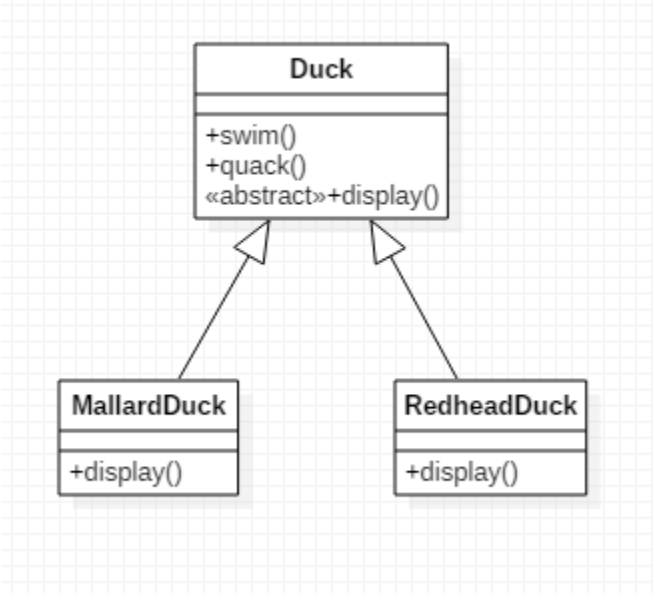
Teacher Notes for the Duck Application Workshop

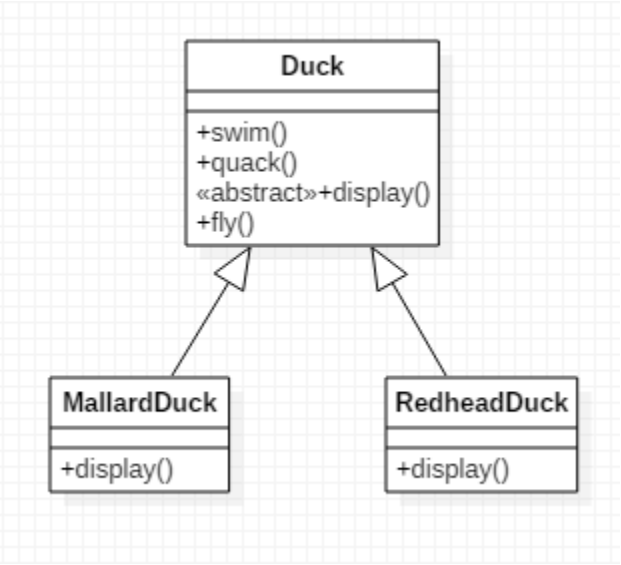
We wish to create a design for a duckpond simulation game. Our design focuses on the basic characteristics of the ducks in the game.

First Try: We start with two behaviors that ducks should have: swimming and quacking. We have in mind some boilerplate implementation of swimming and quacking for ducks. We also expect to have an abstract display operation that will handle details for drawing particular types of ducks. We also have in mind two types of ducks as we begin: mallard ducks and redhead ducks. These exhibit the usual swim and quack behaviors that we expect of ducks, and they each have a unique appearance, so each of these ducks has its own implementation of the display operation.



Considerations for Expanding. We may wish to introduce flying behavior, which would also have boilerplate implemenation. We could add the fly() operation to Duck.

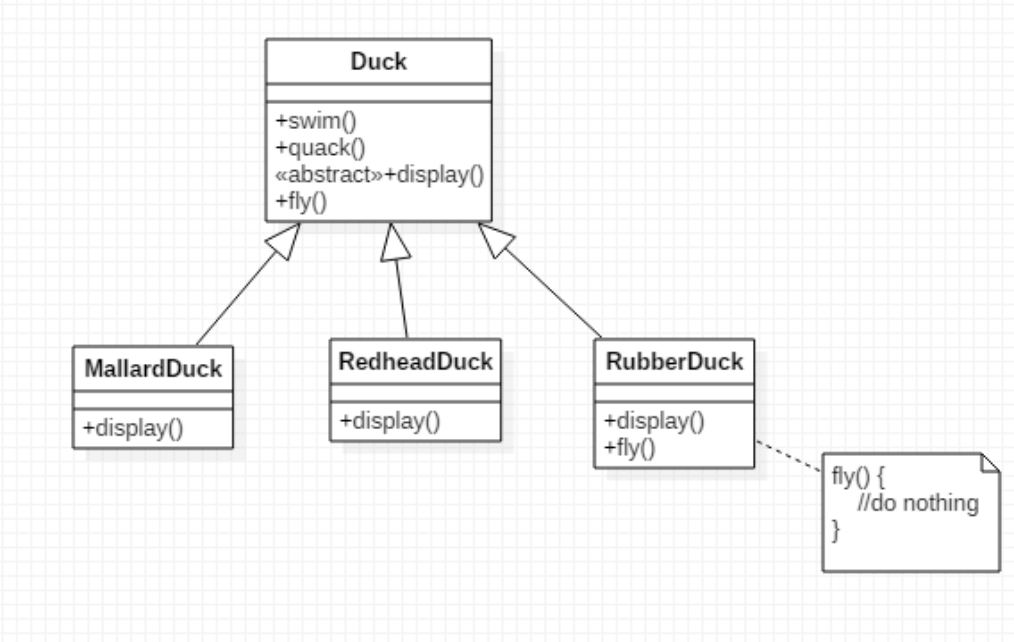
Second Try: We add fly() behavior to Duck



Considerations for further expansion. We also anticipate creation of new Duck types.

Problem. What if we add a new type that does not have all expected behaviors? We could add a RubberDuck which cannot fly. In that case, RubberDuck inherits the standard fly behavior from Duck but this is not appropriate. We could override fly() and give it no behavior.

Third Try: We have added RubberDuck and overridden the fly() operation so that it does nothing.

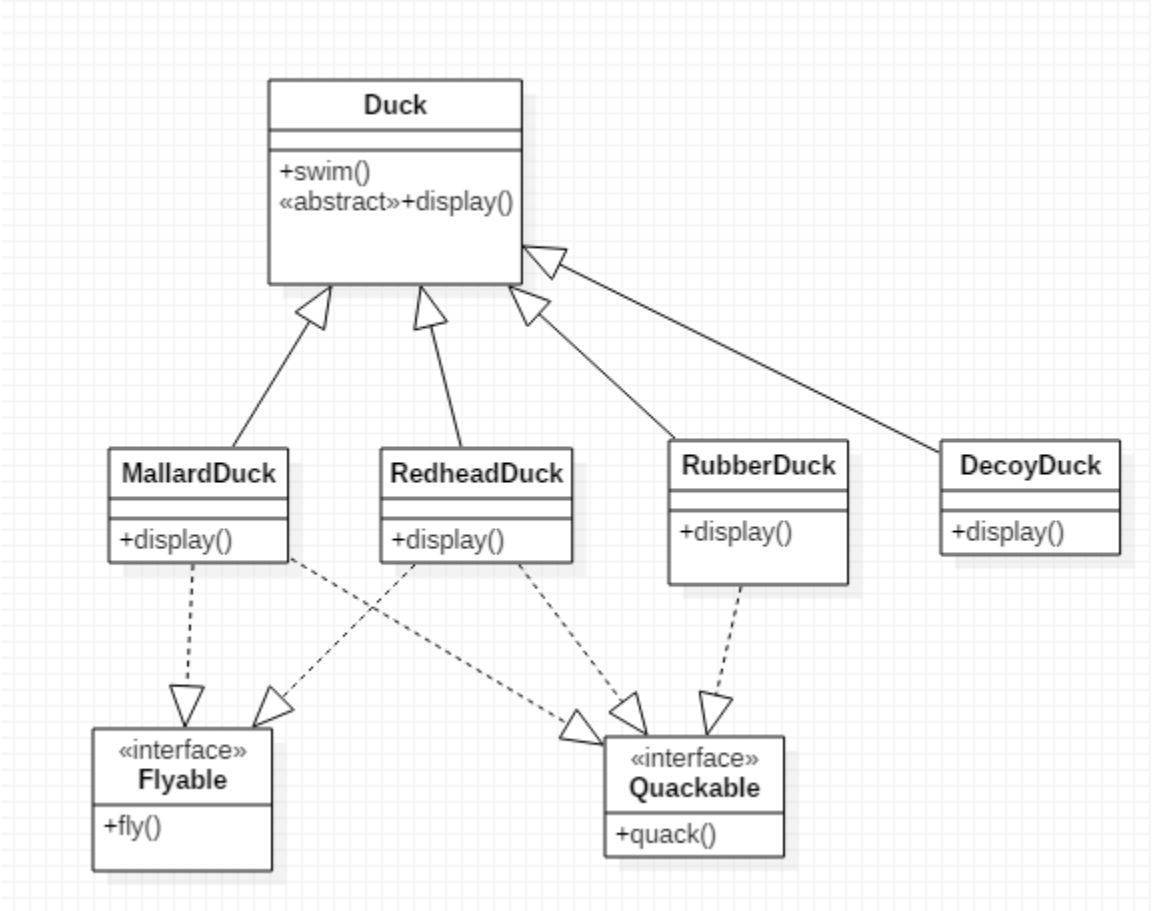


Considerations for further expansion. What if we add more duck types that not only don't fly, but don't quack either? An example is DecoyDuck (a kind of wooden duck). We could override both fly() and quack() with empty implementation.

Problem. As we add more Duck types, we are introducing ducks that may or may not exhibit the fundamental behaviors that are shown on the public interface of Duck. This makes one question whether these behaviors should really be listed as the public methods of Duck. If one or two such methods are occasionally overridden with empty implementation, it would be ok, but it may be that a significant portion of our ducks won't fly, and many others won't quack. What would be a better way to handle the fly and quack operations?

We attempt to solve this by introducing interfaces for each of the behaviors that may or may not belong to particular ducks.

Fourth Try: We have introduced Flyable and Quackable interfaces. Those ducks that have these behaviors can then implement these interfaces.



Problem. With this new design, a new problem arises: The fly behavior for a Duck that does fly is always the same. Therefore, every time the Flyable interface is implemented, the implementing class must implement the fly method with exactly the same code. The same holds true of implementers of the Quackable interface. We have introduced a situation whereby we are forced to use the same code over and over. This redundancy makes the code hard to maintain and therefore introduces an undesirable pattern in our design.

Solution. We separate things that change from things that don't change through the use of composition. We define a cluster of classes that represent Fly Behavior and another cluster of classes that represent Quack Behavior, and then require Duck to *compose* these behaviors rather than inherit them. Fly Behavior is indicated by a FlyBehavior interface that is implemented by different types of fly behaviors; similarly for Quack Behavior.

Fifth Try: We implement our strategy from the previous page.