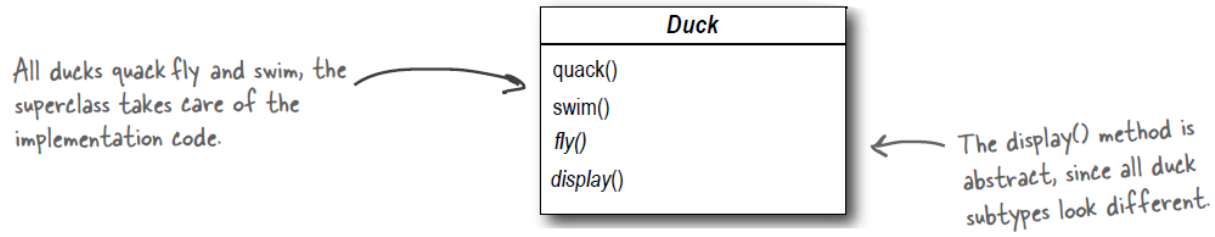
Background and Motivation

In class we discussed Object Oriented programming and design. We discussed how OOP ties together our data and functionality and we reviewed the concepts of inheritance and polymorphism, which you were already familiar with from Java and your intro courses. We also looked at C++, comparing and contrasting it to both procedural C and object-oriented Java. This assignment is designed to give you a basic working experience with C++, while enforcing a key lesson in object-oriented design and the considerations involved. The assignment is comprised of multiple small stages, each building off of the previous one.

The Duck Simulation App

You work for a company that makes a highly successful duck pond simulation game. The game can show a large variety of duck species swimming, flying and making quacking sounds. The initial designers of the system used standard OO techniques and created one Duck superclass from which all other duck types inherit. Recently, the company was bought by a visionary, but controversial CEO, prompting half the engineers to leave and the other half to be fired, leaving no one who even knows where to find the code. Luckily, the design documents were found drawn on the glass walls of the CTO's office. In yet another controversial decision, the new CEO demands that the application be rewritten in C++ and has hired you specifically for this task.

The Current Design



Release #1

Requirements

For this release you will implement the Duck class, as per the spec above, along with two subclasses – the Marbled duck and the White Pekin duck.

All classes must be within the DuckSim namespace.

Since it is a prototype, we will abstract out the complex algorithm and simply print something to the screen.

- The **quack**() method prints to the standard output: `quack!`
  - While the majority of ducks quack like above, some ducks may make a slightly different sound. Therefore, the Duck subclasses must be able to override the quack implementation
  - In particular, the White Pekin duck quacks a lot, so it will print: `quack, quack!`
- The **swim**() method prints to the standard output: `Look, I am swimming`
- The **fly**() method prints to the standard output: `I can fly`
- As indicated above, the **display**() method does something different for each duck implementation. In our case, you will print the following to the standard output:
  - Marbled duck: `Displaying a Marbled duck on the screen`
  - White Pekin duck: `Displaying a White Pekin duck on the screen`

1) Begin by checking out the git from this repository and looking within **DuckApp/release1**
2) You will create three classes: **Duck, MarbledDuck** and **WhitePekinDuck**.
3) The **Duck** class will be declared in a file called **Duck.hpp** and defined in a file called **Duck.cpp**
4) The duck subclasses will both be declared in **AllMyDucks.hpp** and defined in **AllMyDucks.cpp**
5) I have provided a main.cpp file that when run with your code should produce exactly as indicated in the file.

(Note: You may be tempted to devise some fancy algorithm for the display() or quack() methods that allows different implementations to reuse and share functionality. Don't do that. These are simplifications of complex code that in reality would have nothing in common. The point here is to focus on the design and for each implementation to output the text directly. Keep it simple.)

Release #2

Congratulations! Your first release was well-received and your clients have started to use your product. Everything is going well, until you receive a phone call from the product manager who is attending the shareholders meeting in San Francisco. "I'm at the shareholder's meeting," she says. "They just gave a demo here and there were rubber ducks flying around the screen. Rubber ducks don't fly!! What's going on?" As you hang up the phone, you notice an email from a big client of yours, who has just extended the library and implemented a wooden decoy duck. He is wondering why his duck is quacking and flying? You jokingly suggest that it's a feature, but he's not amused.

What happened? The designer failed to notice that not all subclasses of Duck should fly or quack. What he thought was a great use of inheritance for the purpose of reuse hasn't turned out so well when it comes to maintenance. By adding behavior to the Duck superclass, you also added behavior that was not appropriate for some Duck subclasses.

You could always tell your clients to override the fly() and quack() methods with code that does nothing, but what would happen if down the road you have a new requirement for a method called **migrate**() and not all existing ducks actually migrate? A localized update to your Duck code will have non-local side effects on all subclasses.

You realize that inheriting from the Duck superclass probably wasn't the correct design, because you know that management will want to update the product every few months in ways that they haven't yet decided on. You need a cleaner way to have *some* but not *all* of the duck types fly or quack, and you need a cleaner way to add future behaviors.

Being a Java developer, you immediately think of an interface. You decide to take the fly() and quack() methods out of the superclass and create **FlyBehavior** and **QuackBehavior** interfaces. In Java, you would use actual interfaces, but in C++ there are only classes. However, we can take advantage of multiple inheritance.

Requirements

1) Remove the **fly**() and **quack**() methods from your Duck superclass and replace them with two methods called **performFly**() and **performQuack**().

2) Create two new classes called **FlyBehavior** and **QuackBehavior** in the **Behaviors.hpp** and **Behaviors.cpp** files.
   - Define the **fly**() method within the **FlyBehavior** class
   - Define the **quack**() method within the **QuackBehavior** class
   - Implement both classes as appropriate to support the Duck subclasses

3) Reimplement the **MarbledDuck** and **WhitePekinDuck** classes, inheriting and overriding the appropriate behaviors as necessary.
   - Note, the Duck API now contains the **performFly**() and **performQuack**() methods.
   - Those methods should be implemented by delegating the responsibility and calling the **fly**() and **quack**() methods as appropriate.

4) Implement two new duck subclasses within the **AllMyDucks** files called **RubberDuck** and **DecoyDuck.**

   - A rubber duck does not fly, and when asked to quack, prints: `squeak`
   - A decoy duck does not fly and does not quack
   - Display a rubber duck: `Displaying a rubber duck on the screen`
   - Display a decoy duck: `Displaying a decoy duck on the screen`

5) Users should **not** be able to call **quack**() or **fly**() directly on any duck object; they are meant to control the duck using only the **performQuack**() and **performFly**() methods.

6) I have provided a main.cpp file that when run with your code should produce exactly as indicated in the file.

7) In a comment written in the Behaviors.hpp file, explain why you needed to write duplicate code across the duck subclasses. Specifically, since 99% or all ducks will fly or quack, why did you find yourself having to rewrite similar code for performQuack() and performFly()? Imagine you had 500 duck classes and needed to add some logging behavior within either of those methods?

8) In a second comment written in the Behaviors.hpp file, discuss whether the issue would be better or worse if you had been coding in Java and had your ducks implemented good old Java Interfaces.

Release #3

Congratulations again! You have fixed the flying rubber duck and quacking decoy duck bug. (Although you still feel it is a cool feature.) However, as you noted in your comments to your manager, you have created a little bit of a maintenance nightmare. The Duck Simulation application is really taking off and new ducks are being created every day. There is only one constant in the software industry – and that called change. You can bet that something will need to change soon, most probably in ways you didn't anticipate.

Inheriting from one superclass didn't work out well, since the duck behavior keeps changing across the subclasses, and it's not appropriate for all subclasses to have those behaviors. Breaking out the behaviors into individual interfaces or classes sounded promising (at least in C++), but it does leave you with some duplication of code. As well, what would happen if a new flying behavior comes onto the scene, or I want to change the quacking behavior of one of my ducks temporarily?

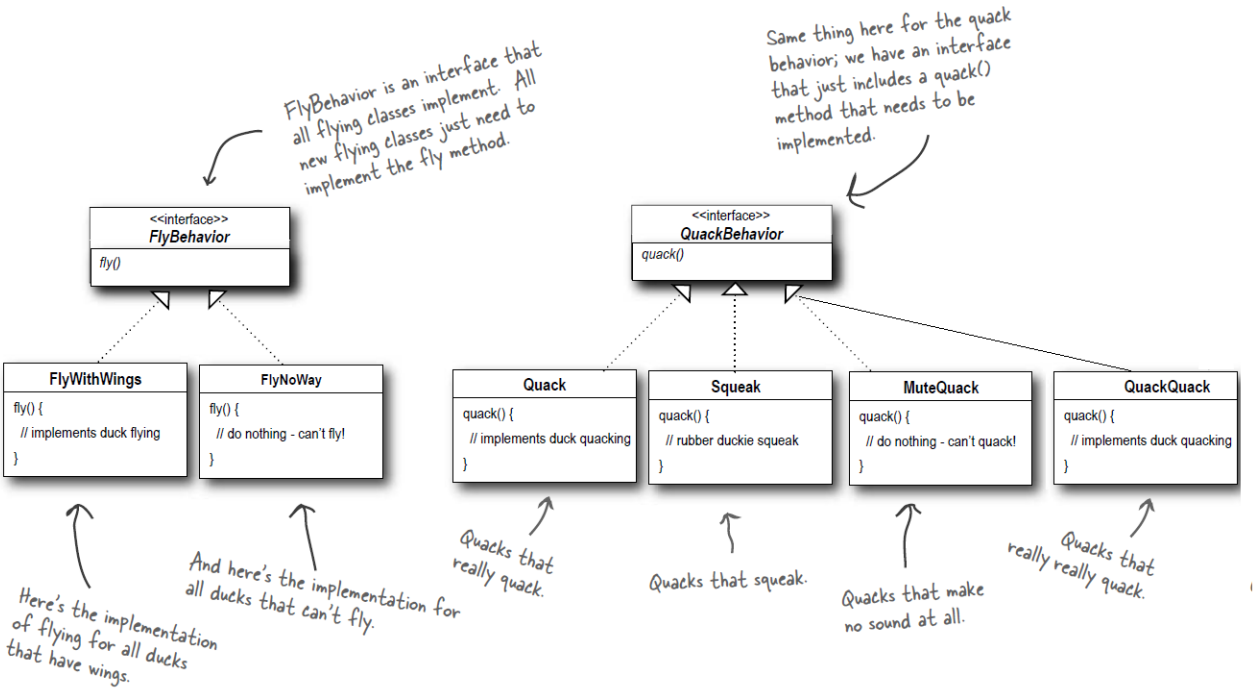
What you are looking for is a way to extract the parts that could change and encapsulate it so that it won't affect the rest of your code. That way, you can alter or extend those parts that vary, without affecting those parts that don't. The result? Fewer unintended consequences from code changes and more flexibility in your systems!

And while we're on the topic of flexibility, next week you agreed to present at your younger brother's career day at their elementary school. You want to show them your application, but knowing your audience, you'd like to instantiate a new rubber duckie and have it fly around the screen. That would really get the kids interested in becoming software engineers. Even better, you'd like the behavior to be able to change at run time, so that the kids could press a button and the ducks could make different sounds.

We already started pulling out the behaviors from the Duck superclass. That was a great start. However, we'd like to keep things flexible and we know that inheriting the behaviors limits our flexibility. When we inherit a behavior, it's built into the class and we can't change it without making a change to the class. Instead, this time we aren't going to have the duck subclasses inherit or implement the behaviors; instead, they will **use** them.

From now on, the Duck behaviors will live in a separate class structure—a class that implements a particular behavior interface. That way, the Duck classes won't need to know any of the implementation details for their own behaviors. When we construct the duck subclasses, we pass in any behavior that we want it to implement. Additionally, we can provide methods to add or remove behaviors from ducks even after the duck is instantiated. With this design, other types of objects can reuse our fly and quack behaviors because these behaviors are no longer hidden away in our Duck classes, as they were in languages without multiple inheritance. And we can add new behaviors without modifying any of our existing behavior classes or touching any of the Duck classes that use flying behaviors.

The key is that a Duck will now **delegate** its flying and quacking behavior, instead of using quacking and flying methods defined in the Duck class (or subclass). Polymorphism is attained through composition, not through inheritance, which models a *has-a* relationship, as opposed to an *is-a* relationship.



COM-3640 Programming Languages
Homework #4: Duck Simulation
Due Thu Dec 22 11:59 PM

Requirements

1) Define and implement the above behavior classes within the **Behaviors.hpp** and **Behaviors.cpp** files.
    - FlyWithWings prints: `I can fly`
    - Quack prints: `quack!`
    - Squeak prints: `squeak`
    - QuackQuack prints: `quack, quack!`
    - FlyNoWay and MuteQuack do not print anything

2) Reorganize your code from release two so that the duck subclasses do not inherit from the behavior classes. Limit code duplication and redundancy.

3) Implement default constructors for the four duck subclasses that will create their respective objects with the correct behaviors. Example, `RubberDuck d;` creates a rubber duck object that squeaks and cannot fly.

4) Implement constructors for the duck subclasses that allow users to instantiate new instances of ducks and specify the fly behavior and quack behavior at the time of construction.

5) Add two methods to the Duck class called **setFlyBehavior** and **setQuackBehavior** that will allow the user to dynamically change the behavior of an existing duck instance.

6) Ensure that all objects that have been dynamically allocated from the heap are deallocated when the object is deleted or goes out of scope.

7) In a comment within the Duck.hpp class, explain what you would need to do if you wanted to be able to handle ducks that could quack in multiple ways. For example, you want your duck to both quack and squeak. (Bonus points if you can suggest a way to do so without changing the interface of any methods).

8) I have provided a main.cpp file that when run with your code should produce exactly as indicated in the file.

Release #4

You have now learned a valuable OO design lesson and have a flexible system to extend and enhance, more safely and at a quicker pace.

For the last release of the year, your manager asks you to implement the following:

1) A unique id for each duck instance, based on an increasing counter for each duck object instantiated. For example, the first duck created is given id=1, the second one id=2, etc. All duck subclasses share the same counter.
    - Implement a method **getIdentifier**() that returns the id (integer value) of the duck.

2) Augment the **FlyWithWings** behavior to allow the user to optionally specify a speed multiplier (and integer from 1-5) upon construction. The multiplier will then be reflected in the fly method.
    - For example, if the user specifies a multiplier of 3, the fly() method will print:
      `I can fly fly fly`
    - If the argument is out if range (i.e. not between 1 and 5) throw the following exception:
      `throw std::out_of_range ("Speed out of range");`

3) I have provided a main.cpp file that when run with your code should produce exactly as indicated in the file.

General HW Submission Instructions

- All submitted assignments must be checked into the YU GitHub system.
- Say that your Git url is https://github.com/Yeshiva-University-CS/SmithBob. When you *git clone* that url to your computer, the result is a directory named SmithBob. That root (Git) directory will be referred to as **$MYGIT**.
- All of your submitted assignments must be rooted in a directory named: **$MYGIT/PL-COM3640/assignments**.
- Each assignment will be associated with a specific subdirectory. We will refer to that directory name as **$DIR.**
- Therefore, all submissions for a given assignment will be checked into and rooted at:
  $MYGIT/PL-COM3640/assignments/$DIR

This HW's Submission

- The subdirectory for this assignment is **DuckApp**.
- The assignment must be checked into: **$MYGIT/PL-COM3640/assignments/DuckApp**
    - This assignment expects multiple versions, which will be under:
    - DuckApp/release1, DuckApp/release2 etc., as the template code is structured.