Design Patterns

Strategy Pattern*

How to design for flexibility?

ebru@hacettepe.edu.tr

ebruakcapinarsezer@gmail.com

http://yunus.hacettepe.edu.tr/~ebru/

@ebru176

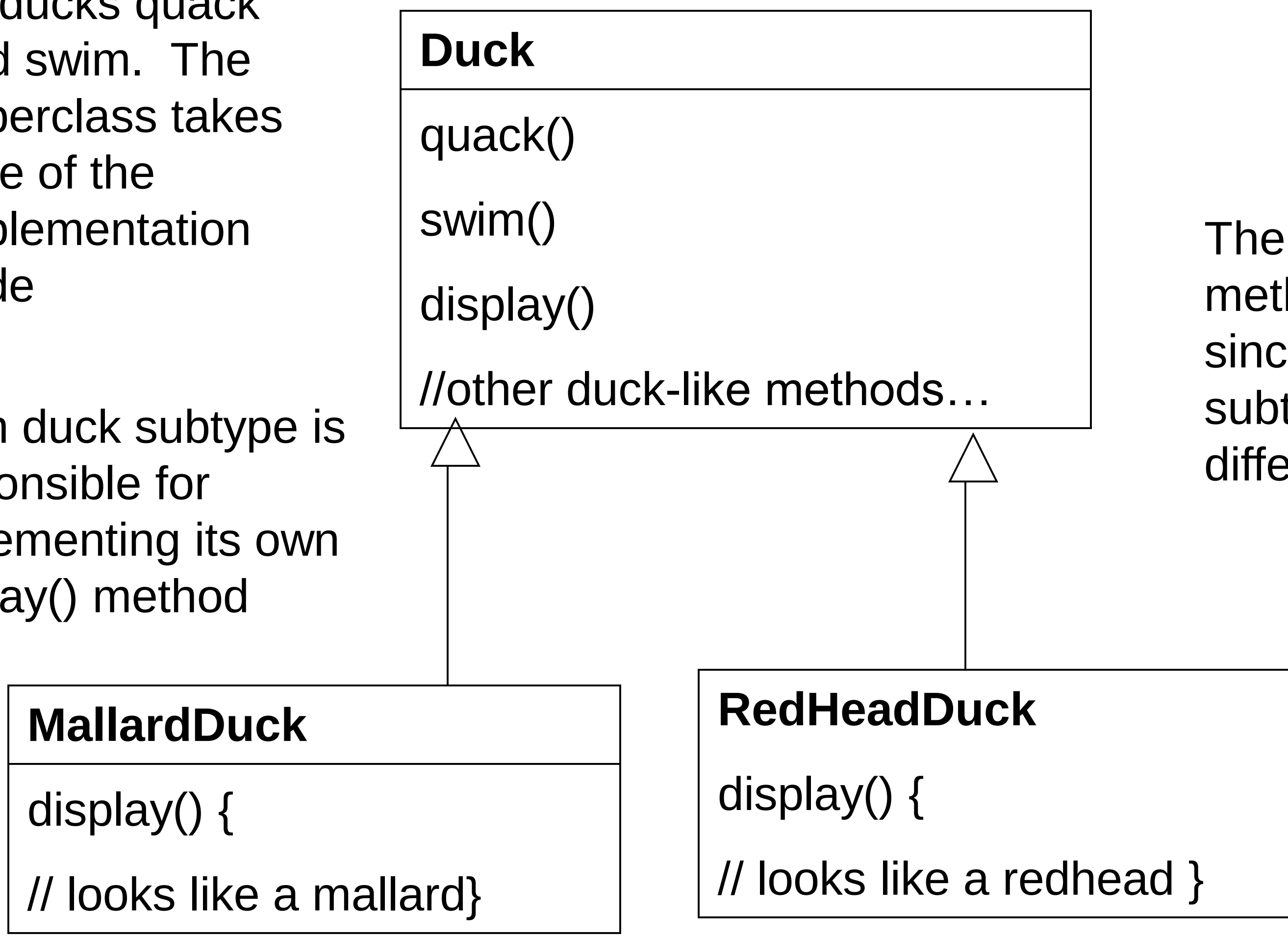
Ekim 2017

revised from, www.uwosh.edu/faculty_staff/huen/262/f09/slides/10_ Strategy_Pattern**.ppt*

Existing Duck application

All ducks quack and swim. The superclass takes care of the implementation code

Each duck subtype is responsible for implementing its own display() method

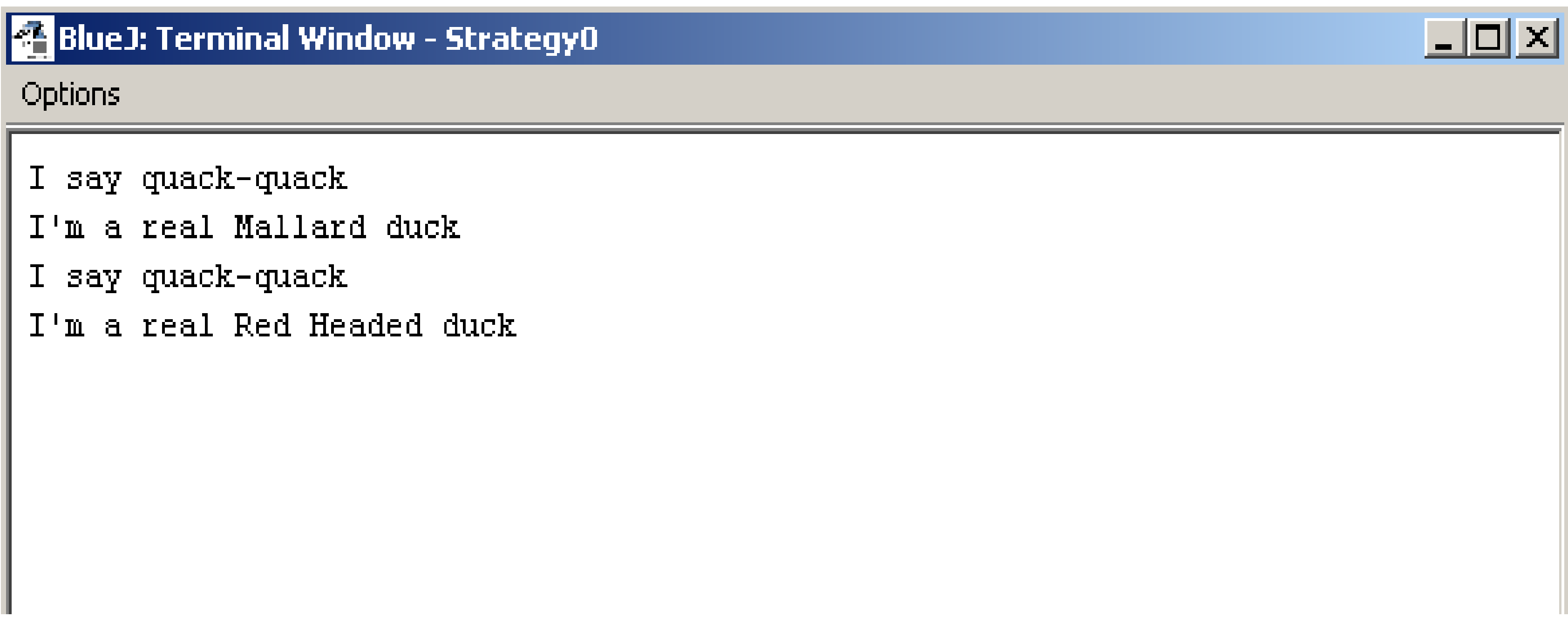


The display() method is abstract, since all duck subtypes look different

Other duck types inherit from the Duck class

. . .

Testing Mallard, RedHeadDuck classes

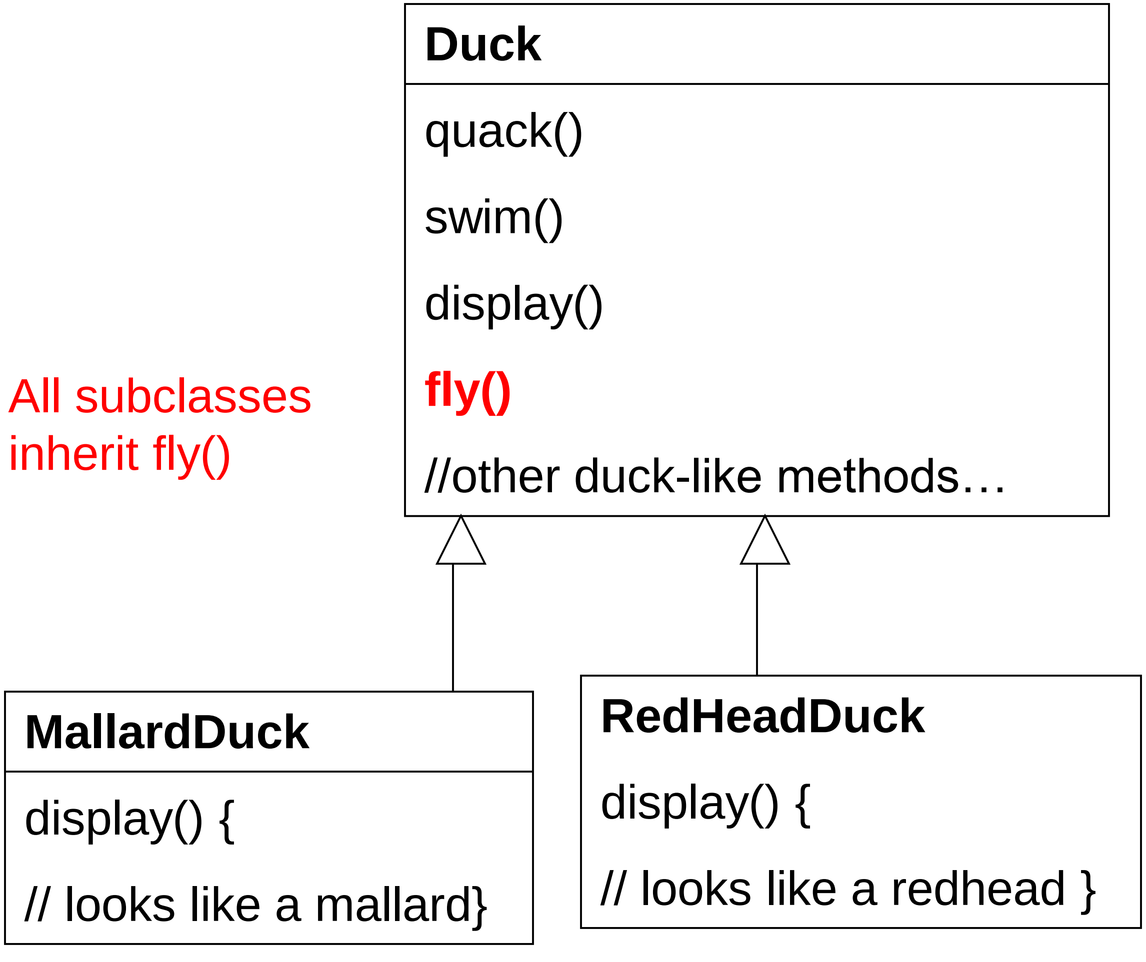


Changing Requirment

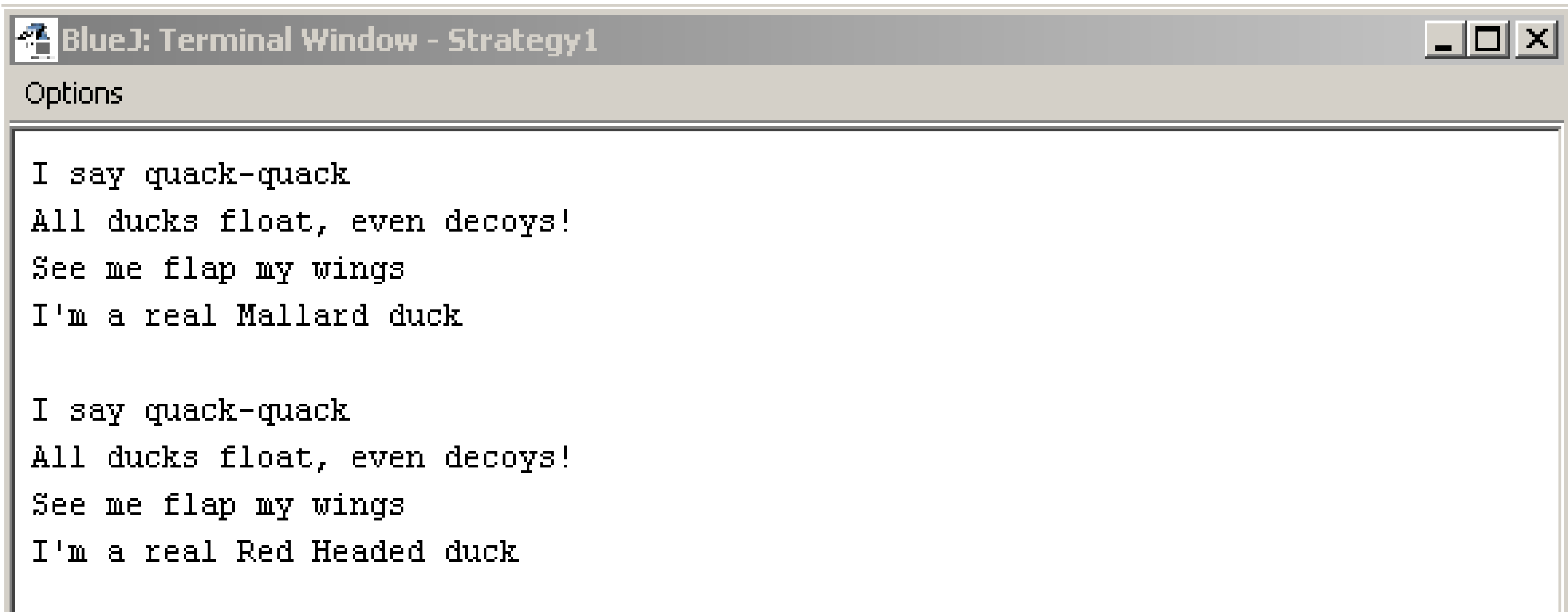
- No sweat!
  - Add a method fly() in Duck
  - Continue to use inheritance

Add a method fly() in Duck

All subclasses
inherit fly()

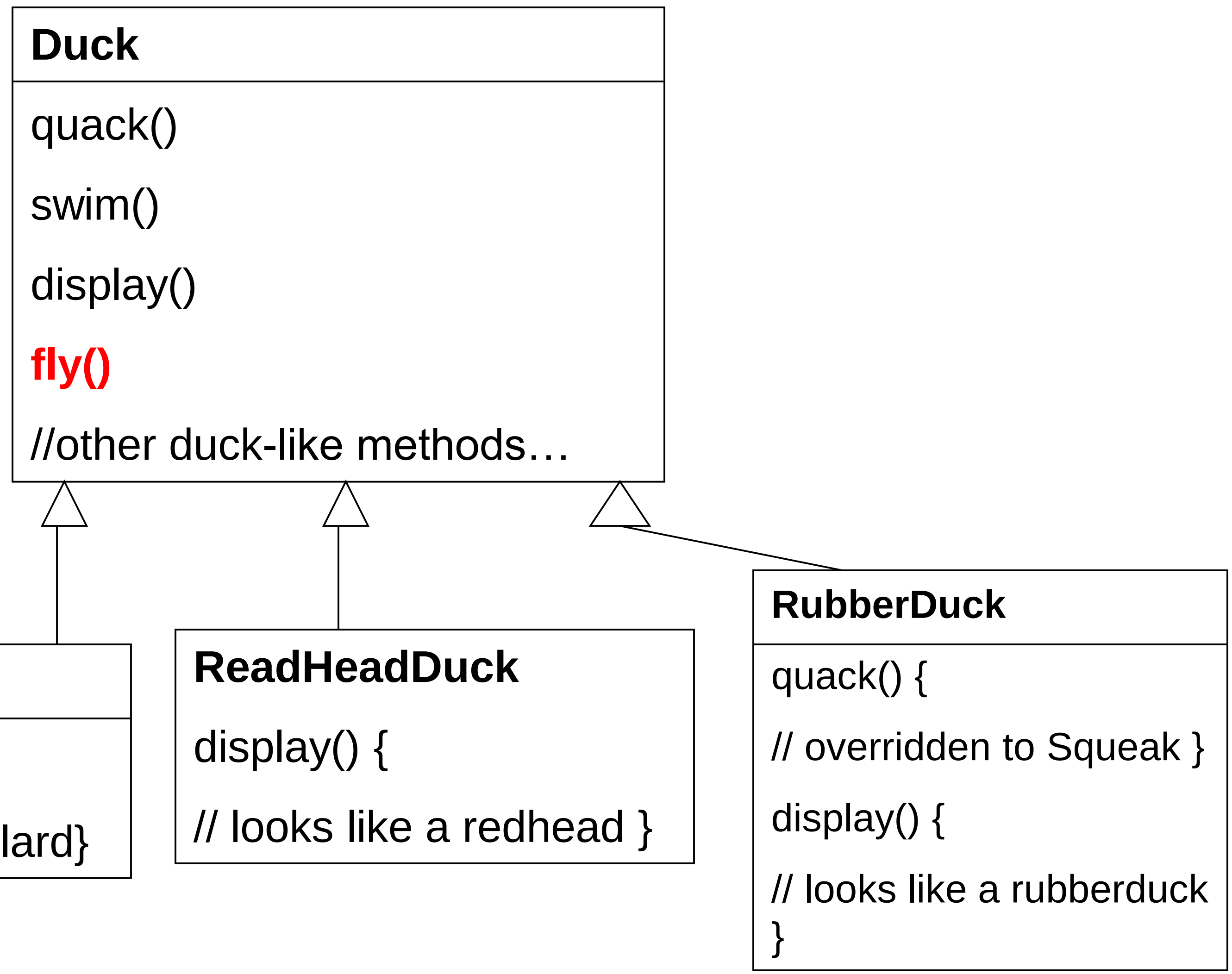


Executing


BlueJ: Terminal Window - Strategy1
Options
I say quack-quack
All ducks float, even decoys!
See me flap my wings
I'm a real Mallard duck
I say quack-quack
All ducks float, even decoys!
See me flap my wings
I'm a real Red Headed duck

Something seriously wrong!

All duck types now can fly including RubberDuck



Executing … What?

BlueJ: Terminal Window - Strategy2

Options

```
I say quack-quack
All ducks float, even decoys!
See me flap my wings
I'm a real Mallard duck

I say quack-quack
All ducks float, even decoys!
See me flap my wings
I'm a real Red Headed duck

I say squeak-squeak
All ducks float, even decoys!
See me flap my wings
I'm a rubber duckie
```

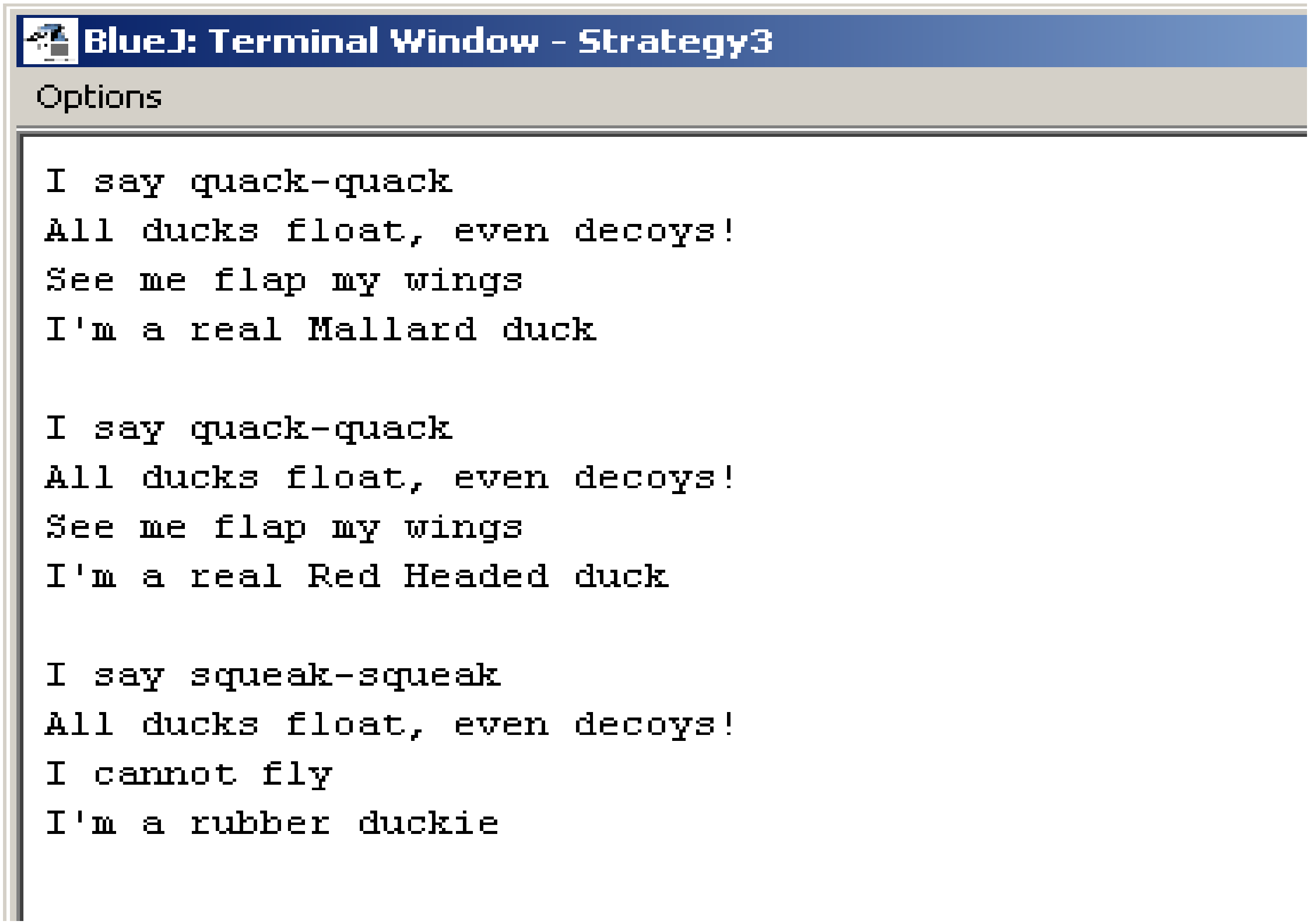
Root cause?

- Applying inheritance to achieve re-use
- Poor solution for maintenance

How do we fix this?

- Using inheritance as before
  - Override the fly() method in rubber duck as in quack()

Executing


BlueJ: Terminal Window - Strategy3
Options
I say quack-quack
All ducks float, even decoys!
See me flap my wings
I'm a real Mallard duck
I say quack-quack
All ducks float, even decoys!
See me flap my wings
I'm a real Red Headed duck
I say squeak-squeak
All ducks float, even decoys!
I cannot fly
I'm a rubber duckie

Is the problem solved?

- Any new problems?

Wait a minute

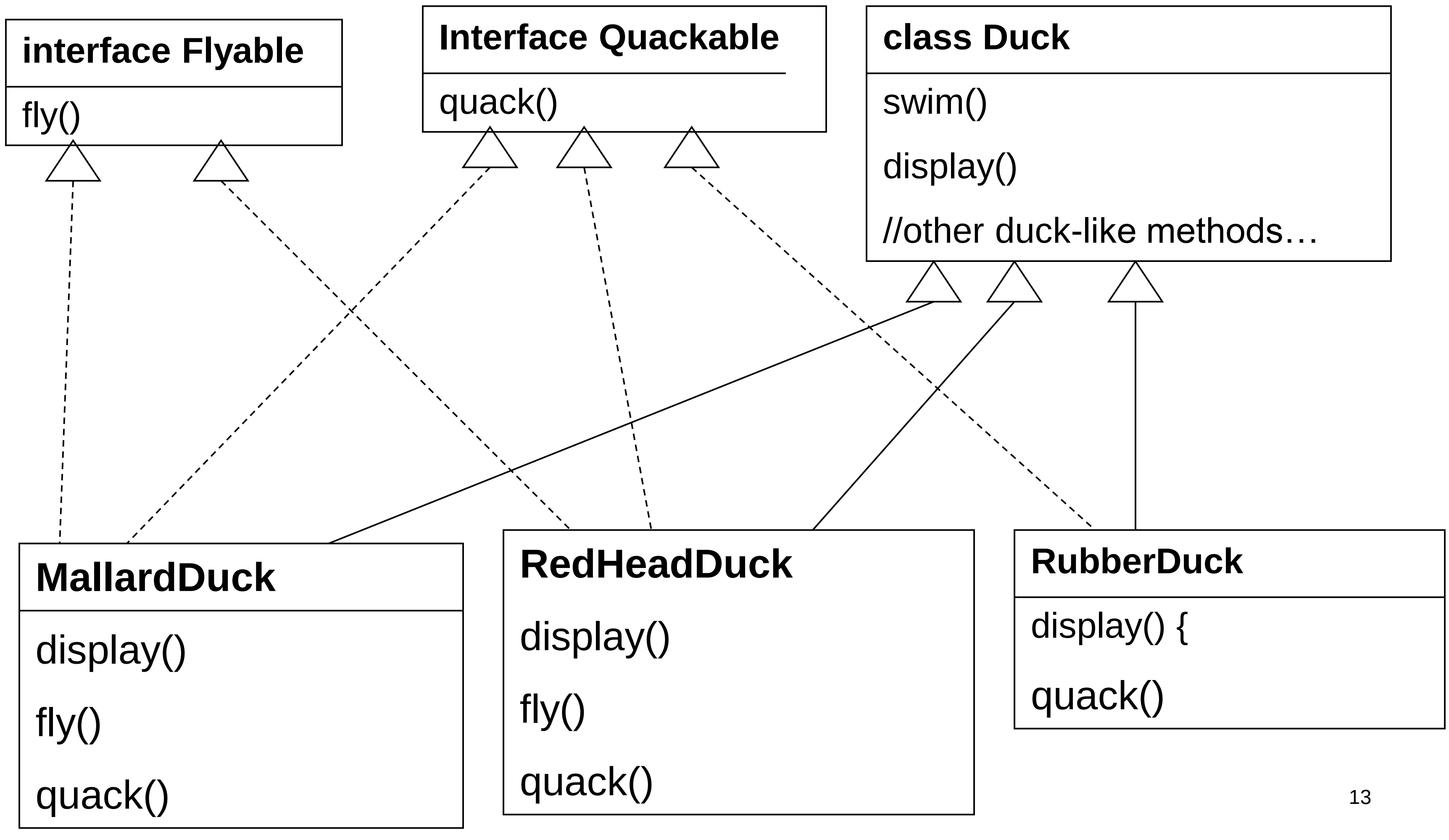
- How about new duck types?
  - Decoy duck?
    - Can’t quack
    - Can’t fly
- How do we solve it?

Summary

- What have we done so far?
- What problems have we solved?
- What problems have we introduced in solving the problems?
- Is there a better way of doing things?

How about Interface?

- Take the fly() method out of Duck superclass
- And make a Flyable() interface
  - Only those ducks that fly are required to implement the interface
- Make a Quackable interface too



But

- You shoot yourself in the foot by <u>duplicating code</u> for every duck type that can fly and quack!
- And we have a lot of duck types
- We have to be careful about the properties – we cannot just call the methods blindly
- We have created a maintenance nightmare!

Re-thinking:

- Inheritance has not worked well because
  - Duck behavior keeps changing
  - Not suitable for all subclasses to have those properties

- Interface was at first promising, but
  - No code re-use
  - Tedious
    - Every time a behavior is changed, you must track down and change it in all the subclasses where it is defined
  - Error prone

#1 Design Principle

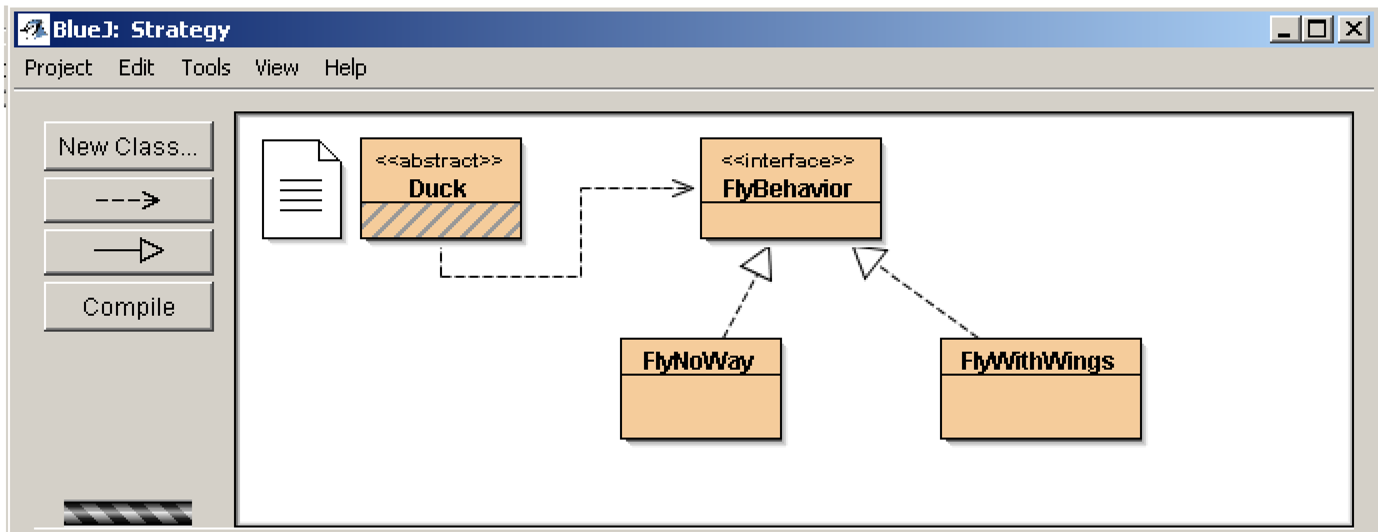
- Identify the aspects of your application that vary and separate them from what stays the same
- So what are variable in the Duck class?
  - Flying behavior
  - Quacking behavior
- Pull these duck behaviors out of the Duck class
  - Create new classes for these behaviors

How do we design the classes to implement the fly and quack behaviors?

- Goal: to keep things flexible
- Want to assign behaviors to instances of Duck
  - Instantiate a new MallardDuck instance
  - Initialize it with a specific type of flying
  - Be able to change the behavior dynamically

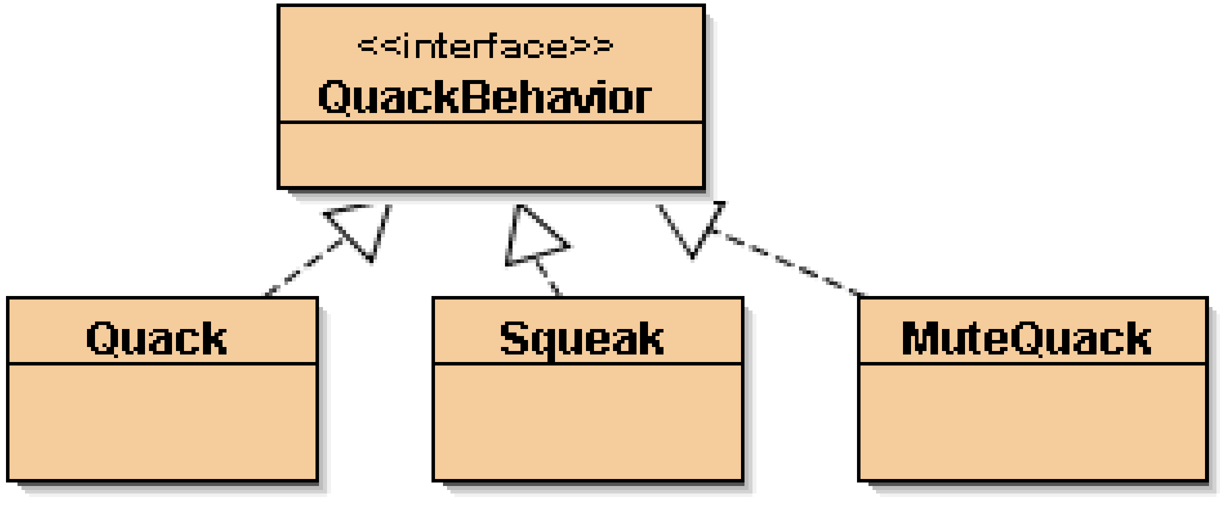
#2 Design Principle

- Program to a supertype, not an implementation
- Use a supertype to represent each behavior
  - FlyBehavior and QuackBehavior
  - Each implementation of a behavior will implement one of these supertypes
- In the past, we rely on an implementation
  - In superclass Duck, or
  - A specialized implementation in the subclass
- Now: Duck subclass will use a behavior represented in a supertype.


BlueJ: Strategy
Project
Edit
Tools
View
Help
New Class...
Compile
<<abstract>>
Duck
<<interface>>
FlyBehavior
FlyNoWay
FlyWithWings

3 classes in code

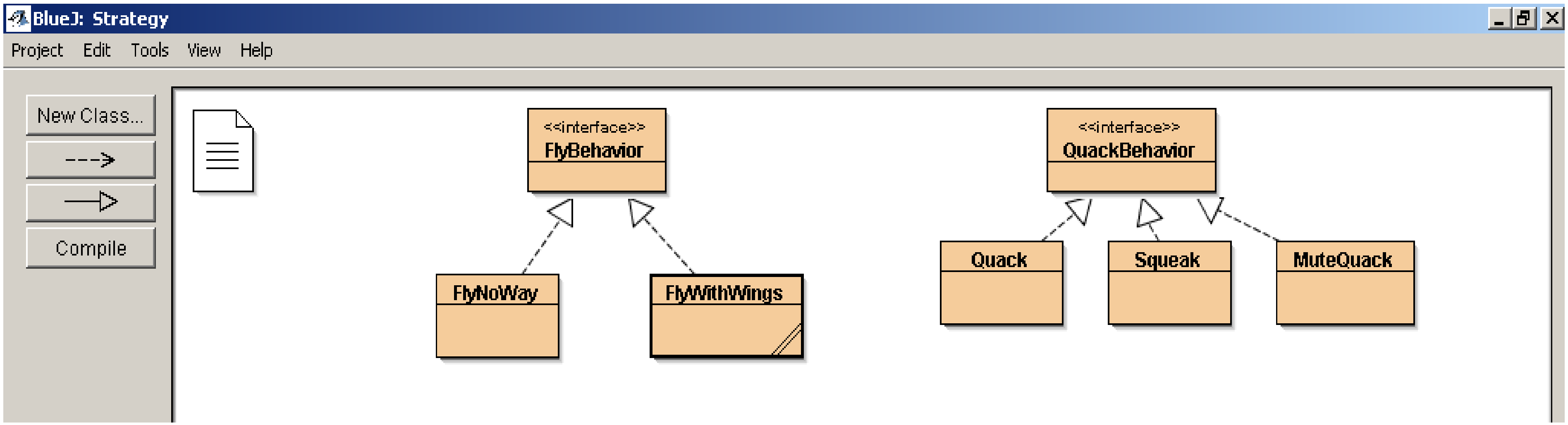
```
public interface FlyBehavior {
   public void fly();
}
----------------------------------------------------------------------------------------------------
public class FlyWithWings implements FlyBehavior {
   public void fly() {
         System.out.println("I'm flying!!");
   }
}
----------------------------------------------------------------------------------------------------
public class FlyNoWay implements FlyBehavior {
   public void fly() {
         System.out.println("I can't fly");
   }
}
```




```
public interface QuackBehavior {
        public void quack();
}
```

Specific behaviors by implementing interface QuackBehavior

```
public class Quack implements QuackBehavior {
   public void quack() {
            System.out.println("Quack");
   }
}
------------------------------------------------------------------------
public class Squeak implements QuackBehavior {
   public void quack() {
            System.out.println("Squeak");
   }
}
-------------------------------------------------------------------------
public class MuteQuack implements QuackBehavior {
   public void quack() {
            System.out.println("<< Silence >>");
   }
}
```


BlueJ: Strategy
Project
Edit
Tools
View
Help
New Class...
Compile
<<interface>>
FlyBehavior
<<interface>>
QuackBehavior
FlyNoWay
FlyWithWings
Quack
Squeak
MuteQuack

Integrating the Duck Behavior

1. Add 2 instance variables:

Behavior variables are declared as the behavior SUPERTYPE

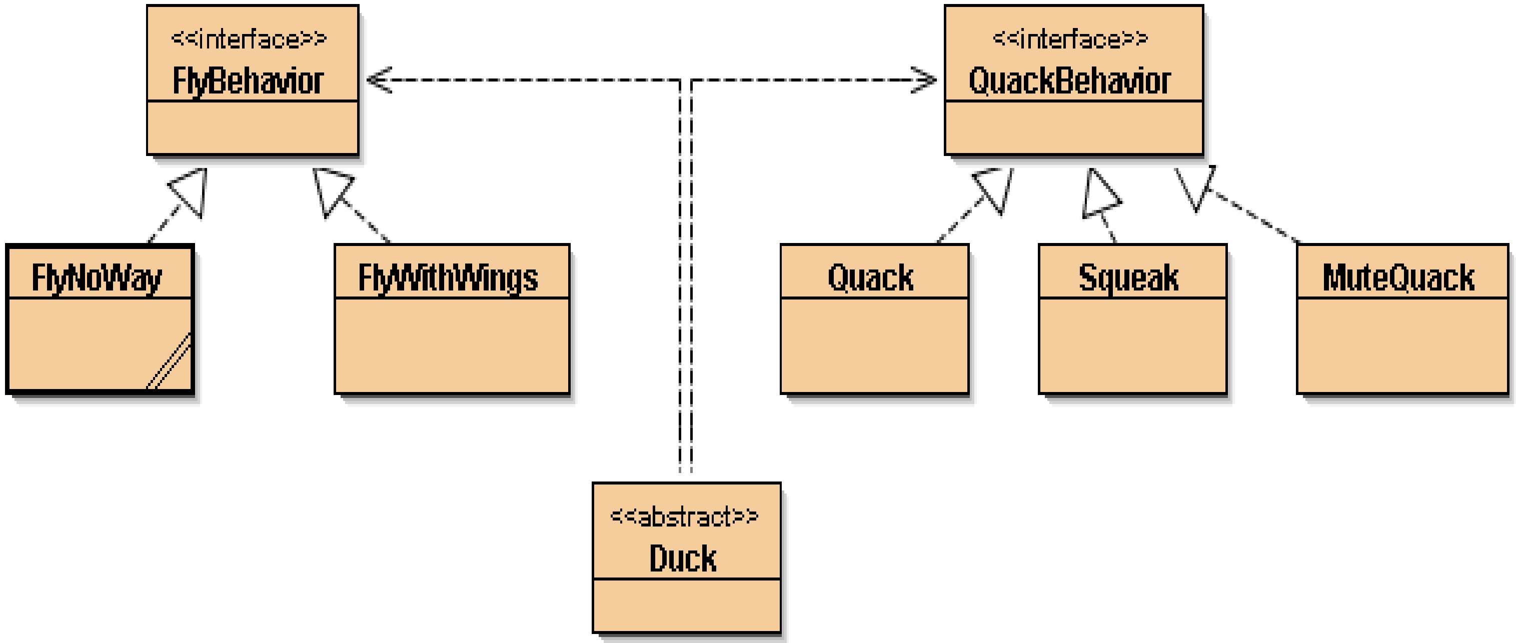
Duck
FlyBehavior flyBehavior **QuackBehavior quackBehavior**
performQuack() Swim() Display() **performFly()** //OTHER duck-like methods

Instance variables hold a reference to a specific behavior at runtime

These general methods replace fly() and quack()

2. Implement performQuack()

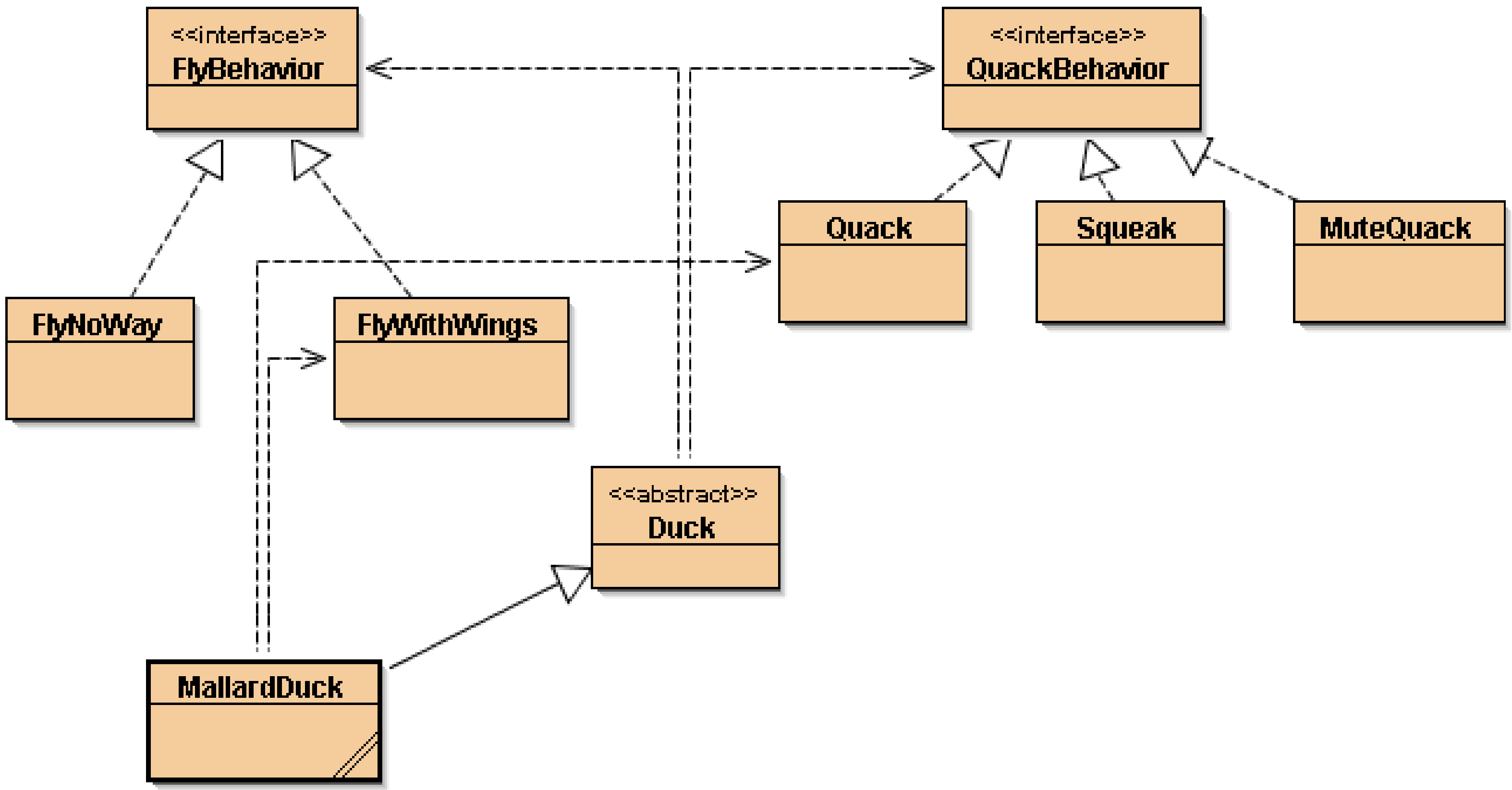
```
public abstract class Duck {
    // Declare two reference variables for the behavior interface types
    FlyBehavior flyBehavior;
    QuackBehavior quackBehavior; // All duck subclasses inherit these
    // etc

    public Duck(FlyBehavior f, QuackBehavior q) {
    }

    public Duck() {
    }

    public void performQuack() {
        quackBehavior.quack();  // Delegate to the behavior class
    }
```



<<interface>>
FlyBehavior
<<interface>>
QuackBehavior
FlyNoWay
FlyWithWings
Quack
Squeak
MuteQuack
<<abstract>>
Duck

3. How to set the quackBehavior variable & flyBehavior variable

```
public class MallardDuck extends Duck {

   public MallardDuck() {

      quackBehavior = new Quack();
            // A MallardDuck uses the Quack class to handle its quack,
            // so when performQuack is called, the responsibility for the quack
            // is delegated to the Quack object and we get a real quack

      flyBehavior = new FlyWithWings();
            // And it uses flyWithWings as its flyBehavior type

   }

   public void display() {
         System.out.println("I'm a real Mallard duck");
   }
```


<<interface>>
FlyBehavior
<<interface>>
QuackBehavior
Quack
Squeak
MuteQuack
FlyNoWay
FlyWithWings
<<abstract>>
Duck
MallardDuck

Testing the Duck code

Type and compile:

- Duck class and the MallardDuck class
- FlyBehavior interface and the two behavior implementation classes (FlyWithwings.java and flyNoWay.java)
- QuackBehavior interface and 3 behavior implementation classes
- Test class (MiniDuckSimulator.java)

// 1. Duck class

```
public abstract class Duck {
    // Reference variables for the behavior interface types
    FlyBehavior flyBehavior;
    QuackBehavior quackBehavior; // All duck subclasses inherit these

    public Duck() { }

   abstract void display();

   public void performFly() {
            flyBehavior.fly();      // Delegate to the behavior class
    }

    public void performQuack() {
        quackBehavior.quack();  // Delegate to the behavior class
    }

    public void swim() {
        System.out.println("All ducks float, even decoys!");
    }
```

Is it possible to manage all duck's sub-object with this super type?

2. FlyBehavior and two behavior implementation classes

```
public interface FlyBehavior {
   public void fly();
}
-----------------------------------------------------------------------------------------------
public class FlyWithWings implements FlyBehavior {
   public void fly() {
          System.out.println("I'm flying!!");
   }
}
-----------------------------------------------------------------------------------------------
public class FlyNoWay implements FlyBehavior {
   public void fly() {
          System.out.println("I can't fly");
   }
}
```

// 3. QuackBehavior interface and 3 behavior implementation classes

```
public interface QuackBehavior {
    public void quack();
}
```

```
public class Quack implements QuackBehavior {
    public void quack() {
            System.out.println("Quack");
    }
}
```

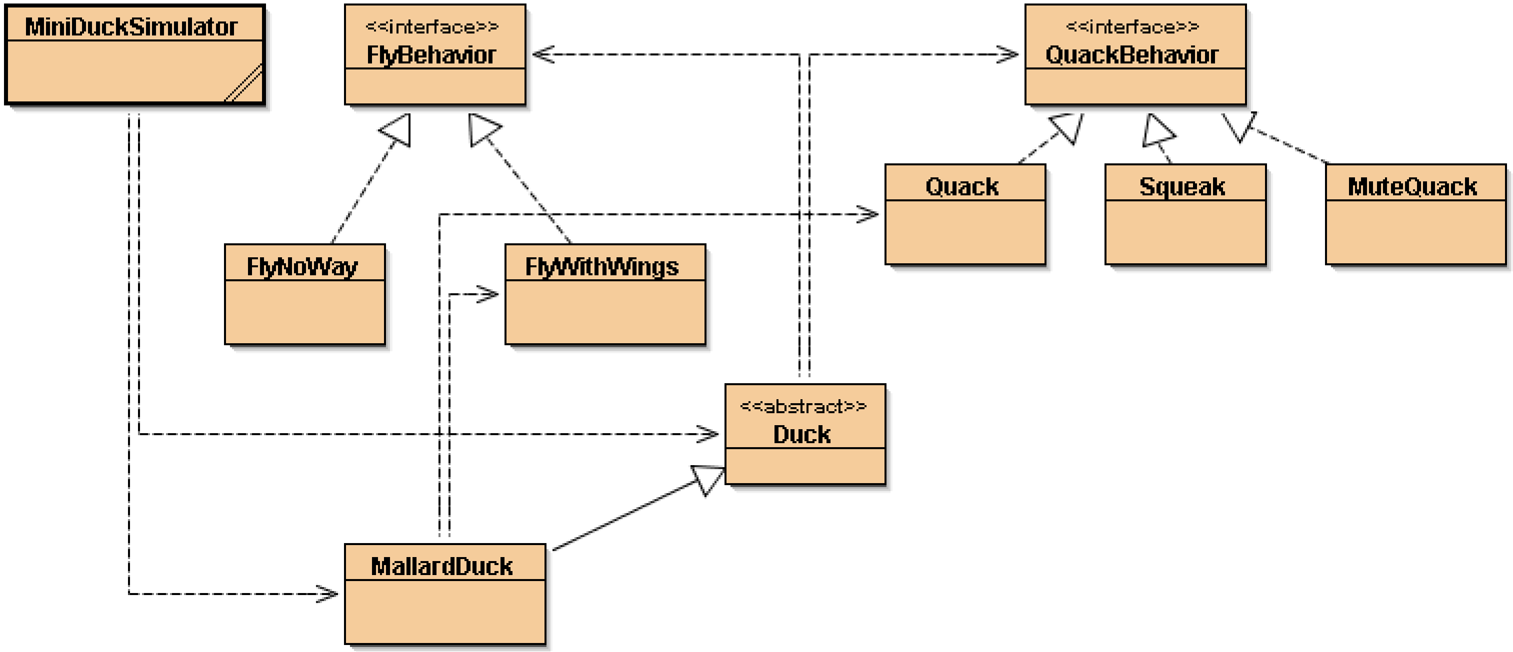
```
public class Squeak implements QuackBehavior {
    public void quack() {
            System.out.println("Squeak");
    }
}
```

```
public class MuteQuack implements QuackBehavior {
    public void quack() {
            System.out.println("<< Silence >>");
    }
}
```

4. Type and compile the test class (MiniDuckSimulator.java)

```
public class MiniDuckSimulator {

    public static void main(String[] args) {

        Duck    mallard = new MallardDuck();
        mallard.performQuack();
          // This calls the MallardDuck's inherited performQuack() method,
          // which then delegates to the object's QuackBehavior
          // (i.e. calls quack() on the duck's inherited quackBehavior
          //  reference)
        mallard.performFly();
           // Then we do the same thing with MallardDuck's inherited
           // performFly() method.
    }
}
```

At the end: Strategy project



Check-in

- We have built dynamic behavior in ducks e.g. a MallardDuck
  - The dynamic behavior is instantiated in the duck's constructor
- How can we change the duck's behavior after instantiation?

Changing a duck's behavior after instantiation

- Set the duck's behavior type through a mutator method on the duck's subclass

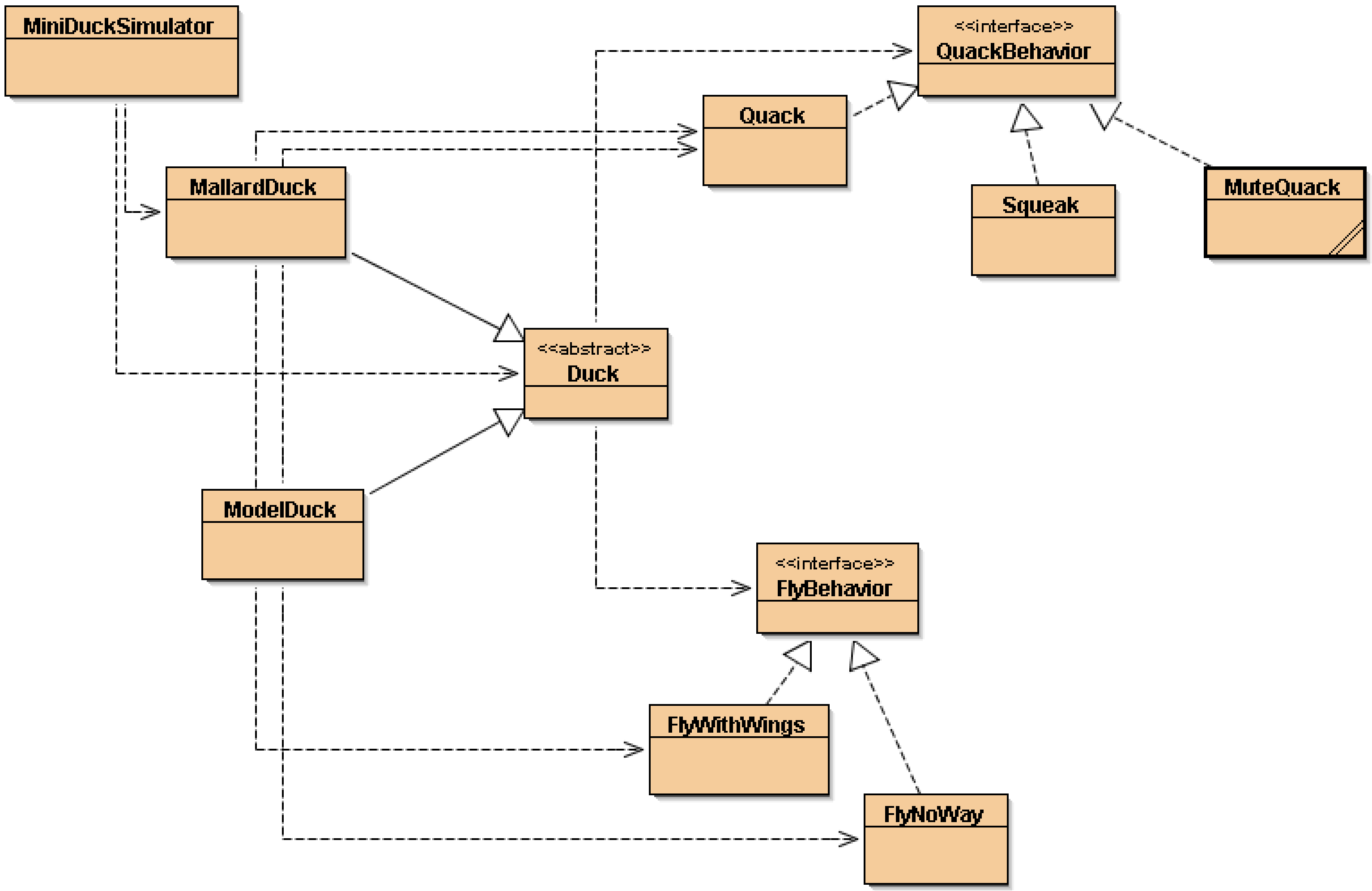
How to set behavior dynamically?

1. Add new methods to the Duck class

```
public void setFlyBehavior (FlyBehavior fb) {
    flyBehavior = fb;
}

public void setQuackBehavior(QuackBehavior qb) {
    quackBehavior = qb;
}
```

2. Make a new Duck type (ModelDuck.java)

```
public class ModelDuck extends Duck {
  public ModelDuck() {
        flyBehavior = new FlyNoWay();
          // Model duck has no way to fly
        quackBehavior = new Quack();
  }

  public void display() {
        System.out.println("I'm a model duck");
  }
}
```


MiniDuckSimulator
<<interface>>
QuackBehavior
Quack
MallardDuck
Squeak
MuteQuack
<<abstract>>
Duck
ModelDuck
<<interface>>
FlyBehavior
FlyWithWings
FlyNoWay

Enabling ModelDuck to fly

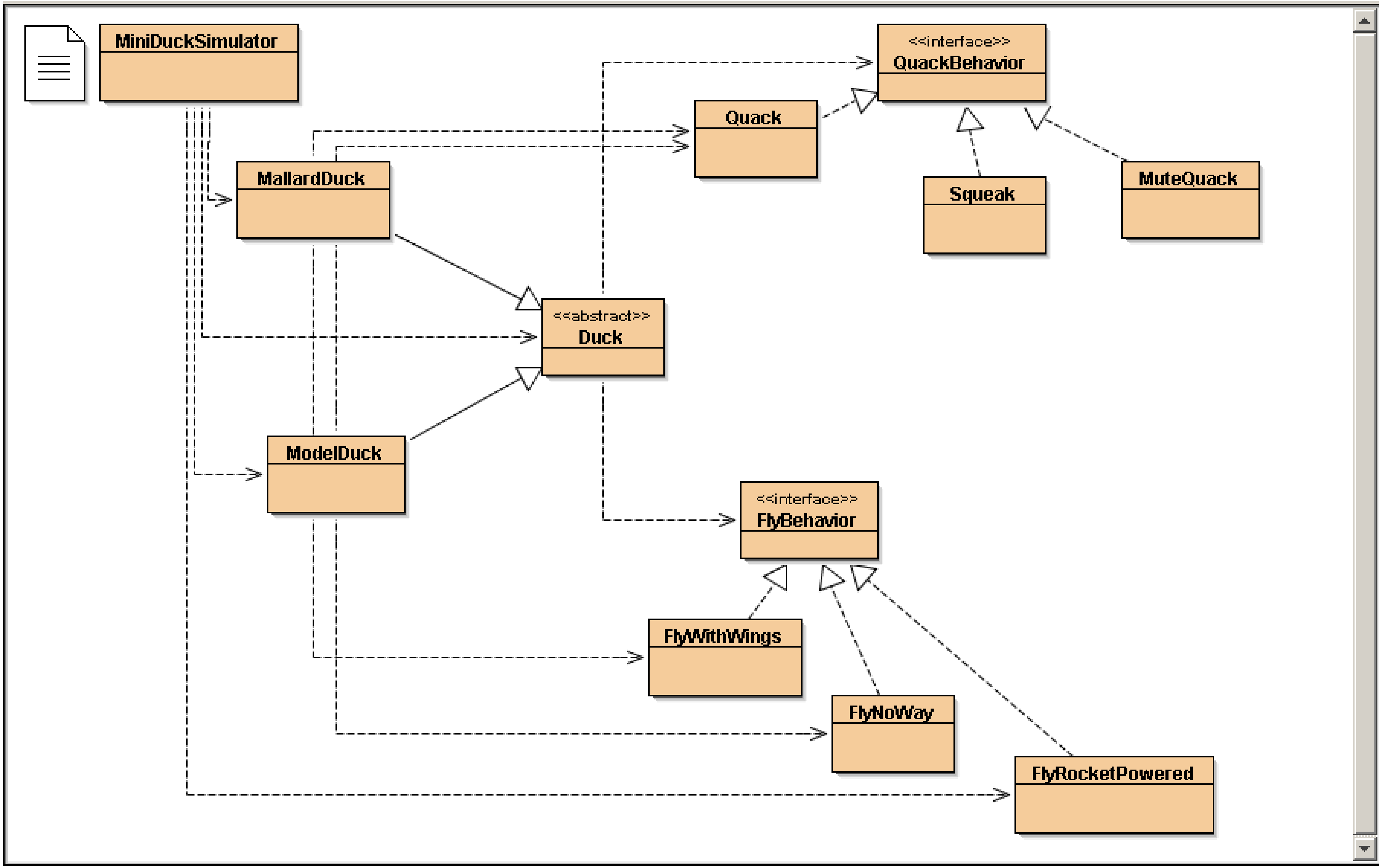
- Use a mutator (setter) method to enable ModelDuck to fly

3. Make a new FlyBehavior type (FlyRocketPowered.java)

```
public class FlyRocketPowered implements FlyBehavior {

  public void fly() {
        System.out.println("I'm flying with a rocket");
  }
}
```

4. Change the test class (MiniDuckSimulator.java), add the ModelDuck, and make the ModelDuck rocket-enabled

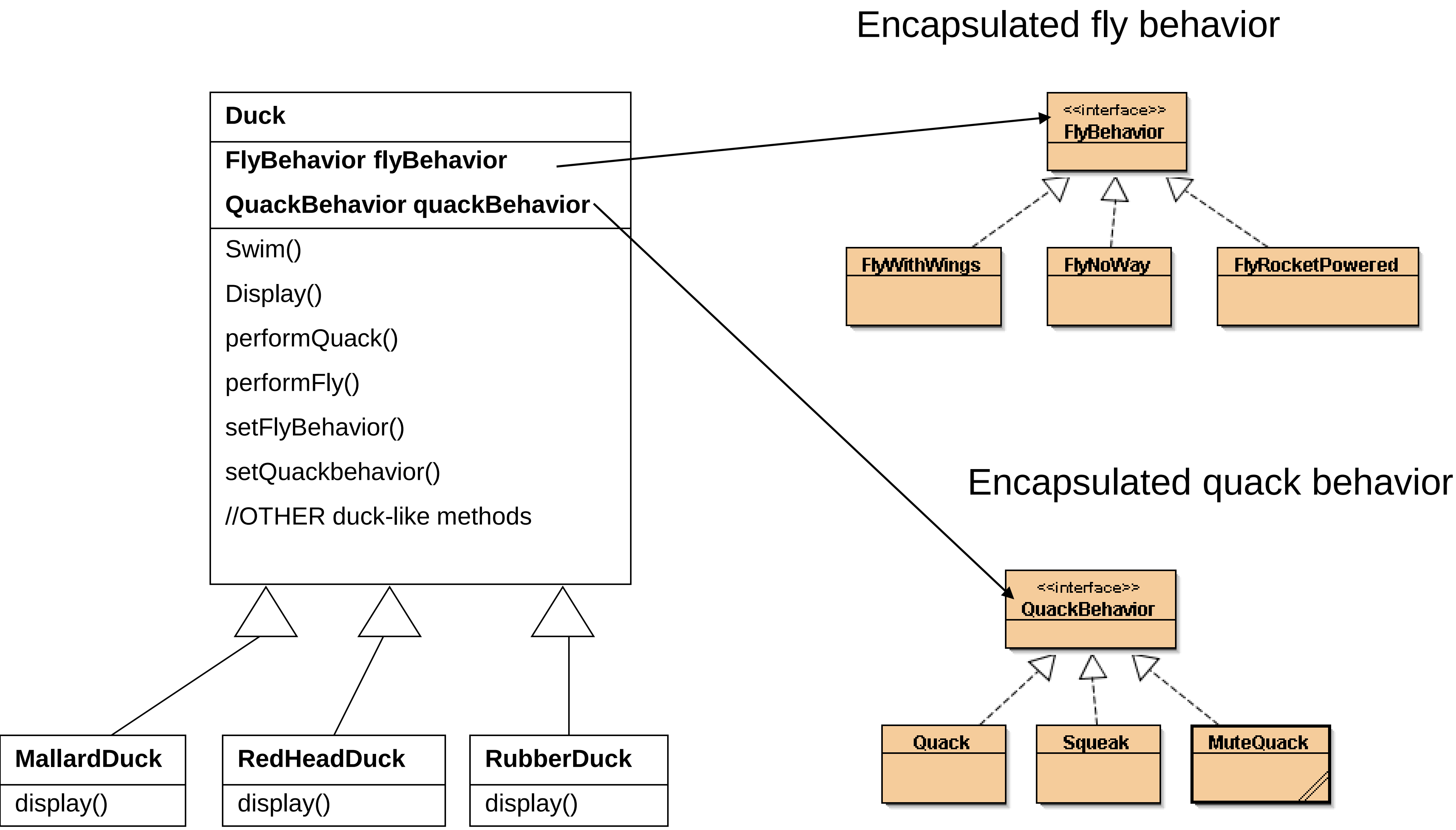
```
Duck    model = new ModelDuck();
    model.performFly();
        // call to performFly() delegates to the flyBehavior
        // object set in ModelDuck's constructor
    model.setFlyBehavior(new FlyRocketPowered());
       // change the duck's behavior at runtime by
       // invoking the model's inherited behavior setter
       // method
    model.performFly();
```



MiniDuckSimulator
<<interface>>
QuackBehavior
Quack
MallardDuck
Squeak
MuteQuack
<<abstract>>
Duck
ModelDuck
<<interface>>
FlyBehavior
FlyWithWings
FlyNoWay
FlyRocketPowered

Big Picture on encapsulated behaviors

Reworked class structure


Encapsulated fly behavior
Duck
FlyBehavior flyBehavior
QuackBehavior quackBehavior
Swim()
Display()
performQuack()
performFly()
setFlyBehavior()
setQuackbehavior()
//OTHER duck-like methods
MallardDuck
display()
RedHeadDuck
display()
RubberDuck
display()
<<interface>>
FlyBehavior
FlyWithWings
FlyNoWay
FlyRocketPowered
Encapsulated quack behavior
<<interface>>
QuackBehavior
Quack
Squeak
MuteQuack

HAS-A can be better than IS-A

- Each duck <u>has a</u> FlyBehavior and a QuackBehavior to which it delegates flying and quacking
- **<u>Composition</u>** at work
  - Instead of inheriting behavior, ducks get their behavior by being *composed* with the right behavior object

Third Design Principle

- Favor composition over inheritance
  - More flexibility
  - Encapsulate a family of algorithms into their own set of classes
  - Able to change behavior at runtime

Strategy

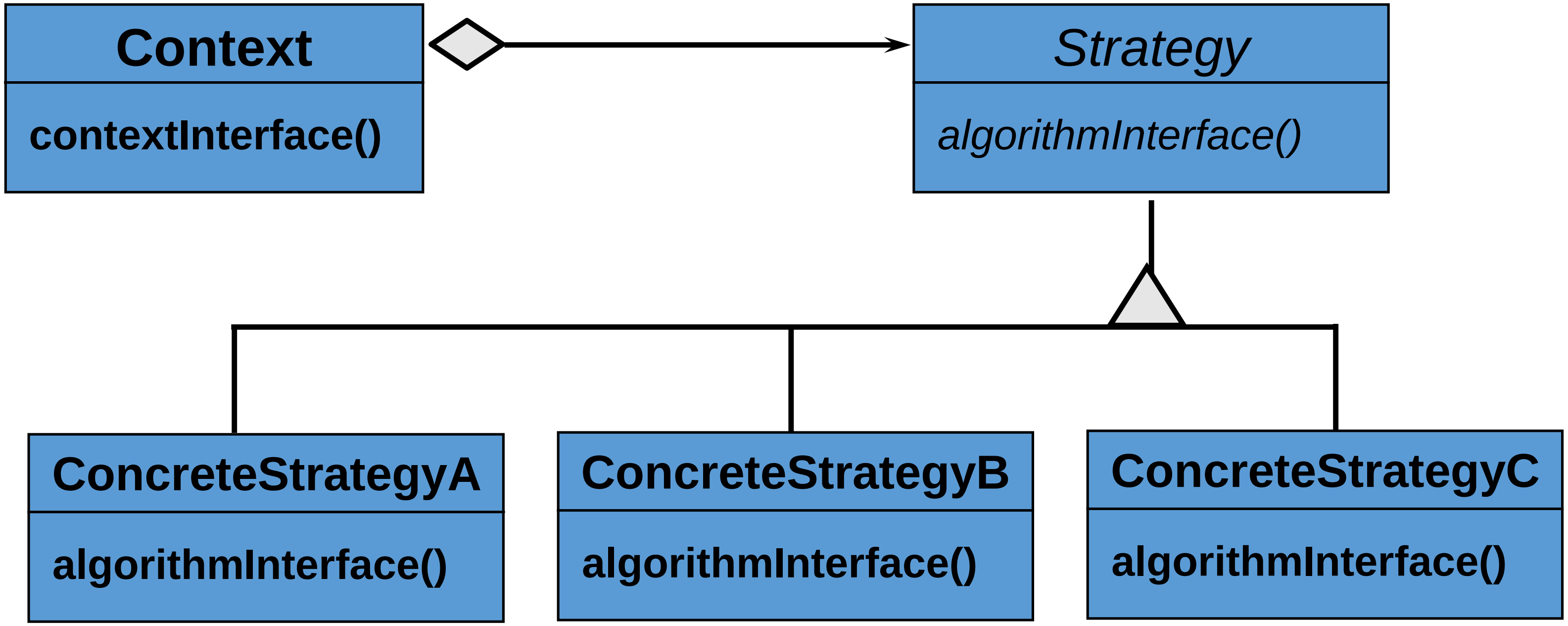
In a Strategy design pattern, you will:

- Define a family of algorithms
- Encapsulate each one
- Make them interchangeable

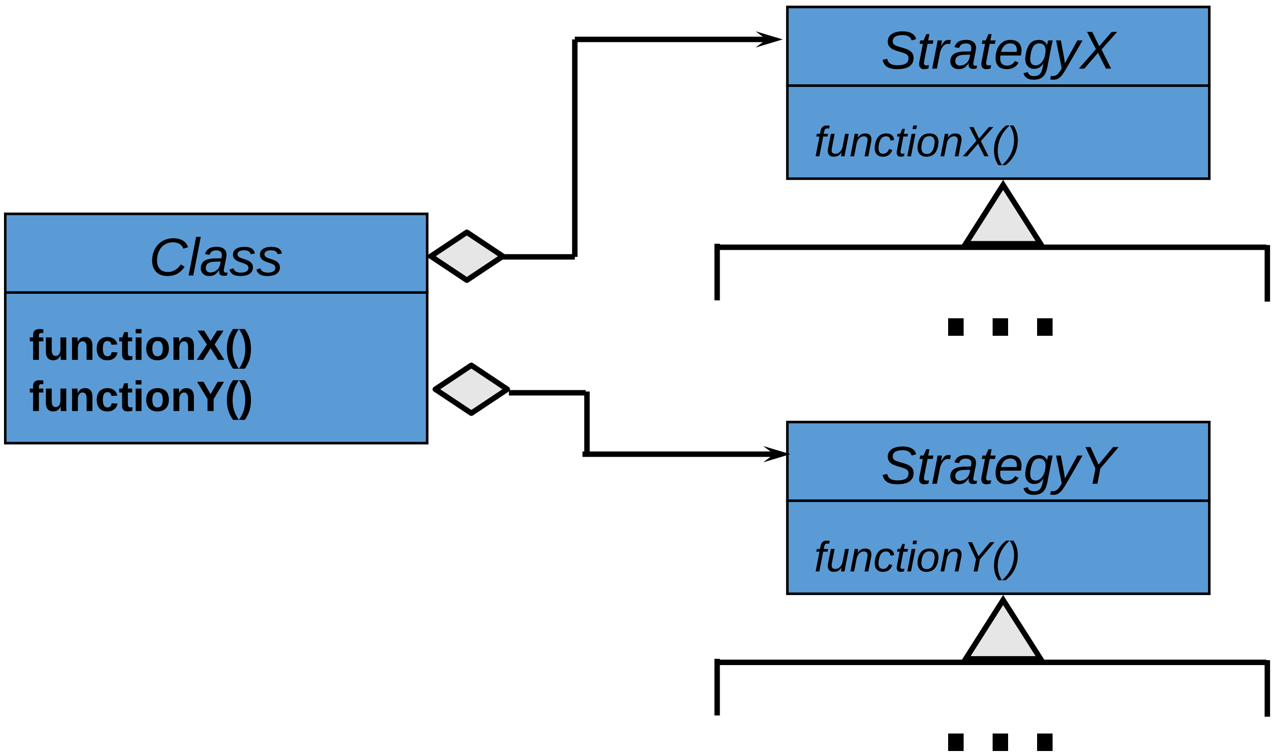
You should use Strategy when:

- You have code with a lot of algorithms
- You want to use these algorithms at different times
- You have algorithm(s) that use data the client should not know about

Strategy Class Diagram



Strategy makes this easy!



Benefits of Strategy

- Eliminates conditional statements
  - Can be more efficient than case statements
- Choice of implementation
  - Client can choose among different implementations with different space and time trade-offs
- Families of related algorithms
- Alternative to subclassing
  - This lets you vary the algorithm dynamically, which makes it easier to change and extend
  - You also avoid complex inheritance structures

Strategy Pattern

- The strategy Pattern
  - Defines a family of algorithms,
  - Encapsulates each one,
  - Makes them interchangeable.
- Strategy lets the algorithm vary independently from clients that use it