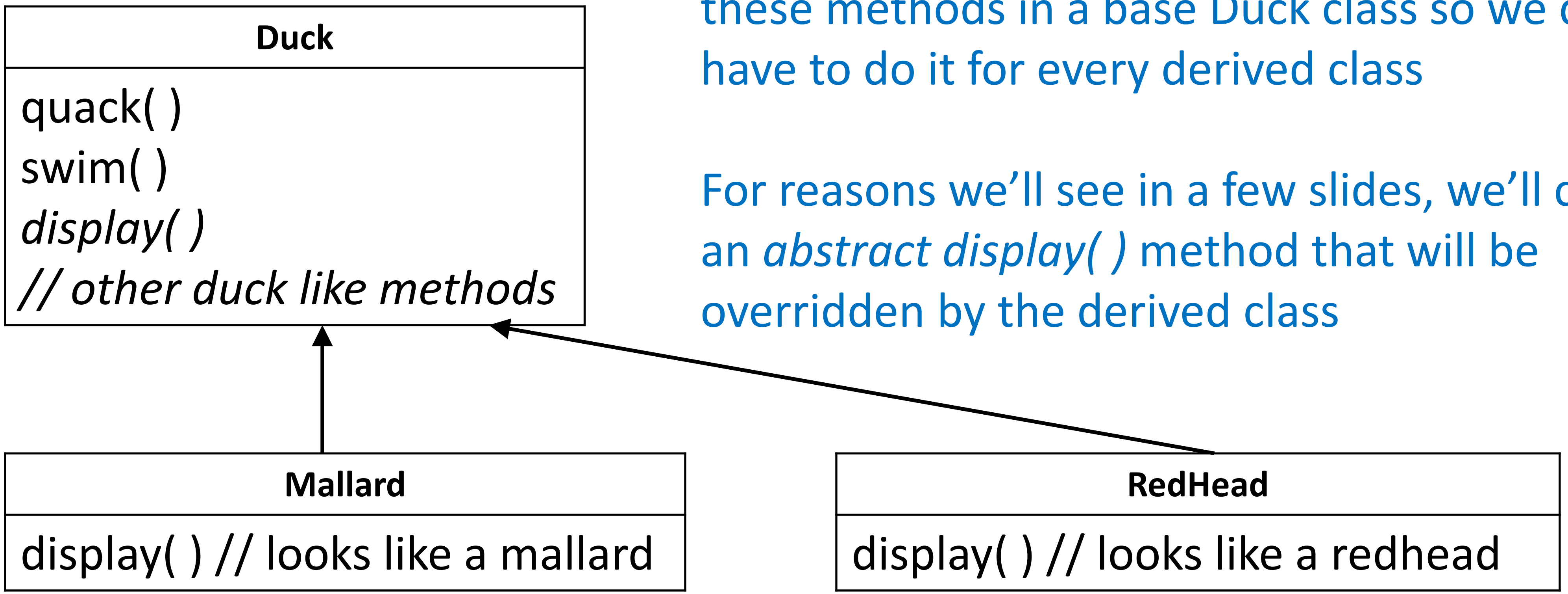
The Strategy Pattern

A duck simulator

- Our goals is to write a duck simulator that simulates ducks (duh!)
- Like real-world code we'll have changing requirements, enhancements will be requested, and there will be modifications needed
- We want to write code that is amenable to change without knowing what those changes will be

A basic duck simulator (UML)



All ducks swim and quack, so we'll implement these methods in a base Duck class so we don't have to do it for every derived class

For reasons we'll see in a few slides, we'll create an *abstract display( )* method that will be overridden by the derived class

Virtual pure functions and abstract classes

- Virtual pure functions and abstract classes are a way to force every inheriting class to implement the virtual pure function
- A virtual pure function is a method that is declared as such in the base class
- No definition, i.e., implementation, is provided
- A class with a virtual pure function is an abstract class
- Because not all methods are defined in an abstract class, we cannot create an object for the abstract class
- A class that inherits from an abstract class will also be abstract if it does not provide a definition for the virtual pure function
- Virtual pure function names are written in italics in UML

Abstract classes

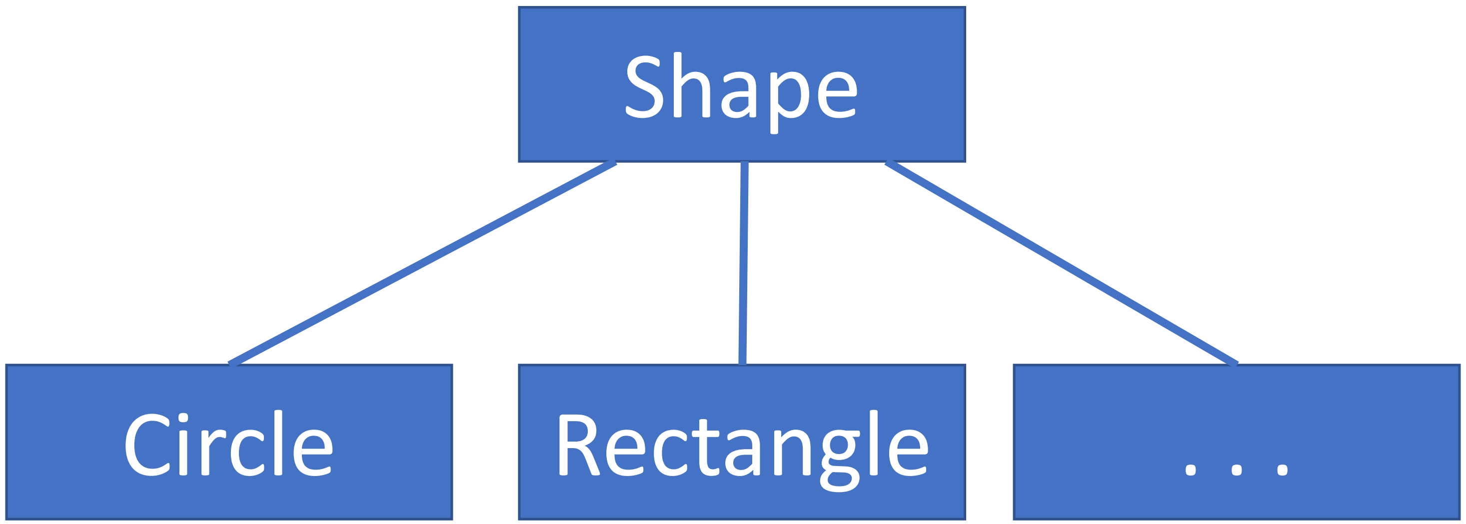
- Abstract classes are classes for which objects cannot be constructed
- They can be derived from, however
- What are they good for?

1. Can lend organization to a class hierarchy,
2. Provides a common base class
3. Can represent a specialized behavior that when mixed with other classes gives a desired behavior

Can help build up an implementation

Let’s look at a concrete example to make these concepts clearer. In particular, let’s look at a shape class such as might be used in a drawing program

A Shape class



- It makes sense to construct a Circle, Rectangle, etc., but not an amorphous shape
- However, it may be useful to have arrays of shapes to hold different kinds of shapes
- And it may be useful to *know* that every shape has a perimeter, area, color, etc., property implemented
- Abstract classes allow this

The Shape abstract class (see code in Exampl1/Good)

```
class Shape {
public:
  virtual double area( ) = 0;
  virtual double circumference() = 0;
};
```

- The Shape abstract class requires any class that inherits from it to implement area and circumference.
- This lets us call these functions on any class that extends Shape.
- C++ abstract classes can also declare variables and non-abstract functions

The Square class, Circle is similar

```
class Square : public Shape {
public:
   Square(float);
   ~Square( );
   double area( );
   double circumference();
private:
   float side;
};
```

```
Square::Square(float s) : side(s) { };
Square::~Square( ) { }
double Square::area( ) {
   return side*side;
}

double Square::circumference() {
   return 4.0*side;
}
```

What happens if Circle doesn't declare and define a pure virtual function? (See Example1/Error code)

```
class Circle : public Shape {
public:
   Circle(float);
   ~Circle( );
   // double area( );
   double circumference();
   static const double
         PI=3.141592653589;
private:
   float radius;
};
```

```
Circle::Circle(float r) : radius(r) { }
Circle::~Circle( ) { }
// double Circle::area( ) {
//   return Circle::PI*radius*radius;
//}

double Circle::circumference( ) {
  return 2*3.PI*radius;
}
```

Circle.cpp compiles ok

The error shows up in the code that tries to instantiate the abstract class

```
g++ main.cpp
main.cpp: In function 'int main()':
main.cpp:11:30: error: cannot allocate an object of abstract type 'Circle'
    shapes[1] = new Circle(4.0);
                             ^
In file included from main.cpp:2:0:
Circle.h:5:7: note:   because the following virtual functions are pure within 'Circle':
 class Circle : public Shape {
       ^
In file included from Square.h:3:0,
                 from main.cpp:1:
Shape.h:5:19: note:     virtual double Shape::area()
    virtual double area( ) = 0;
```

```
class Duck {
public:
  virtual void quack( );
  virtual void display( ) = 0;
  virtual void preen( ) = 0;
};

void Duck::quack( ) {
  std::cout << "Quack, quack" << std::endl;
}

class RedHeadDuck : public Duck {
public:
  void display( );
};

void RedHeadDuck::display( ) {
  std::cout << "I’m a redhead duck" << std::endl;
}
```

```
class SuperDuck : public RedHeadDuck {
public:
  void preen( );
};

void SuperDuck::preen( ) {
  std::cout << "Preen, preen" << std::endl;
}

int main(int argc, char *argv[]) {
  Duck duck = new Duck( )
  RedHeadDuck redHead = new RedHeadDuck( );
  SuperDuck super;
  super.quack( );
  super.display( );
  super.preen( );
}
```

See Example2 code

Virtual pure functions and abstract class example

(See Example2 code)

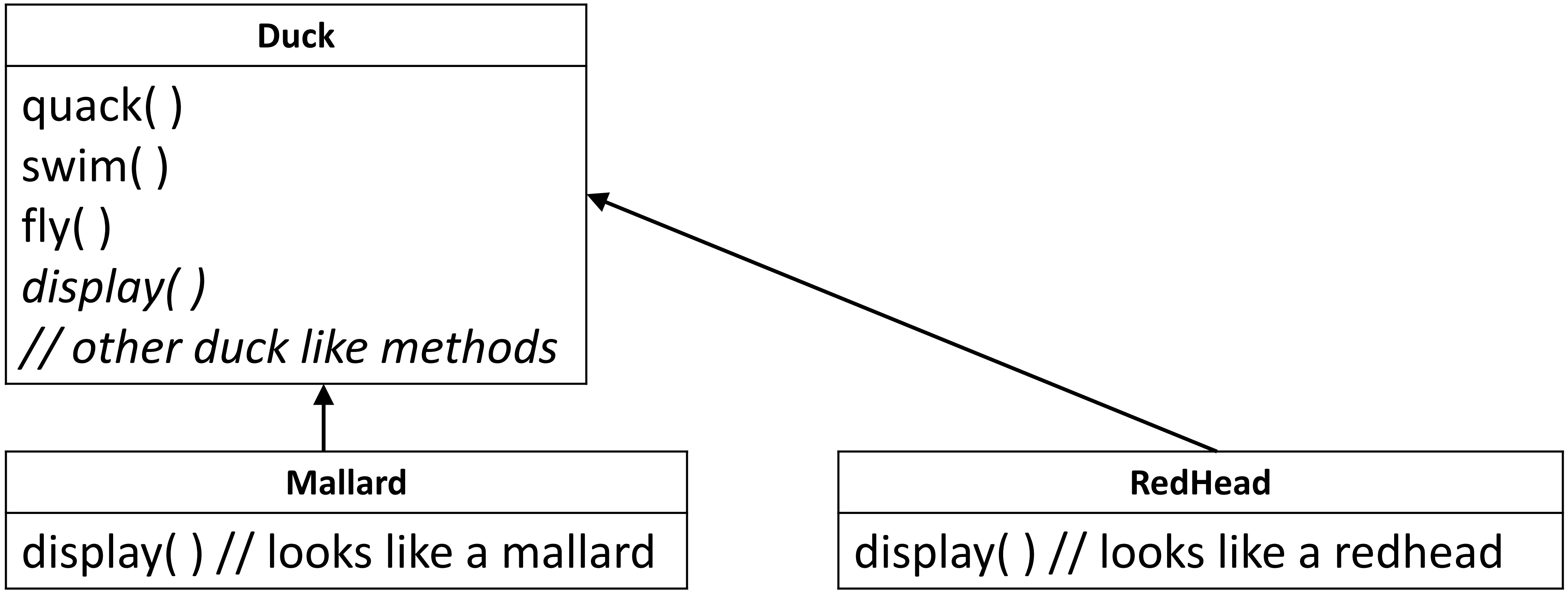
```
class Duck {
public:
  virtual void quack( );
  virtual void display( ) = 0;
};

void Duck::quack( ) {
  std::cout << "Quack, quack" << std::endl;
}

class RedHeadDuck : public Duck {
public:
  void display( );
};

void RedHeadDuck::display( ) {
  std::cout << “I’m a redhead duck" << std::endl;
}
```

```
int main(int argc, char *argv[]) {
  Duck duck = new Duck( );
  RedHeadDuck r1;
  Duck* r2 = new RedHeadDuck( );
  r1.quack( );
  r1.display( );
  r2->quack( );
  r2->display( );
}
```

We fix the compile time error, test, and everything is fine

- Then management says, *add flying to the behaviors*
- We don’t want to change every derived class, so we assume every duck will fly and add flying to the base class.
- Inheritance allows this behavior to be shared by all of the derived classes

The new Duck subclass – all ducks can fly!



But there's a problem

- Unbeknownst to us, a RubberDuck class inherits from our Duck class
- Rubber ducks don't fly, but our code gives them a fly behavior
- Inheritance is good when all of the derived (subclasses) need the inherited behavior
- When they don't need it, we have actions associated with objects that don't make sense
    - This is a bug

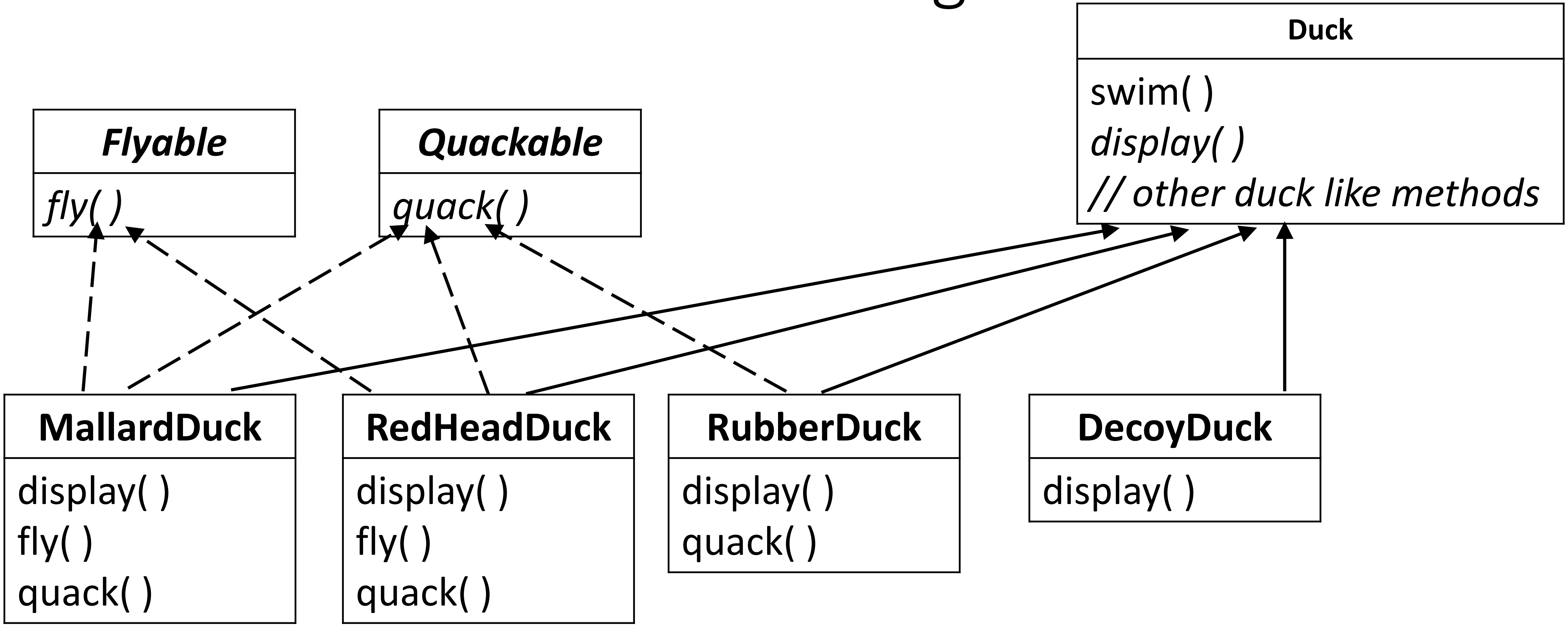
A possible solution

- Override the fly method in the RubberDuck class
- This overriding can be used to eliminate the erroneous fly behavior
- Inheritance is good when all derived classes need the inherited behavior – but causes problems when some derived classes don't need it
  - Why is the behavior in the base class if it is not fundamental to what the base class is?
  - Every derived class with different behavior has to override it, even if two such classes have the same behavior
    - This leads to code duplication as multiple classes implement the same overriding behavior
    - Duplication increases the overhead of maintenance and debugging

Why not make fly virtual pure like display?

- fly( ) implements a behavior that not all derived classes representing different kinds of ducks need
- With fly( ), multiple classes may need the same fly behavior
- display( ) implements a behavior that we want to require every duck to have
  - With display, every duck is different, every derived class representing a different duck will have a different implementation because every kind of duck has a unique appearance
  - Because every kind duck needs a different implementation of display( ), requiring every kind of duck to implement display will not lead to duplicated code
  - ***fly( ) will be the same for most ducks, however – which leads to code duplication, a terrible thing.***

A new duck simulator design



The good about this design

- Only ducks that can fly inherit the Flyable abstract class
- Only ducks that make noise inherit the Quackable abstract class
- Base Duck class functionality no longer need to be overridden because they are incompatible with the properties of some duck, helping maintenance

The bad about this design

- Lots of code duplication
  - Every duck that flies needs to implement a fly( ) *concrete* method
  - Code duplication will mean lots of cut-and-pasting during the initial development
  - Bugs will crop up in one version and not the other
  - Common bugs will need to be found and fixed across all versions
  - Code drift will make functionality change
  - This will be a maintenance nightmare
- Not all ducks implement all of the functionality
  - We cannot have a single Duck pointer (or reference) that can point to all kinds of ducks and perform all of the behavior of those ducks

```
class Duck {
public:
  void swim( );
};

void Duck::swim( ) {
  std::cout << "Paddle, paddle" << std::endl;
}

class Flyable {
public:
  virtual void fly( ) = 0;
};

class Quackable {
public:
  virtual void quack( )=0;
};
```

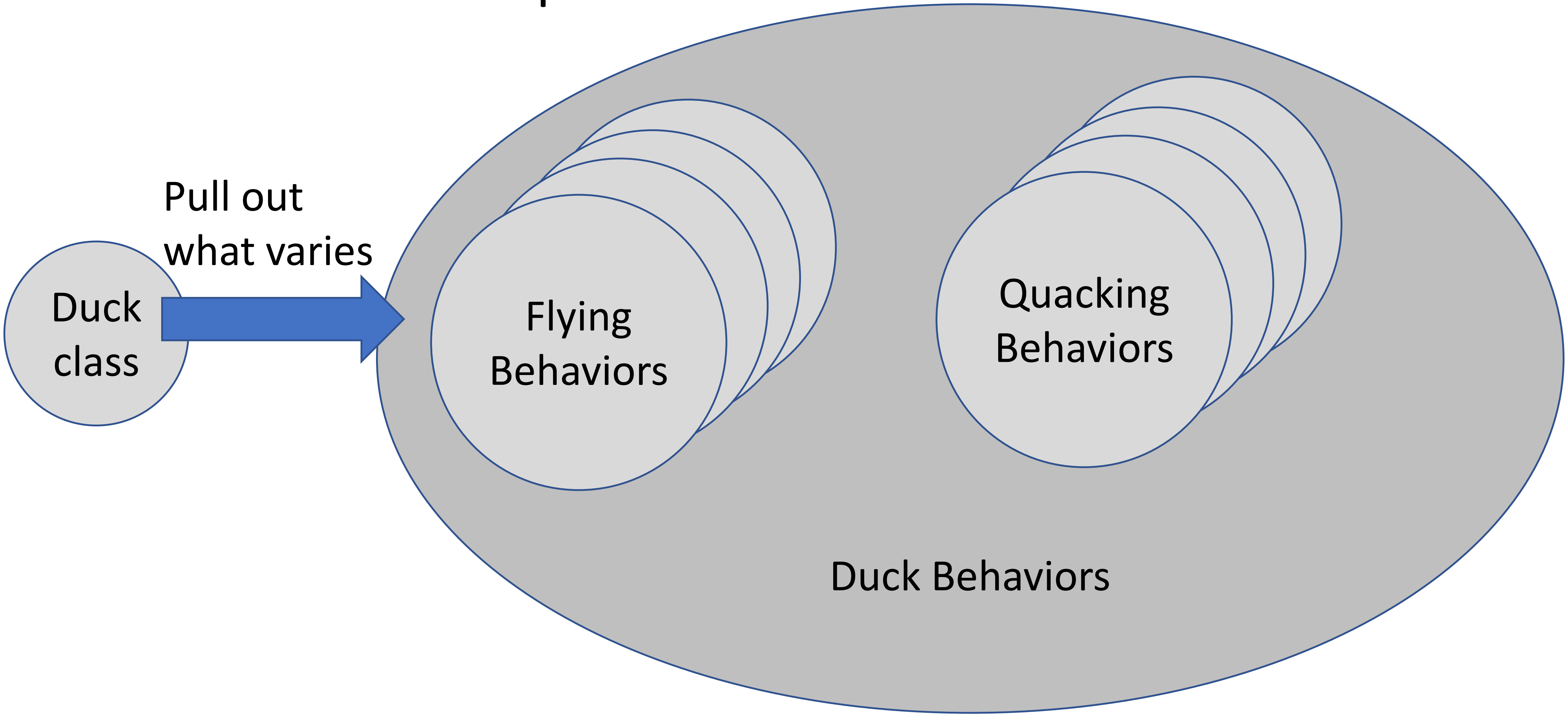
```
class RedHeadDuck : public Duck,
                    public Flyable, public Quackable {
  void fly( );
  void quack( );
};

void RedHeadDuck::fly( ) {
  std::cout << "I am flying!" << std::endl;
}

void RedHeadDuck::quack( ) {
  std::cout << "Quack, quack!" << std::endl;
}

int main(int argc, char *argv[]) {
  Duck duck = new RedHeadDuck( );
  // duck.fly( );
  // duck.quack( );
  duck.swim( );
}
```

See Example4 code

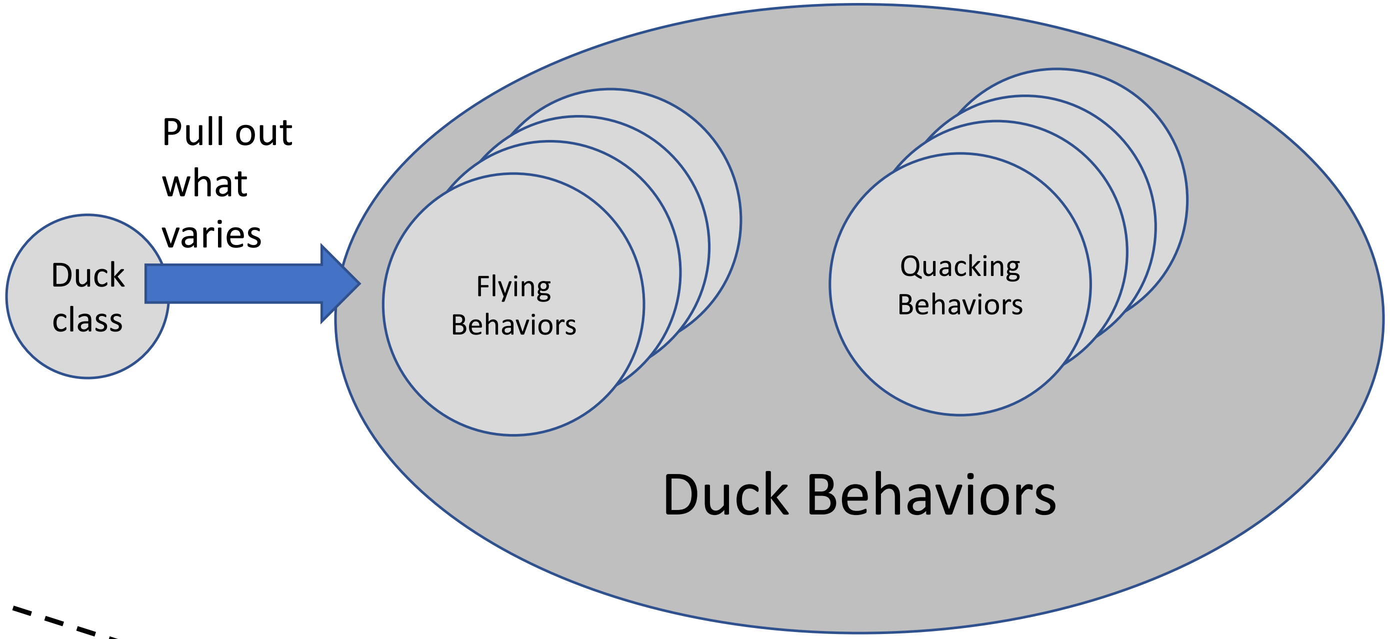
Let’s be guided by a *Design Principle*

- Identify the aspects of your application that vary, and separate them from what stays the same
- Stated differently: *take the parts that vary and encapsulate them, so that later you can alter or extend the parts that vary without affecting those that don’t*
- This forms the basis of most design patterns
    - Patterns provide a way to let some part of a system vary independently of other parts

Let’s do this separation



Let’s do this separation



FlyBehavior

fly( )

FlyWithWings

```
fly( ) {
  // implements duck flying
}
```

FlyNoWay

```
fly( ) {
  // do nothing – can’t fly
}
```

Design Principle

- *Program to an interface, not an implementation*
- We have the *Flyable* abstract class (interface), but FlyWithWings and FlyNoWay implementations
- Benefits of this design principle
  - We can assign any behavior that implements the interface to a specific duck
- We have one copy of the code that implements the behavior
  - This reduces code to be written and simplifies maintenance
- We can change the behavior at runtime by assigning a new behavior to the duck
- Previously we programmed to concrete implementations
  - Either the implementation in the super (base) class, or
  - An implementation within each duck that *implemented* a FlyBehavior or QuackBehavior

If you know Java . . .

- An interface is not necessarily a Java interface
- An interface is any base class such that we don’t hard code a specific implementation into our code
- Interfaces are often either Java interfaces or Java/C++ abstract classes
- Thus, for an interface, we can use
  - A Java interface (obviously, only when programming in Java)
  - An abstract class
  - A base class whose derived classes provide the concrete implementations

Consider the following code example

Animal

makeSound( )

Abstract supertype/Base class (either an abstract class or an interface)

Dog

```
class Dog : public Animal {
public:
  virtual void makeSound( );
private:
  virtual void bark( );
};

void Dog::makeSound( ) {
  bark( );
}

void Dog::bark( ) {
  std::cout << "barking
sound" << std::endl;
}
```

Cat

```
class Cat : public Animal {
public:
  virtual void makeSound( );
private:
  virtual void meow( );
};

void Cat::makeSound( ) {
  meow( );
}

void Cat::meow( ) {
  std::cout << "meowing
sound" << std::endl;
}
```

See Example5 code

Programming to an implementation

```
Dog* d = new Dog( );
d->bark( );
```

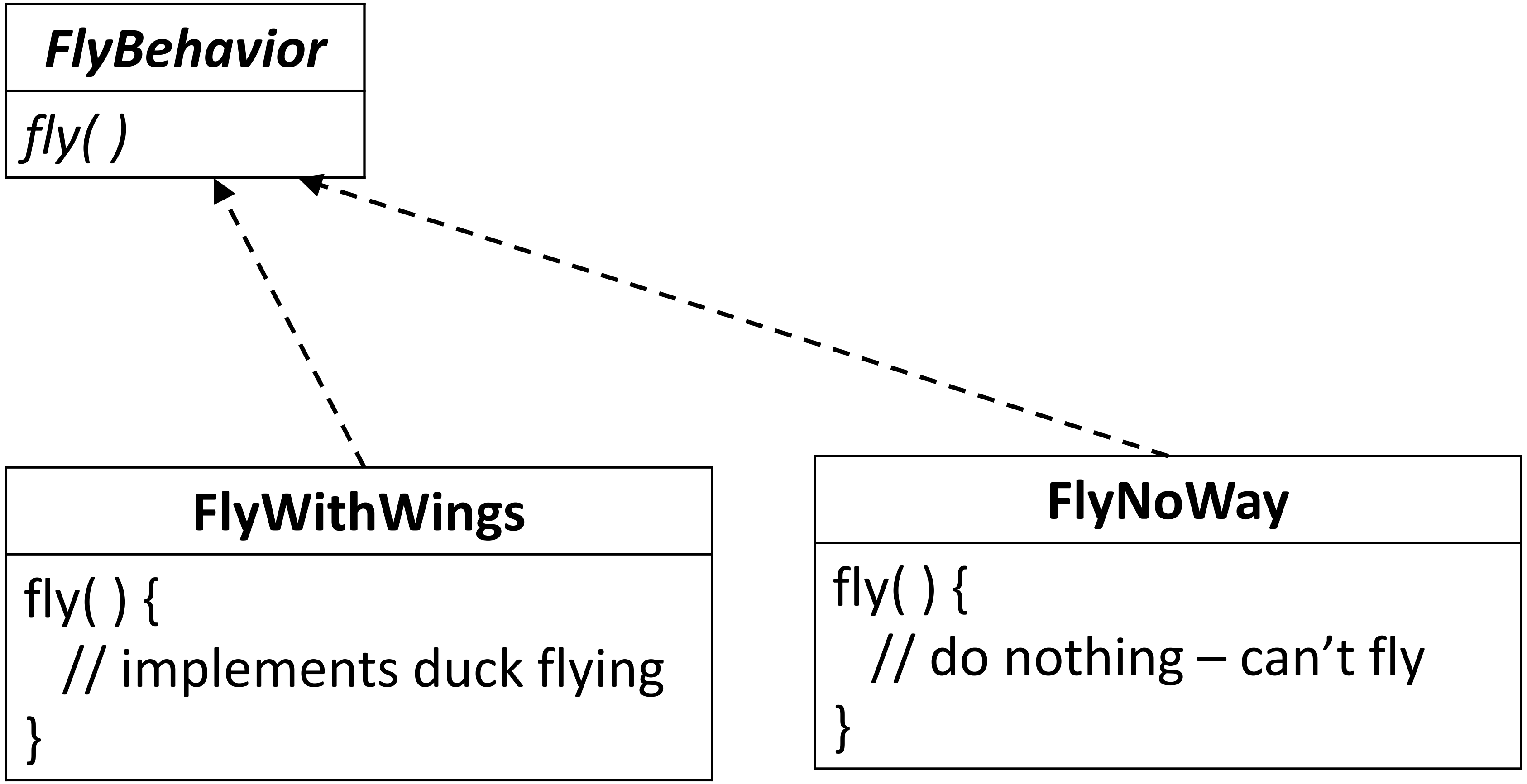
Programming to an interface or a supertype

```
Animal* animal = new Dog( );
animal->makeSounds( );
```

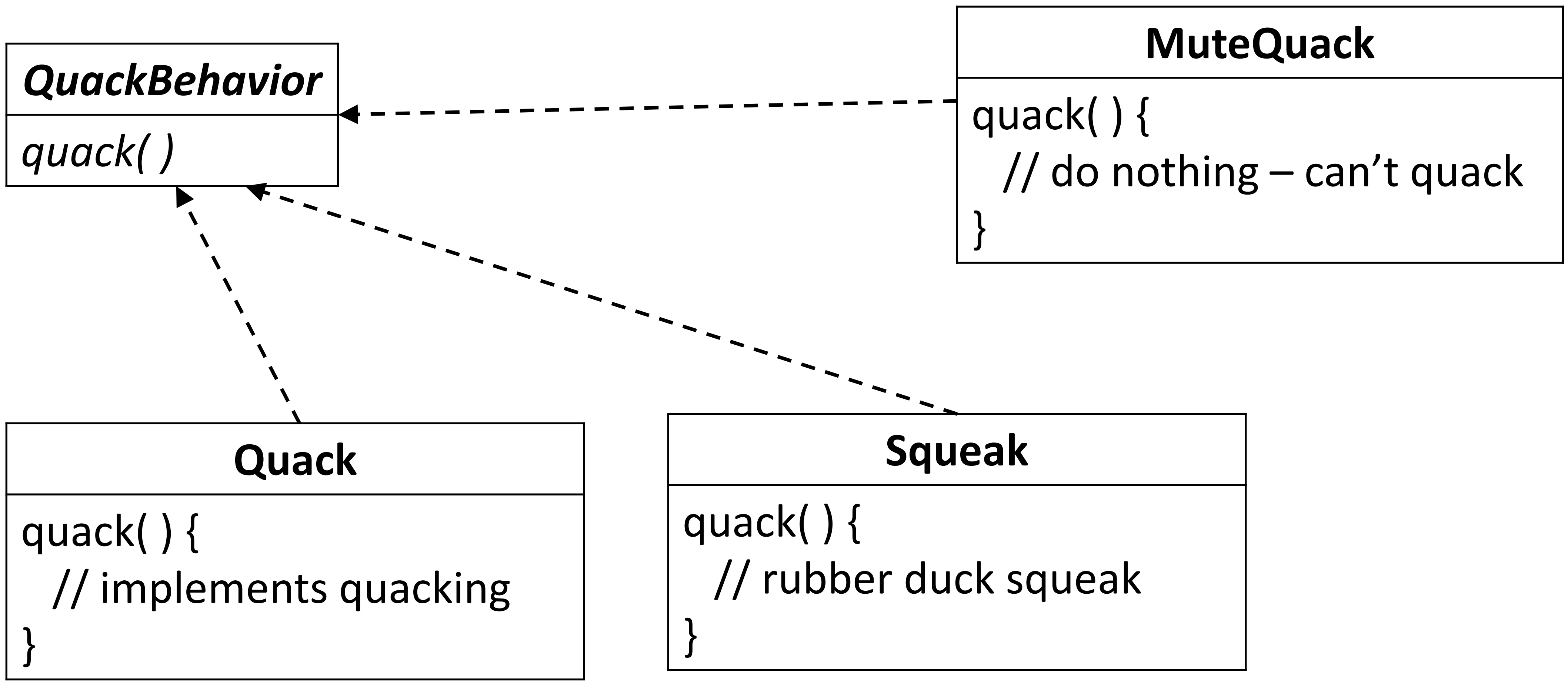
Or possibly even better

```
Animal* a = getAnimal( );
a->makeSound( );
```

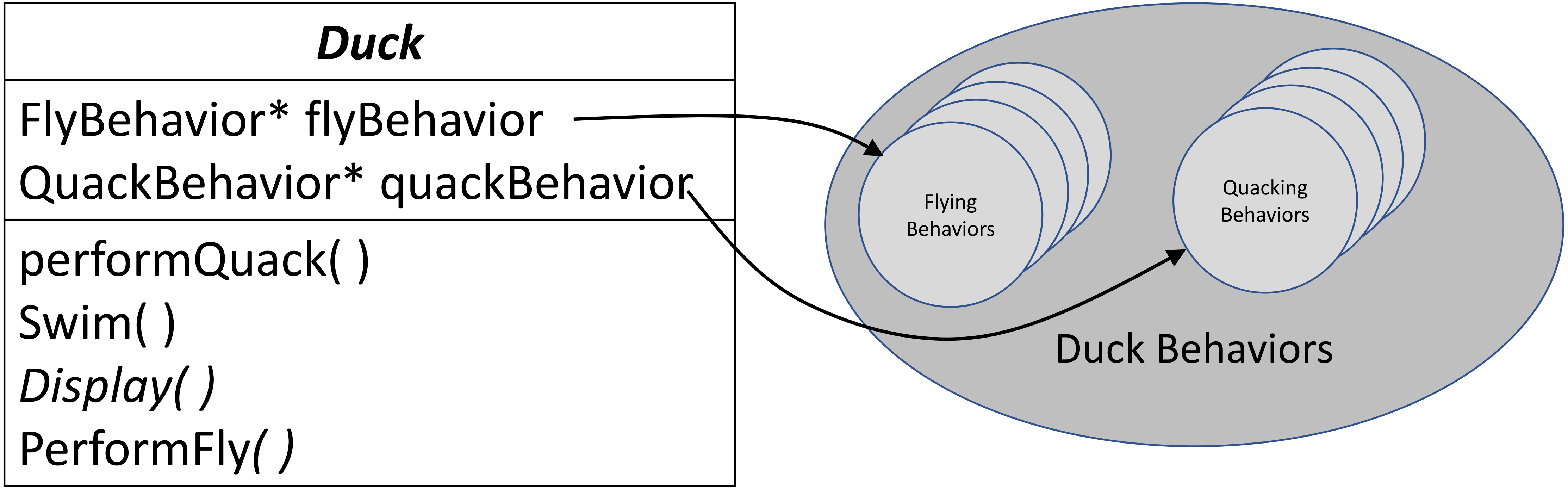
Implementing the Duck behaviors -- FlyBehavior



Implementing the Duck behaviors -- QuackBehavior



The new Duck class



```
class Duck {
public:
  Duck( );
  virtual void performQuack ( );
  virtual void performFly ( );
  // other Duck stuff
protected:
  QuackBehavior* quackBehavior;
  FlyBehavior* flyBehavior;
};

void Duck::performQuack ( ) {
  quackBehavior->quack( );
}

void Duck::performFly( ) {
  flyBehavior->fly( );
}
```

Via inheritance, every kind of duck (mallard, redhead, rubber, …) has a pointer to something that implements the QuackBehavior interface

The Duck object delegates the handling of a behavior to the object reference by quackBehavior

See Example6 code

```
class MallardDuck : public Duck {
public:
  MallardDuck( );
  void display( );
};

MallardDuck::MallardDuck( ) {
  quackBehavior = new Quack( );
  flyBehavior = new FlyWithWings( );
}

void MallardDuck::display( ) {
  std::cout << "I’m a real mallard duck" << std::endl;
}
```

We’re not completely adhering to our design principles, however

- We are still programming to an implementation
- Look at the lines:
  - quackBehavior = new Quack( );
  - flyBehavior = new FlyWithWings( );
- We’ll deal with this soon with a different pattern (the *Factory* pattern)

```
class Duck { // complete code in Example6
public:
  Duck( );
  virtual void setFlyBehavior(FlyBehavior*);
  virtual void setQuackBehavior(QuackBehavior*);
  virtual void performQuack ( );
  virtual void performFly ( );
  virtual void swim( );
  virtual void display( )=0;
protected:
  QuackBehavior* quackBehavior;
  FlyBehavior* flyBehavior;
};

Duck::Duck( ) { }

void Duck::swim( ) {
  std::cout << "All ducks float" << std::endl;
}
```

```
void Duck::performQuack ( ) {
  quackBehavior->quack( );
}

void Duck::performFly( ) {
  flyBehavior->fly( );
}

void Duck::setFlyBehavior(FlyBehavior* fb) {
  flyBehavior = fb;
}

void Duck::setQuackBehavior(QuackBehavior* qb) {
  quackBehavior = qb;
}
```

```cpp
class FlyBehavior {
public:
  virtual void fly( ) = 0;
};

void FlyWithWings::fly( ) {
  std::cout << "Fly, fly!" << std::endl;
}

void FlyNoWay::fly( ) {
  std::cout << "No can fly." << std::endl;
}
```

```cpp
class QuackBehavior {
public:
  virtual void quack( ) = 0;
};

class MuteQuack : public QuackBehavior {
public:
  void quack( );
};

void MuteQuack::quack( ) {
  std::cout << ". . ." << std::endl;
}
```

```
class DecoyDuck : public Duck {
public:
  DecoyDuck( );
  void display( );
};
DecoyDuck::DecoyDuck( ) {
  flyBehavior = new FlyNoWay( );
  quackBehavior = new MuteQuack( );
}

void DecoyDuck::display( ) {
  std::cout << "I’m a decoy duck" << std::endl;
}
```

HASA is often better than ISA

- Instead of inheriting behaviors from another class, we can *compose* a class with another class
- The HASA relation enables this
- Composition allows us to both
  - Encapsulate a set of algorithms in a set of classes
  - Change the algorithm that another class uses at runtime, as long as the algorithm implements the correct behavior interface

Design Principle

Favor composition over inheritance

Our First pattern

- With the Duck class and the associated subclasses and interfaces we've implemented the *Strategy* pattern
  - The strategy pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from the clients that use it
- Design principles followed in the Strategy pattern
  - Identify the aspects of your application that vary and separate them from what stays the same
  - Program to an interface, not an implementation
  - Favor composition over inheritance

Commentary

- Are patterns nothing more than good object oriented design?
  - It is often not obvious what the right way to do things is
  - The best practices have often been discovered by hard work and trial and error
  - Patterns hope to capture these best practices to solve a problem, but are not just best practices
- If you cannot find a pattern to do what you want to do, apply the design principles we have seen and you will learn

Summary

- Knowing OO basics and an OO language doesn’t make you a good OO programmer
  - But it’s hard to be a good OO programmer if you don’t know the language, so that is an important fundamental – it’s just not the whole story
- Good OO designs are reusable, extensible and maintainable
- Patterns show how to build good OO designs
- Patterns and design principles address change in software systems
- Most patterns allow some parts of a system to vary independently of the other parts
  - Often this is done by encapsulating the various parts
- Patterns provide a shared language for communicating with other programmers – saying you use the Strategy pattern provides high level information