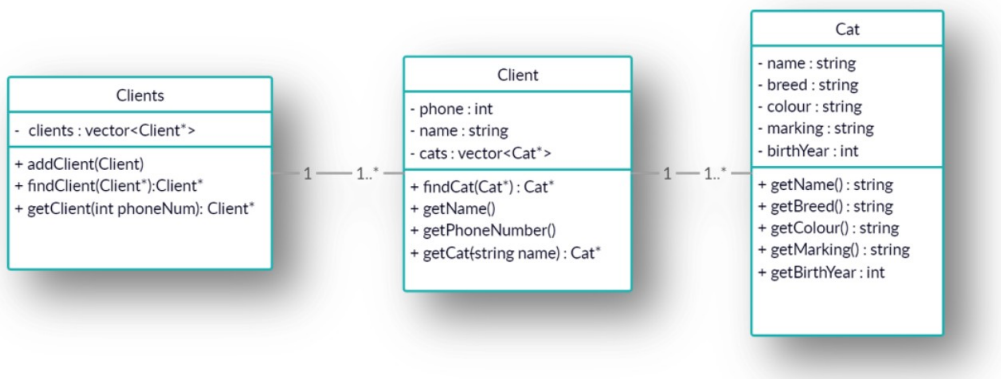
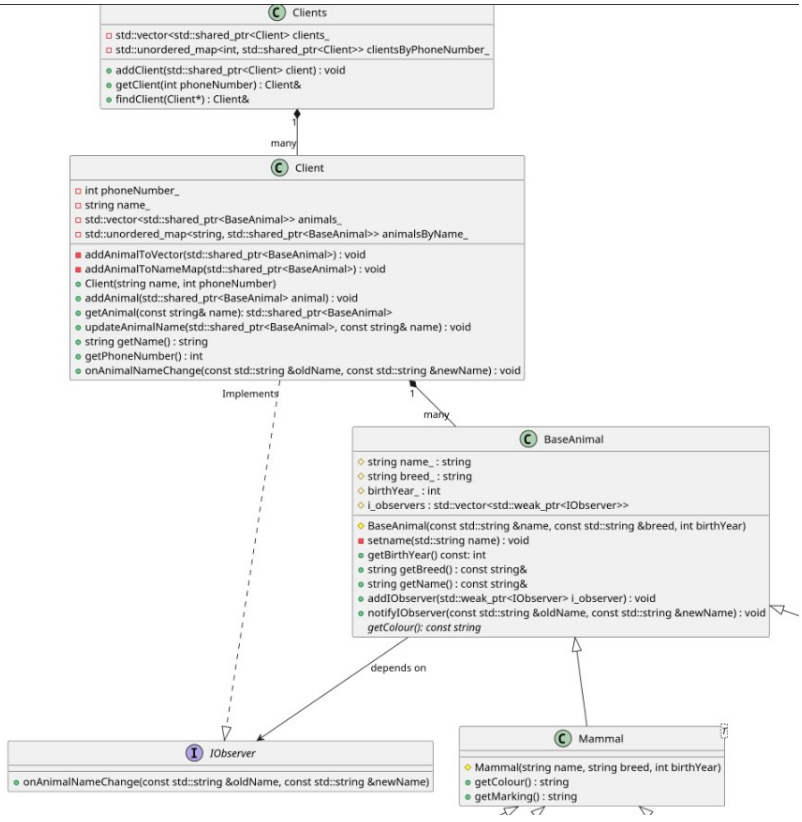
Assignment 2
Revising Vet Application
John Holloway
CPSC-2720 Fall 2024

The original client-management system was a simple design that met the need when the clinic was only attending to cats, however it was not designed to handle dogs let along other times of animals.



In response, the design of the system was overhauled to follow SOLID design principles, as well as to future proof the system for further enhancements should they arrive.



The Clients class has been left largely untouched, but it now uses smart pointers and maps. This change was made primarily for the author's enjoyment and practice with modern C++ techniques[1].

The Client class, however, has undergone significant changes. Previously, it was responsible for adding animals to its collections and keeping track of animals by their names using a map. This introduced a critical issue: if an animal's name changed, the key in the map would become inaccurate, leading to inconsistencies. Consequently, the Client class became tightly coupled to the BaseAnimal class, violating the SOLID design principles.

To address this, the Observer Design Pattern was introduced. With this pattern, BaseAnimal objects notify Client objects whenever their name changes. An interface, IObserver, was created for this purpose, which the Client class implements. The BaseAnimal class now maintains a collection of IObserver

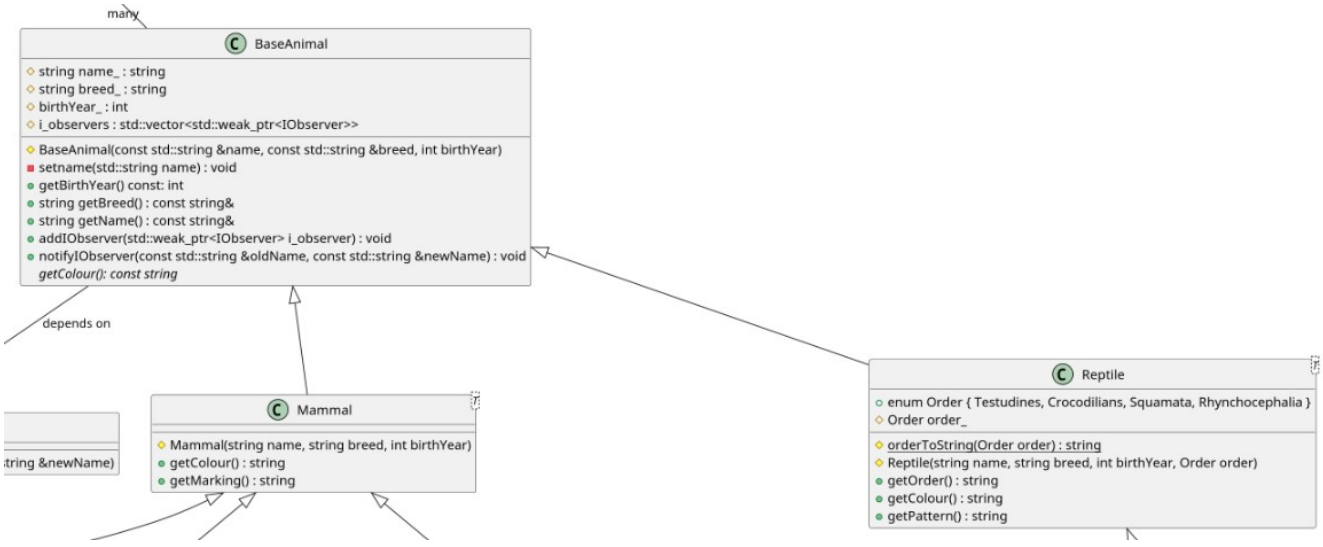
1 There is an incredible story about the author learning the differences in C++ regarding references, pointers, unique and shared pointers, and creating a hazard involving dangling references to a unique pointer!

objects (in this case, only the Client class implements IObserver, but the design allows for future extensibility). Whenever an animal's name changes, BaseAnimal notifies all registered observers, passing the relevant information (e.g., old and new names). The Client class, upon receiving this notification, updates its map to ensure consistency.

This design eliminates the tight coupling between Client and BaseAnimal. Instead, BaseAnimal now depends on the IObserver interface, while the concrete implementation of IObserver remains encapsulated. This adheres to the Dependency Inversion Principle (DIP), as BaseAnimal depends on an abstraction (IObserver) rather than a concrete implementation. Furthermore, it demonstrates the Interface Segregation Principle (ISP) by ensuring that IObserver contains only the specific methods required for observer functionality, without imposing unnecessary responsibilities.

The next major change to the animals being tracked by the system was to create a BaseAnimal class. It is assumed that all animals in the system will have common information, namely a name, breed and birth year. The three variables are encapsulated from all other objects except those that inherit from the BaseAnimal class. This ensure that other objects cannot read or write the private data without going through the appropriate getters or setters. From this base class more and more methods and fields are added to the object through inheritance.

The client objects now have a collection BaseAnimals. In keeping with the Liskov Substitution Principle, all of the derived classes from BaseAnimal may be substituted in for any of the methods that accept an object of type BaseAnimal, or return a reference to an object of type BaseAnimal. Should a object receive an object of type Cat (a derivative of BaseAnimal), the object will only be able to access those methods that are common to BaseAnimal, and not the specific attributes or behavior unique to the Cat class. Additionally, you will see that the functions in BaseAnimal follow the *Single Responsibility Principle.*



The classes which extend BaseAnimal are those that extend the concept of animals themselves. Following the classification of the different types of animals by similar attributes in biology, the child classes have also adopted a similar methodology, creating different types of animals such as mammals and reptiles . A mammal *is an* animal and a reptile *is an* animal. This allows the application to add in additional types of animals that may have far different features than those found in cats in dogs.

Reptiles for example have scales that can form patterns, while similar of an idea to markings, it is a different attribute of the genus and would therefore have to be its own distinct field. By breaking down the objects in such way it makes it easier to add or remove attributes that are unique to the type of animal. If in the future the vet clinic wanted to track additional information on animals, such as information on the condition of the nipples of animals that give live birth, the appropriate fields could be added to the mammal class as such a concept would only be applicable to mammals. Adding this additional field would not effect the objects that extend from the reptile class, or any additional children added to BaseAnimal. By doing so, we follow the Open-Close Principle of SOLID, allowing the classes to be open to extension and closed to modifcation

```
template<typename Child>
class Mammal : public BaseAnimal {
protected:


public:
    Mammal(const std::string &name, const std::string &breed, int birthYear)
            : BaseAnimal(name, breed, birthYear) {};

    ~Mammal() = default;

    std::string getColour() const {
        const Child *child = static_cast<const Child *>(this);
        return Child::colourToStringImplementation(child->colour_);
    }

    std::string getMarking() const {
        const Child *child = static_cast<const Child *>(this);
        return Child::markingToStringImplementation(child->marking_);
    }
};
```

The Mammal class extends the BaseAnimal class. It does not add any additional variables to the object. However, it does include two new methods, getColour() and getMarking().

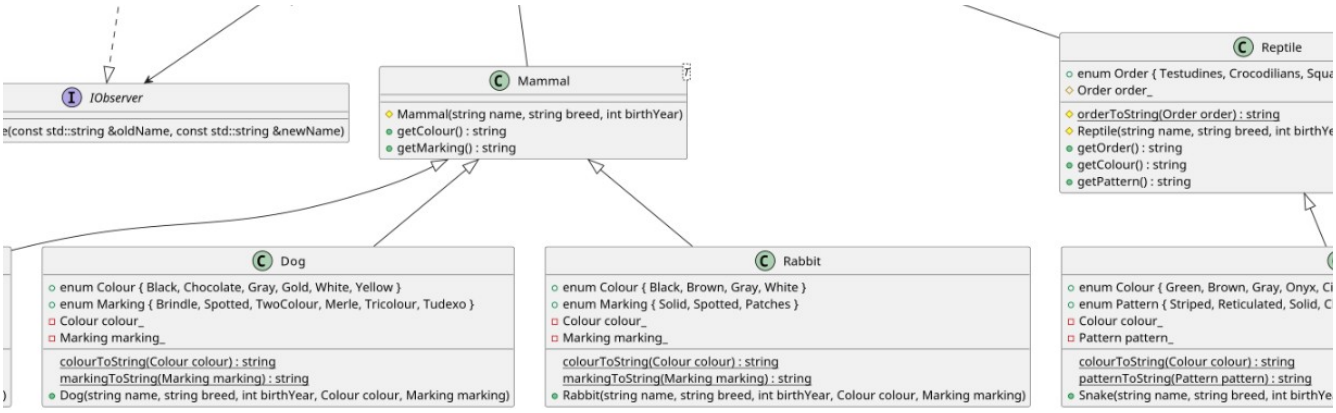
The mammal class is actually a template class that calls static methods inside of its children. By returning the value from the child's unique static function, we achieve compile time polymorphism using the *Curiously Recurring Template Pattern*[2].

Unlike traditional virtual functions, this technique requires the child class to implement the function markingToStringImplementation(). This enforcement occurs at compile time rather than runtime polymorphism, ensuring that the function is provided by the immediate child rather than allow it to be passed further down the inheritance tree. The child must have a function named *markingToStringImplementation()* that has as its parameters an object of type colour. It is up to the child to implement this following the Dependency Inversion Principle; so long as the child has something that implement to this interface the program will successfully compile

Furthermore, the return type of markingToStringImplementation() is not determined by the function declaration as it would be with a virtual function. In this example, the compile time check sees that value returned by getMarking() is of type std::string, and therefore check is if the return value from Child::markingToStringImplementation() matches this data type. Were the function doing something different, such as assigning the return value from Child::markingToStringImplementation() to an auto variable and printing it to the console, there would be no way for this function to tell if the child

2 Though learning to do was not needed for the assignment and difficult for someone new to C++, John did enjoy learning this trick!

function returned a string, an int, or something else different. In such an event would require an additional check to determine the type returned[3].




```
#ifndef ASSIGN2_INCLUDE_CAT_H_
#define ASSIGN2_INCLUDE_CAT_H_

#include "Mammal.h"

#define CAT_COLOURS \
  X(Black)          \
  X(Chocolate)      \
  X(Gray)           \
  X(Cream)          \
  X(Cinnamon)       \
  X(Red)

#define CAT_MARKINGS \
  X(Brindle)         \
  X(Spotted)         \
  X(Tortoise)        \
  X(Calico)          \
  X(Striped)         \
  X(Tabby)

class Cat : public Mammal<Cat> {
public:
    enum class Colour {
#define X(name) name,
        CAT_COLOURS
#undef X
    };

    enum class Marking {
#define X(name) name,
        CAT_MARKINGS
#undef X
```

The final child classes may appear to have repetitive methods at first glance. However, each class implements its own unique logic for methods like colourToString() and similarly named functions.

To enforce species-specific constraints on colours and markings, enumerations were used to define the allowed values, as outlined in the appendix of the Assignment 2 documentation. This design ensures that users cannot assign or retrieve a colour or marking that is not specified in the enumerations.

Macros were utilized to centralize the list of colours and markings in the class header file. The C++ preprocessor leverages these macros for both the enumeration declarations and the static functions that convert enumeration values to their corresponding string representations. This approach ensures that any updates to the colour and marking macros are automatically reflected in the class definitions, reducing redundancy and the risk of inconsistencies.

3 A way of doing this would be to use std::is_same for static time checking. But that's beyond this assignment, as is this whole section about dynamic vs static polymorphism. The author just found it fun!