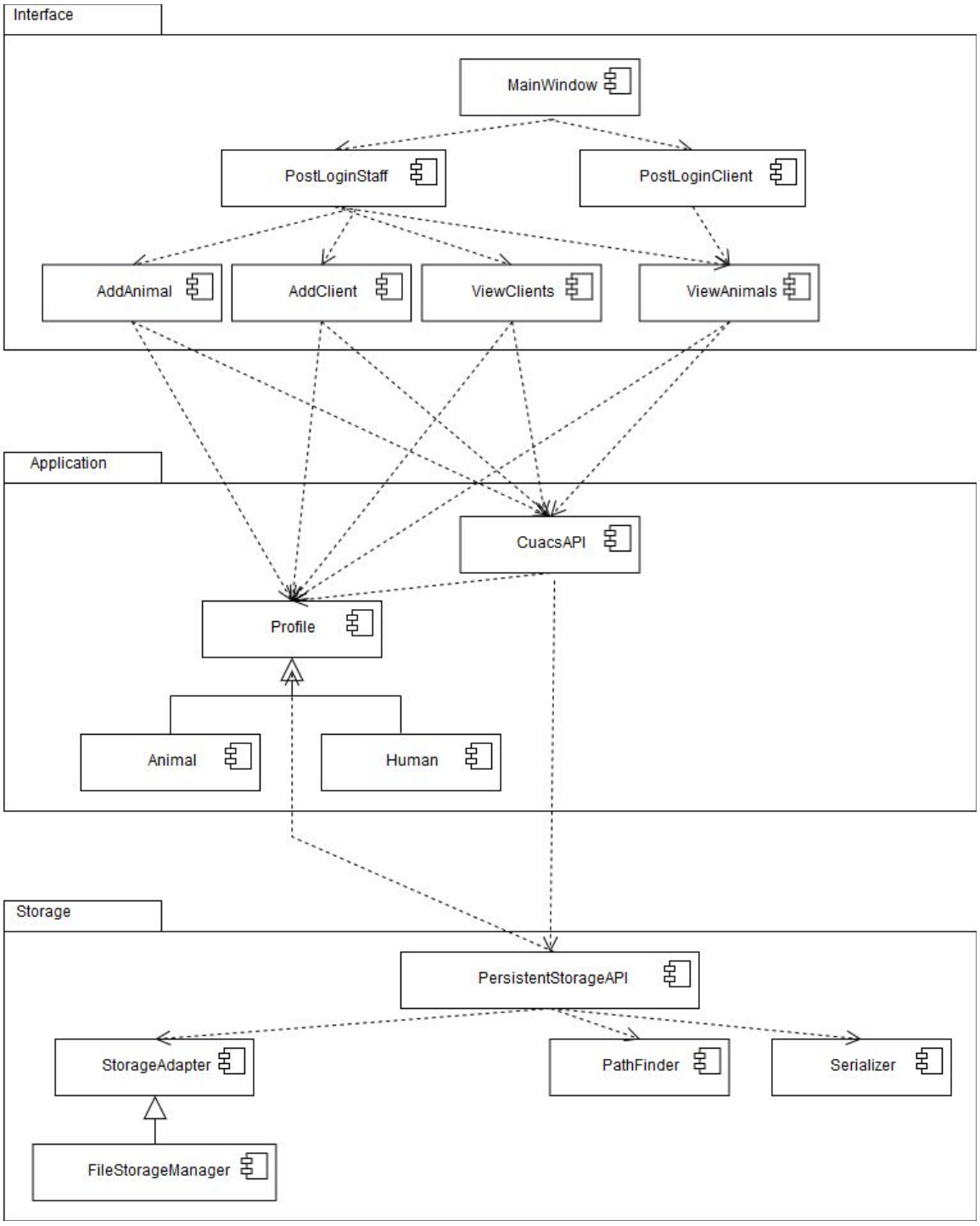

Figure 2.2 - Preliminary Subsystem Decomposition
(UML Component Diagram layers shown as UML packages)

2.2.3 Shortcomings

Certain problems arise from the above image (figure 2.2). The subsystems are not as loosely coupled as they could be, specifically, every subsystem must access the Profile class and the CuacsAPI class. Additionally, the bottom layer(Storage subsystem) is accessing the middle layer(Application Logic subsystem) here, this is not ideal for our design because lower layers should not have knowledge of higher-level layers. This design also relies heavily on the facade class CuacsAPI to store information and implement the application logic. The constructors used for the Animal and Human objects are quite large (using large amounts of CSV). This should be simplified to avoid error.

2.3 Full System Decomposition

2.3.1 Full System Overview

There are several shortcomings of the original design of the cuACS system. These have been fixed in the final system design. The design has been refactored in order to promote loose coupling and high cohesion. Although the three basic subsystems remain the same, the dependencies from lower layers to higher layers have been removed (see figures 2.1 and 2.3).

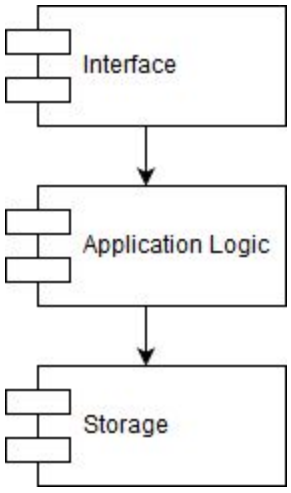

Figure 2.3 - System Overview

2.3.2 Full Subsystem Description

As shown in figure 2.3 (and further broken down in figure 2.4), the three basic subsystems are again Interface, Application Logic, and Storage. These subsystems are now loosely coupled and each subsystem has only a single access point used by any subsystem above.

The Interface subsystem contains the objects that are used in the graphical user interface. These are the boundary objects that the user interacts with. For example, a client can access the system through the MainWindow and access the ClientView interface. From there they are able to access the ViewAnimals (view all animals in the system), ViewProfile (view their own profile), and EditProfile (edit their own profile) interfaces. The classes on the bottom layer of this subsystem are registered with the CuacsObserver class which will notify them of any change in the data model in the facade CuacsAPI class in the subsystem below, since they rely on information stored there.

The Application Logic subsystem contains the entity objects (Animal and Human classes, as well as their attributes), and the classes they interact with. The AnimalCollection and ClientCollection classes use CuacsAPI to relay information about the entity objects to and from the PersistentStorageAPI in the storage subsystem. Alongside these classes, the Application Logic subsystem handles the ACMAlgorithm and its concrete implementations SmallSampleAlgorithm (A dynamic combinatorial algorithm) and LargeSampleAlgorithm (A hill-climbing-type algorithm), which utilize Profile data to formulate ideal matchings of the clients and animals obtained from the AnimalCollection and ClientCollection classes.

The Storage subsystem handles the storage and retrieval of data from the persistent storage, flat files in our case. The facade class PersistentStorageAPI receives the data from the AnimalCollection and ClientCollection and utilizes the other classes in this subsystem to store and retrieve the data persistently. The StorageAdapterFactory class allows for very easy switching between flat files and an SQL database. The StorageAdapter class is an adapter that allows storage and retrieval of client and animal profiles from the persistent storage. This generic adapter can be implemented further by a concrete class to facilitate a specific kind of storage, for instance flat files or database tables. In the case of our cuACS software system, the class FileStorageManager implements from the StorageAdapter and provides the functionality to store and retrieve flat files for the client and animal information. The PathFinder class is responsible to find the right directory to store the flat file based on the profile type, animal or client.

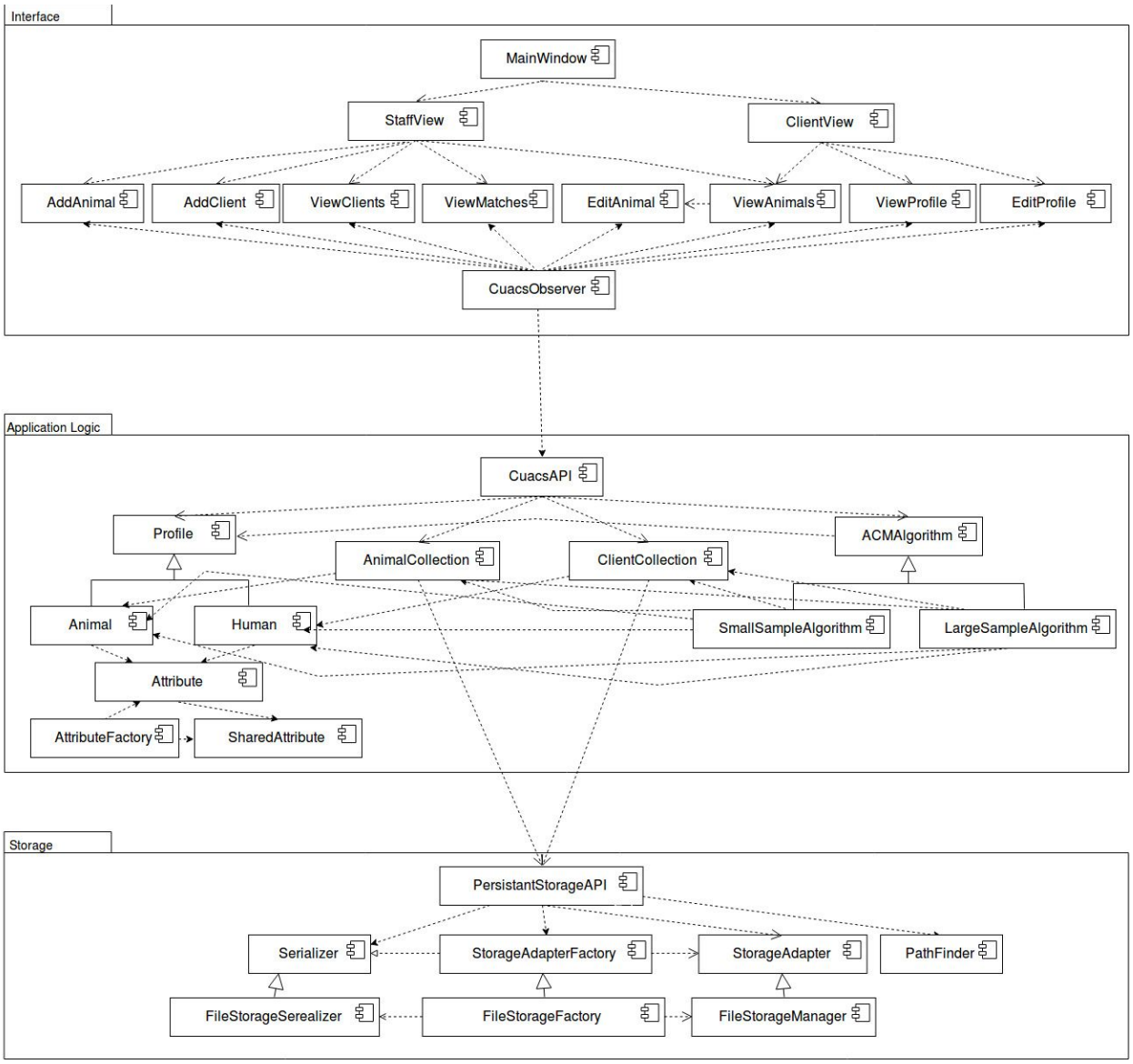

Figure 2.4 - Full Subsystem Decomposition
(UML Component Diagram layers shown as UML packages)

2.4 Design Evolution

After carefully evaluating the code base for the second deliverable and considering the shortcomings of its subsystems, we are now aiming to evolve our design strategies to improve those shortcomings. Let's start by looking at the shortcomings briefly, how they violate our design goals, and whether the evolved design addresses these issues.

Firstly, the preliminary subsystems were not very loosely coupled. Specifically, every subsystem is accessing the Profile class and the CuacsAPI class. This means that modifying one subsystem will have an impact and demand modifications on the other. This means that the cost of administering the codebase increases. To resolve this problem, we minimized the relationship a class has with a non facade class from another subsystem and this eventually minimized the number of relationships between classes from different subsystems. Now the Facade class is the only one responsible for interacting with the Interface subsystem. For instance in the new design, the classes from the Interface subsystem have associations only with the CuacsAPI from the Application Logic subsystem and have no idea about (or need an association with) the Profile class. We can also see that there is an extremely high level of cohesion in the new subsystem decomposition

Additionally, the layers in the preliminary subsystem decomposition are poorly designed. As you will notice, the bottom layer(Storage subsystem) is accessing the middle layer(Application Logic subsystem) which is not ideal. This may lead to duplication of the code and incorrect use the of the Object Oriented Programming techniques. Duplication of code leads to poor readability and higher maintenance cost. Therefore, the new design ensures that the CuacsAPI from the Interface subsystem gives Profile access to the PersistenStorageAPI in Storage subsystem. Now, there is no association between the PersistenStorageAPI and the profile class which not only makes a better layer design but also makes the coupling loose.

Furthermore, our preliminary subsystem decomposition shows that the facade class CuacsAPI has to store information and implement a lot of the application logic. CuacsAPI must maintain a count of the current available client id and animal id. This approach of using a Facade class may not be fully correct and it violates the principle that classes should serve a single purpose. To clarify, the logic must not be implemented in the facade as it must aim to provide a layer for the UI to interact with the backend. This is solved by abstracting this functionality into individual collection classes to store the clients and the animals and retrieve them from the persistent storage.

Finally, we have noticed that client and animal attributes are more volatile than previously anticipated. That is, the specific attributes we store for each tend to be prone to change. To facilitate this change, we decided to convert the specific attributes of each class to instead be held in a collection of attribute objects, each containing the name of the attribute, its data-type as well as its value. These objects can be used to limit the amount of code which needs to be updated in order to add or remove

attributes. For example, we can automatically generate an interface from a list of attributes instead of hardcoding each attribute into a view. Since these objects would introduce a lot of repetition in their name and type attributes, we implement this part of the attribute object as a flyweight. This will save much space in the system as many attribute objects will be created (around 20 for any profile).

These were the essential changes we made in the process of evolving the design of our software system so that it better aligns with the design goals.

3. Design Strategies

3.1 Persistent Storage

An Important aspect of our system is how we have opted to store persistent information. This information consists of two basic categories of objects: client profiles, and animal profiles. The storage is accomplished through the storage subsystem described in section 2.3.

As the application is designed to be run locally, we have opted to use a flat-file solution rather than using a database, as our purposes do not warrant the development overhead that comes with implementing a database system. As maintaining a synchronized store of data accessible to multiple applications and preventing race conditions between them is unnecessary in this context, flat-files can suffice. Do note that we have implemented this system in such a way that, if the need may arise, transfering to a database model should pose minimal inconvenience. More on this in section 3.2.

The two types of persistent objects which we store, client profiles and animal profiles, are very similar in nature. Each consists of a specific set of attributes with corresponding values. Clients have their animal preferences, basic information, and a few personality traits. Animals have both physical and non-physical traits. For more information about the specific attributes of each class, please refer to our previous document, *Algorithm Analysis Document* [Animal Farm].

As both these types of objects are very similar, the storing procedure is the same for both of them. This procedure is executed upon every edit or addition of a profile. When such an action is undertaken, the PersistentStorageAPI class is called in action through its storeProfile method. Due to our design goal of making the system as modular as possible, the procedure is somewhat complex, using quite a few objects to accomplish its task. This procedure is illustrated in figure 3.1.

When PersistentStorageAPI is told to store a profile, it begins by creating a PathFinder object. This object's getDirectory method will then be called with the profile to save passed as a parameter. Depending on the profile type (Animal or Client), the PathFinder will return the path to the appropriate directory. The PersistentStorageApi will then create a FileStorageSerializer and a FileStorageManager object through the use of a StorageAdapterFactory object. The profile is then asked for a vector representing the

current state of its attributes, which is passed to the FileStorageSerializer in order to serialize the profile into a storable format (in this case, a string representation) which is done by noting down the value of each attribute in order, starting with the profile ID, as a string, with a newline between each value. This serialized form of the object is then passed to the FileStorageManager, along with the path to the appropriate directory, through its save method in order to be saved as a flat file. This is done by opening a file in the passed directory, with the name of the file being the id of the profile to store, and outputting the string we obtained to the file directly.

This maximally avoids saving any duplicate data. This is because only the value of every attribute is stored in every profile. The attribute to which the value corresponds to is inferred from the order of the values in the stored file. Furthermore, profiles are only stored in a single location, their appropriate directory. Hence, there is no duplication of any data using this method of storing persistent information.

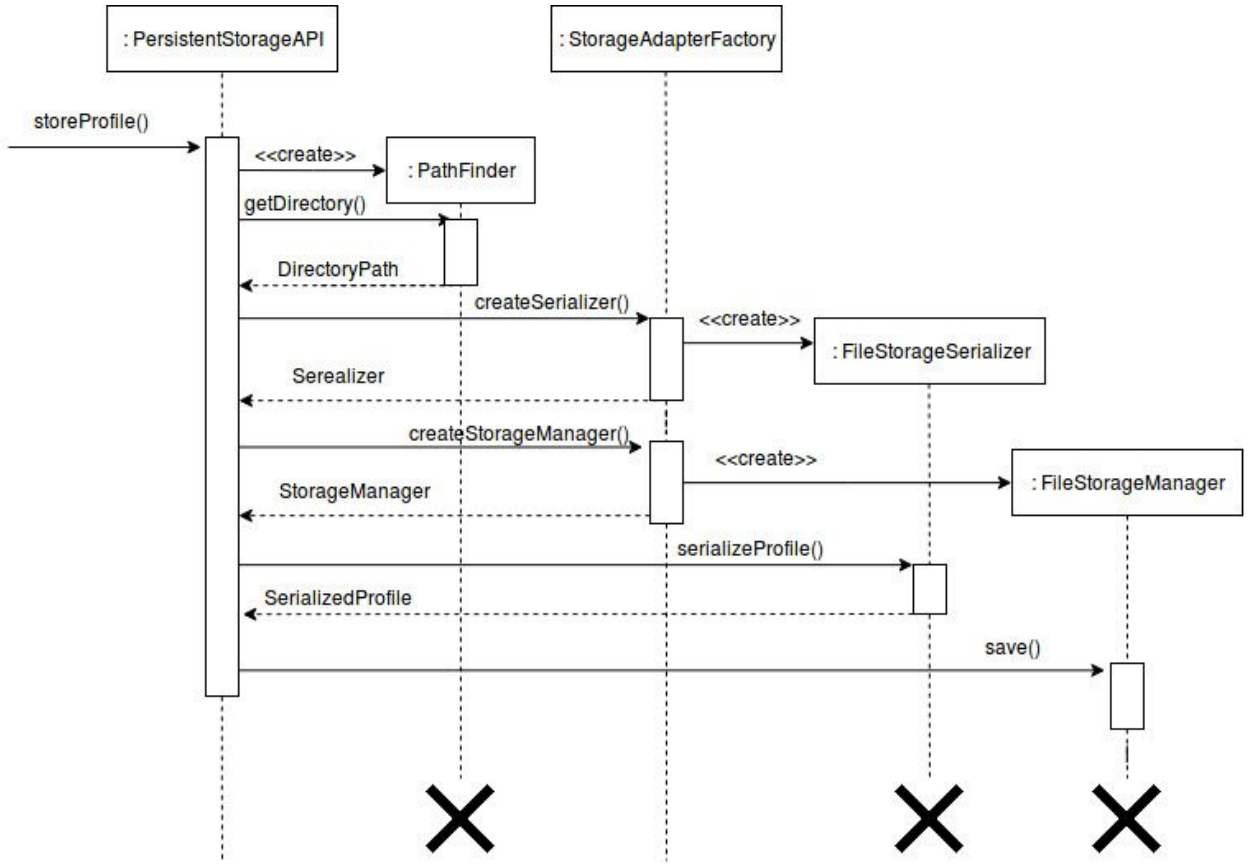

Figure 3.1 - Sequence Diagram for Storing a Profile

3.2 Design Patterns

In order to maximize the modularity of our code, and its flexibility, we have used several design patterns in the design of our system. Below we will discuss the potential problems we've discovered, what design pattern we've used to address these problems and how these design patterns address the problems.

A problem could arise if multiple animal/client collections were declared in our system. As these classes obtain their information from persistent storage, as well as update it, race conditions could occur if multiple saves and loads happened from different lists. To prevent this problem, we wanted to ensure that only one of each collection could be declared. The solution to this problem was to implement these classes as **Singleton**s[GOF]. This ensures that only one instance of the class will ever be created, and that this instance will be returned if there is ever an attempt to create a second instance.

A potential source of implementation overhead is the constant updating of view classes whenever they are displayed. Forgetting to update the view according to the current state of the animal/client collections can result in hard-to-track bugs. To nip this problem in the bud, we've decided to use an **Observer**[GOF] design pattern. Any active view can be registered with the CuacsObserver class, which will update these views upon any change in the animal/client collections. This greatly limits the number of places we must check if a certain view is not updating properly, to either the observer(really a notifier) or the view itself (if it fails to register itself with the observer).

In our preliminary system design, we noticed that our subsystems had a high level of coupling between them. This needed to be reduced as high coupling can lead to a monolithic system which becomes very hard to change. To address this issue, multiple **Facade**[GOF] classes were used to handle communication between subsystems. This allows the subsystems to interact with each other through the use of an interface, rather than directly communicating with the implementing classes. This results in much lower coupling between the subsystems, which greatly facilitates changing their implementation. The CuacsAPI and the PersistentStorageAPI classes are such Facades.

We also expected problems when using large samples of clients/animals with our chosen ACM algorithm, as it is highly sensitive to input size. Past around 175 clients and animals, it begins to perform radically slower. To prevent this from affecting the staff members, we decided it would be best to use a different algorithm when the

sample size reaches this number (this would be a hill-climbing algorithm, please see our previous paper, *Algorithm Design Document*[Animal Farm] for more details). To facilitate switching between the two algorithms, we use a **Strategy**[GOF] pattern. This allows us to encapsulate each algorithm and use a simple policy to determine which one to use at run-time. Thus the staff members will not be stuck waiting an inordinate amount of time for their matches to appear should they ever be in a situation where shelter capacity grows tremendously and unexpectedly.

Our decision to store the individual attributes of profiles as objects introduced a certain overhead. This was the fact that many of the objects would have duplicated data in their name attribute and data-type attribute. To help alleviate this memory burden, we have opted to implement these parts of those objects as **Flyweight**s[GOF]. This allows us to store the name and data-type for an attribute only once per global profile attribute, instead of once for every attribute in every profile. As there are about 20 attributes in total per profile type, and 50 to 100 total profiles during normal operations, this results in a substantial amount of memory saved.

While we have explained the reasons behind our choice to use flat-files as a storing mechanism, there could arise a situation which necessitates using a database instead. It was necessary to put some thought into reducing the implementation costs resulting from such a situation. In order to accomplish this, we decided to use an **Abstract Factory**[GOF] design pattern. This factory is used to generate the serializer and manager objects needed to store files. As it stands, this factory only produces one family of products, the FileStorage (flat file) family. However, this can be used as a dock to “plug-in” another type of file storage. For example, a serializer generating SQL insert statements along with a manager interacting with the SQL database could be added to permit the use of a SQL database. In this case, the developers need not change a multitude of classes in the code, but only create a new factory to generate the SQL family of storage classes and implement these classes. With this, we minimize the hassle of changing our file storage system.

While these are the patterns we have decided to include in our system, we have also considered other patterns to the extent with which they could help us implement additional features. Below are some honorable mentions.

While not currently part of our system, we considered it an interesting idea to add undo/redo operations on staff operations, such as adding or editing an animal. Had we done this, we would have implemented each of these operations as **Command**s[GOF]. These commands, when executed, would be stored in a dequeue and implement both

an execute and undo functionality. The undo feature could work by executing the undo operation of the last command that was executed, and moving a pointer to point to the previous command. Redo would simply execute the command again.

We also considered implementing our animal/client collections as more convoluted data structures (such as binary search trees) to allow for more complex search and find operations than simply listing all animals. In this case, we would have had to implement **Iterator**s[GOF] for these collection to enable easy traversal, as looping over them would cease to be trivial in this case. This would also allow us to decouple the iteration of the collection form its implementation, which would be a good thing regardless of whether we switched the data structure, as it would allow us to modify the data structure without worrying about affecting other parts of the code.