“System Design”

4our People

Kaleb Tesfay - 101048170
Adam Farah - 100966918
Japinder Sandhu - 101021899
Samson Teklay - 100972872
03/14/19

1. Introduction

Animal shelter staff often face issues in the processes of animal adoption. One of the main issues is allowing clients to adopt animals which they are not compatible with. A client may want to adopt a pet to enjoy the pets comfort and companionship, however if the pets' attributes are not compatible with the clients' preferences, it would most likely lead to a situation where the pets could get mistreated. Optimal matches bring about lasting bonds with clients. Therefore, to avoid this issue, our team has come up with a system called cuACS (Carleton University Animal-Client System) that runs an algorithm which automatically matches an animal with a client based on their compatibility.

This document will cover the system design which includes system decomposition and design strategies. Developers define the design goals of the project and decompose the system into smaller subsystem that can be realized by our team. We will also select strategies for building the system, such as hardware/software strategy. The result of system design is a model that includes a subsystem decomposition and a clear description of each of these strategies. System design is decomposed into several activities, each addressing part of the overall problem of decomposing the system.

2. Subsystem Decomposition

2.1. D2 Implementation Decomposition

2.1.1. Logical Subsystem Descriptions

In this section, we will briefly discuss how the implemented D2 feature design is decomposed into subsystems as well as describe every logical subsystem of the feature implementation. In order to reduce complexity of the solution domain, the system is decomposed into smaller and simpler parts called subsystems. Subsystems are group of related classes that have well-defined interfaces. The subsystems used in this design are relatively independent of each other. Thus, they are created in a way that they can be assigned to a single developer or group of developers.

We used the three-tier architectural style in the D2 feature implementation. The three-tier architectural style organizes the subsystems into three layers. The first layer, **Interface**, includes a User Interface subsystem. This subsystem is responsible of providing communication between the cuACS and the user. It includes the use of mouse, keyboard and displaying screens to allow the user interact with the system. The user interface subsystem contains MainWindow, Control and Shelter classes.

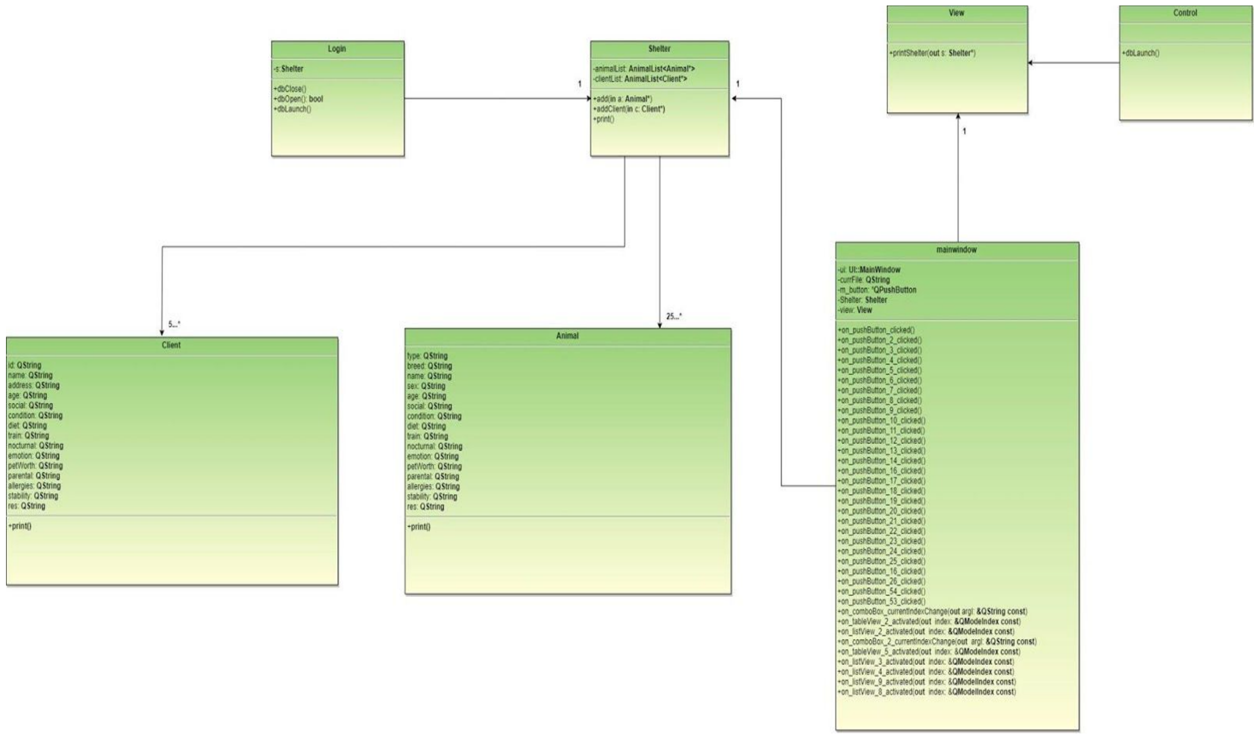
The second layer, **Application**, includes four subsystems: UserManagement, ClientManagement, AnimalManagement and Notification. These subsystems are relatively independent of each other; where a change in one has little to no impact on the other subsystems. The **UserManagement** subsystem assists users (shelter staff or client) log in to their accounts to manage and access specific information pertaining to those accounts. For example, the shelter staff can add or remove clients as well as the client can access or modify their own profiles. This subsystem contains Shelter and Client classes.

The **AnimalManagement** subsystem allows managing animals in the shelter. It includes Animal and Shelter classes.

The **Notification** subsystem notifies other subsystems when an event has occurred. It provides service to other subsystems as it only calls the subsystem which had a change or update. For example, the Notification subsystem notifies the UserManagement subsystem when a shelter staff adds a new client. This subsystem is associated with classes such as the MainWindow, Shelter, Client and View classes.

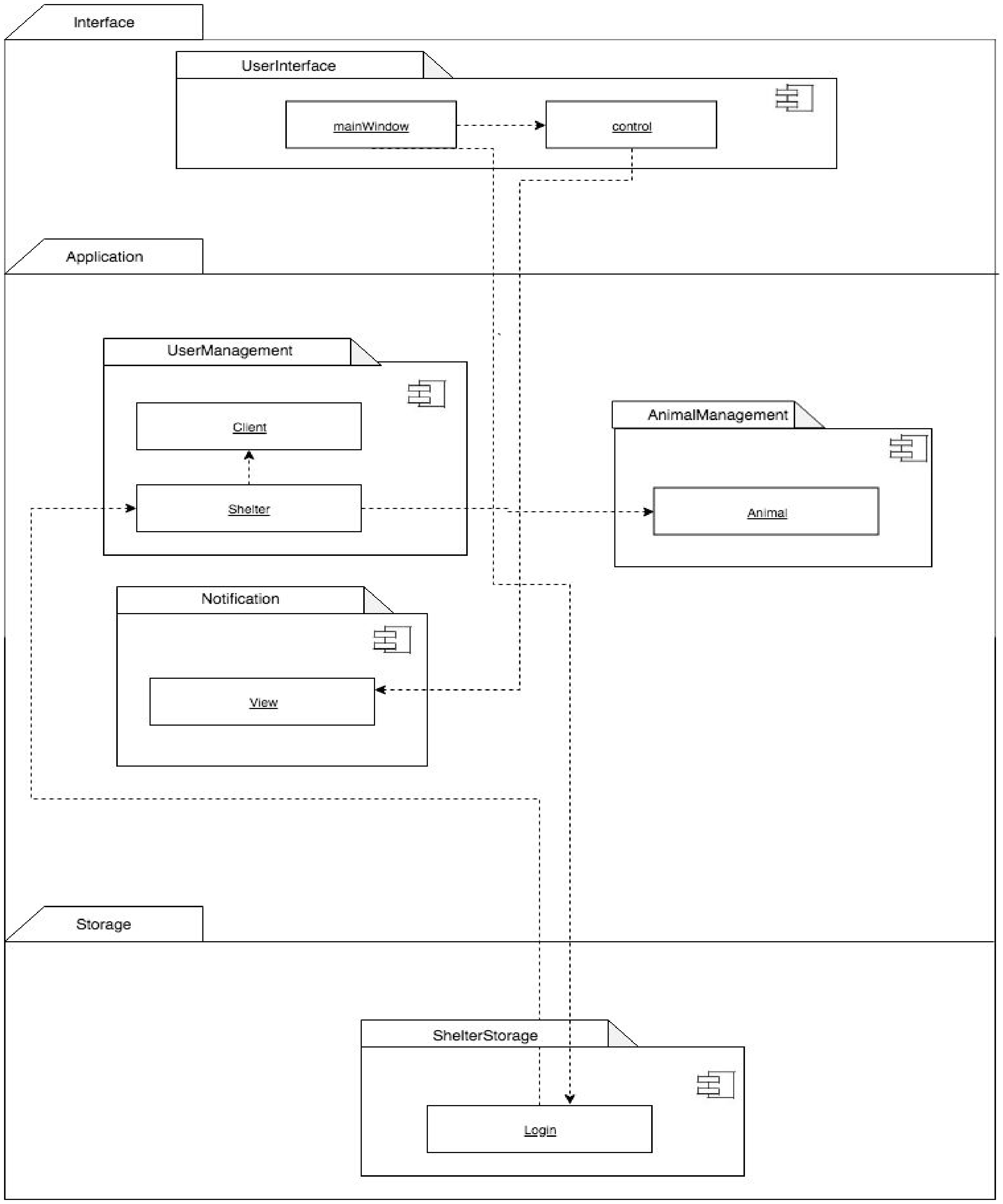
The **ShelterStorage** subsystem is found in the third layer. It's responsible of storing the observer's (animals and clients) into memory. A LogIn class is used by the shelter staff to access the database.

Class Diagram



Login
-s: Shelter
+dbClose()
+dbOpen(): bool
+dbLaunch()
1
Shelter
-animalList: AnimalList<Animal*>
-clientList: AnimalList<Client*>
+add(in a: Animal*)
+addClient(in c: Client*)
+print()
1
View
+printShelter(out s: Shelter*)
Control
+dbLaunch()
1
5...*
Client
id: QString
name: QString
address: QString
age: QString
social: QString
condition: QString
diet: QString
train: QString
nocturnal: QString
emotion: QString
petWorth: QString
parental: QString
allergies: QString
stability: QString
res: QString
+print()
25...*
Animal
type: QString
breed: QString
name: QString
sex: QString
age: QString
social: QString
condition: QString
diet: QString
train: QString
nocturnal: QString
emotion: QString
petWorth: QString
parental: QString
allergies: QString
stability: QString
res: QString
+print()
mainwindow
-ui: UI::MainWindow
-currFile: QString
-m_button: *QPushButton
-Shelter: Shelter
-view: View
+on_pushButton_clicked()
+on_pushButton_2_clicked()
+on_pushButton_3_clicked()
+on_pushButton_4_clicked()
+on_pushButton_5_clicked()
+on_pushButton_6_clicked()
+on_pushButton_7_clicked()
+on_pushButton_8_clicked()
+on_pushButton_9_clicked()
+on_pushButton_10_clicked()
+on_pushButton_11_clicked()
+on_pushButton_12_clicked()
+on_pushButton_13_clicked()
+on_pushButton_14_clicked()
+on_pushButton_16_clicked()
+on_pushButton_17_clicked()
+on_pushButton_18_clicked()
+on_pushButton_19_clicked()
+on_pushButton_20_clicked()
+on_pushButton_21_clicked()
+on_pushButton_22_clicked()
+on_pushButton_23_clicked()
+on_pushButton_24_clicked()
+on_pushButton_25_clicked()
+on_pushButton_16_clicked()
+on_pushButton_26_clicked()
+on_pushButton_54_clicked()
+on_pushButton_53_clicked()
+on_comboBox_currentIndexChange(out argl: &QString const)
+on_tableView_2_activated(out index: &QModelndex const)
+on_listView_2_activated(out index: &QModelndex const)
+on_comboBox_2_currentIndexChange(out argl: &QString const)
+on_tableView_5_activated(out index: &QModelndex const)
+on_listView_3_activated(out index: &QModelndex const)
+on_listView_4_activated(out index: &QModelndex const)
+on_listView_9_activated(out index: &QModelIndex const)
+on_listView_8_activated(out index: &QModelIndex const)

UML Component Diagram



2.2. Full System Decomposition

2.2.1. Subsystem Descriptions

This section discusses how the entire system is decomposed into several subsystems. It is more complex and advance than the D2 implementation decomposition. Decomposition of the subsystems result in various independent subsystems which allow complexity reduction. It also allows assigning implementation of these subsystems to individual groups. The subsystems of the full system decomposition have loose coupling where an impact on one subsystem has little to no impact on other subsystems. On the other hand, the classes within the subsystems have high cohesion. The classes are closely related to each other to the limit where a change in one class could cause an impact on the other class. Similarly, the full system decomposition also has a three-tier architectural style.

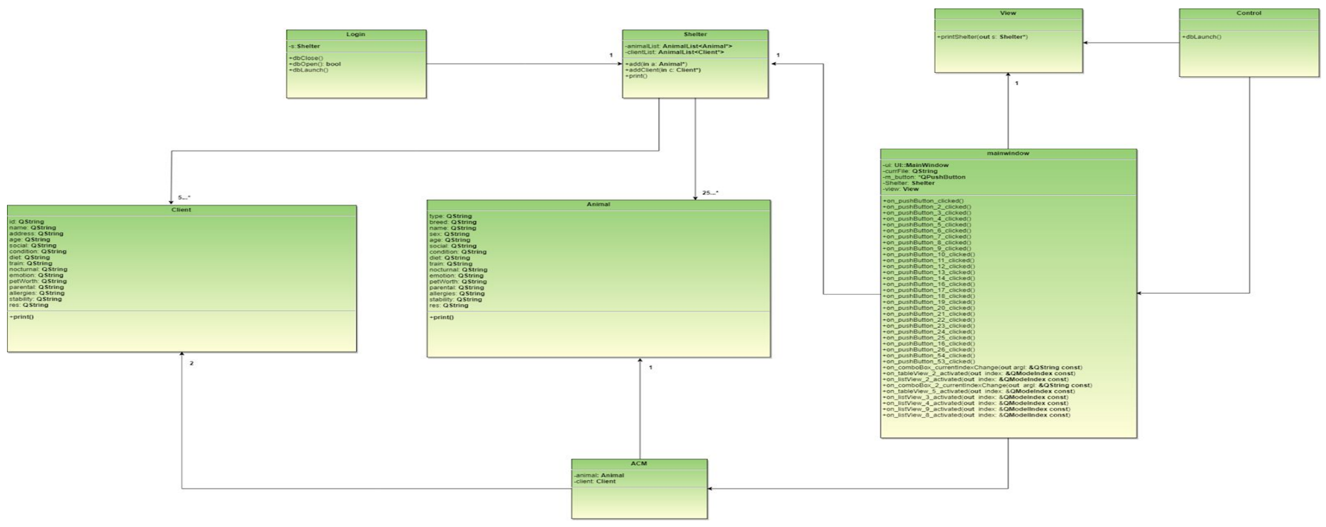
The **UserInterface** subsystem allows interaction between the users and the system. The UserInterface Subsystem provides the means of communicating between the user interface. The subsystem will connect with the QT GUI framework which has multimodal input, including keyboard and mouse. With The UserInterface Subsystem, MainWindow.cpp is the major class associated, this class is used to for button functionality and GUI functionality. MainWindow.cpp is using Control.cpp to help notify the DB when certain updates are made. MainWindow.cpp will also communicate with the ACM.cpp class allowing users to launch the cuACS matching algorithm. High cohesion is relevant within this Subsystem, when the MainWindow.cpp class is updated it directly affects Control.cpp entity objects. This has low coupling with other Subsystems.

The **UserManagement** subsystem contains classes that allow both shelter staff and clients log into and manage their accounts. The classes included in this subsystem are Shelter and Client classes. The shelter class deals with what the shelter staff do with the system while the client class deals with what the clients do with the system. For example, a shelter staff can add a client to the system, and clients can edit their own profile once they are added by the shelter. These classes are highly cohesive because if a client is added to the system or edits their profile, then there would be a change in the Shelter class. The Algorithm subsystem will be affected when the Shelter is updated. The Algorithm subsystem has loose coupling with UserManagement.

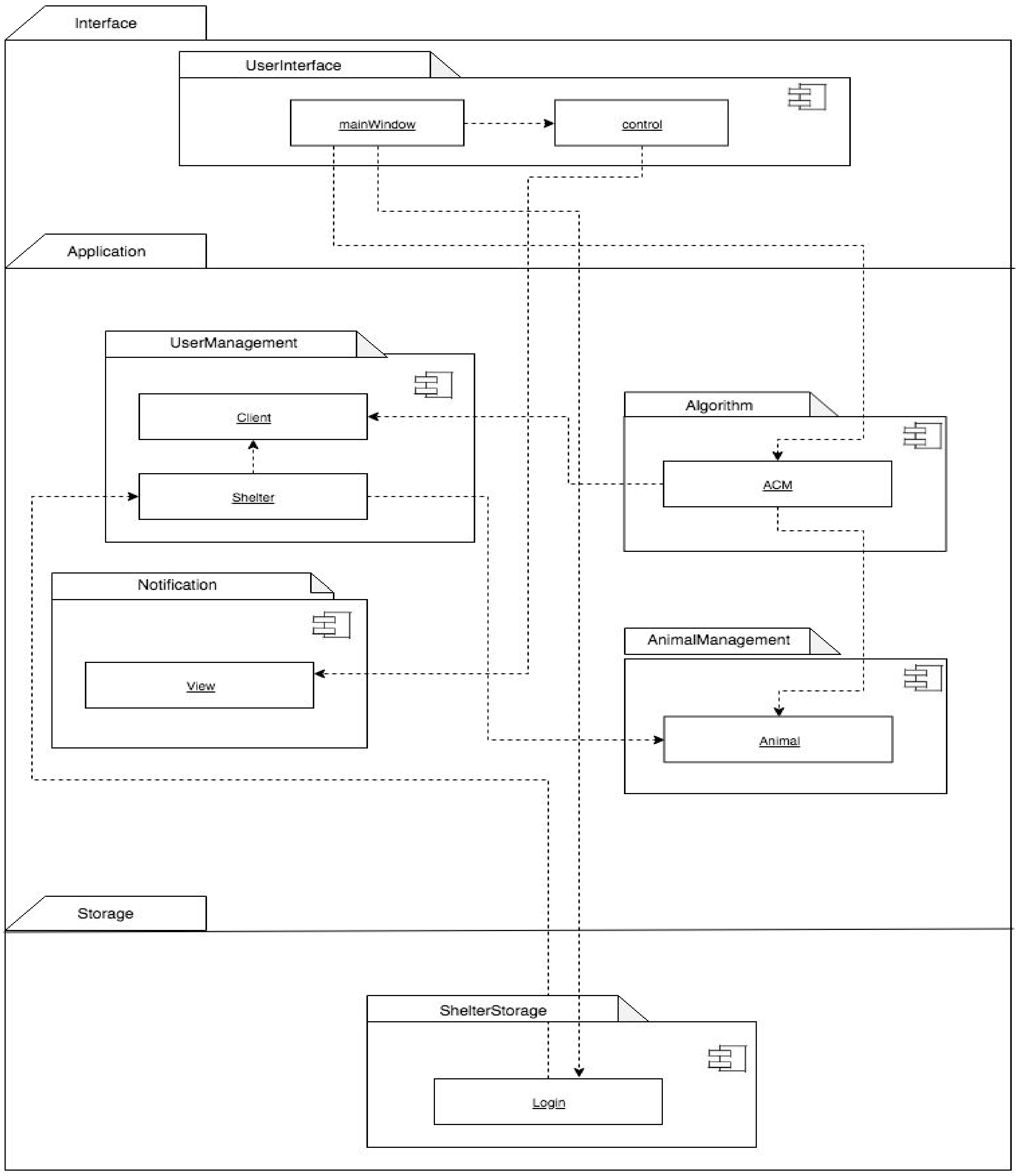
The **Algorithm** subsystem consists of ACM class. The Algorithm Subsystem provides a service through which cuACS algorithm can be run using the information provided by the users. The shelter staff uses the client class and animal class to compute a matching algorithm to find the most compatible match. When a compatible animal-client match is found, the view class is used to update the database result. The Algorithm subsystem has coupling with UserManagement, AnimalManagement and UserInterface. Low coupling is considered because there is minimal change, only simple data reading.

The **ShelterStorage** subsystem provides services for storage and retrieval of any persistent data in the cuACS project. These services are provided through a class Login.cpp DB is established through this. Login establishes the ShelterStorage after the Login occurs. The Notification subsystem which contains the View.cpp file will notify the ShelterStorage with particular changes.

2.2.2. Class Diagram


Login
-s: Shelter
+dbClose()
+dbOpen(): bool
+dbLaunch()
Shelter
-animalList: AnimalList<Animal*>
-clientList: AnimalList<Client*>
+add(in a: Animal*)
+addClient(in c: Client*)
+print()
1
1
View
+printShelter(out s: Shelter*)
Control
+dbLaunch()
1
5...*
Client
id: QString
name: QString
address: QString
age: QString
social: QString
condition: QString
diet: QString
train: QString
nocturnal: QString
emotion: QString
petWorth: QString
parental: QString
allergies: QString
stability: QString
res: QString
+print()
25...*
Animal
type: QString
breed: QString
name: QString
sex: QString
age: QString
social: QString
condition: QString
diet: QString
train: QString
nocturnal: QString
emotion: QString
petWorth: QString
parental: QString
allergies: QString
stability: QString
res: QString
+print()
mainwindow
-ui: UI::MainWindow
-currFile: QString
-m_button: *QPushButton
-Shelter: Shelter
-view: View
+on_pushButton_clicked()
+on_pushButton_2_clicked()
+on_pushButton_3_clicked()
+on_pushButton_4_clicked()
+on_pushButton_5_clicked()
+on_pushButton_6_clicked()
+on_pushButton_7_clicked()
+on_pushButton_8_clicked()
+on_pushButton_9_clicked()
+on_pushButton_10_clicked()
+on_pushButton_11_clicked()
+on_pushButton_12_clicked()
+on_pushButton_13_clicked()
+on_pushButton_14_clicked()
+on_pushButton_16_clicked()
+on_pushButton_17_clicked()
+on_pushButton_18_clicked()
+on_pushButton_19_clicked()
+on_pushButton_20_clicked()
+on_pushButton_21_clicked()
+on_pushButton_22_clicked()
+on_pushButton_23_clicked()
+on_pushButton_24_clicked()
+on_pushButton_25_clicked()
+on_pushButton_16_clicked()
+on_pushButton_26_clicked()
+on_pushButton_54_clicked()
+on_pushButton_53_clicked()
+on_comboBox_currentIndexChange(out argl: &QString const)
+on_tableView_2_activated(out index: &QModelIndex const)
+on_listView_2_activated(out index: &QModelIndex const)
+on_comboBox_2_currentIndexChange(out argl: &QString const)
+on_tableView_5_activated(out index: &QModelIndex const)
+on_listView_3_activated(out index: &QModelIndex const)
+on_listView_4_activated(out index: &QModelIndex const)
+on_listView_9_activated(out index: &QModelIndex const)
+on_listView_8_activated(out index: &QModelIndex const)
2
1
ACM
-animal: Animal
-client: Client

2.2.3. UML Component Diagram



2.3. Design Evolution

2.3.1. Decomposition Difference Description

One of the main differences between the D2 feature implementation and the full system decomposition is the fact that the animal-client matching algorithm is included in the full system decomposition. Although both the system decompositions are very similar to each other, addition of the ACM has a great impact on the system. The ACM also has loose coupling with the other subsystems in the application layer. Other than that difference mentioned above, both D2 implementation feature and the full system decomposition follow the same architectural style (three-tier).

2.3.2. Design Strategies

The design for both D2 feature implementation and the full system decomposition is relatively the same except that the full system decomposition has an animal-client matching algorithm added to it. Since the beginning of the project, our team have been following the same system strategy where we only had to make a little change from our D2 feature implementation. Considering the fact that our full system decomposition includes the animal-client matching algorithm, we could believe that it makes it superior over the initial D2 system decomposition.

2.3.3. Design Evolution

D2 system decomposition evolved in a way that it has now an ACM it the application layer of the three-tier. There isn't much of evolution on the system decomposition with the exception of the animal-client matching algorithm being added to the full system decomposition.

3. Design Strategies

3.1. Persistent Storage

Persistent data is what outlives a single execution of the system, as said in class. Given that, our system needs to have finite client and animal data that will be in the database for all executions of this program.

Linked list (c++) and database (sqLite) are connected to each other. Whenever an object is added to the database, it also gets dynamically allocated into memory by the linked list. This helps to allow concurrent access and detailed access.

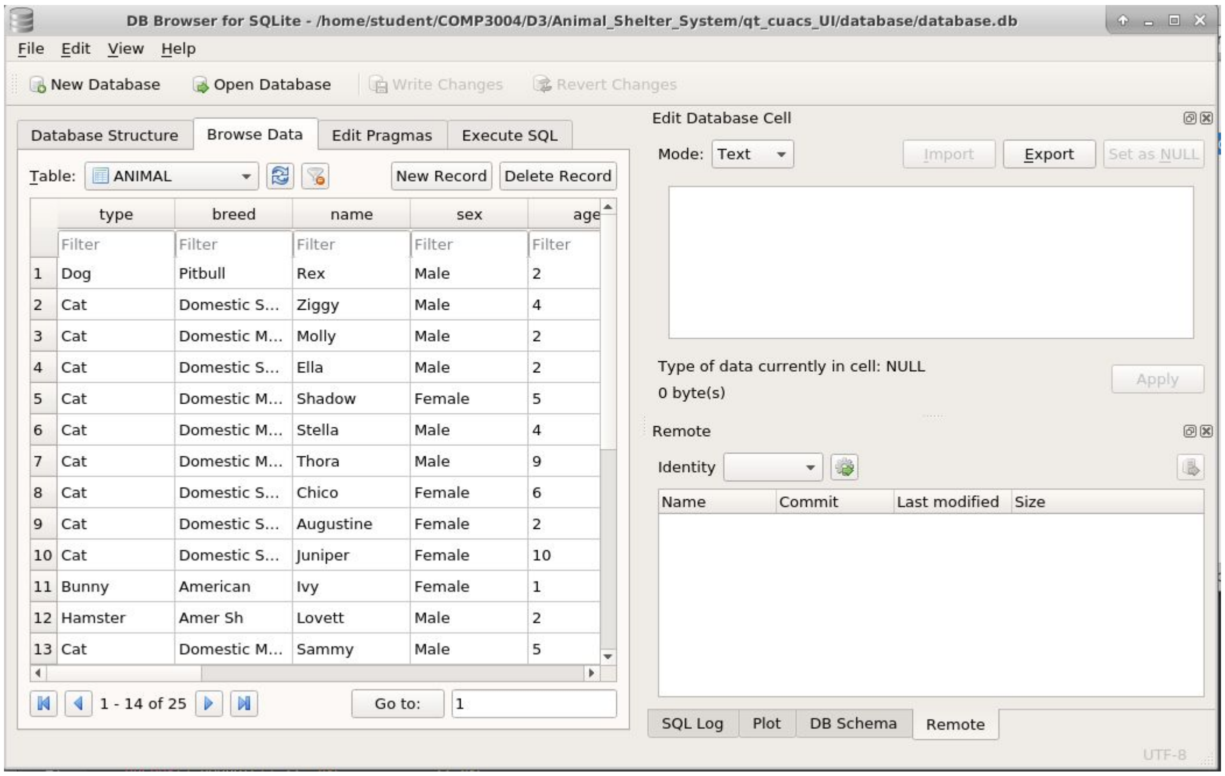


Table: ANIMAL

	type	breed	name	sex	age
	Filter	Filter	Filter	Filter	Filter
1	Dog	Pitbull	Rex	Male	2
2	Cat	Domestic S...	Ziggy	Male	4
3	Cat	Domestic M...	Molly	Male	2
4	Cat	Domestic S...	Ella	Male	2
5	Cat	Domestic M...	Shadow	Female	5
6	Cat	Domestic M...	Stella	Male	4
7	Cat	Domestic M...	Thora	Male	9
8	Cat	Domestic S...	Chico	Female	6
9	Cat	Domestic S...	Augustine	Female	2
10	Cat	Domestic S...	Juniper	Female	10
11	Bunny	American	Ivy	Female	1
12	Hamster	Amer Sh	Lovett	Male	2
13	Cat	Domestic M...	Sammy	Male	5

1 - 14 of 25 Go to: 1

Mode: Text

Type of data currently in cell: NULL
0 byte(s)

Name	Commit	Last modified	Size

From the image above (and logged on a designated terminal) our database is organized for view and modification, in .

This database has two main objects (Client, Animal) whose attributes are can be transformed.

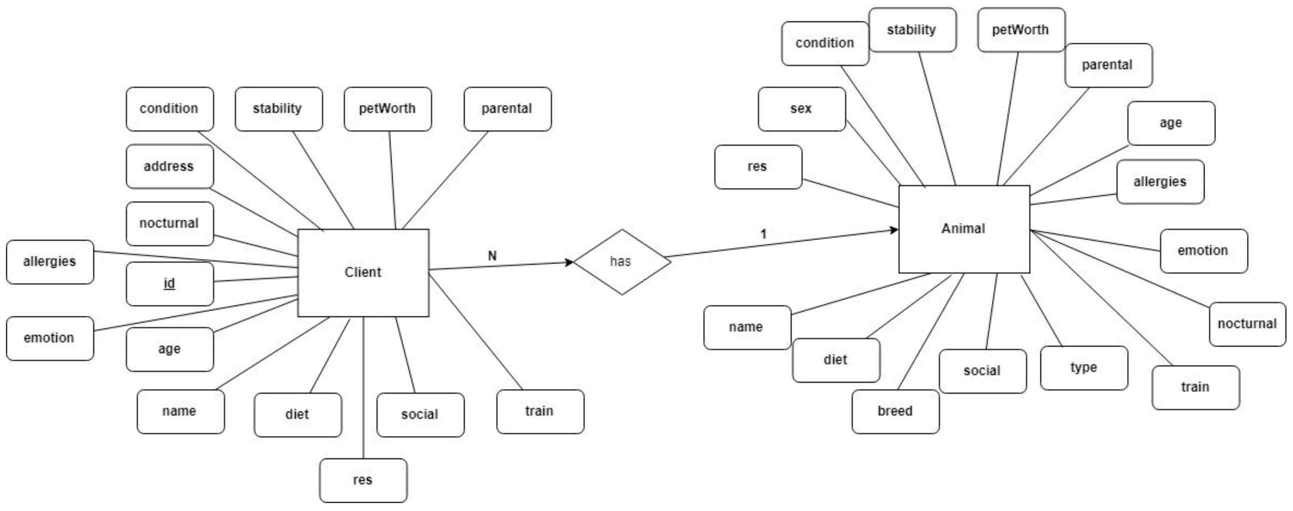
Client:

Client object plays pivotal role in matching, this takes the various types of animalistic values in-conjunction to preference returning the best matches. Holding values of primary key value for id and regular attributes for name and address. The rest of the values fall under matching preferences which are highlighted below in the Animal section. The values

Animal:

Attributes are type, breed, name, sex, age, social, condition, diet, train, concturnal, emotion, petWorth, parental, allergies, stability and res.

The minimum amount of clients, animals, and staff consistently are held in the database and adjustments on these objects will still hold true on re-executions of the system.



3.2. Design Patterns

Design patterns in our system decomposition provides solutions to common software design problems. In the case of object-oriented programming, design patterns are generally aimed at solving the problems object generation and interaction, rather then the large scale problems of overall software architecture. They give generalised solutions in form of templates that may be applied to real-world problems. The design pattern we utilized in our cuACS system are the Observer, Bridge and Facade.

In the case for the observer design pattern, this design pattern is categorized as a behavioral pattern because it defines a manner for controlling communication between classes or entities. The observer pattern is used to allow single object, known as the subject, to publish changes to its state. Many other observer objects that depend upon the subject can subscribe to it so that they immediately and automatically notified of any changes to the subject state. In terms of our implementation of how we are using the observer design pattern, the observer in our scenario are the staff and client interfaces and our base class is the mainwindow class while our subject is the Shelter class. The Staff and Client interfaces are the observers because they are getting updated by our base class *(mainwindow.cpp)*. These interfaces are getting updated every time a new Client or Animal is added or Edited. The Shelter class is the subject because it is utilizing an add function which is implemented by our Linked List class *(AnimalList.cpp)*. The subject class separates the lists into two functions. One function is for adding animal (+add(in Animal)), and the other function is for adding client (+addClient(in Client)). This stores the objects into dynamic memory. The view class can be considered as the concrete subject class has it inherits from the subject class and gets the state of the current Shelter. The pattern gives loose coupling between the subject and it's observers. This subject holds a collection of observers that are set only at runtime. Each observer may be of any class that inherits from a known base class or implements a common interface. The actual functionality of the observers and their use of the state data need not be known by the subject.

In the case for the Bridge Design Pattern, this design pattern is categorized as a structural pattern because it defines a manner for creating relationships between classes or entities. The bridge pattern is used to separate the abstract elements of a class from the implementation details. In terms of our implementation of how we are using the Bridge Design Pattern, we have a Staff and Client interfaces that use an abstraction class which is the GUI. The implementer in this case is the database which gets implemented in the abstraction. The function that get used by the Staff and Client interfaces are the concrete implementers which changes the state of the database which is determined based the functionality of the concrete implementers. For example, add animal, add client, edit animal, edit client, view animals, and view clients. When using the bridge design pattern, far fewer classes are necessary for all but the simplest scenarios. This is due to the pattern removing platform dependencies from the abstraction. Another benefit of the bridge pattern is that it introduces the possibility of changing the implementation details at runtime.

In the case for the Facade Design Pattern, this Design Pattern is categorized as a structural Design Pattern because it defines a manner for creating relationships between classes or entities. The facade design pattern is used to define a simplified interface to a more complex subsystem. In terms of our implementation of how we are using the Facade Design Pattern, the Facade class is considered to be our GUI and utilize the packages to be the Client and Staff interfaces. The facade class contains the set of simple functions that are made available to it's users and that hides the complexities if the difficult-to-use subsystems. The complexity of the functionality is accessed by the packages in which they are the Staff and Client interfaces. The path the Staff interface takes is the adding, viewing, and editing of clients and animals and the path the Client interface is taking is the viewing of animals, and the editing of their personally information. The Facade class is a wrapper that contains a set of members that are easily understood and simple to use. These members access the subsystem on behalf of the Facade user, hiding the implementation details.