# cuACS Project

## Algorithm Design Document

Team 070 – Segmentation Fault

Weihang Chen

Liyongshi Chen

Richard Xia

Xiran Zhou

Submitted to:

Dr. Christine Laurendeau

COMP3004 Object-Oriented Software Engineering

School of Computer Science

Carleton University

# Table of Content:

# 1. Introduction

## 1.1. Overview of the Project

Animal adoption software plays an important part in the process of adopting sheltered animals. A well-designed software system for animal adoption should not only be easy to use, but also be beneficial to all parties that are involved in this system, i.e., the adopters (Clients), the sheltered animals (Animals), and possibly the shelter staffs (Staffs). The Carleton University Animal Care System (cuACS) is designed and currently being implemented to provide such a system. The goal of the cuACS is to generate an optimal set of animal-client matches, for which the clients' preferences are satisfied with more sheltered animals being adopted.

The system provides users with several main features. Clients can view the information of animals available for adoption, as well as update their matching preferences or other personal information by editing their own profile. Shelter staff can view, add, and edit animal information. They can also view existing clients and add new clients. Moreover, they can launch the execution of an animal-client matching algorithm and view the resulting matches.

The creation of the cuACS aims to improve the existing animal adoption systems. One of the drawbacks that the existing systems have is that it sometimes allows animals to be adopted by humans with whom they are not fully compatible. This incompatibility results in many disadvantages, including a mismatch of needs, expectations, etc. Another flaw that animal adoption systems currently have is that, some systems sacrifice the interests of sheltered animals, in order to fulfill the clients' preferences and needs. Such unresolved issues are considered when designing the cuACS. With a successful implementation, the cuACS would be capable of managing animal and human information, evaluating animal-client compatibility, as well as optimizing animal-client matches.

## 1.2. Overview of Document

This document describes the system design of the Carleton University Animal Care System (cuACS) created by Team 70 – Segmentation Fault, a group of four Computer Science students at Carleton University.

The structure of this document includes an introduction section that describes the overview of the cuACS project and the overview of the document; a subsystem decomposition section detailing the D2 implementation decomposition, the full system decomposition, and the design evolution; and a design strategies section that discusses persistent storage and the design patterns of the cuACS system.

# 2. Subsystem Decomposition

## 2.1. D2 Implementation Decomposition

According to D2 feature implementation, the entire Staff feature and part of the Client feature are divided into several subsystems based on Repository architecture structure. The subsystems are classified into two types:

1. Subsystem:

    OptionResponse, AnimalManage, and ClientManage.

2. Repository:

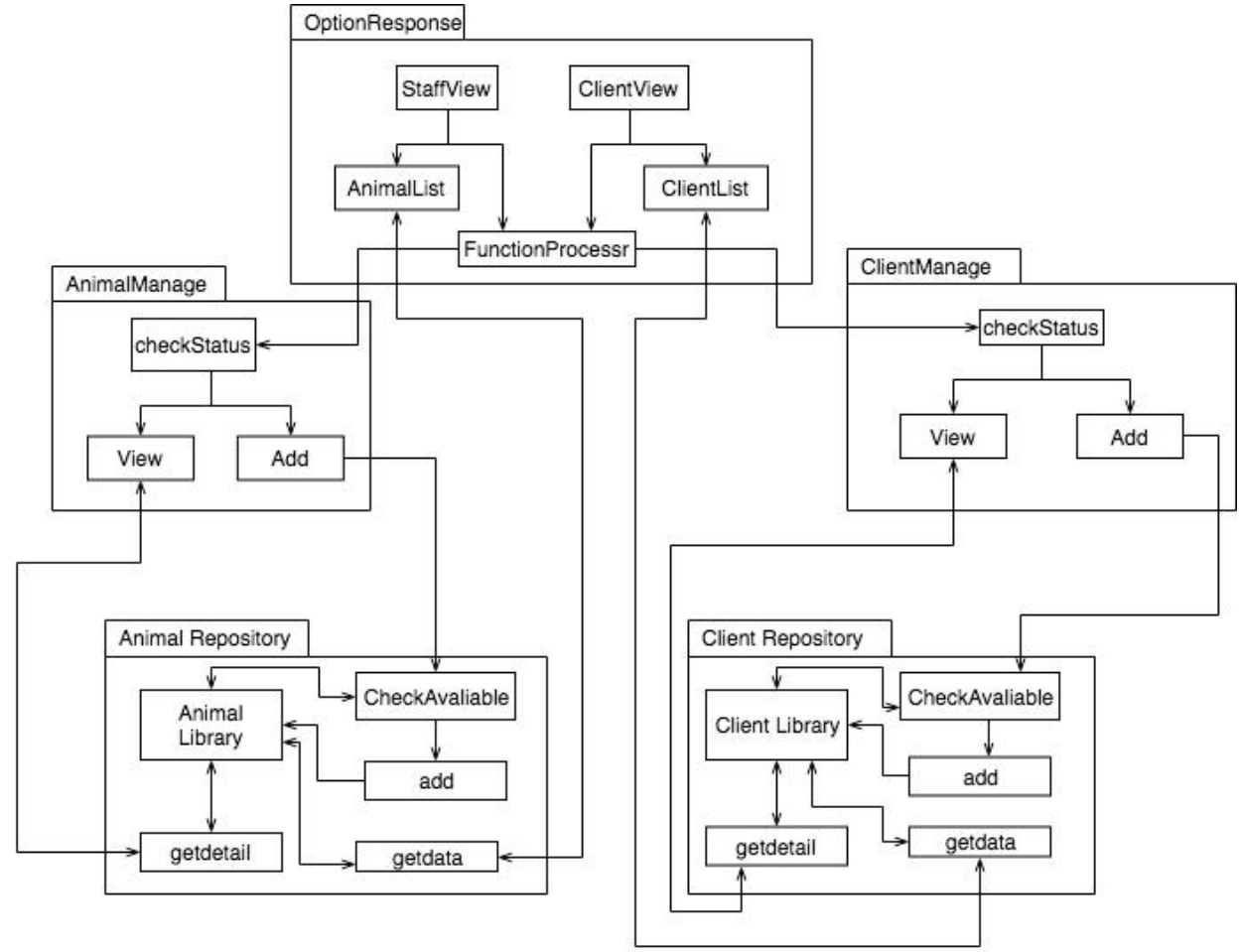
    ClientRepository, and AnimalRepository.

The table below describes the logical subsystem implemented in D2.

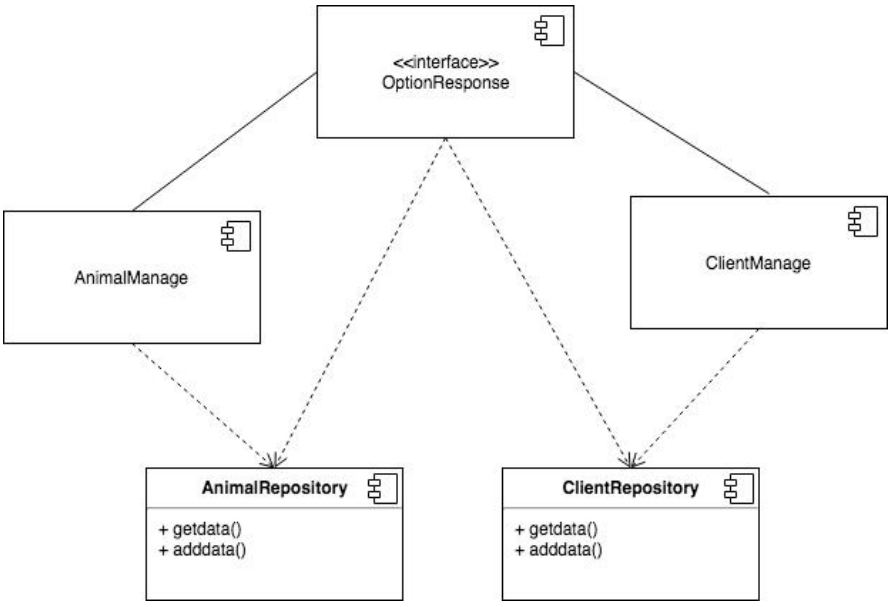
**Table 1 – Logical Subsystem Descriptions**

| Logical Subsystem Name | Description |
|---|---|
| AnimalManage | The subsystem to process AnimalInfo details as write or view status. |
| ClientManage | The subsystem to process ClientInfo details as write or view status. |
| ClientRepository | The repository subsystem to store and handle the actual data of ClientInfo as a flat file at a permanent memory location. |
| AnimalRepository | The repository subsystem to store and handle the actual data of AnimalInfo as a flat file at a permanent memory location. |
| OptionResponse | The subsystem to repose the interactions of the users. |

## Figure 1 – UML Class Diagram with Subsystem as Packages




## Figure 2 – UML Component Diagram



## 2.2. Full System Decomposition

Full system decomposition is an essential part of system design. It is developed by decomposing the system into smaller parts based on system functionalities and requirements.
The table below describes the subsystems required for the entire cuACS system.

**Table 2 – Subsystem Descriptions**

| Subsystem Name | Descriptions |
|---|---|
| StaffViewSubsystem | The StaffViewSubsystem is responsible for presenting information to the Staff and providing an interface to interact with the Staff. |
| ClientViewSubsystem | The ClientViewSubsystem is responsible for presenting information to the Client and providing an interface to interact with the Client. |
| StaffOperationManagementSubsystem | The StaffOperationManagement Subsystem is responsible for managing a set of operations that can be initiated by the Staff. |
| ClientOperationManagementSubsystem | The ClientOperationManagement Subsystem is responsible for managing a set of operations that can be initiated by the Client. |
| AnimalManagementSubsystem | The AnimalManagement Subsystem is responsible for managing and processing Animal information. |
| ClientManagementSubsystem | The ClientManagement Subsystem is responsible for managing and processing Client information. |
| MatchSubsystem | The MatchingSubsystem is responsible for computing and generating a set of optimized Animal-Client pairs. |
| AnimalStorageSubsystem | The AnimalStorageSubsystem is responsible for storing and managing a list of existing Animals that are available for adoption, as well as their detailed Animal profiles. |
| ClientStorageSubsystem | The ClientStorageSubsystem is responsible for storing and managing a list of existing Clients, as well as their detailed Client profiles. |

**Figure 3 – UML Component Diagram**

The figure below depicts the logical subsystems of the cuACS as well as their dependencies using UML component diagrams.

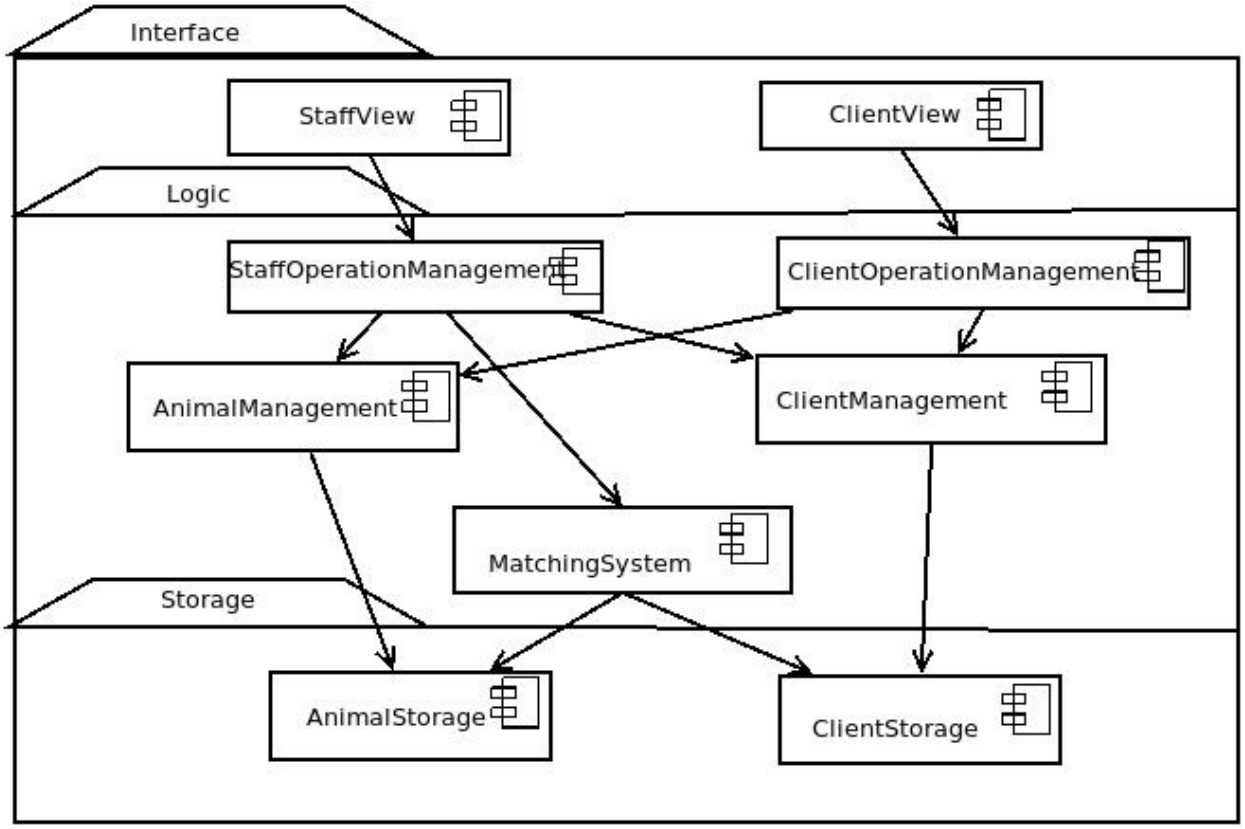


**Figure 4 – UML Class Diagrams and Packages**

The figure below illustrates the UML class diagrams with their packages included in every logical subsystem.

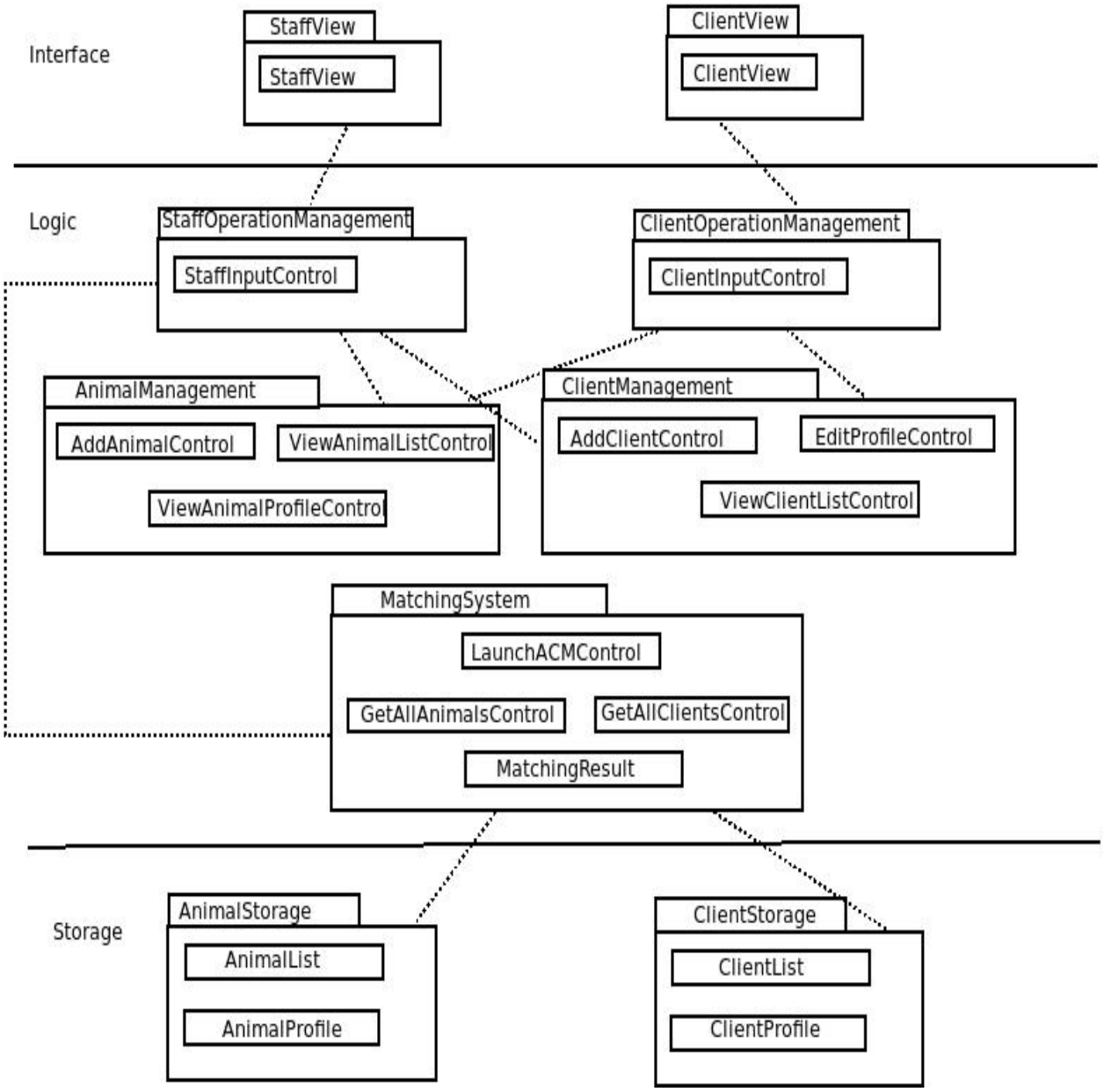
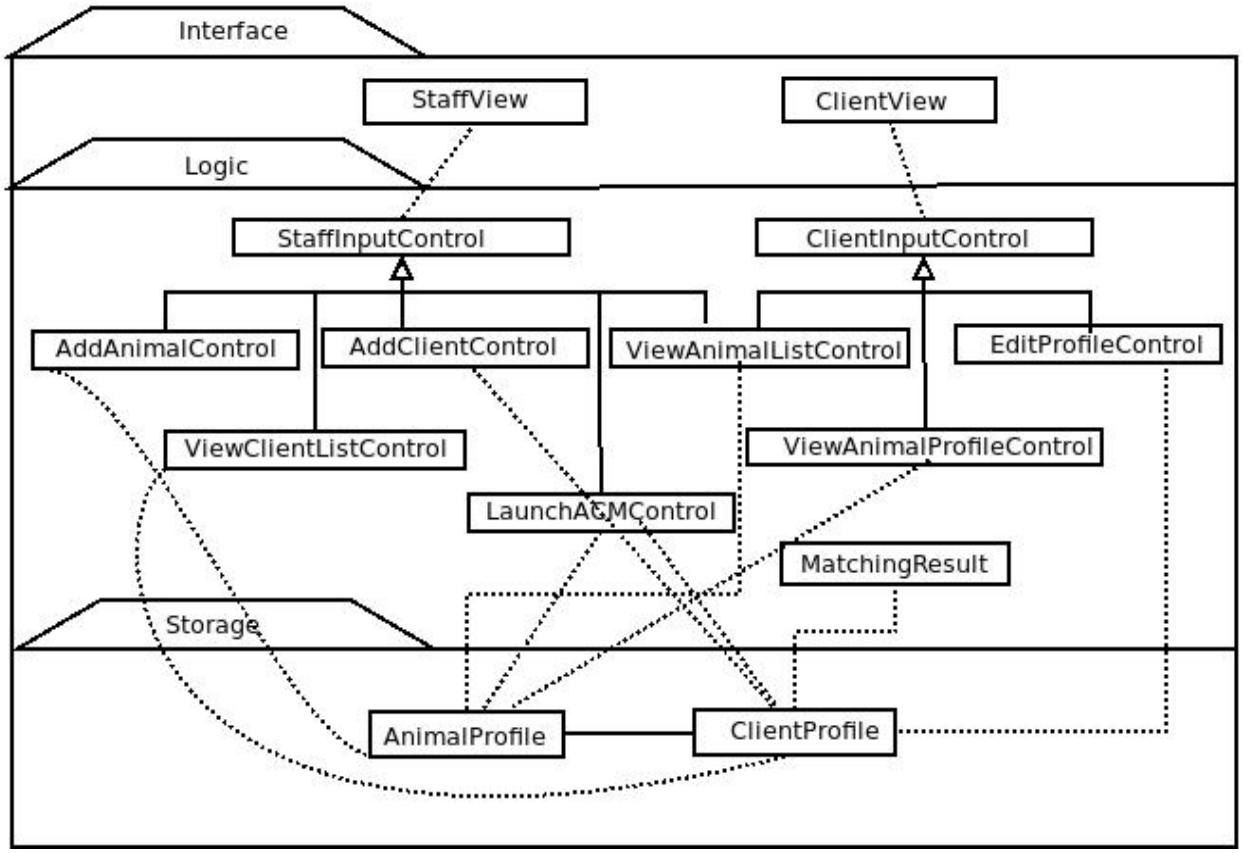


**Figure 5 – Full System UML Class Diagrams**



## 2.3. Design Evolution

The Difference between the decomposition for the D2 feature implementation and the entire system:

According to the previous description, the entire system is designed by 3-Tier Architecture which organizes subsystems as three different layers. In the D2 implementation decomposition, the subsystems have been distributed as Repository Architecture. The design between the full system and the D2 feature implementation is obviously different.

The way to process data:

In the entire system, data has been stored in three different profile subsystems in the storage layer. The processing data is responsible for subsystems in the logic layer. After the subsystem in the logic layer completes the operation, data is then transfered to the database. In contrast to the entire system, the D2 feature implementation uses repositories as a central data processor. Database and the computational processes are independent and are triggered by a single request. Once receiving the request from other subsystems, the repository starts to process data passively.

The dependency between subsystems and database:

The design of D2 feature implementation makes subsystems visit repositories independently. Because the repository is passive, the relationship between subsystems and database is close. Therefore, the subsystem is easily affected once the repository has been changed. The design of the entire system separates the interface layer and the logical layer. Each layer is independent, therefore separate physical location of these layers are allowed.

The full design of system decomposition is more complete than the D2 feature implementation. The architecture of D2 feature implementation is limited, with the whole structure influences by the repository. It is too high of a dependency between the data structure of the data store and the corresponding subsystem. It is hard to control the spread range if data have any problems. Choosing to use the 3-Tier architecture can make the relationship between the subsystem and data storage flexible. Because each tier is an individual, it is easy to manage the staff-client side and the animal-client side. The development also is friendly and efficient for us, each team member can work with

different tier at the same time. It can save a lot of time on consistency with a different part, especially for information management subsystems and storage subsystems.

Before design the full system decomposition, we use the repository structure to organize the D2 feature implementation. We found a lot of problem and limitation during the process. Firstly, it is possible to replicate or duplicate data when the system is doing multiple operations. For example, if the staff open more than one window to add an animal, it is possible to add the same animal more than one time. Secondly, it is too hard for the evolution of data, because the subsystem is closely related to the change of data. If we want to make some change with the repository, we probably also need to change all of the subsystems which connect with the repository. Then we found it is more efficient working with 3-Tier architecture. It solves the high dependency problem effectively. Furthermore, it easy to find the problem resource when we develop the system. It is clear to distinguish different subsystem.

# 3. Design Strategies

## 3.1. Persistent Storage

Persistent data refers to information that is saved after all processes are finished executing. Persistent storage is required to make data usable in other programs, such as opening a .txt file on notepad that was saved using an output stream. The most common physical types of persistent storage include hard disk drives and solid-state drives. Volatile storage consists of most Random Access Memory and CPU memory.

Animal and client data will be the two main objects of persistent storage in our program.Animal data consists of physical attributes, non-physical attributes, animal name and animal id. Physical attributes and non-physical will be essential to computing our ACM algorithm.

In our program clients should be able to load and save their own data while admin users are able to load and save animal data. The control is responsible for managing these functions which are defined in the the animalmanagement and clientmanagement classes. Each class has a loadData and a saveData function which uses a QFile to store data.

We chose to use the Qt file input output datastream to avoid typecasting issues with the environment.

Consider how our animalManagement class loads data from a .txt file:

We begin our program by calling clear() to animalList to make sure we have a new list of animals to add data to. Next, we define our Qt variables (data, myfile and txtInput) to make file storage accessible with the Qt environment. To load data from the text file we call open on myfile and specify our QIODevice as ReadOnly. We also set a pointer to the head of the AnimalList node and declare our animal class. Now we will begin parsing data in the while loop for as long as the end of file. Each line of data is read and animal id, name, physical attributes and non physical attributes are set in that order.

Here is the corresponding code from the animalmanagement class:

```
int AnimalManagement::loadAnimalData(string fileName){
   animalList.clear();
   QString data;
   QFile myfile(QString::fromStdString(fileName));
   myfile.open(QIODevice::ReadOnly);
   QTextStream txtInput(&myfile);
   node<Animal>* temp=animalList.getHeadNode();
   Animal* animal;
while(!txtInput.atEnd()){
     animal = new Animal();
     data = txtInput.readLine();
     animal->setAnimalID(data.toInt());                  //set Animal id
     data = txtInput.readLine();
     animal->setAnimalName(data.toStdString());     //set Animal name
     data = txtInput.readLine();
     animal->pa.setGender(data.toInt());                 //set Animal physical attribute
     data = txtInput.readLine();
     animal->npa.setAggresivity(data.toInt());          //set Animal non physical attribute
     animalList.add(animal);    }
```

Saving data is similar to loading data but instead we are saving with the bitwise << shift operator. We begin by opening a connection to the QFile as a WriteOnly QIODevice. Next we will create a QTextStream to output a text file. We must also initialize the temp node to the head of the clientList. While there are still clients in temp we will parse through the list of clients and store each value as a new line. Once there are no more clients to save we must be sure to close our file. Here is an example of how our clientmanagement class saves data to a .txt file:

```
int ClientManagement::saveClientData(string fileName){
   QFile myfile(QString::fromStdString((fileName)));
   myfile.open(QIODevice::WriteOnly);
   QTextStream txtOutput(&myfile);
```

```
    node<Client>* temp=clientList.getHeadNode();
    Client* client;
    while (temp != NULL){
        client = temp->obj;
        txtOutput<<client->getClientID()<<"\n";
        txtOutput<<QString::fromStdString(client->getClientName())<<"\n";
        txtOutput<<QString::fromStdString(client->ci.getPhone())<<"\n";
        temp = temp->next;
    }
    myfile.close();
    return 0;
}
```

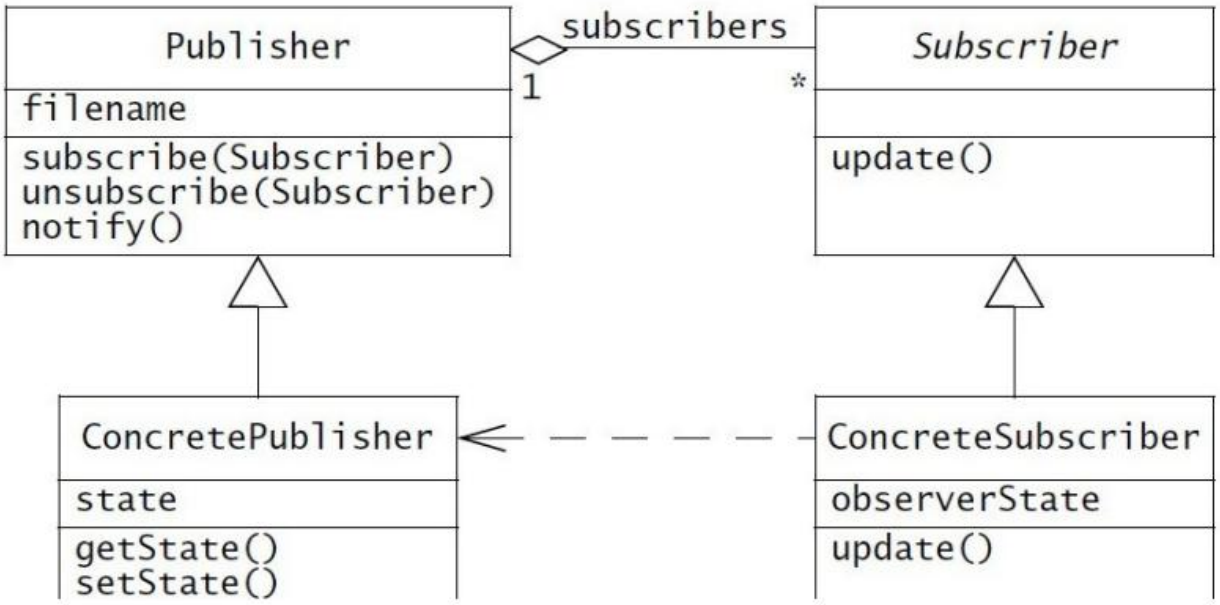
Duplication of data is an important topic, each animal and client should only be entered once with their own unique id.

Adding or editing client and animal information is a privilege for staff and cannot be accessed by regular users. Staff privileges are handled by the requestLogIn() function which determines the type of user account to decide whether to process as staffView or clientView.

If there are multiple entries of the same client or animal then we can separate them by id and delete the extra objects. We must remember to overwrite the file if we want these changes to stay in persistent storage..

## 3.2. Design Pattern

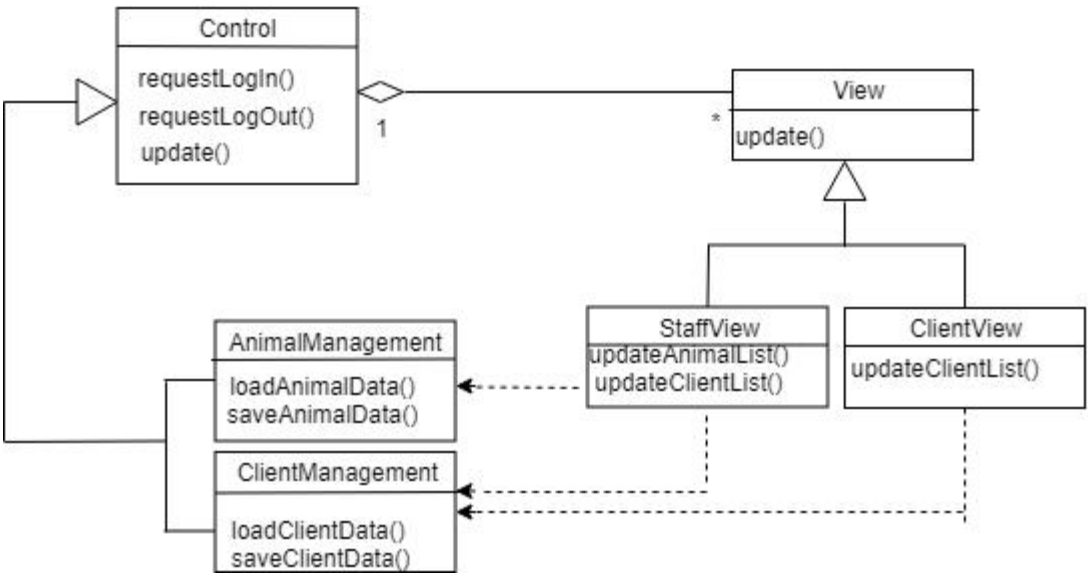
The Gang of Four design pattern that will be used in our algorithm is the Observer pattern. The observer pattern is a behavioural pattern that describes relationships between classes and objects. In this design pattern the subject publishes changes to its state and the observer is notified.




This design pattern is suitable for our project because adding animals or editing client preferences will change the state of the ACM algorithm. The Observer pattern best handles this scenario as we are able to update the algorithm based on any changes that affects matching.

In our algorithm we are concerned with two concretePublishers and concreteSubscribers, client and staff. If a client changes their preferences this will cause our algorithm's result to change. If a staff adds new animals, edits animal information or adds clients this will also cause our algorithm to change. Our Publisher, control, handles these changes and notifies the view of any updates.

Clients and Staff also have the option to view a list of animals or view the detailed description of an animal. Both these changes will be updated in view, the publisher. Staff have the option of viewing a list of clients, adding or removing clients will cause changes in StaffView and ClientView.




**Control**:
The control is responsible for maintaining the view. It is an abstract class containing the operations for adding and removing views. requestLogIn() will create a new ClientView or StaffView depending on the user's credentials. requestLogOut() will simply close the view. update() will call saveAnimalData() in AnimalManagement class and saveClientData() in the ClientManagement class. These changes are also sent to StaffView with updateAnimalList() and ClientView using updateClientList().

**View:**
View is the abstract class that is used to update StaffView and ClientView. When the state of either AnimalManagement or ClientManagement is changed StaffView and ClientView is updated. If staff or client chooses to view a list of animals or an animal's detailed profile the View will be updated.

**StaffView:**
The staff concrete observer. When saveAnimalData() or loadAnimalData() is called, StaffView will handle changes to the staff observer with updateAnimalList(). Staff can also add or edit clients so changes to AnimalManagement will also call updateAnimalList() in StaffView.

**ClientView:**
The client concrete observer. When saveClientlData() or loadClientData() is called, ClientView will handle changes to the client observer with updateClientList().

**AnimalManagement**:
The animal concrete subject. AnimalManagement is the state of the list of animals. Animals can be read from storage using loadAnimalData() and can be updated using saveAnimalData(). Staff can make changes to AnimalManagement in StaffView.

**ClientManagement**:
The client concrete subject. ClientManagement is the state of the list of clients. Clients can be read from storage using loadClientData() and can be updated using saveClientData(). Staff and clients can make changes to ClientManagement in StaffView and ClientView respectively.