CMPSCI 220 Programming Methodology

05: Inheritance vs Composition

Objectives

Composition versus Inheritance

- Inheritance in OO
- Unified Modeling Language
- IS-A and HAS-A relationships
- Composition as an alternative to inheritance
- Delegation

SimUDuck Application

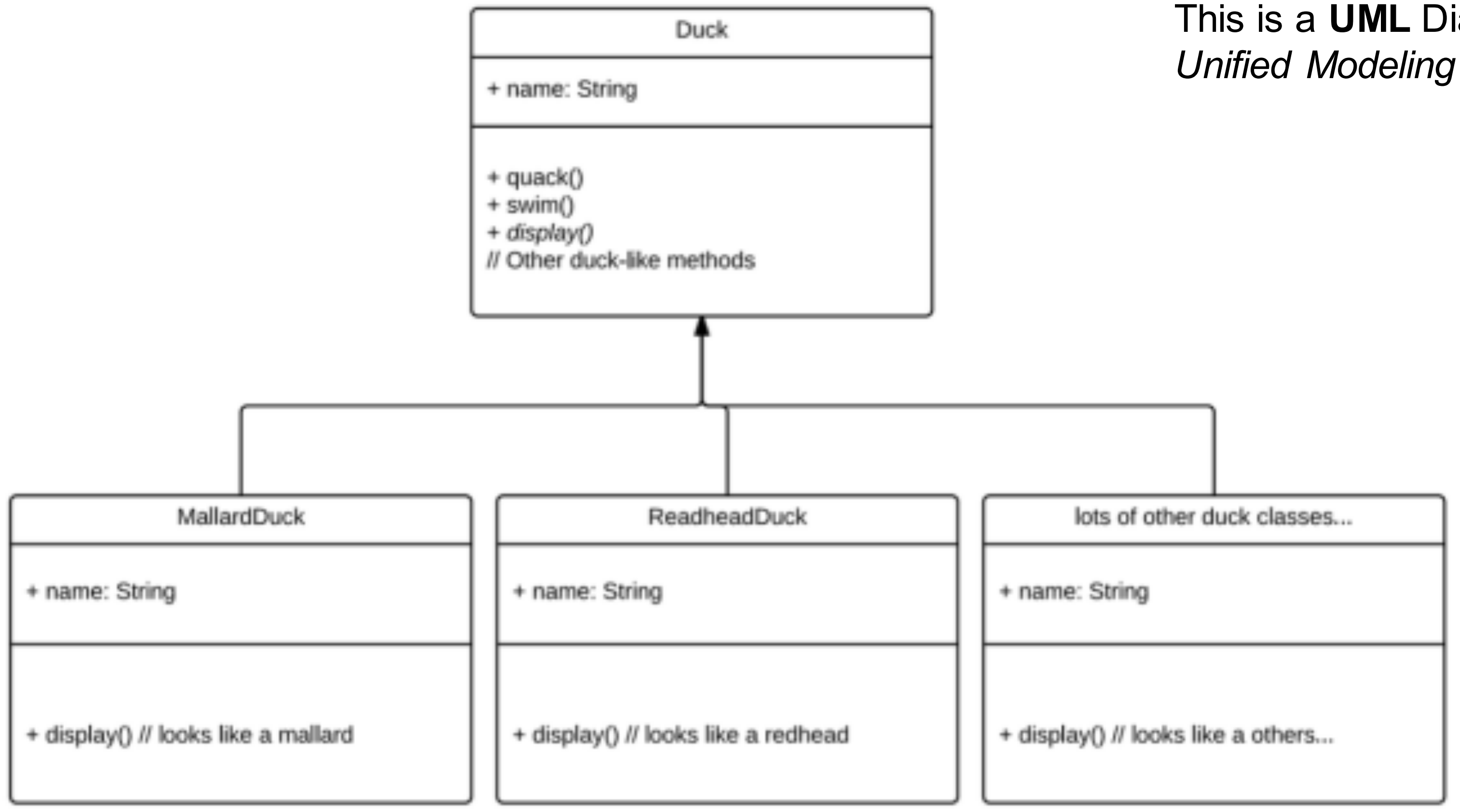
Amazing Game Company

Imagine that you work for a company that makes a highly successful game *SimUDuck*. The game can show a large variety of duck species swimming and making quacking sounds. The initial designers of the system used standard OO techniques and created on Duck superclass from which all other duck types inherit.

This is example is taken from head first design patterns

Very real looking graphics, don't you think!

SimUDuck Class Hierarchy



Duck
+ name: String
+ quack()
+ swim()
+ *display()*
// Other duck-like methods
MallardDuck
+ name: String
+ display() // looks like a mallard
ReadheadDuck
+ name: String
+ display() // looks like a redhead
lots of other duck classes...
+ name: String
+ display() // looks like a others...

This is a **UML** Diagram!
Unified Modeling Language

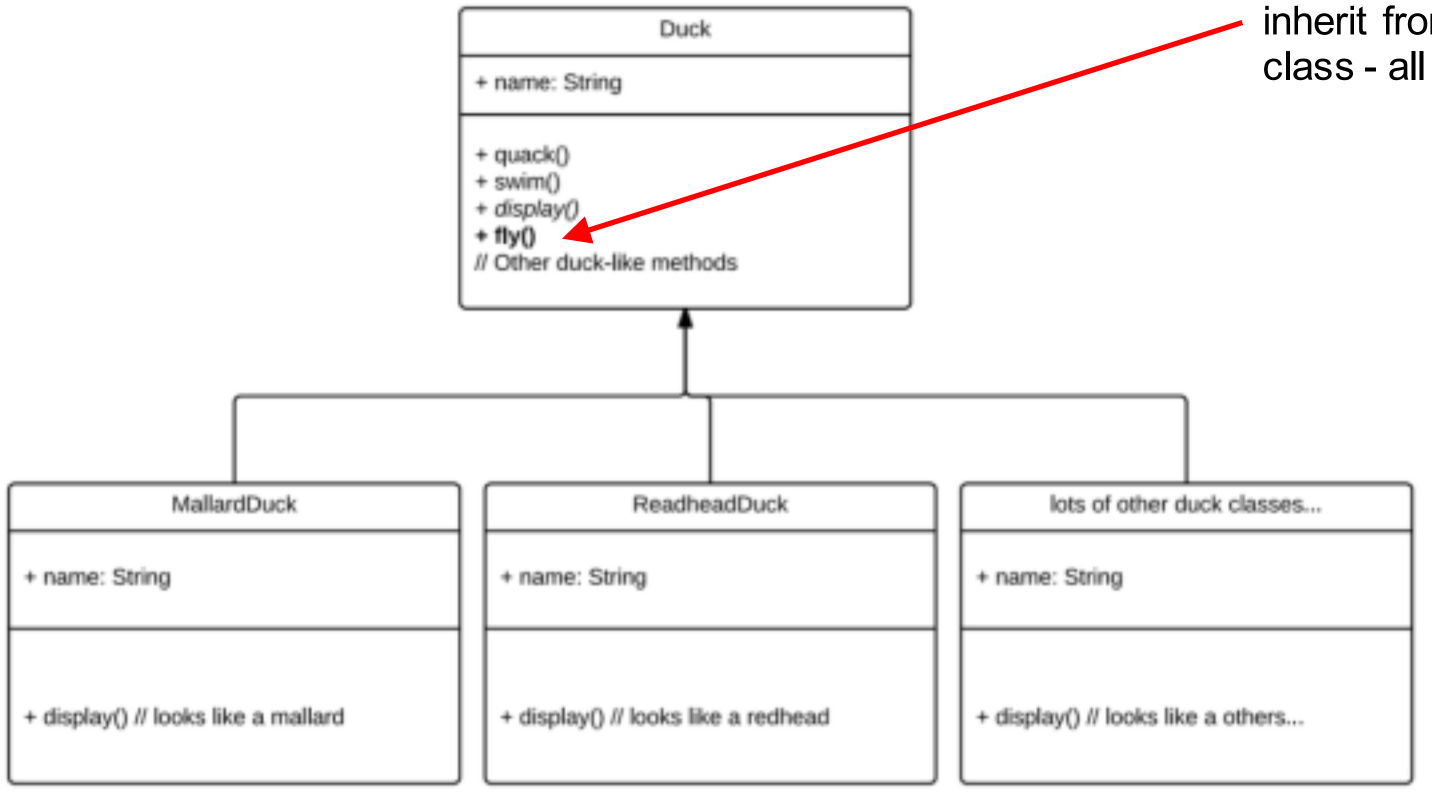
SimUDuck Application: Next Version!

Amazing Game Company
To stay competitive, the executives decide that flying ducks is just what SimUDuck needs to blow away the competitors!

You know what to do because you are a true OO genius. We will simply add a new method that will allow ducks to fly!

Very real looking graphics, don't you think!

SimUDuck Class Hierarchy Flying!


Duck
+ name: String
+ quack()
+ swim()
+ display()
+ fly()
// Other duck-like methods
MallardDuck
+ name: String
+ display() // looks like a mallard
ReadheadDuck
+ name: String
+ display() // looks like a redhead
lots of other duck classes...
+ name: String
+ display() // looks like a others...

Now, all subclasses will inherit from the Duck super class - all ducks can now fly!

SimUDuck Application: Next Version!

Amazing Game Company
BIG PROBLEM!

The executives contact you from a shareholders meeting. They just gave a demo of the game and their were **rubber ducks** flying all over the screen!

What happened?

Very real looking graphics, don't you think!

SimUDuck Flying Fix!

Well, what can we do to make this work?

SimUDuck Flying Fix!

Well, what can we do to make this work?

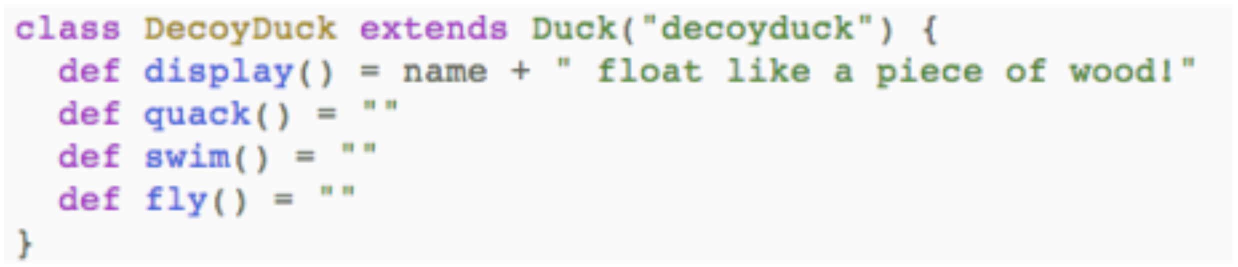
Sure, we can override the fly method to do nothing in our **RubberDuck** class.

But, what if we add a **DecoyDuck**?

Example!

DecoyDuck Can Work, But...

We have several empty methods...



```
class DecoyDuck extends Duck("decoyduck") {
  def display() = name + " float like a piece of wood!"
  def quack() = ""
  def swim() = ""
  def fly() = ""
}
```

i-clicker Question

Which of the following are disadvantages of using *inheritance* to provide Duck behavior? Choose one out of possibly many...

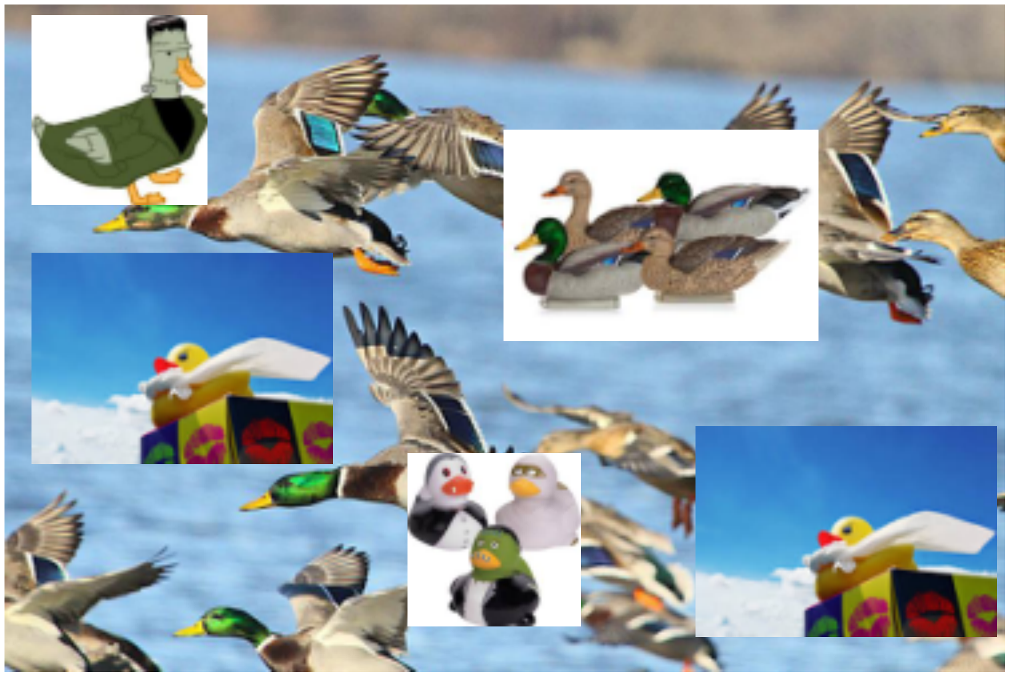
A. Code is duplicated across subclasses
B. Runtime behavior changes are difficult
C. We can't make ducks dance
D. Changes can unintentionally affect other ducks
E. Ducks can't fly and quack at the same time

SimUDuck Application: Iterative Releases!

Amazing Game Company

The executives are now telling you that they will be having tight release schedules and want to update the game every 2 months! Each update will include at least 6 new ducks.

Is there another approach to solving the problem?


Very real looking graphics, don't you think!

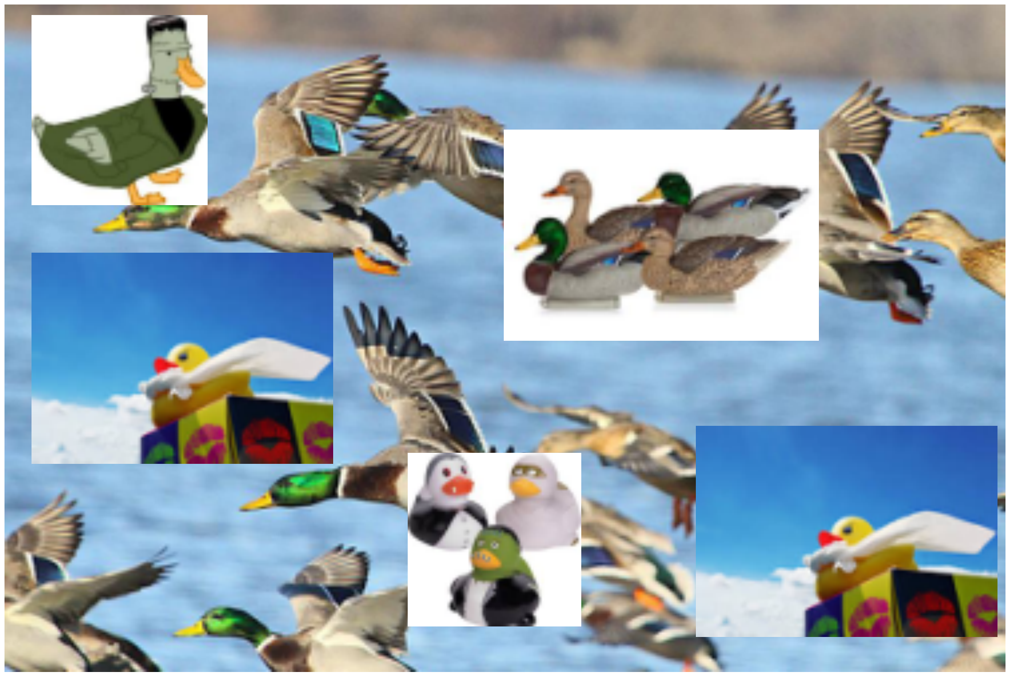
SimUDuck Application: Iterative Releases!

Amazing Game Company

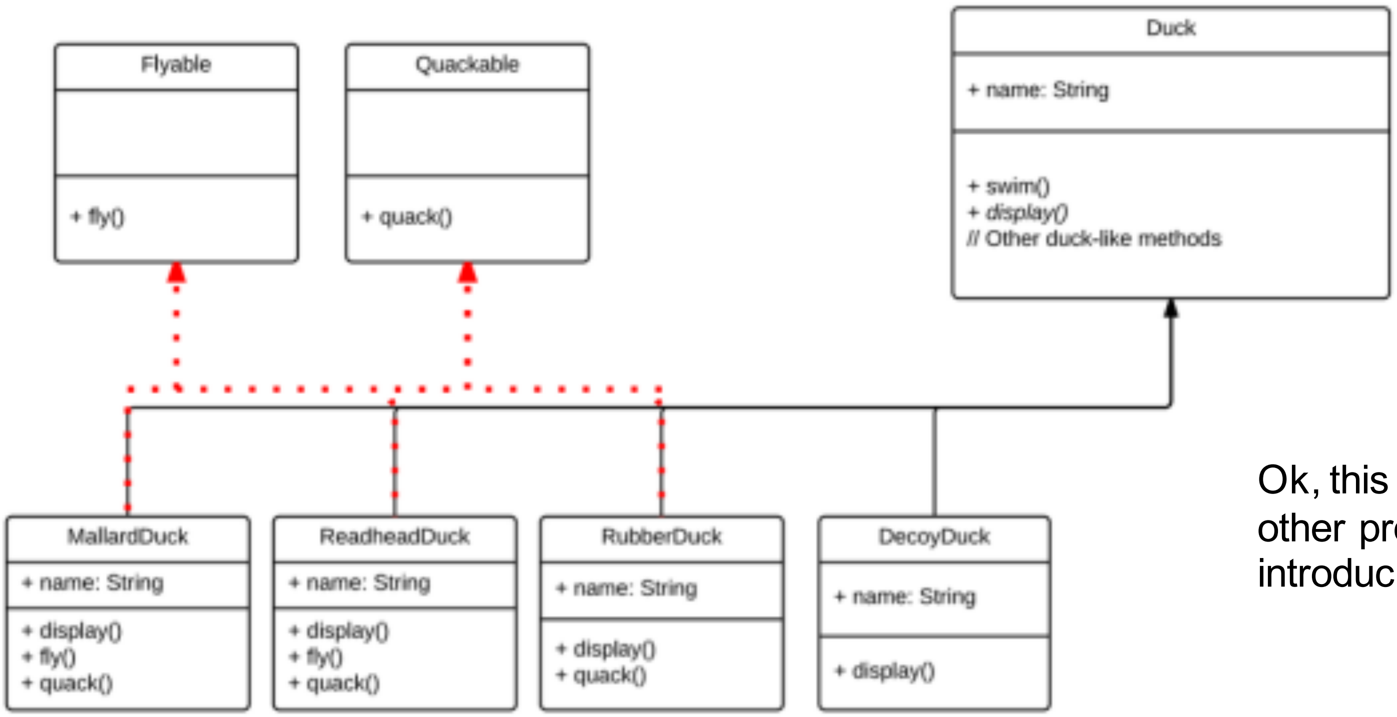
The executives are now telling you that they will be having tight release schedules and want to update the game every 2 months! Each update will include at least 6 new ducks.

Is there another approach to solving the problem?

What about interfaces/traits?


Very real looking graphics, don't you think!

SimUDuck With Interfaces/Traits!



Ok, this works - but what other problems are introduced by this?

Example!

Problem With Interfaces/Traits?

What is the biggest problem introduced here?

A. It makes subclasses impossible to implement
B. You are constrained by the number of ducks
C. You are forced to duplicate lots of code
D. You can't introduce other animals
E. There are no problems

Code Duplication

So, using interfaces forces lots of duplicated code to be written. This is an indicator of **bad design**.

So, perhaps inheritance is not all that it *quacks* up to be?

Wouldn't it be cool if there was a way to build software so that when we need to change it, we could do so with the least possible impact on existing code?

Zeroing in on the problem...

Design Principle: *Identify the aspects of your application that vary and separate them from what stays the same*.

Take what varies and "encapsulate" it so it won't affect the rest of your code.

The result: fewer unintended consequences from code changes and more flexibility in your systems!

Inheritance IS-A Relationship

Inheritance forms an "is-a" relationship between classes. In our fixed version of the SimUDuck hierarchy we had…

MallardDuck IS-A Duck IS-A Flyable IS-A Quackable
RubberDuck IS-A Duck IS-A Quackable
DecoyDuck IS-A Duck

SimUDuck: Separate the ducks from the, um, ducks

How could we better design this application so that we can *separate* out the changes so that our ducks can be more easily implemented and eliminate redundant code?

Is there an alternative to the IS-A relationship?

Example!

Ducks HAS-A Behavior

The last solution shows the HAS-A relationship

MallardDuck IS-A Duck
    HAS-A FlyableBehavior
    HAS-A QuackableBehavior

RubberDuck IS-A Duck
    HAS-A FlyableBehavior
    HAS-A QuackableBehavior

DecoyDuck IS-A Duck
    HAS-A FlyableBehavior
    HAS-A QuackableBehavior

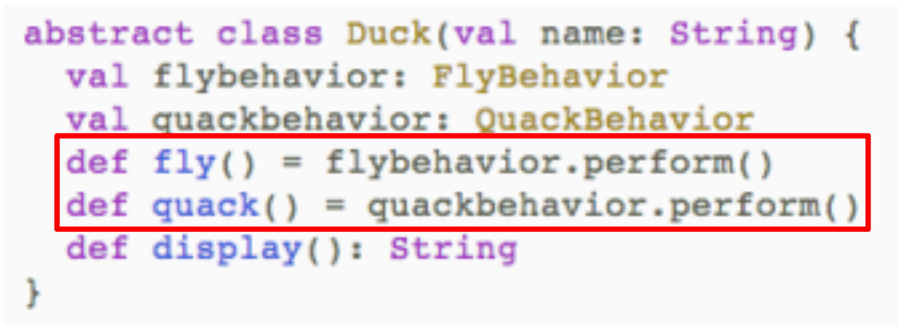
We have carefully encapsulated those aspects of a duck that can change, and will change, across many different kinds of ducks.

You can see this quite clearly with the HAS-A relationship.

HAS-A leads to a more flexible design that can be easily extended without lots of code duplication.

This is called **composition.**

Delegation



```
abstract class Duck(val name: String) {
  val flybehavior: FlyBehavior
  val quackbehavior: QuackBehavior
  def fly() = flybehavior.perform()
  def quack() = quackbehavior.perform()
  def display(): String
}
```

We encapsulate behavior in other classes to isolate changes and form a HAS-A relationship.

The base class uses *delegation* to "delegate" the behavior (e.g., fly, quack) to other classes that implement the specific behavior.

Composition Over Inheritance

Design Principle:
Favor *composition* over *inheritance*

This will lead to better design!

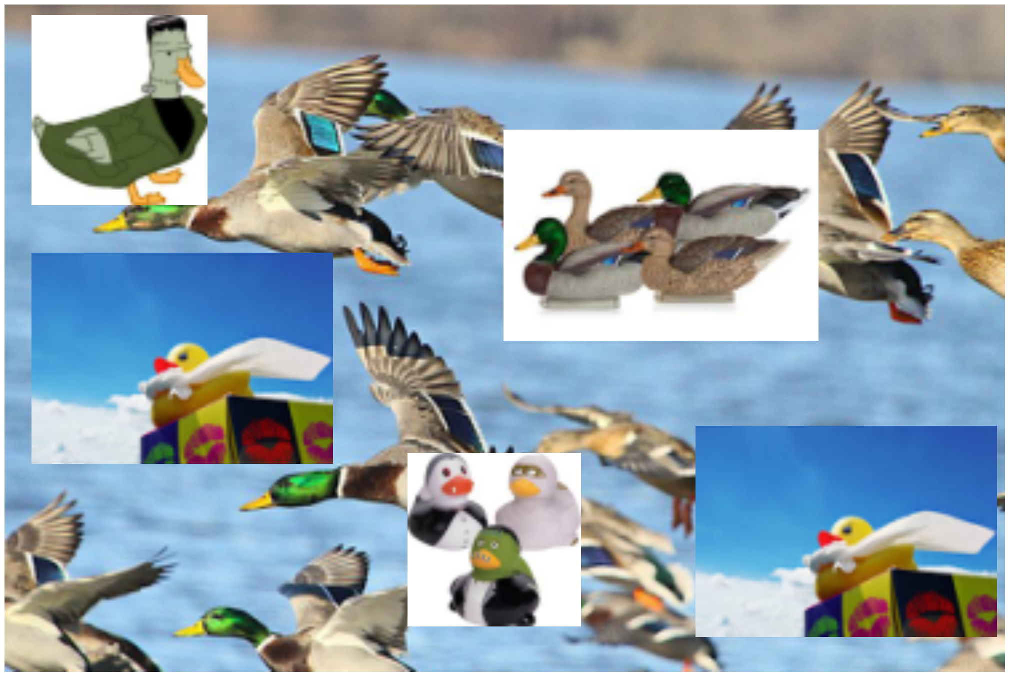
SimUDuck Application: New Idea!

Amazing Game Company

Now, the executives want to add "magic" to the game. That is, they want ducks to be able to change the behavior of other ducks they come in contact with. They want ducks to be able to teach other ducks new tricks.

Can we teach a rubber duck to fly?

What changes might we need to make to allow this capability?


Very real looking graphics, don't you think!