CARLETON UNIVERSITY

cuACS

System Design Document

Team QuackJaws
Jake Bauer
Ashlee Foureyes
Skyler Gubbels
Will Watt

Submitted to:
Dr. Christine Laurendeau
COMP3004 – Object-Oriented Software Engineering
School of Computer Science
Carleton University
March 21, 2019

Contents

1 Introduction....2
1.1 The cuACS System....2
1.2 Document Overview....3
2 Subsystem Decomposition....4
2.1 Deliverable 2 Subsystem Decomposition....4
2.2 Full Subsystem Decomposition....10
2.3 Design Evolution....18
2.3.1 Deliverable 2 Explanation....18
2.3.2 Deliverable 2 Limitations....18
2.3.3 Goals for Improvement of the Entire System....18
2.3.4 Explanation/Evolution of Design Choices....19
3 Design Strategies....20
3.1 Persistent Storage....20
3.1.1 Explanation of the Operation and Design of Persistent Storage in cuACS....20
3.1.2 ER Diagram and Table Schema....23
3.2 Design Patterns....26
3.2.1 Adapter....26
3.2.2 Abstract Factory....27
3.2.3 Bridge....28
3.2.4 Composite....29
3.2.5 Command....29
3.2.6 Observer....30
3.2.7 Proxy....30
3.2.8 Strategy....30
4 Index of Tables....31
5 Index of Figures....31
6 Sources....31

1 Introduction

1.1 The cuACS System

Animal shelters are tasked with the responsibility of providing care to animals in need while they await adoption into a loving home. Humans that are seeking the companionship of a pet can visit a shelter and choose an animal to adopt. However, with this process an issue often arises where a pet is adopted by a human with whom they are not fully compatible. This mismatch can be a result of a variety of reasons, including temperament, conflicting lifestyle

requirements, and the physical and non-physical needs of both the pet and the potential adopter.

Carleton University Animal Care System (cuACS) aims to alleviate the issue of mismatching by providing a tool that automatically matches a pet to a potential owner based on compatibility. This compatibility measurement is based on matching the numerous physical and non-physical traits applicable to the animals in the shelter as well as the traits and desires of potential owners. cuACS enables a smooth adoption process, and ensures the experience will prove positive and match an animal which both fulfills a clients expectations and suits their lifestyle and needs.

Within the cuACS system, animals and clients are able to be added and edited. When either is added, the user is required to enter information on both physical and non-physical traits to create the profile. cuACS provides support for cats, dogs, rabbits and lizards. The system takes into account over 20 physical and non-physical traits of the animal, as well as client traits in order to provide an optimal match. An algorithm is utilized to determine which animal and which client are the most suited for each other.

1.2 Document Overview

This document aims to address the transformation of the analysis model into a system design model. This includes the primary activity of designing the subsystem decomposition. Subsystem decomposition is where the system is decomposed into smaller parts based on the use case and analysis model. Subsystems are replaceable parts of the system that encapsulate the state and behavior of the classes contained within defined interfaces.

This document includes the implementation decomposition of the second deliverable (D2). It will describe each logical subsystem implemented with our D2 features, and include class diagrams and packages. This document will also detail the decomposition of the full system. A description of each subsystem within the entire cuACS system will be provided.

Design evolution is also explored to present the differences between the decomposition for the D2 feature implementation and the decomposition for the entire cuACS system. Design choices will be discussed in detail, including the reasoning of their implementation.

Persistent storage will be described in detail, including the strategy for storing and the reasoning behind the organization of the data. Each object in persistent storage will be detailed. It will detail the sqlite3 database used in cuACS, while providing the reasoning behind

choosing this type. This section will also detail the structure of the database, and explain in-depth the way in which data is created, processed and stored.

Finally, an explanation of design patterns will be provided. Design patterns are the partial solutions to common problems. In this section, the Adapter, Abstract Factory, Bridge, Composite, Command, Observer, Proxy and Strategy design patterns will be explore. It will give an overview of each, including their intent, and also provide reasoning on why each was or was not chosen to be implemented in the cuACS system.

2 Subsystem Decomposition

A subsystem decomposition describes the system as a collection of subsystems which interact and interface with each other. It details all of the individual parts that make up the system. In this section, the subsystem decomposition is performed for the code-base delivered for the second deliverable as well as an up-to-date, refined decomposition for the system as a whole. Following that, there is a comparison between the two decompositions detailing the shortcomings of the design of the second deliverable code-base and how the new system design maximizes cohesion while reducing coupling.

2.1 Deliverable 2 Subsystem Decomposition

The code for the second deliverable was written without the wisdom of design patterns and the concepts of coupling and cohesion and is therefore not as effective as it could be at being a maintainable, simple (as opposed to overly-complex), and efficient system. Below is *Figure 1—Deliverable 2 Subsystem Decomposition UML Class Diagram* detailing the different subsystems and the classes that make up those subsystems that were decomposed from the deliverable 2 code-base. Following that are tables with brief descriptions of each subsystem and class. Following that, *Figure 2—Deliverable 2 Subsystem Decomposition UML Package Diagram* gives an overview of how each of the subsystems and classes interact with each other. This diagram describes the level of coupling and cohesion present in the system between and within the subsystems respectively. It can be observed from the above-noted figure that the deliverable 2 subsystem decomposition exhibits relatively strong coupling and weak cohesion which is not ideal. It is also a very complicated diagram because of how the subsystems interact with one-another.

Finally, *Figure 3—Deliverable 2 Subsystem Decomposition UML Component Diagram* details how each subsystem interacts with other subsystems using a UML component diagram with

ball-and-socket notation where the balls represent the services that a subsystem provides and the sockets represent the services upon which the subsystem is dependent. This figure serves to show, in a more simplified and easier-to-parse format, the ways that the subsystems interact with one-another.

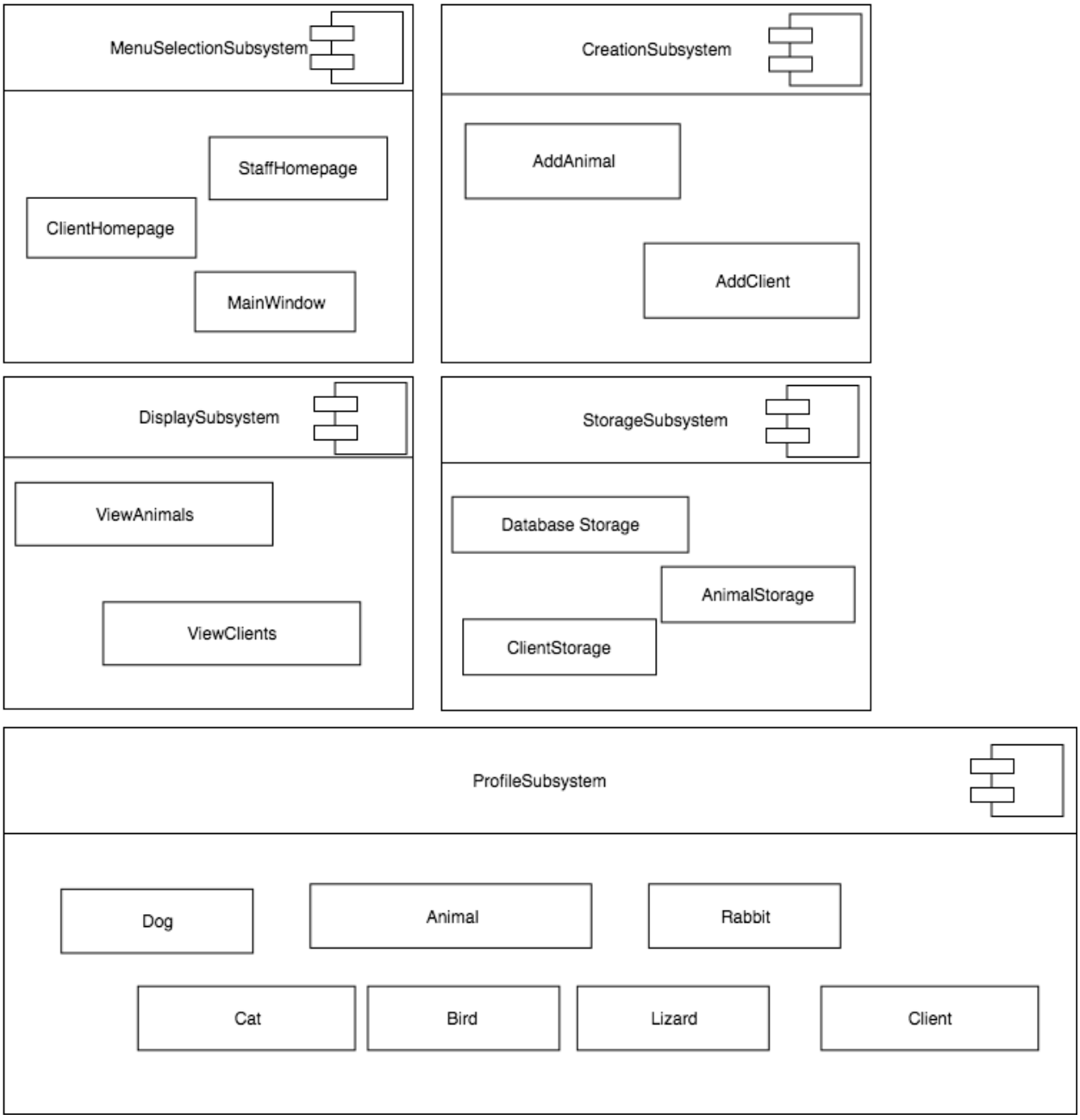

Figure 1—Deliverable 2 Subsystem Decomposition UML Class Diagram

Table 1—Deliverable 2 MenuSelectionSubsystem

MenuSelectionSubsystem
• The MenuSelectionSubsystem is responsible for allowing the user to choose how they want to interact with the system.
MainWindow
• The MainWindow allows a user to log into either the StaffHomepage or ClientHomepage. It checks for valid email if user is trying to log in as an existing Client.
StaffHomepage
• The StaffHomepage allows the user to initiate the Animal/Client creation process. Allows the user to initiate the viewing of all Animals and Clients.
ClientHomepage
• The ClientHomepage allows the user to view all Animals.

Table 2—Deliverable 2 CreationSubsystem

CreationSubsystem
• The CreationSubsystem is responsible for creating Animal and Client profiles. It is this subsystem that gathers all of the users' inputs and stores them in new objects.
AddAnimal
• The AddAnimal class gets all required information from the user and creates a new Animal object.
AddClient
• The AddClient class gets all the required information from the user and creates a new Client object.

Table 3—Deliverable 2 DisplaySubsystem

DisplaySubsystem
• The Display subsystem is responsible for viewing all the Animals and Clients in the StorageSubsystem.
ViewAnimals
• ViewAnimals displays all animals in the StorageSubsystem along with all their attributes. Users are able to filter results by Animal species.
ViewClients
• ViewClients displays all clients in the StorageSubsystem along with their contact information.

Table 4—Deliverable 2 StorageSubsystem

StorageSubsystem
• The StorageSubsystem is responsible for local and persistent storage of all Animal and Client objects. Responsible for assigning unique Animal and Client id's.
AnimalStorage
• AnimalStorage is responsible for storing all 5 species of Animal objects created by AddAnimal. Once AnimalStorage assigns id Animal object information is added to DatabaseStorage.
ClientStorage
• ClientStorage is responsible for storing the Clients. Once ClientStorage assigns a unique id the Client information is added to DatabaseStorage.
DatabaseStorage
• DatabaseStorage is persistent storage for all Animal and Client attributes. Used by MainWindow at startup to create new Animal/Client objects from database data.

Table 5—Deliverable 2 ProfileSubsystem

ProfileSubsystem
• The ProfileSubsystem is responsible for locally storing information entered by the user in the form of Animal or Client objects.
Animal
• The Animal class includes 5 sub-classes (Dog, Cat, Bird, Lizard, Rabbit). Used to store information input by user regarding Animals in the shelter. These are then stored in AnimalStorage.
Profile
• Profiles are used to store information input by user regarding their own profile. These are then stored in ClientStorage.

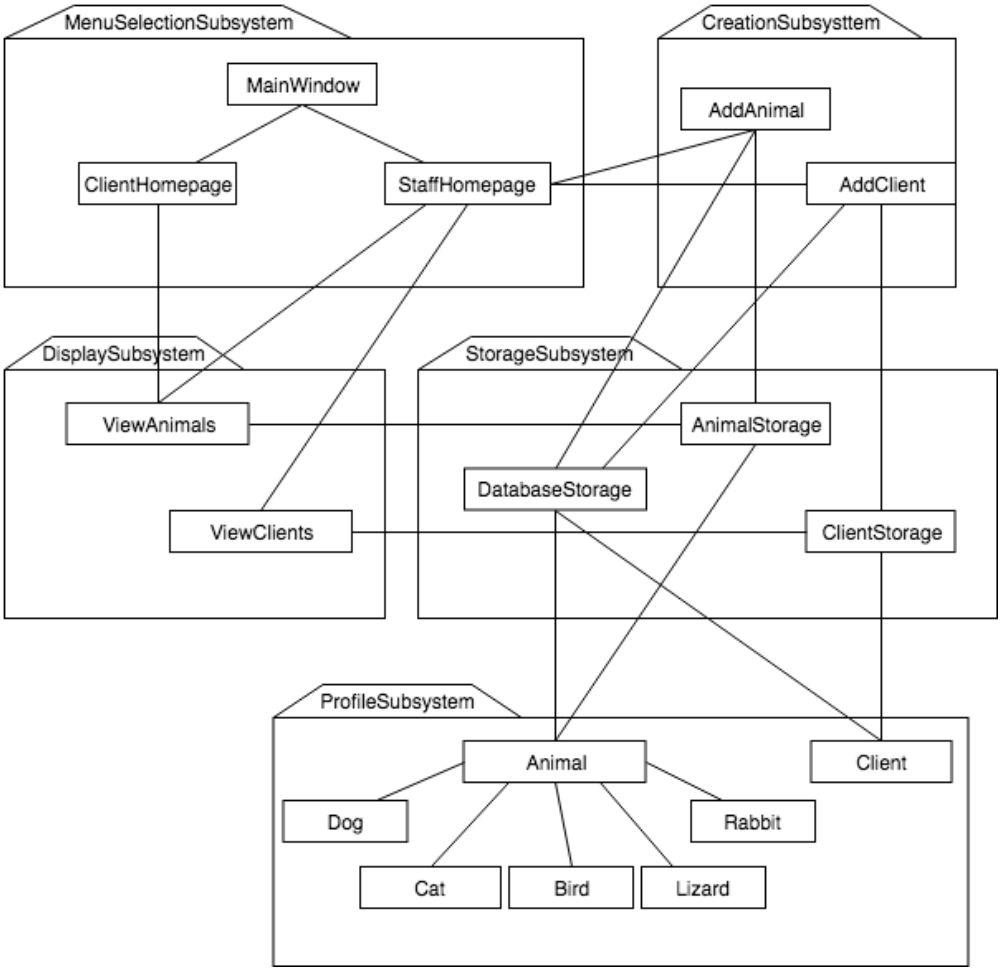

Figure 2—Deliverable 2 Subsystem Decomposition UML Package Diagram

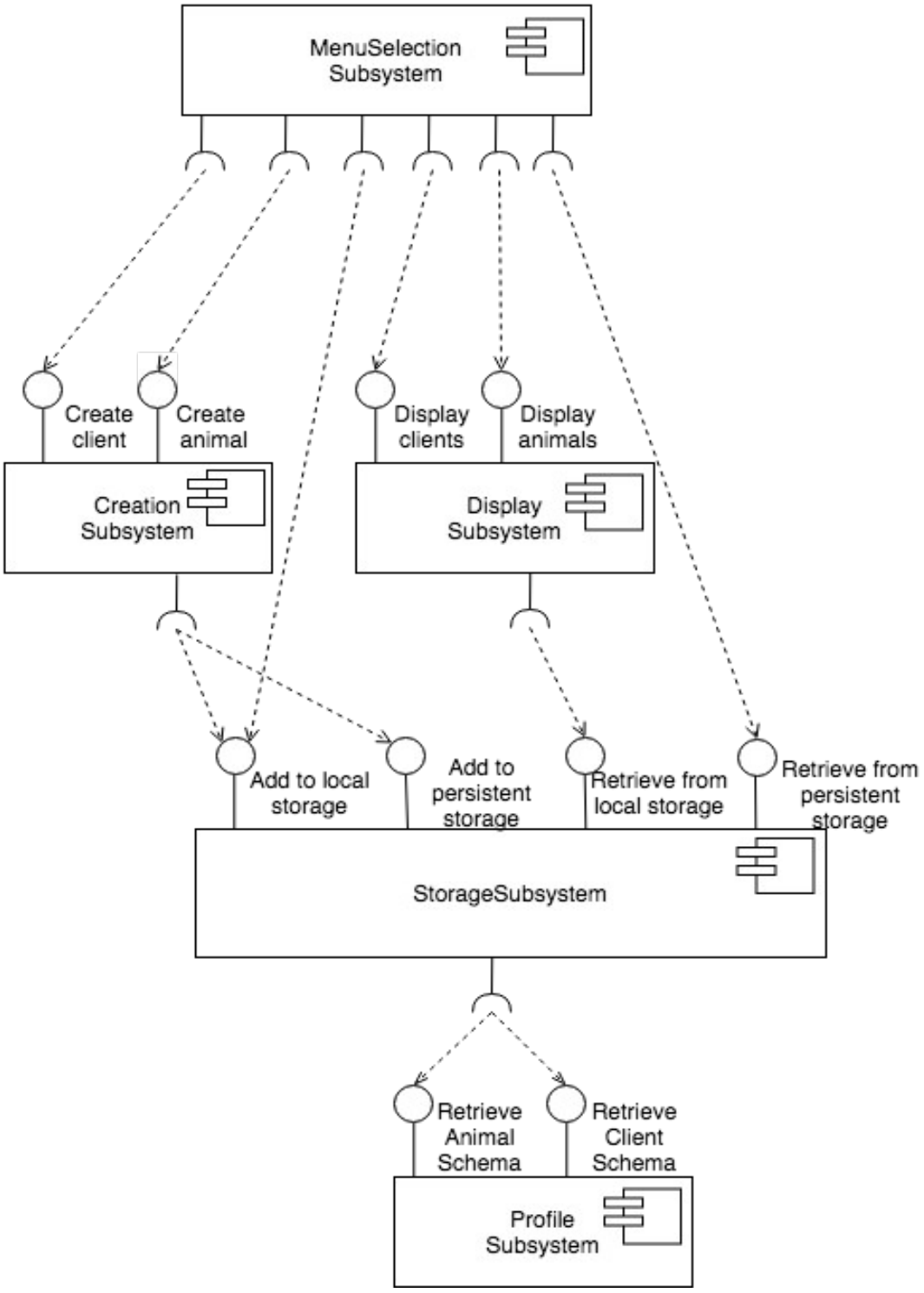

Figure 3—Deliverable 2 Subsystem Decomposition UML Component Diagram

2.2 Full Subsystem Decomposition

Taking into account the concepts of cohesion and coupling as well as design patterns, a much more effective and maintainable system was devised for moving forward. The following full subsystem decomposition details the system as it should be upon completion with all of it's requisite components present. Figure 4—Full System Decomposition UML Class Diagram details the subsystems and classes that make up this new and improved system design. Accompanying this figure are a series of tables with brief descriptions of each subsystem and class involved. These descriptions are also far more involved and detailed than those of the deliverable 2 design which is indicative of the refinement of the system design.

Following that, Figure 5—Full System Subsystem Decomposition UML Package Diagram gives an overview of how each of the subsystems and classes interact with each other. This diagram describes the level of coupling and cohesion present in the system between and within the subsystems respectively. It can be observed from the above-noted figure that this new refined subsystem decomposition exhibits relatively weak coupling and much stronger cohesion which is more ideal. It is also much easier to understand and follow each path through the system.

Finally, Figure 6—Full System Subsystem Decomposition UML Component Diagram details how each subsystem interacts with other subsystems using a UML component diagram with ball-and-socket notation. This figure serves to show, in a more simplified and easier-to-parse format, the ways that the subsystems interact with one-another. This refined system has a flow which is much cleaner and easier to understand where the subsystems form a structure akin to layers of abstraction.

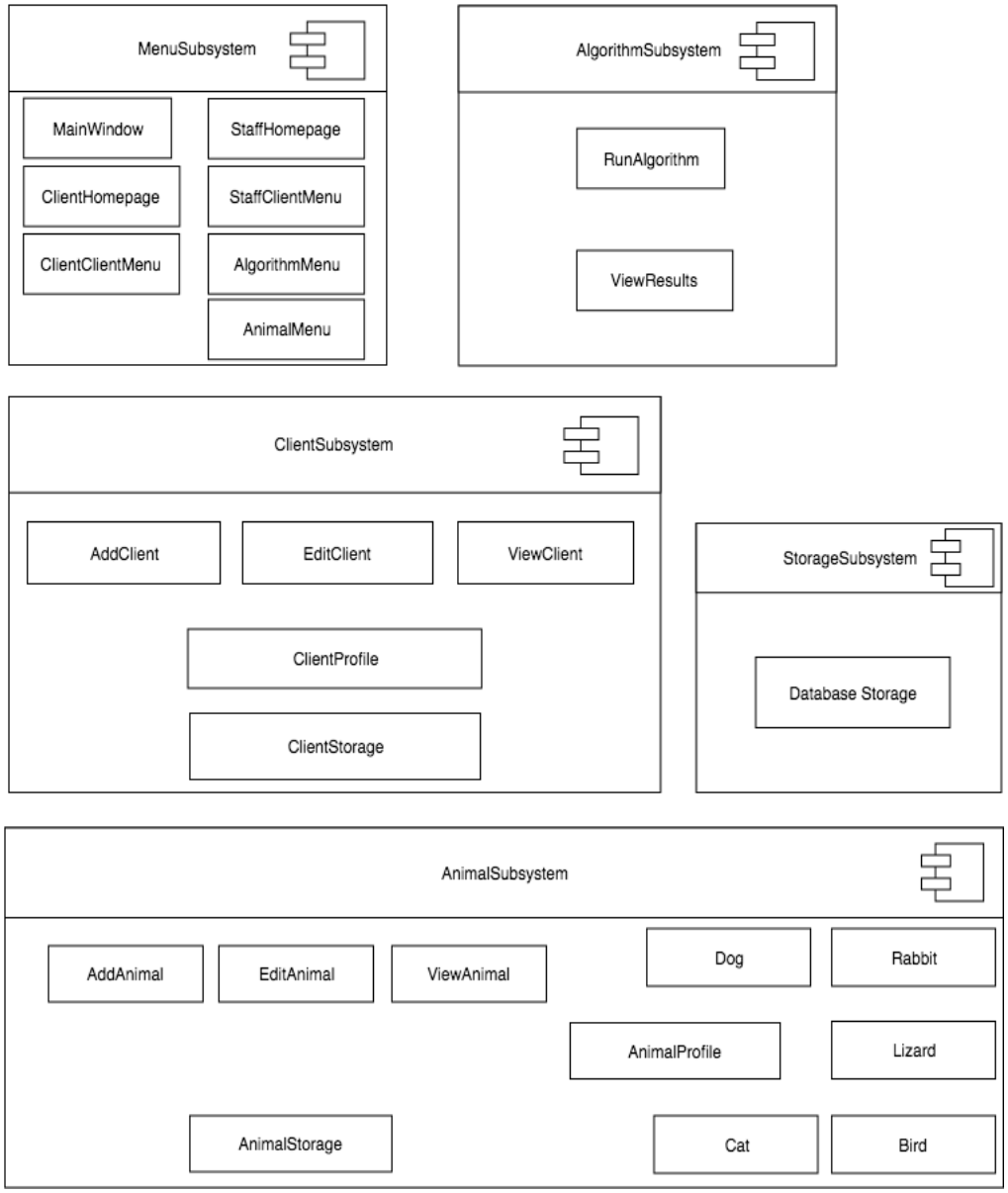

Figure 4—Full System Decomposition UML Class Diagram

Table 6—Full System MenuSubsystem

MenuSubsystem
• The MenuSelectionSubsystem is responsible for allowing the user to choose how they want to interact with the system. They choose which menu to access to perform certain actions.
MainWindow
• The MainWindow allows the user to decide if they want to log in to StaffHomepage or ClientHomepage. It checks for a valid email if the user is trying to log in as an existing Client.
StaffHomepage
• The StaffHomepage provides access to several menus which further encapsulate the functionality that staff members can execute.
StaffClientMenu
• The StaffClientMenu allows the staff member access to viewing and adding ClientProfiles.
AlgorithmMenu
• The AlgorithmMenu allows the staff member access to running the algorithm and viewing the results output by the algorithm.
AnimalMenu
• The AnimalMenu allows the staff member access to viewing and adding AnimalProfiles.
ClientHomepage
• The ClientHomepage provides access to a menu which further encapsulates the functionality that Clients can execute. It also provides access to the ViewAnimal class, allowing clients to view the AnimalProfiles of the animals in the shelter.
ClientClientMenu
• The ClientClientMenu allows a client access to viewing and editing their own ClientProfile.

Table 7—Full System ClientSubsystem

ClientSubsystem
• The ClientSubsystem is responsible for handling everything relating to the Client type of user of the cuACS system. It is specifically responsible for loading ClientProfiles from persistent storage, handling adding clients and allowing those clients to edit their profiles, and viewing the clients.
AddClient
• The AddClient class provides the functionality for adding a new ClientProfile to the system.
ViewClient
• The ViewClient class provides functionality for displaying a ClientProfile's information.
EditClient
• The EditClient class provides the facilities for editing the information contained in a ClientProfile.
ClientProfile
• The ClientProfile class represents the locally stored (non-persistent) client profiles which are acted upon by AddClient, ViewClient, and EditClient and are loaded into the system on startup.
ClientStorage
• The ClientStorage class is responsible for communicating between the locally stored ClientProfiles loaded at run-time and the persistent storage where all client profiles are saved to and loaded from.

Table 8—Full System AnimalSubsystem

AnimalSubsystem
• The AnimalSubsystem is responsible for handling everything relating to the Animals of the cuACS system. It is specifically responsible for loading AnimalProfiles from persistent storage, handling adding animals and editing their profiles, and viewing the animals.
AddAnimal
• The AddAnimal class provides the functionality for adding a new AnimalProfile to the system.

ViewAnimal
• The ViewAnimal class provides functionality for displaying an AnimalProfile's information.
EditAnimal
• The EditAnimal class provides the facilities for editing the information contained in an AnimalProfile.
AnimalProfile
• The AnimalProfile class represents the locally stored (non-persistent) animal profiles which are acted upon by AddAnimal, ViewAnimal, and EditAnimal and are loaded into the system on startup. It includes five sub-classes for handling profile information for each type of animal.
Rabbit
• The Rabbit class represents the unique animal profile for rabbits containing rabbit-specific attributes in addition to basic animal ones.
Lizard
• The Lizard class represents the unique animal profile for lizards containing lizard-specific attributes in addition to basic animal ones.
Bird
• The Bird class represents the unique animal profile for birds containing bird-specific attributes in addition to basic animal ones.
Cat
• The Cat class represents the unique animal profile for cats containing cat-specific attributes in addition to basic animal ones.
Dog
• The Dog class represents the unique animal profile for dogs containing dog-specific attributes in addition to basic animal ones.
AnimalStorage
• The AnimalStorage class is responsible for communicating between the locally stored animal profiles loaded at run-time and the persistent storage where all animal profiles are saved to and loaded from.

Table 9—Full System StorageSubsystem

StorageSubsystem
• The StorageSubsystem is responsible for the persistent storage of all ClientProfiles and AnimalProfiles.
DatabaseStorage
• DatabaseStorage is the persistent storage for all AnimalProfiles and ClientProfiles. Used by ClientStorage and AnimalStorage at startup to create new Animal/Client objects from database data, by those same classes when storing newly added or edited profiles, and by the Algorithm when reading profiles to determine matches.

Table 10—Full System AlgorithmSubsystem

AlgorithmSubsystem
• The AlgorithmSubsystem is responsible for handling the operation of the Animal-Client matching (ACM) algorithm.
RunAlgorithm
• The RunAlgorithm class controls the running of the ACM algorithm. It is used to launch an instance of the matching algorithm to compute the optimal matches between Clients and Animals.
ViewResults
• The ViewResults class displays results of the most recent run of the ACM algorithm.

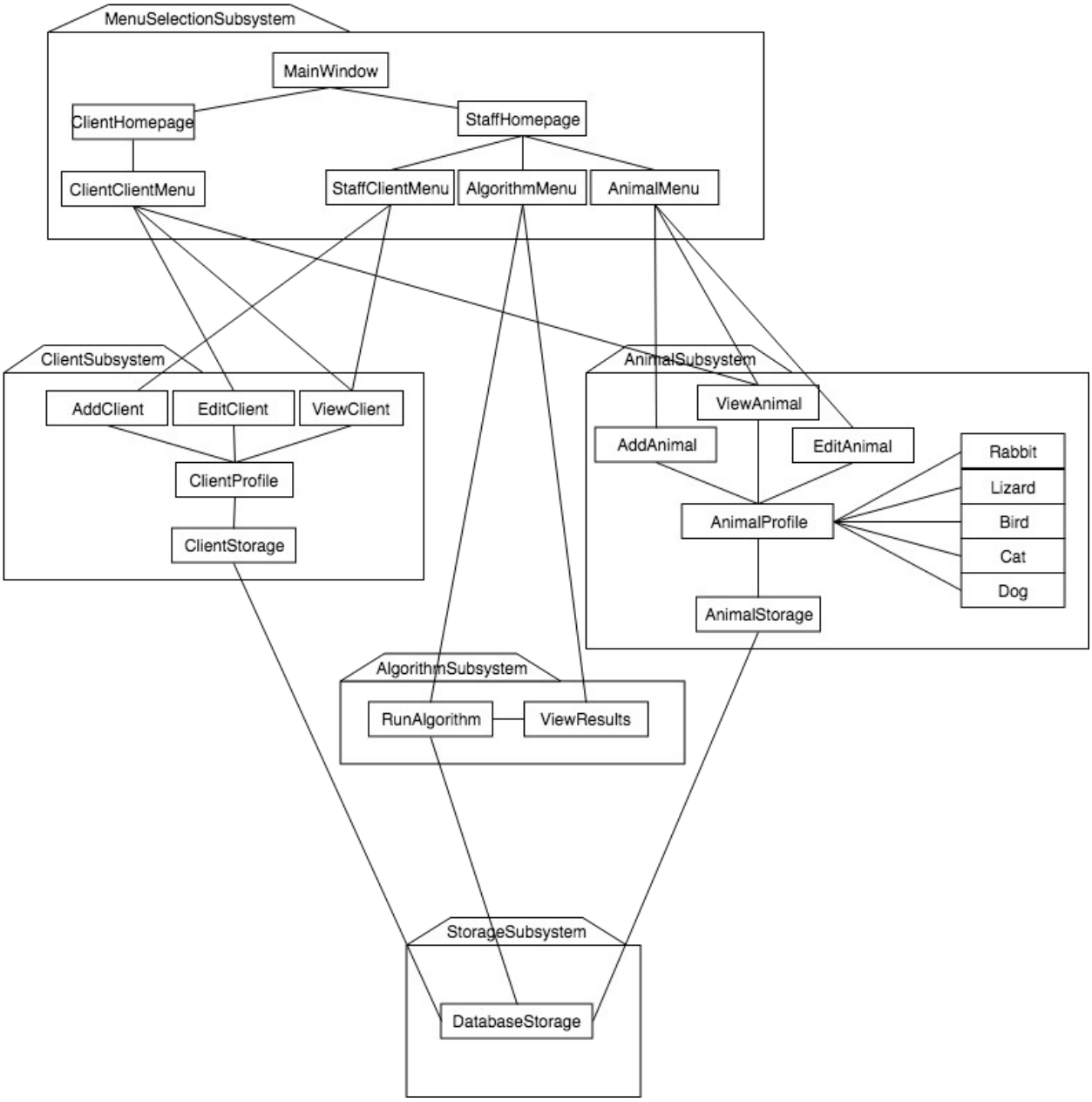

Figure 5—Full System Subsystem Decomposition UML Package Diagram

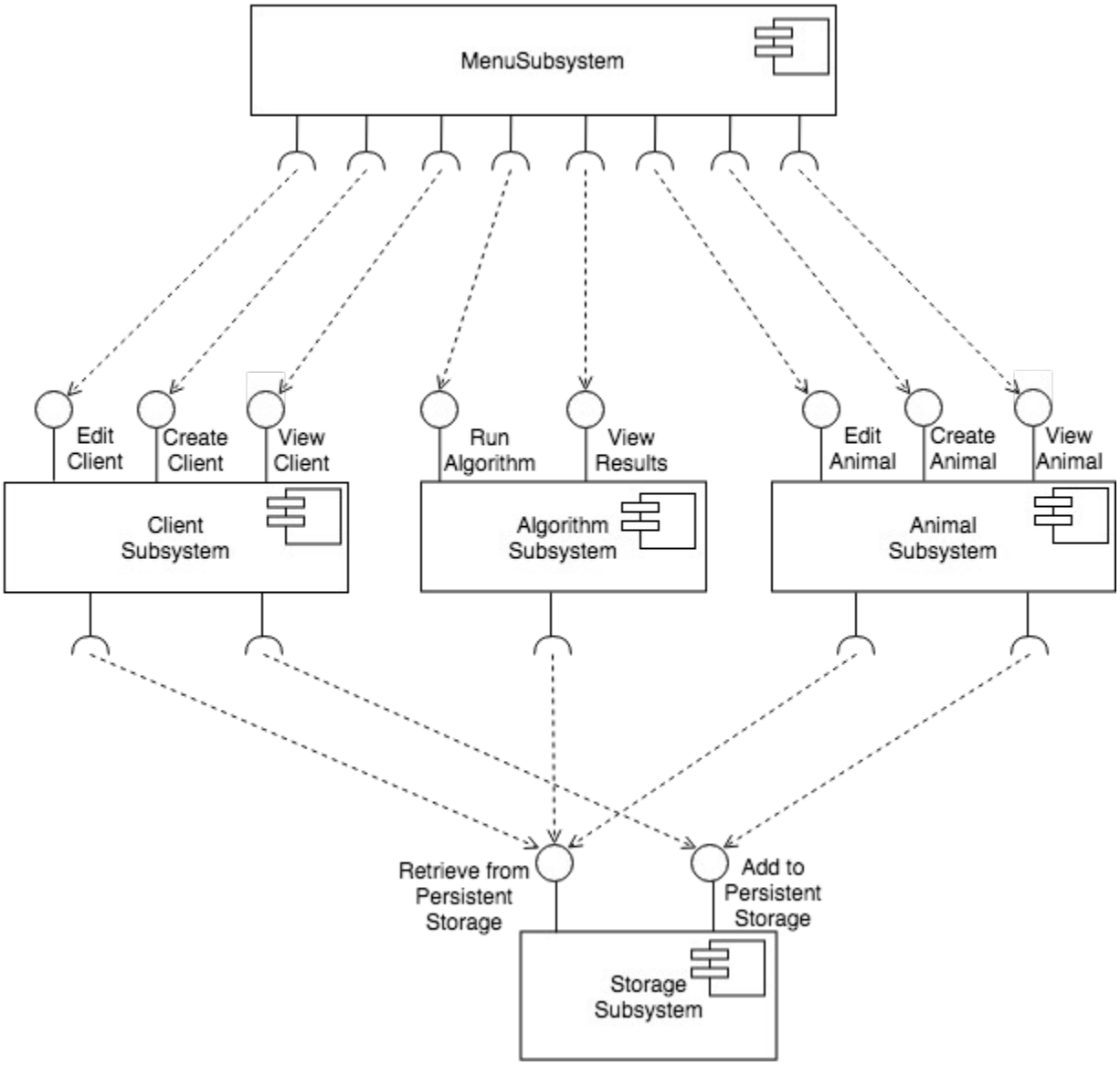

Figure 6—Full System Subsystem Decomposition UML Component Diagram

2.3 Design Evolution

2.3.1 Deliverable 2 Explanation

For the initial subsystem decomposition for the deliverable 2 code it seemed logical to organize the subsystems based on their functionality. Regardless of whether we are working with Client or Animal profiles, we would always be using a similar approach for creating, viewing and adding them to storage. For this reason, creating subsystems dedicated to Adding, Viewing and Storing these profiles seemed to make the most sense.

Furthermore, having the storage subsystem dedicated entirely to the local and persistent storage of these profiles appeared to be the obvious solution considering they all offered some kind of storage functionality.

2.3.2 Deliverable 2 Limitations

While this approach seemed to work initially, when mapping the interactions between subsystems it became apparent that there were some significant drawbacks in decomposing the system in this manner.

Firstly, even though the classes in each subsystem performed the same task, the classes had absolutely no relation to each other. Therefore, we had AddAnimal and AddClient in the CreationSubsystem even though the adding an Animal had absolutely no relation to adding a Client and vice versa. This resulted in our subsystems having very low cohesion with the majority of subsystems not having any functional relations between classes.

Secondly, because we had such low cohesion the relations between our classes ended up happening across subsystems. This led to situations where the execution of even the simplest task would make use of many subsystems. This meant that as a result of the low cohesion within our subsystems, our system also had exceptionally high coupling.

2.3.3 Goals for Improvement of the Entire System

Considering the deliverable 2 subsystem decomposition had some major problems with regards to cohesion and coupling, our main goals for the subsystem decomposition for the final product was to create subsystems that perform similar tasks in such a way where the objects would be

functionally related. This way the system could operate with minimal between-subsystem interaction.

2.3.4 Explanation/Evolution of Design Choices

When overlooking the deliverable 2 subsystem decomposition we noticed that a significant cause of low cohesion within our subsystems was that Animal-specific classes and Client-specific classes never interacted. Utilizing this observation we were able to conclude a better approach would involve organizing the subsystems based on the type of Profile with which they would be interacting.

For this reason, we decided to create an AnimalSystem that included Animal, AddAnimal, ViewAnimal and AnimalStorage. Similarly, we also created a ClientSubsystem that included Client, AddClient, ViewClient and ClientStorage. Both Add and View objects made significant use of their respective Storage objects and by taking this approach we are able to keep all these relations contained to only one subsystem. This severely limited the amount of relations that occurred between subsystems which resulted in much looser coupling and much greater cohesion within these specific subsystems.

We also created a PersistentStorageSubsystem that is separate from the AnimalStorage and ClientStorage objects. The majority of our system makes use of the AnimalStorage and ClientStorage objects while the DatabaseStorage object is rarely used. Therefore, creating a separate StorageSubsystem that contained exclusively these three Storage objects resulted in the majority of our system having to interact with the StorageSubsystem at some point in order to accomplish even the simplest of tasks. By separating PersistentStorage from AnimalStorage and ClientStorage, interaction between other systems and the Storage-specific subsystem only ever occurs when a Profile is created, edited or at program startup to load all of the database information.

Finally, with regards to the MainMenuSubsystem, we expanded the objects into a variety of subcategories based on the services they will offer. This significantly increases code maintainability because we can make a modification to any given subsystem and have it only affect a single subsection of the MainMenuSubsystem. This allows us to significantly reduce coupling between subsystems.

3 Design Strategies

The design strategies used in designing the system in order to make it maintainable and efficient can be split up into two sections. First, the design strategies behind the persistent storage mechanism of cuACS, and also the overall design strategies governing the structure and layout of the system in terms of how the subsystems interact with each other and the end-users.

This section will begin with detailing the persistent storage strategies employed by the system and will then discuss the overall design patterns used in the design of the system.

3.1 Persistent Storage

3.1.1 Explanation of the Operation and Design of Persistent Storage in cuACS

The persistent data structure used in the cuACS system is a sqlite3 database. Using a database connected to QtCreator allows the program to efficiently and swiftly save data without the hassle of having to parse a plain-text file. Using a database also significantly increases the scalability of the program, allowing for many other animals and clients to be stored. When the program is loaded, a check for a database is performed, searching the operator's host desktop directory for an existing database to load from. If a database does not exist, one is created and placed on the desktop, with the name *"cuACS.db"*. Providing the database on the desktop allows for simple access, if needed, for the operator or in-house IT staff; it also provides the operator with assurance that a database has been created for them, in a spot that they are familiar with. After the database is created, 6 tables are created– clientStorage, dogStorage, catStorage, lizardStorage, birdStorage and rabbitStorage, with corresponding schemas. (*See section 3.1.2 ER Diagram and Table Schema to view each table's associated schema.)* A check for these 6 tables is also performed if an existing database is present, with the correct schema. After the database and tables have been established, the tables are populated with clients (25) and animals (27).

Each client/animal profile is represented in the database as a row, in the appropriate table – i.e. dogs are stored in the table titled 'dogStorage', clients in clientStorage, etc. The columns for each of the tables represent a client/animal's attributes, such as for animals: name, breed, history, hypoallergenic, etc.; and for clients: name, address, wantsBird, has children, etc. Each

of the attributes are stored in the database tables as a sqlite INTEGER type or sqlite TEXT type. For the case of an animal's breed, the associated vector of breeds is converted into the a string in the format: [($breed_1$)($breed_2$)(...)($breed_n$)], for the n breeds which are associated with an animal, this newly formed string is saved as a sqlite TEXT type. Booleans are stored as integers.

As the tables are read, and the data is loaded into the cuACS system, each table is read, row by row. When a row is being read, an object is created, and the corresponding column value is assigned to the objects attribute. For example, when lizardStorage is being read, the lizard object's "space" attribute – denoting how much space the lizard needs to survive – is set by the value stored in the database for the column titled "space." This is done in the exact same way for all animals, and all clients. When a row is finished being read, the object is complete. Next, the newly created object is pushed into the internal Animal or Client list. Using an internal list speeds up processing time and prevents having to query the database for every time a profile is viewed.

When adding a new client or animal to the persistent storage, all of the appropriate text-fields, radio-buttons and check boxes are read, and an animal or client object is created. When created, the object is added to the appropriate internal list (client or storage), as well as sent to the database class to be added to the cuACS database. Here, the object is deconstructed into its attributes for the object, and are stored into the database, by column, based on the profile they are (i.e. client, dog, bird, etc.). A unique identifying number is created for each client or animal in the system, called *idNum*. The idNum is used for distinguishing animals and clients from each other – this is used as the primary key in the database. A primary key is the key that is unique for each record (row) in the database. Assigning each animal and client a unique idNum prevents the data from being over written when a new animal or client is added to the appropriate table. It also prevents duplication of a profile or other data as every primary key in a table must be unique.

Adding a new client or animal to the persistent storage creates the appropriate object, (with a new, unique idNum) and send it's to the databaseStorage class, to add it to the database. Here, we deconstruct the object into its attributes, storing each of them inside the table with columns representing the attributes. When it comes to editing a profile for a client or animal, we refer to the profile by its unique idNum identifier, to ensure that we only edit the profile desired and not any others – this is ensured because the idNum is used as the primary key, and cannot be replicated, for an entry into the database. An idNum is composed of 2 parts, the profile code, followed by an identifying number. Profile codes are as follows: Client = 1, Dog = 2, Cat = 3, Bird = 4, Lizard = 5, Rabbit = 6, and is the first digit for the idNum. The identifying number is

calculated by determining how many clients or objects currently exist in the system, and for a newly created profile is incremented by one. By default, the identifying numbers start at 100. As an example, the first animal in the system would have idNum = *profilecode+*100, adding another animal would create the idNum *profileCode*+101, and so on. An example of the clients' idNum is as follows: if there are 12 clients in the system and another one is being added, the last client idNum would have been 1112, the resulting idNum of the new client will be 1113. Creating and utilizing an idNum in this way allows assurance that no 2 keys are the same, maintaining uniqueness between all animals and clients.

Having these unique idNum keys, allows for easy edits of clients or animals to occur. Simply querying the database table for the idNum of an edited animal or client, allows the row to be easily updated. In order to ensure that all user changes are updated, the all profile data is pushed to the database table and stored for the given idNum.

3.1.2 ER Diagram and Table Schema

In database design, entity-relationship (ER) diagrams are commonly used as a graphical description of the structure and restrictions of a database. Below is the ER diagram for the cuACS database. Since there are very many attributes for each client and animal, the relevant non-uniquely-identifying attributes are listed in tables below this diagram.

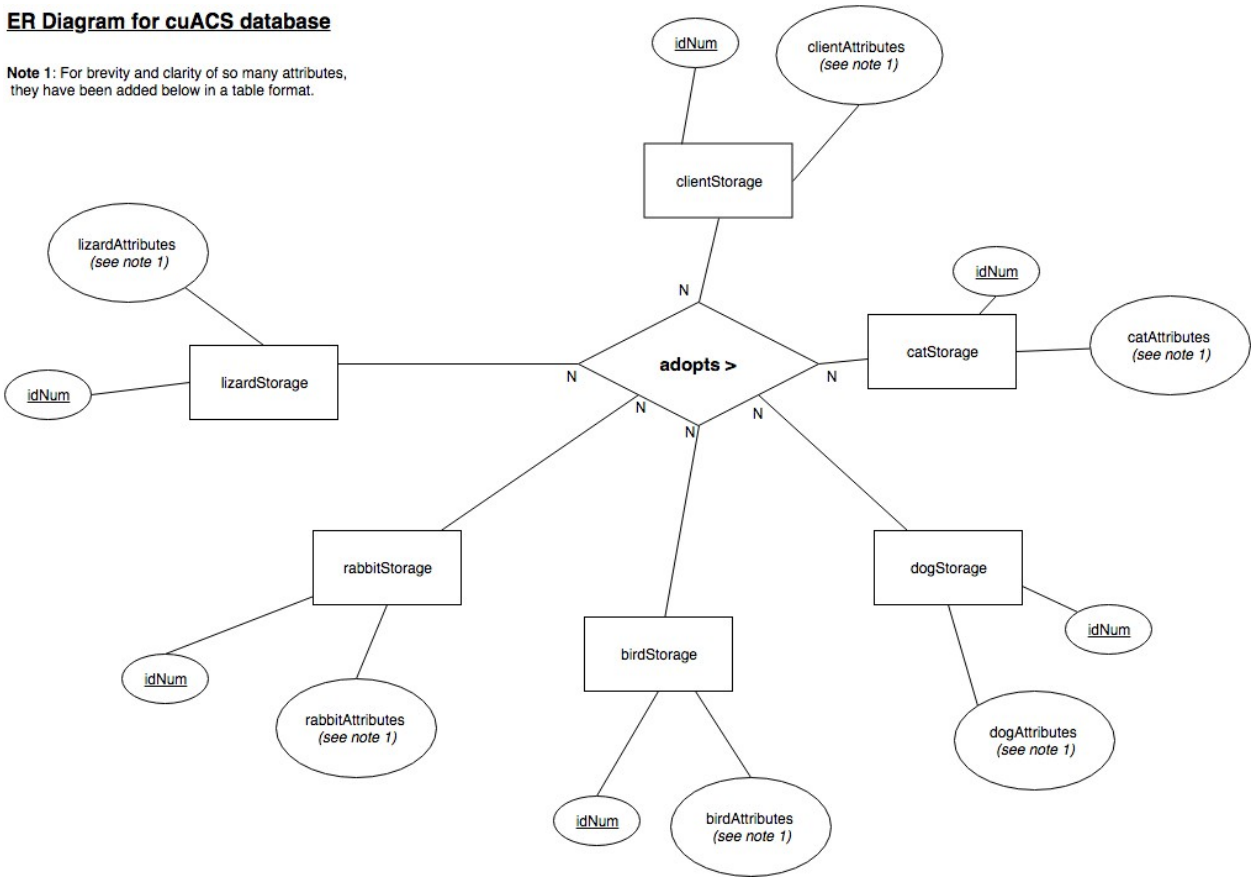

Figure 7—cuACS Database ER Diagram

Table 11: cuACS Database Client Attributes

Client Attributes			
idNum	protector	followCommandsCat	wantsRabbit
fName	energy	doesntShed	hasRabbitAllergies
lName	fearful	wantsBird	rabbitBreeds
address	affection	hasBirdAllergies	rabbitAge
phone	messy	birdBreeds	rabbitSize
email	wantsDog	birdAge	rabbitGender
city	hasDogAllergies	birdSize	isSocialRabbit
prov	dogBreeds	birdGender	needsGrooming
dwelling	dogAge	isQuietBird	rabbitColour
location	dogSize	isSocialBird	dogFur
workSchedule	dogGender	birdColour	catFur
activity	followsCommandsDog	wantsLizard	birdFur
hasChildren	houseTrained	hasLizardAllergies	lizardFur
hasAnimals	wantsCat	lizardBreeds	rabbitFur
travels	hasCatAllergies	lizardAge	quietness
children	catBreeds	lizardSize	age
goodWAnimals	catAge	lizardGender	
strangers	catSize	easyToFeed	
crowds	catGender	simpleLiving	
noises	isCurious	lizardColour	

Table 12: cuACS Database Rabbit Attributes

Rabbit Attributes	
idNum	energy
breed	fearful
name	affection
size	messy
age	nocturnal
gender	hypo
fur	lifeStyle
travels	history
children	pattern
goodWithAnimals	colour
strangers	grooming
crowds	attention
noises	filepath
protector	

Table 13: cuACS Database Dog Attributes

Dog Attributes	
idNum	energy
breed	fearful
name	affection
size	messy
age	nocturnal
gender	hypo
fur	lifeStyle
travels	history
children	barks
goodWithAnimals	training
strangers	bathroomTrained
crowds	goodBoy
noises	filepath
protector	

Table 14: cuACS Database Cat Attributes

Cat Attributes	
idNum	energy
breed	fearful
name	affection
size	messy
age	nocturnal
gender	hypo
fur	lifeStyle
travels	history
children	curiosity
goodWithAnimals	trained
strangers	shedding
crowds	filepath
noises	
protector	

Table 15: cuACS Database Bird Attributes

Bird Attributes	
idNum	energy
breed	fearful
name	affection
size	messy
age	nocturnal
gender	hypo
fur	lifeStyle
travels	history
children	loud
goodWithAnimals	social
strangers	colour
crowds	filepath
noises	
protector	

Table 16: cuACS Database Lizard Attributes

Lizard Attributes	
idNum	energy
breed	fearful
name	affection
size	messy
age	nocturnal
gender	hypo
fur	lifeStyle
travels	history
children	diet
goodWithAnimals	colour
strangers	feed
crowds	space
noises	light
protector	filepath

3.2 Design Patterns

Design patterns provide a partial solution to common problems. They offer a robust and modifiable solution when inheritance and delegation are utilized within a small number of classes. The design pattern descriptions are based on those provided in *Object-Oriented Software Engineering*, *3rd Edition* (Bruegge B., & Dutoit A.).

3.2.1 Adapter

The intentions of the Adapter design pattern is to convert the interface of a legacy class into a different interface – one that is expected by the client. This is done so that the client and legacy class can work together without changes. It can be helpful to compare the Adapter design pattern to an electrical plug. We can not use a Canadian plug in Europe without some sort of intermediary (adapter). This design pattern includes creating that intermediary abstraction that maps the “old” class to the new interface.

cuACS does not utilize the Adapter design pattern. This is because no legacy code exists that is not easily modifiable. The system is still rather early in its implementation – the back-end is not

something that requires an intermediary when dealing with the rest of the components (at this time). We employed Bridge because it makes things work before design, while adapter makes them work after.

3.2.2 Abstract Factory

The Abstract Factory design pattern aims to shield the client from varying platforms that provide different implementation for the same concepts. A platform is represented as a set of abstract products, where each abstract product represents a concept supported by every platform. An AbstractFactory class declares operations for creating each individual product, in this case clients and animals (and then different types of animals). A specific platform is then realized by a ConcreteFactory, and a set of concrete products (the different types of animals). The ConcreteFactory is dependent only on its related concrete products. The client dependent only on the AbstractProduct and AbstractFactory classes, hence making it easier to substitute platforms.

cuACS uses the Abstract Factory design pattern when creating a profile for clients and animals. Here, the AbstractFactory class could be presented by StaffHomepage (where the user then navigates to either the StaffClientMenu or AnimalMenu to create a profile), where one ConcreteFactory class would be AddClient and the other AddAnimal. The AbstractProduct classes would be the AnimalProfile and ClientProfile, where the concrete Products would be the types of animals, in this case Dog, Cat, Rabbit and Lizard.

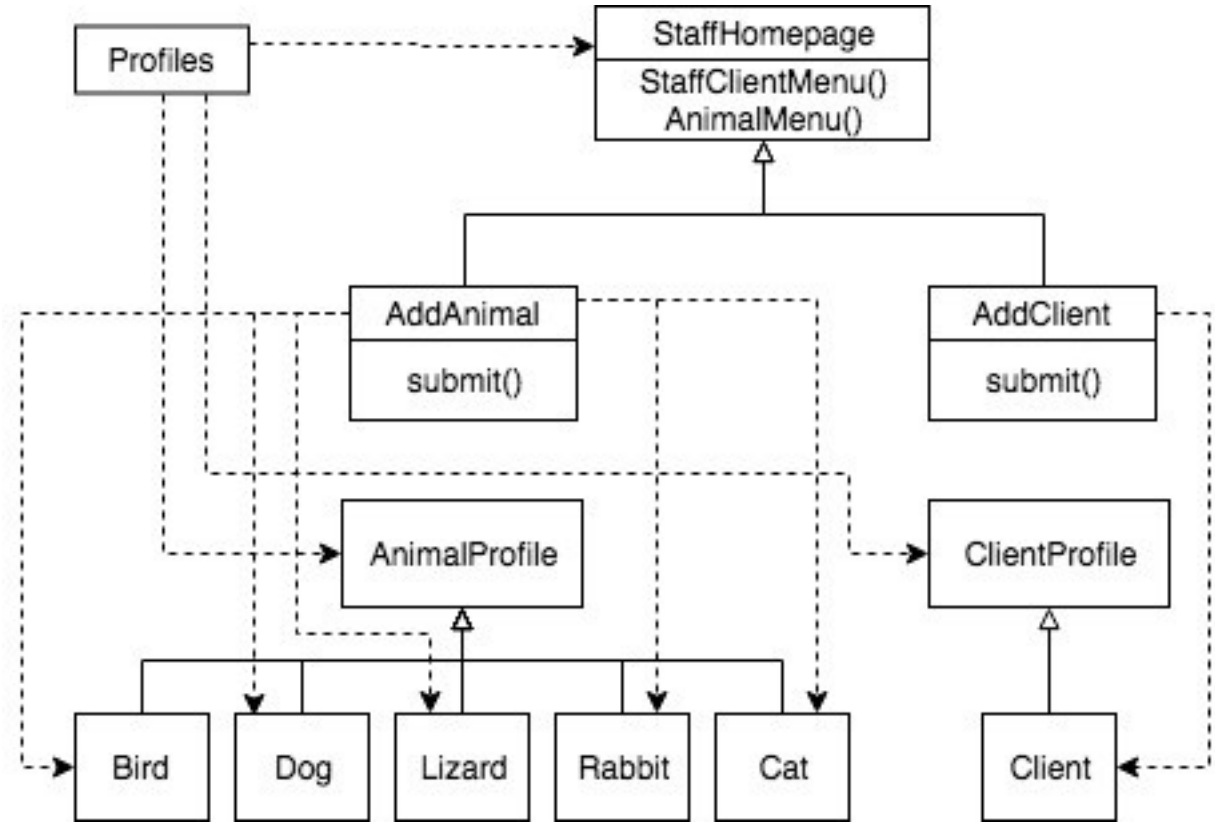

Figure 8—Use of the Abstract Factory Design Pattern

3.2.3 Bridge

The Bridge design pattern aims to decouple an interface from an implementation so that implementations can be substituted at different times (runtime, for an example). Utilizing the Bridge design pattern increases loose coupling between the class abstraction and its implementation.

There are four main elements to this design pattern:

1. Abstraction – a class that defines the interface visible to the client. It contains a reference to the implementer.
2. Implementor – an abstract class that defines the lower-level methods for Abstraction. It is not required to correspond directly to Abstraction and can largely vary.
3. Refined Abstraction – exists to hide the more detailed elements from implementors.
4. Concrete Implementors – Implements the implementor (above) by providing actual, concrete implementation.

cuACS utilized the Bridge design pattern, particularly for the storage and utilization of different profiles. The Bridge pattern decouples an abstraction from its implementation. This means that each can vary. Different implementations exist when new animals and clients are created and then stored, therefore it is necessary to employ the Bridge design pattern.

3.2.4 Composite

The Composite design pattern aims to compose all objects into tree-like structures to represent hierarchies. These hierarchies are of variable width and depth – that is, they can contain "directories" that can be made up of other "directories".

The Component interface specifies the services that are shared among Leaf and Composite. A Composite possesses an aggregation association with Components and implements each service by iterating over each Component contained within. The Leaf services perform the actual request. The result of this pattern allows the Client to use the same code when dealing with both Leaves and Composites. Leaf-specific behaviour can also be modified without changing the hierarchy, and new classes of Leaves can be added without changing as well.

The Composite design pattern is utilized within the cuACS system. This is particularly prevalent within the UI design (QT framework and design elements). An example of this being when an user selects a certain animal, attributes associated with that animal are displayed while others are not, or greyed-out.

3.2.5 Command

The Command design pattern encapsulates a request as an object. This means they can be executed, undone, or queued independently of the request. A Command abstract class declares the interface supported by all Concrete Commands. These ConcreteCommands encapsulate a service to be applied to a Receiver. The Client creates these ConcreteCommands and binds them to specified Receivers, while the Invoker executes or undoes a command.

This results in the Receiver and the algorithm being decoupled. The Command design pattern allows requests to be encapsulated as objects, so this means the clients are "parametrized" by different requests. The control flow of adding and viewing profiles within the cuACS system is an utilization of the Command Design pattern. An animal (or client) profile is created by

identifying the physical and non-physical attributes. These attributes (or traits) make up the animal profile; the profile is an encapsulation of the attributes.

3.2.6 Observer

The Observer design pattern aims to maintain consistency across the states of one Publisher and numerous Subscribers. That is, it defines a one-to-many dependency between different objects. When an object changes state, its dependents are updated automatically.

Since QT combines the view and entity, cuACS does not currently use the Observer Design Pattern. The operations are completed within the same code that the view is updated in. Due to this, the Observer design pattern is not necessary.

3.2.7 Proxy

The intentions of the Proxy Design pattern are to improve the performance and/or security of a system by delaying the expensive computations, using memory only as needed and by checking access before loading an object into memory. The ProxyObject class acts on behalf of a RealObject class. Both classes implement the same interface, while the ProxyObject stores a subset of the attributes of the RealObject. The ProxyObject deals with only certain requests to completion. Other requests are delegated to RealObject. Post-delegation, the RealObject is created, and then loaded in memory.

cuACS does not utilize the Proxy design pattern at this time. There may be future opportunities for its implementation. The pattern avoids duplication of large-sized objects, so especially as the UI becomes more advanced and graphical (in terms of images and other multimedia forms, perhaps) the proxy pattern may become very advantageous.

3.2.8 Strategy

The Strategy design pattern aims to decouple a policy-deciding class from a set of mechanisms in order to ensure different mechanisms can changed transparently from a client. A Client accesses services provided by a Context. The Context services are realized using a mechanism (one of several) as decided by the Policy Object. An abstract Strategy class describes the common interface to all mechanisms that Context can use. The Policy class then creates a ConcreteStrategy object, and then configures the Context to use it.

cuACS does not employ the Strategy design pattern. This is due to the system only utilizing the main algorithm – the clients are matched to the animals using only one algorithm, therefore there are no decisions to make regarding which to use.

4 Index of Tables

Table 1—Deliverable 2 MenuSelectionSubsystem....6
Table 2—Deliverable 2 CreationSubsystem....6
Table 3—Deliverable 2 DisplaySubsystem....7
Table 4—Deliverable 2 StorageSubsystem....7
Table 5—Deliverable 2 ProfileSubsystem....8
Table 6—Full System MenuSubsystem....12
Table 7—Full System ClientSubsystem....13
Table 8—Full System AnimalSubsystem....13
Table 9—Full System StorageSubsystem....15
Table 10—Full System AlgorithmSubsystem....15
Table 11: cuACS Database Client Attributes....24
Table 12: cuACS Database Rabbit Attributes....25
Table 13: cuACS Database Dog Attributes....25
Table 14: cuACS Database Cat Attributes....25
Table 15: cuACS Database Bird Attributes....26
Table 16: cuACS Database Lizard Attributes....26

5 Index of Figures

Figure 1—Deliverable 2 Subsystem Decomposition UML Class Diagram....5
Figure 2—Deliverable 2 Subsystem Decomposition UML Package Diagram....8
Figure 3—Deliverable 2 Subsystem Decomposition UML Component Diagram....9
Figure 4—Full System Decomposition UML Class Diagram....11
Figure 5—Full System Subsystem Decomposition UML Package Diagram....16
Figure 6—Full System Subsystem Decomposition UML Component Diagram....17
Figure 7—cuACS Database ER Diagram....23
Figure 8—Use of the Abstract Factory Design Pattern....28

6 Sources

Bruegge, B., & Dutoit, A. H. (2014). *Object-oriented software engineering: Using UML, Patterns, and Java*. Harlow (UK): Pearson.