

Design Patterns

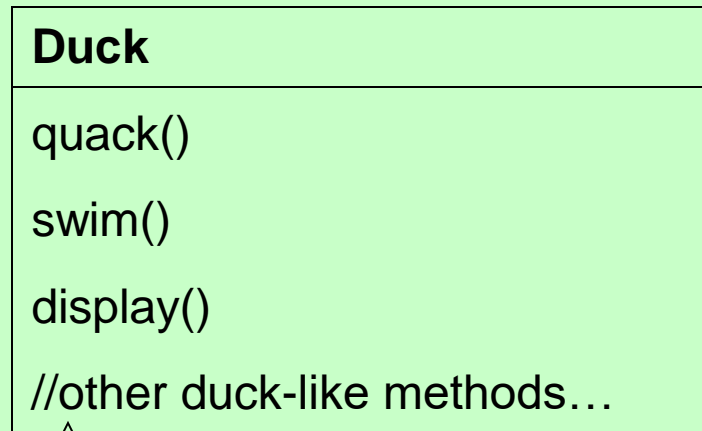
1. Someone has already solved your problems.

SimUDuck

- Joe works for a company that makes a highly successful duck pond simulation 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 **one Duck superclass** from which all other duck types **inherit**.

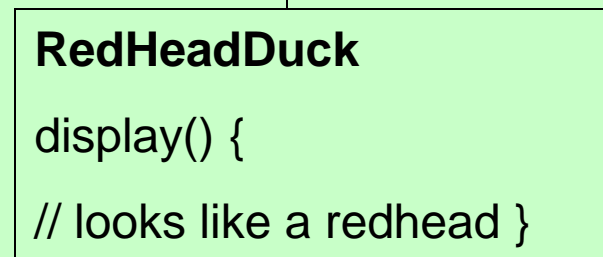
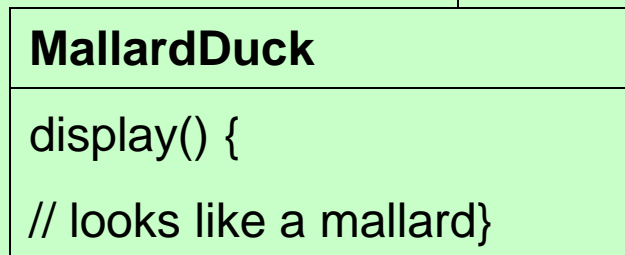
Existing Duck application

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



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

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



Other duck types inherit from the Duck class

...

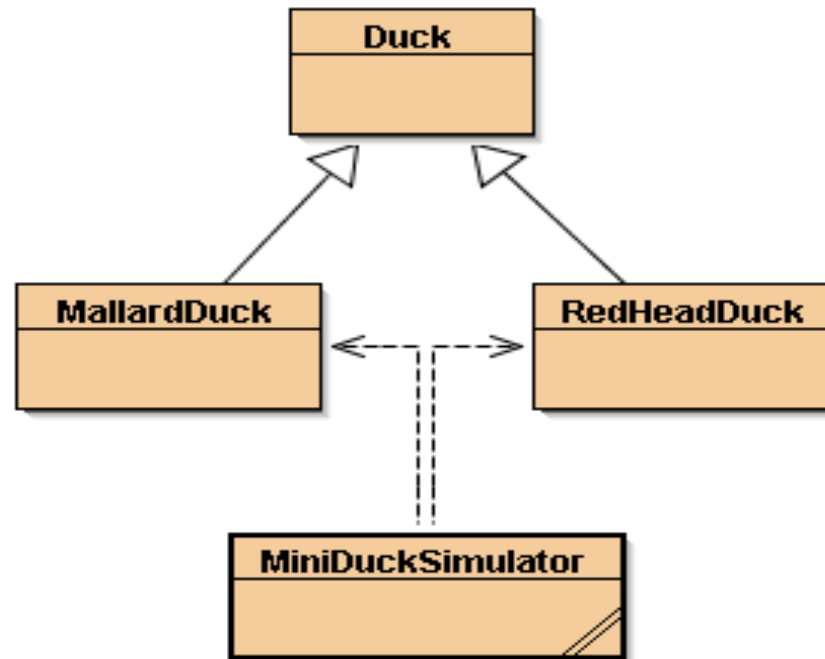


Strategy Pattern

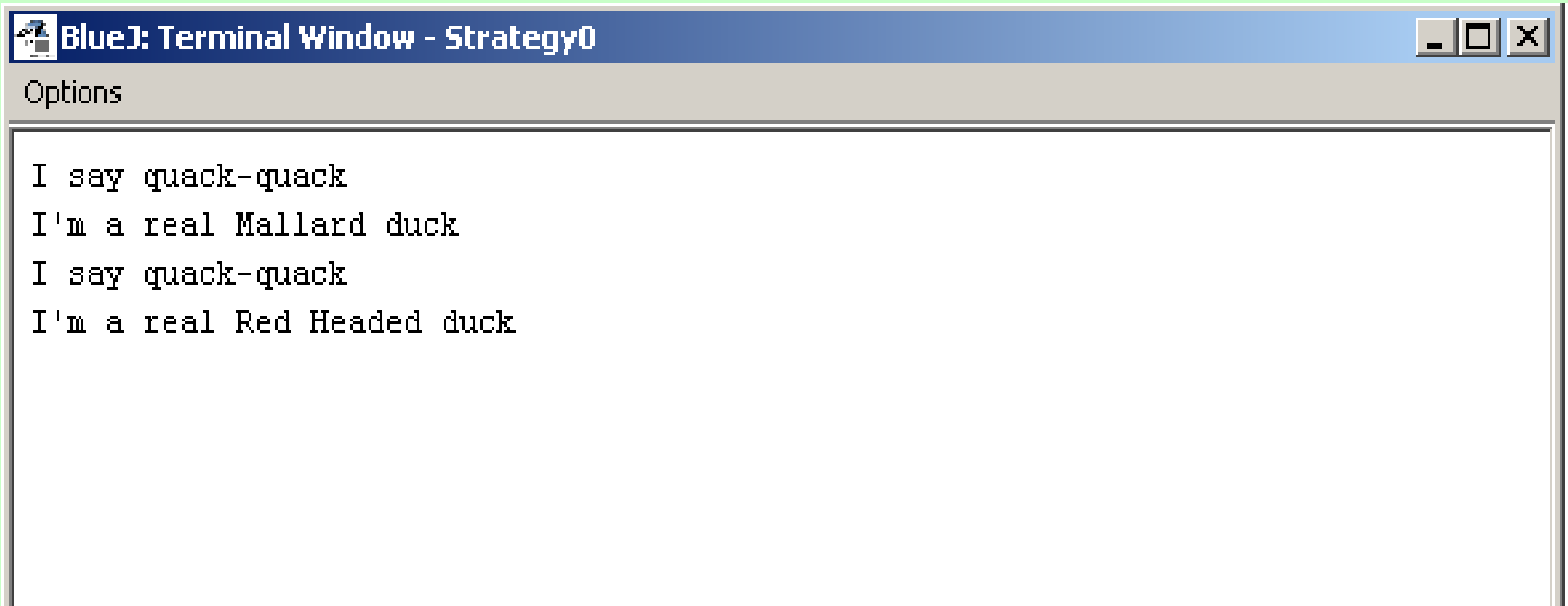
Inheritance

- All subclasses inherit the superclass' properties
 - Variables and methods

Inheritance UML



Testing Mallard, RedHeadDuck classes



```
BlueJ: Terminal Window - Strategy0
Options

I say quack-quack
I'm a real Mallard duck
I say quack-quack
I'm a real Red Headed duck
```

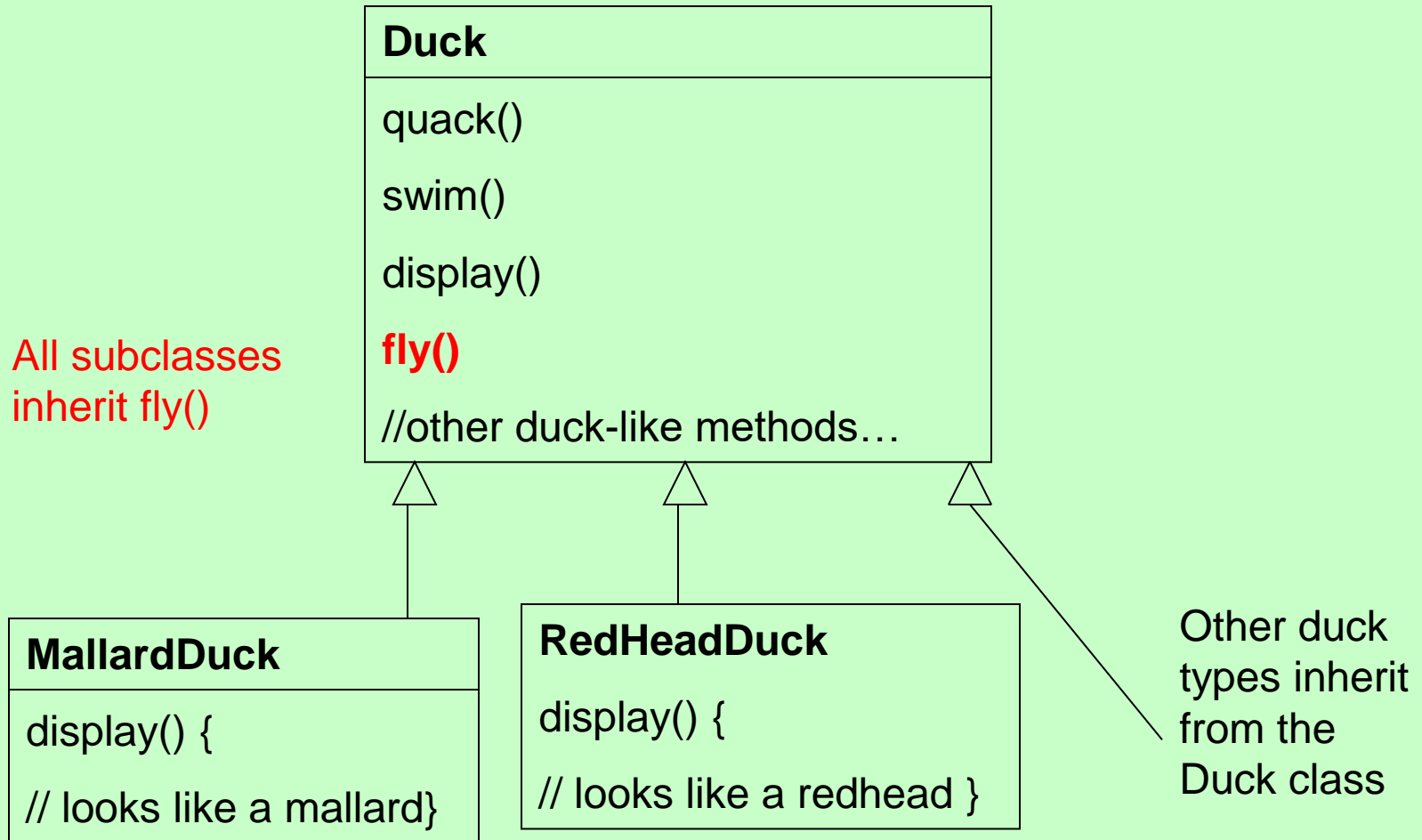
Adding New Behaviour

- The company has been under increasing pressure from competitors
- The company executives think its time for a big innovation to show at the upcoming shareholders meeting **next week**.
- The executives decided that **flying ducks** is just what the simulator needs
- And of course Joes manager told them it will be no problem for Joe to just whip something up **in a week**

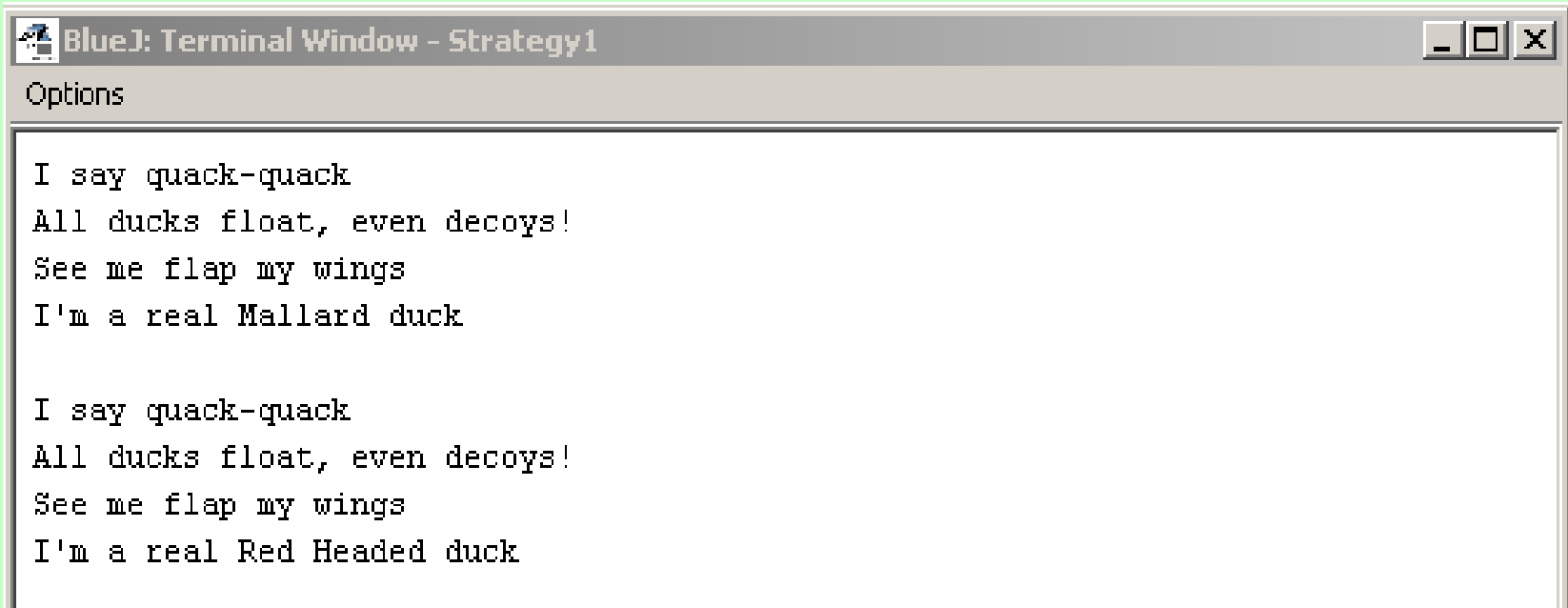
Joe's Solution

- **Joe:** I just need to **add a fly() method in the Duck class** and then **all the ducks will inherit it**. Now it's my time to really show my true OO genius.

Add a method fly() in Duck



Executing Tests

A screenshot of a Java IDE terminal window titled "BlueJ: Terminal Window - Strategy1". The window has a standard OS-style title bar with minimize, maximize, and close buttons. Below the title bar is a tab labeled "Options". The main area of the window contains two blocks of text, each representing a test execution. The first block shows the output for a "Mallard duck" test, and the second block shows the output for a "Red Headed duck" test. Both tests pass, as indicated by the "I'm a real" statement at the end of each block.

```
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
```

Everything OK?

- Think harder.

Something went horribly wrong

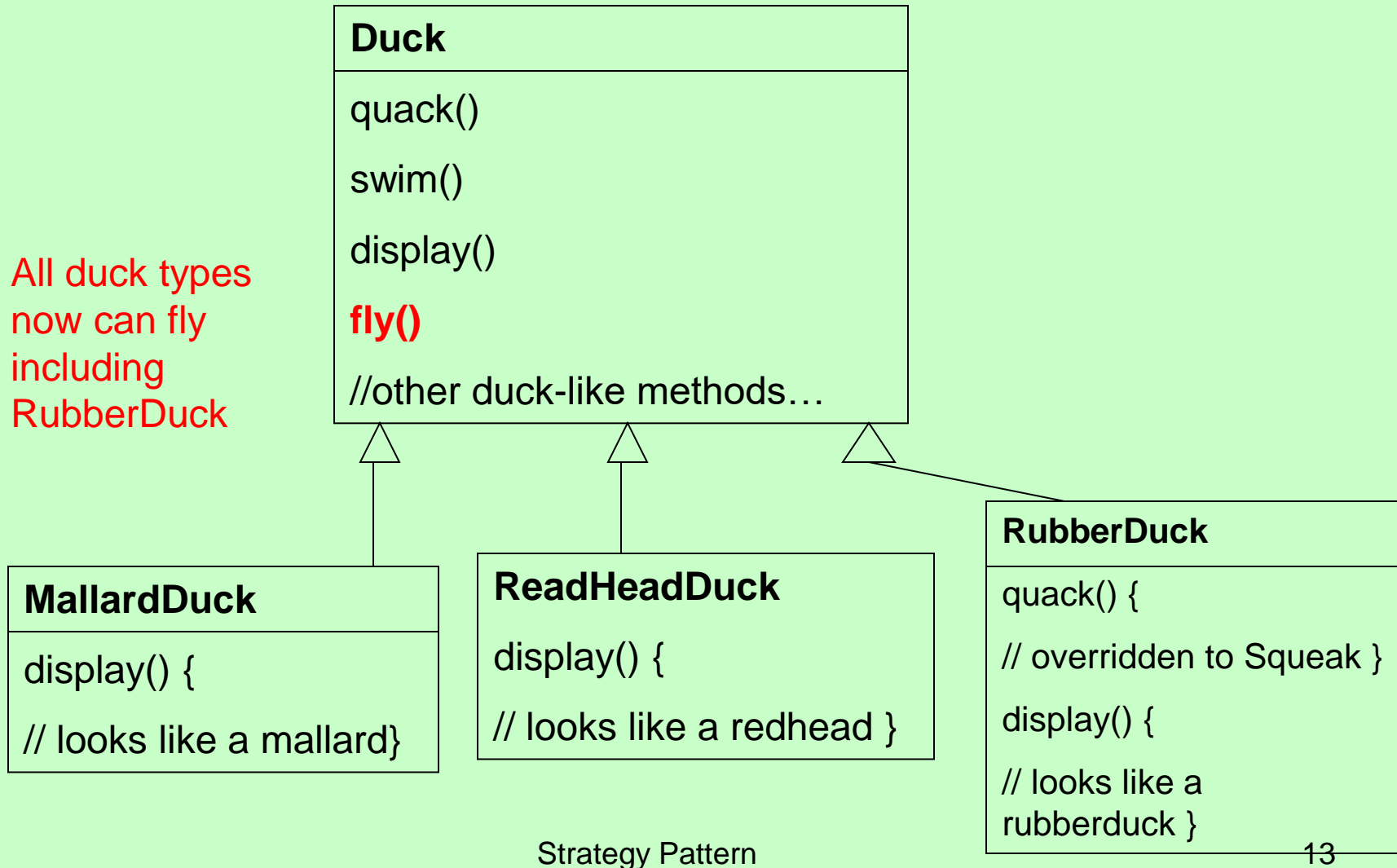
- Joe receives a phone call from his manager
 - "Joe, I'm at the shareholder's meeting. They just gave a demo and there were **rubber duckies** flying around the screen."
- Joe failed to notice that not all subclasses of Duck should fly.
- A localized update to the code caused a nonlocal side effect (flying rubber ducks)!



Strategy Pattern



Something seriously wrong!



Root cause?

- Applying inheritance to achieve re-use
- Poor solution for maintenance
- A great use of inheritance for the purpose of reuse hasn't turn out so well when it comes to maintenance.

How do we fix this?

- Using inheritance as before
 - Override the fly() method in rubber duck as in quack()
- Will it fix the problem?

Is the problem solved?

- Any new problems?

Wait a minute

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



Summary

- Which of the following are disadvantages of using inheritance to provide Duck behavior? (Choose all that apply.)
 - A. Code is duplicated across subclasses.
 - B. Runtime behavior changes are difficult.
 - C. We can't make ducks dance.
 - D. Hard to gain knowledge of all duck behaviors.
 - E. Ducks can't fly and quack at the same time.
 - F. Changes can unintentionally affect other ducks.
- Is there a better way of doing things?

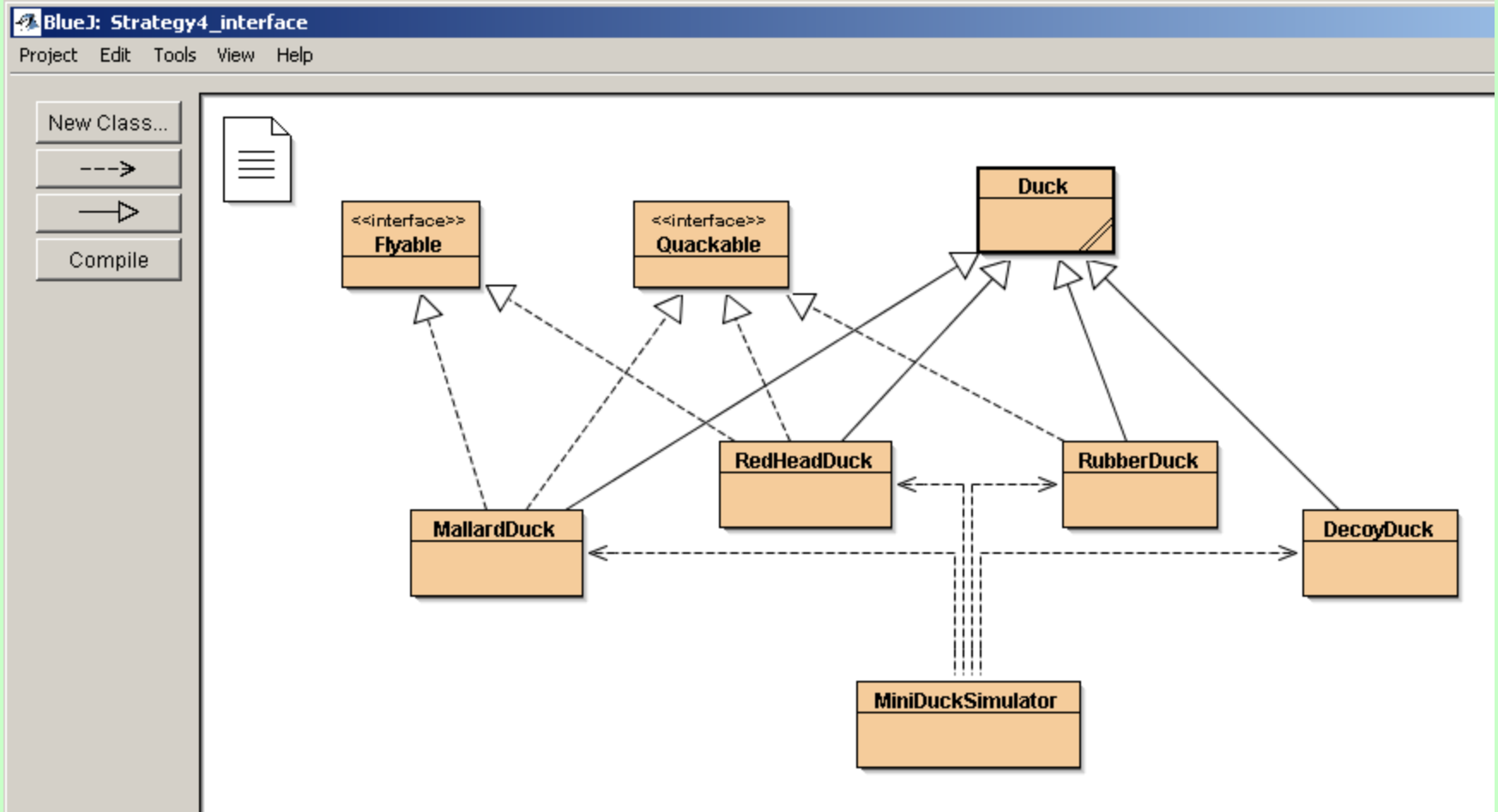
How about Interface?

- Joe realized that inheritance probably wasn't the answer, because he just got a memo that says that the executives now want to update the product every six months (in ways they haven't yet decided on).
- Joe knows the spec will keep changing and he'll be forced to look at and possibly override `fly()` and `quack()` for every new Duck subclass that's ever added to the program... forever.

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
 - Since not all ducks can quack.

How about this solution?



But

- You shoot yourself in the foot by duplicating code 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

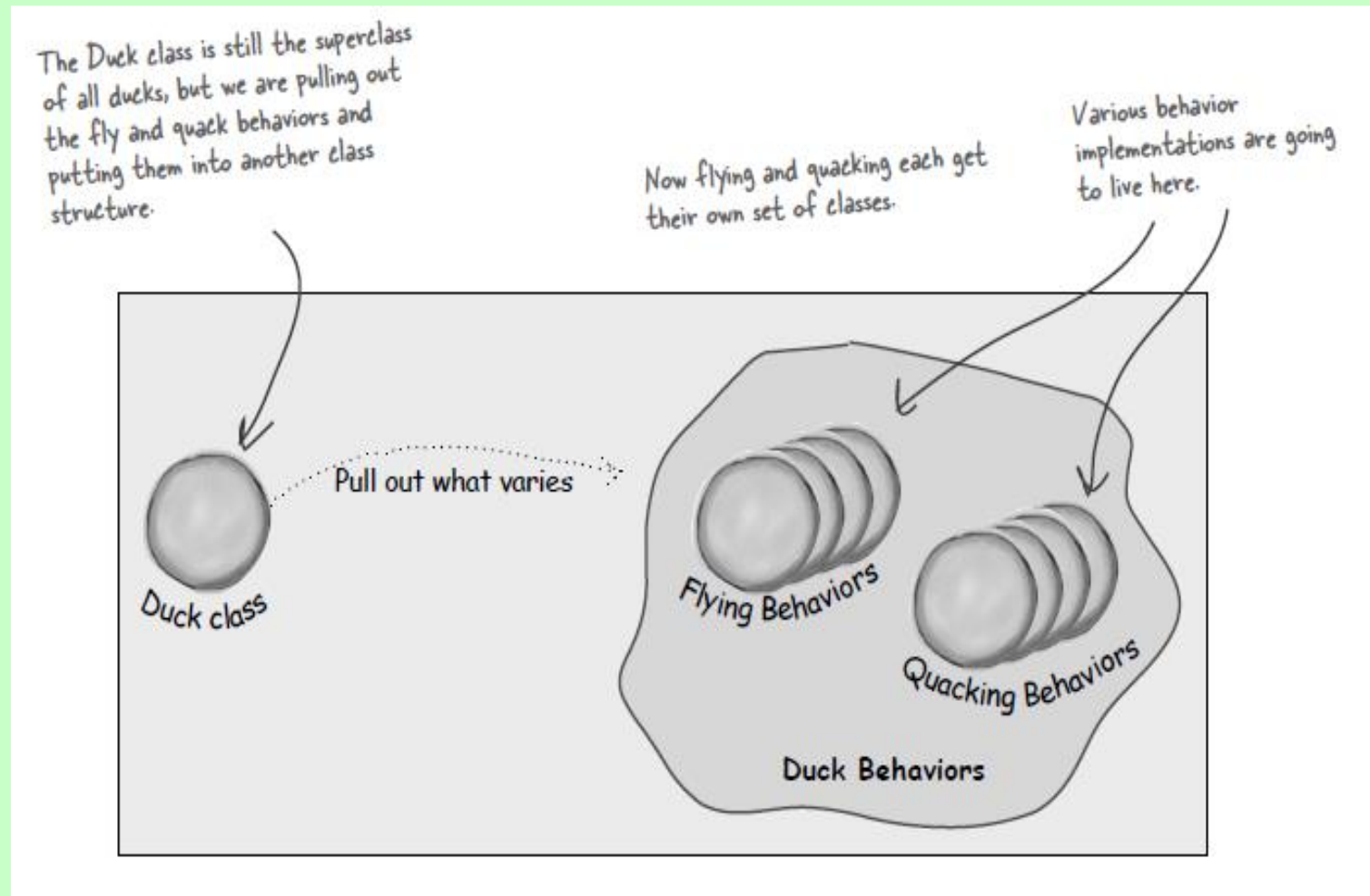
#1 Design Principle

- Identify the aspects of your application that vary and separate them from what stays the same
- In other words:
 - take the parts that vary and encapsulate them, so that later you can change or extend the parts that vary without affecting those that don't

#1 Design Principle

- 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

#1 Design Principle

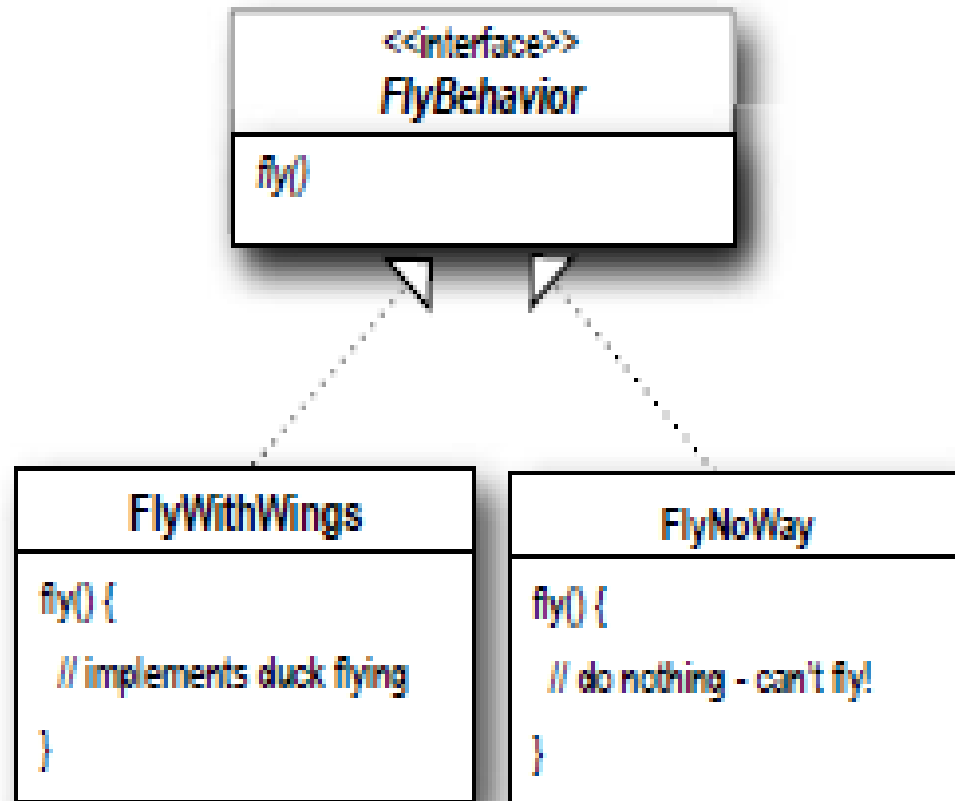


How do we design the classes to implement the fly and quack behaviors?

- 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
 - include behavior setter methods in the Duck classes so that we can, say, change the MallardDuck's flying behavior at runtime.

#2 Design Principle

- Program to an interface, not an implementation
- Use a interface to represent each behavior
 - FlyBehavior and QuackBehavior
 - Each implementation of a behavior will implement one of these interfaces
- In the past, we rely on an implementation
 - In superclass Duck, or
 - A specialized implementation in the subclass

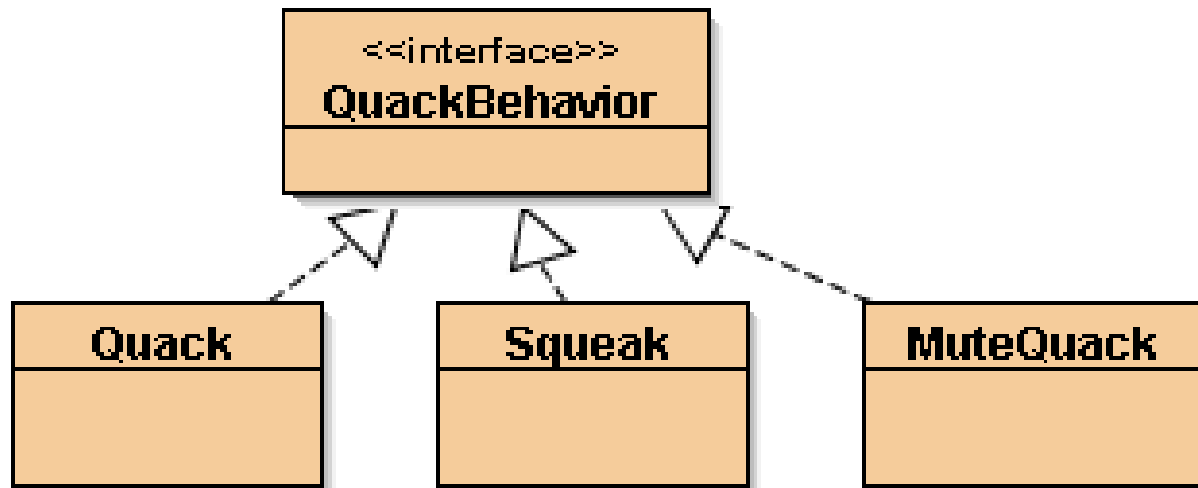


3 classes in code

```
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");  
    }  
}
```



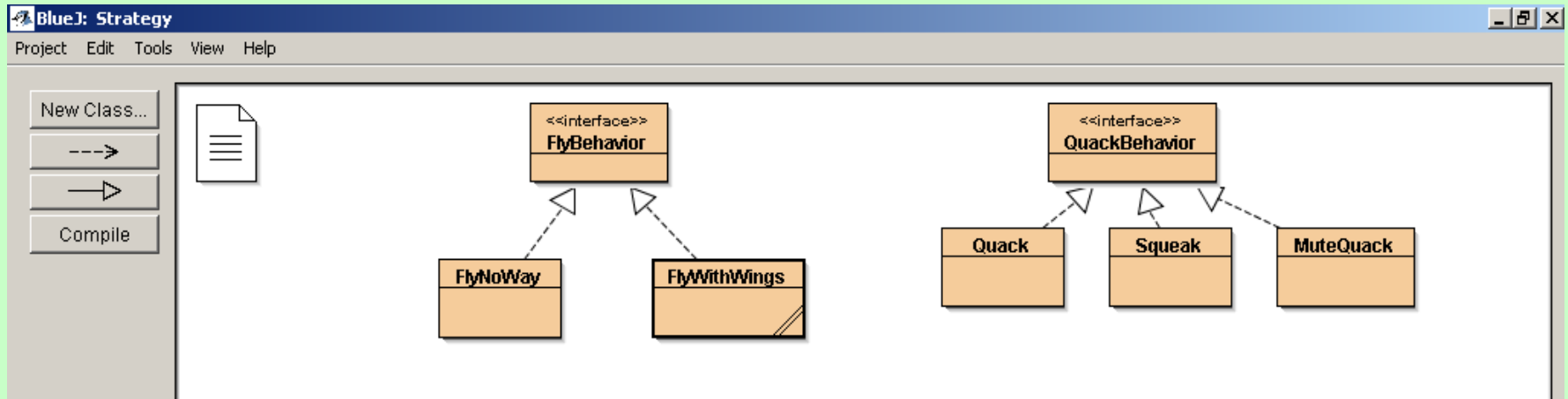
```
public interface QuackBehavior {  
    public void quack();  
}
```

Specific behaviors by implementing interface QuackBehavior

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

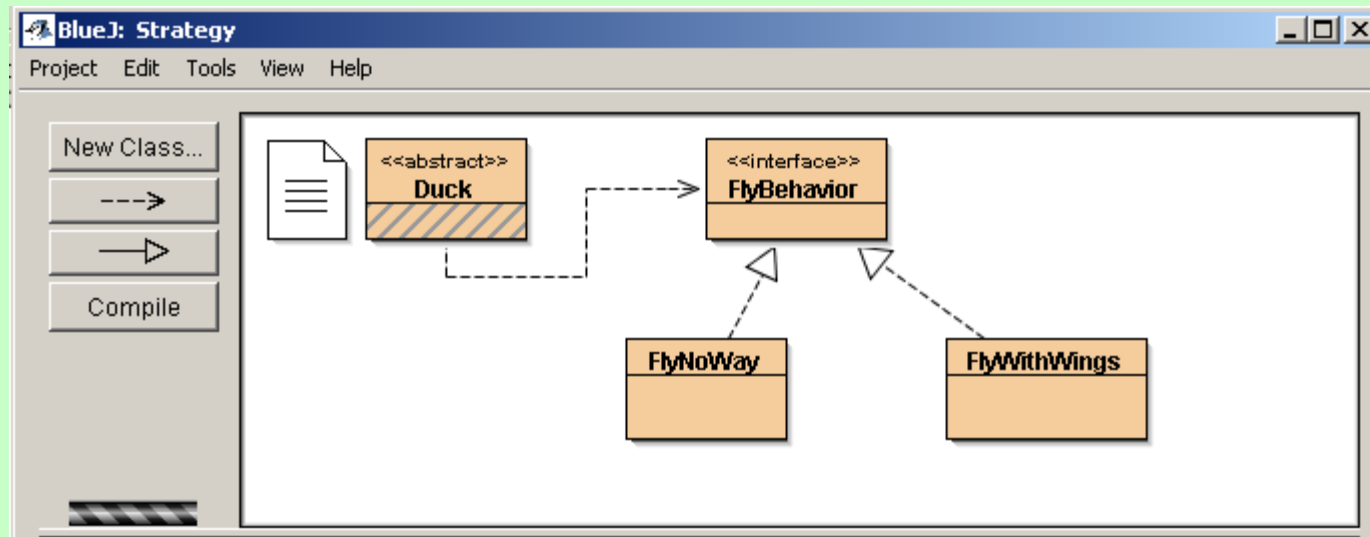
```
-----  
public class Squeak implements QuackBehavior {  
    public void quack() {  
        System.out.println("Squeak");  
    }  
}
```

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

With this design, other types of objects can reuse our fly and quack behaviours because these behaviours are no longer hidden away in our Duck classes.

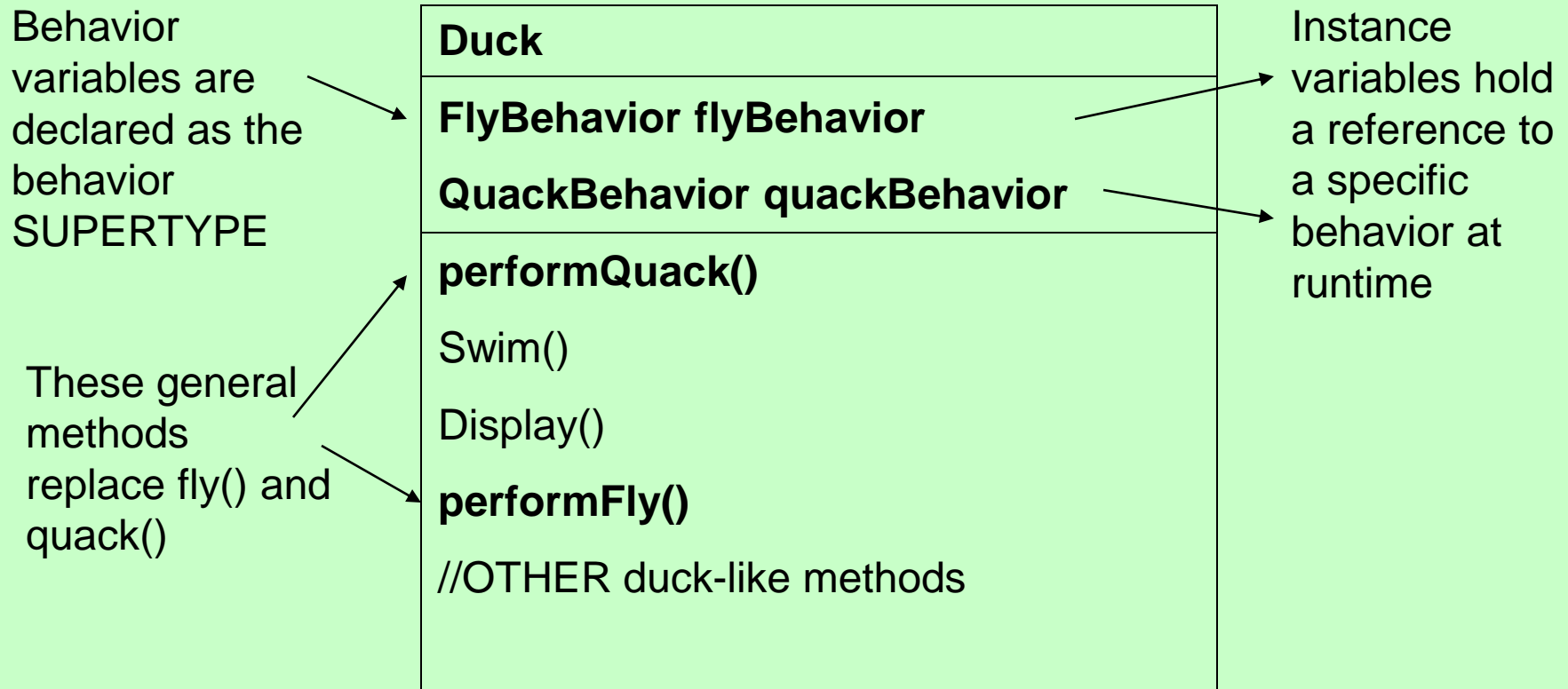
And we can add new behaviours without modifying any of existing behaviour classes or touching any of the Duck classes that use flying behaviours.



Now: Duck subclass will use a behavior represented by an interface (FlyBehaviour and QuackBehaviour).

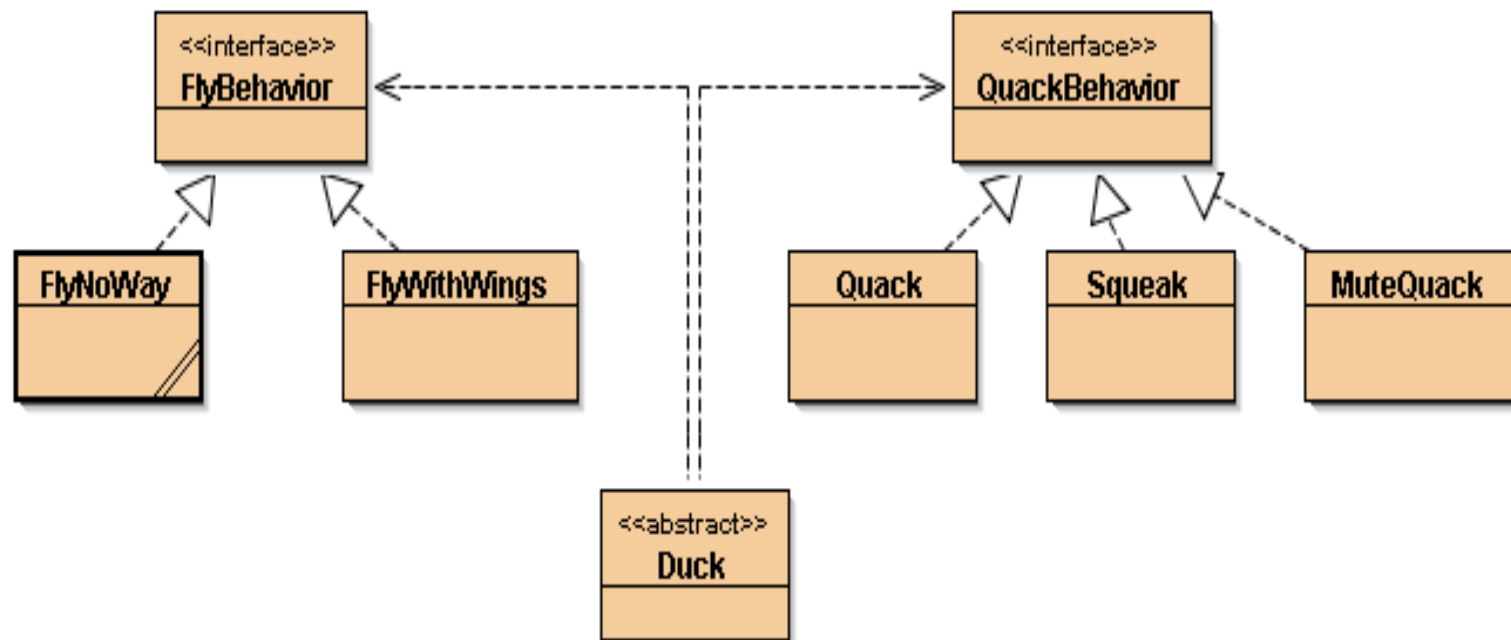
Integrating the Duck Behavior

1. Add 2 instance variables:



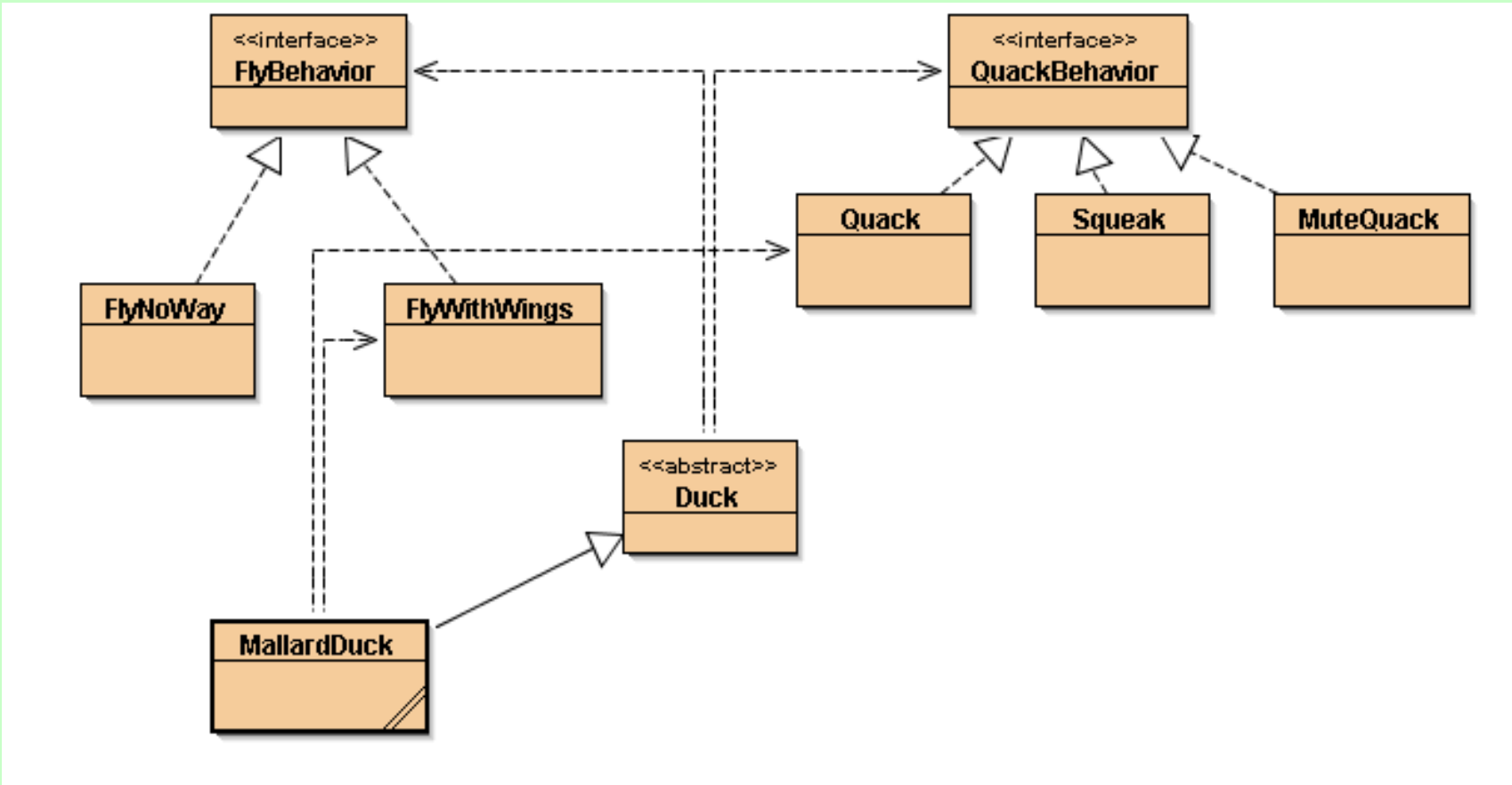
Implement performQuack()

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



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");  
    }  
}
```



// 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!");  
    }  
}
```


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");  
    }  
}
```

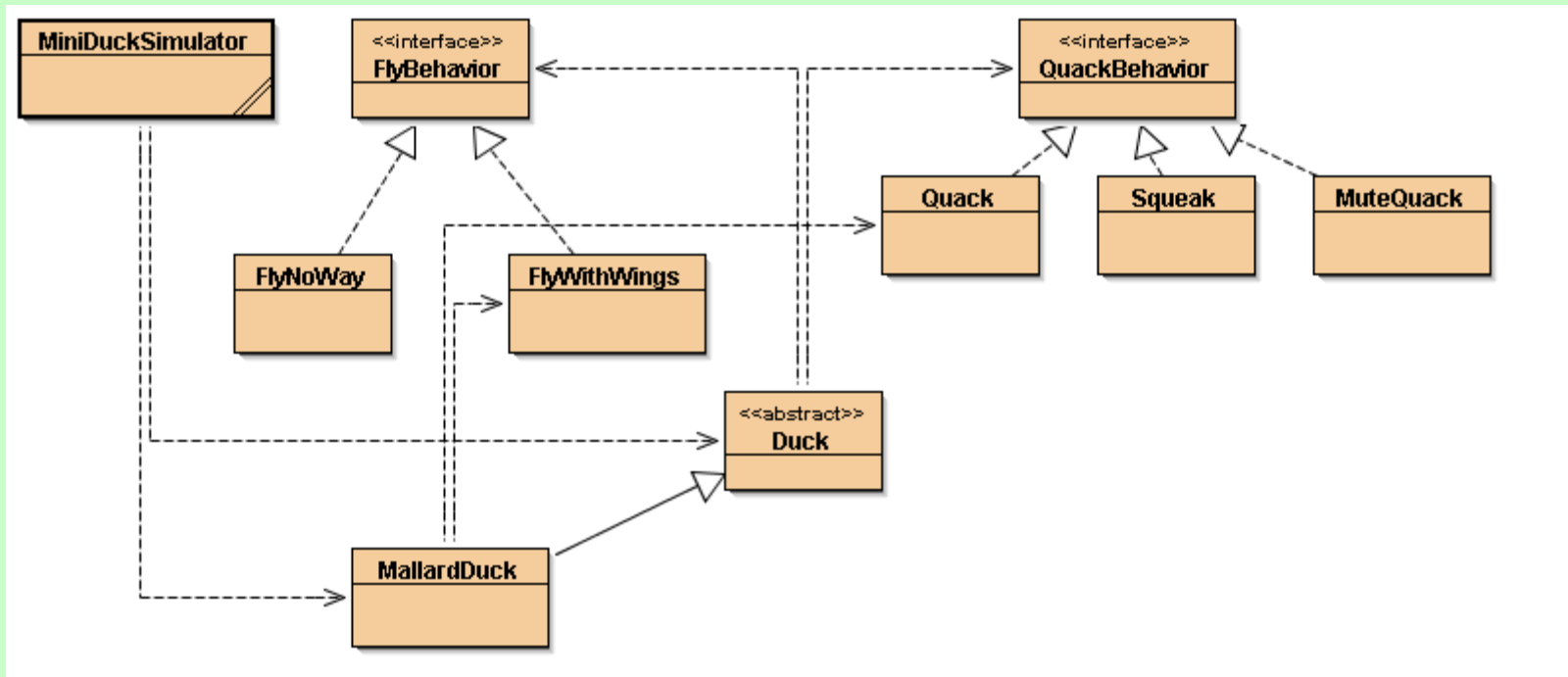
```
-----  
public class Squeak implements QuackBehavior {  
    public void quack() {  
        System.out.println("Squeak");  
    }  
}
```

```
-----  
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.  
    }  
}
```

Strategy project



Dynamic Behaviours

- 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?

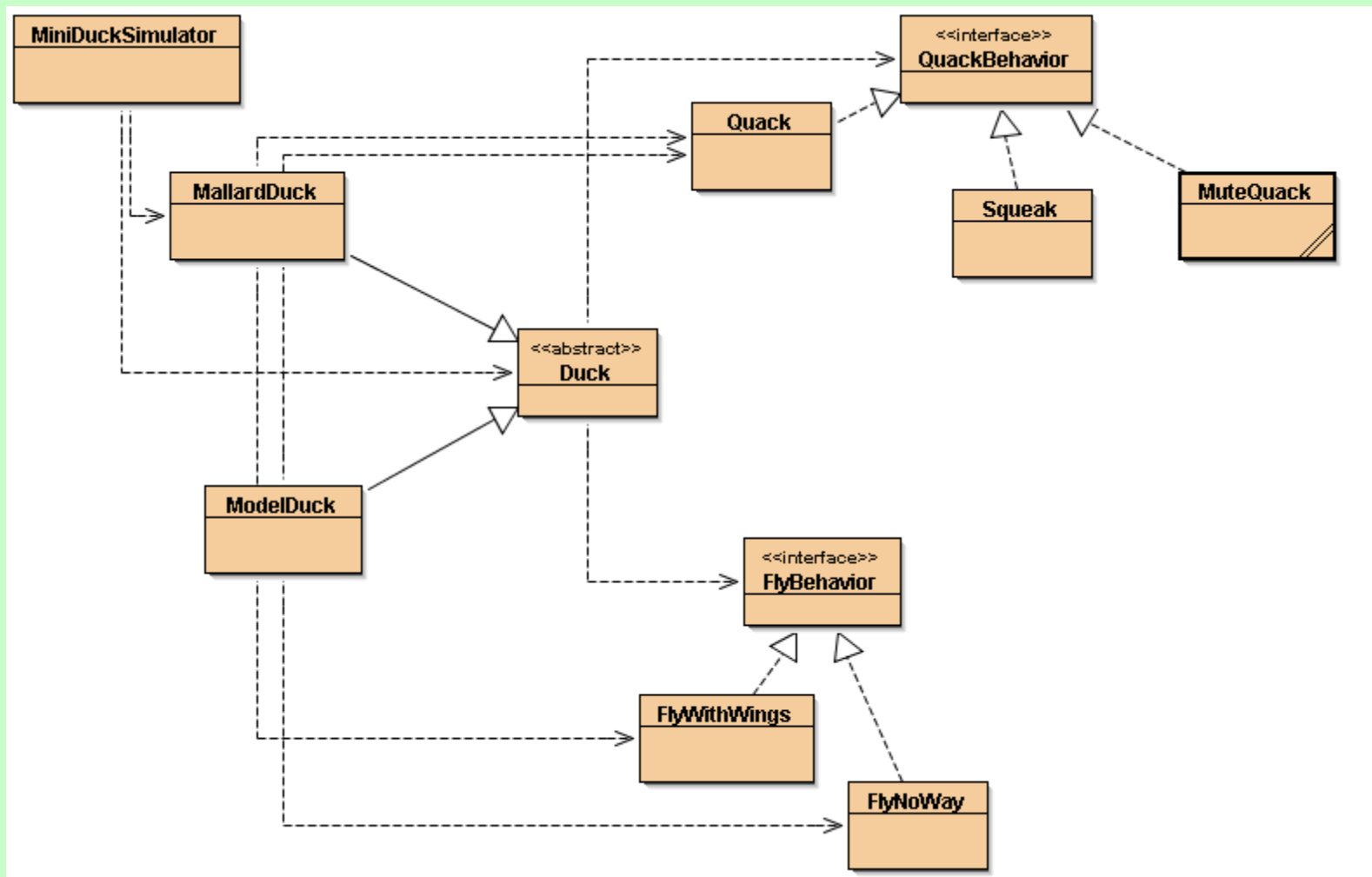
1. Add new methods to the Duck class

```
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");  
    }  
}
```

Enabling ModelDuck to fly

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

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

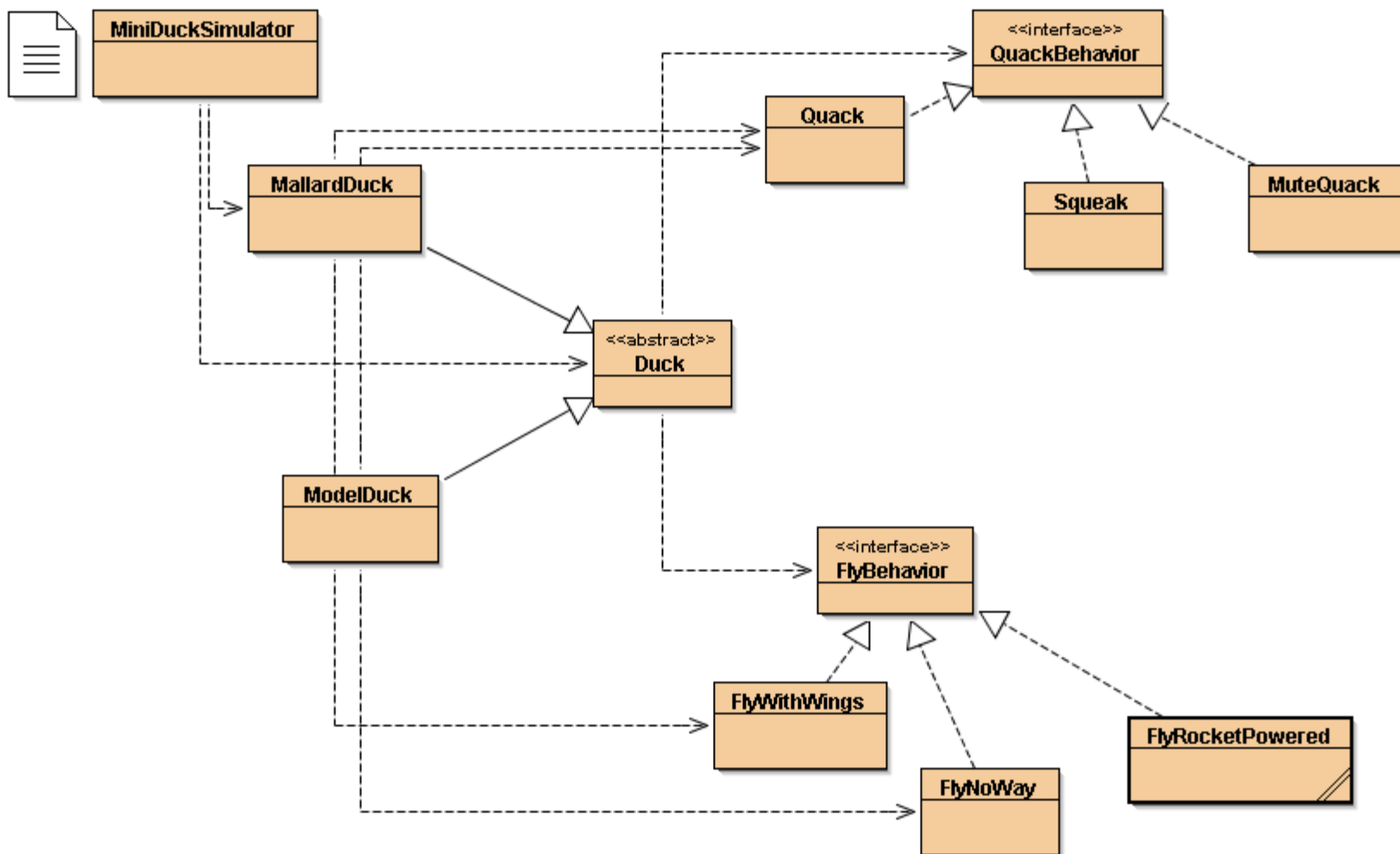
```
public class FlyRocketPowered implements FlyBehavior {  
  
    public void fly() {  
        System.out.println("I'm flying with a rocket");  
    }  
}
```

New Class...

--->

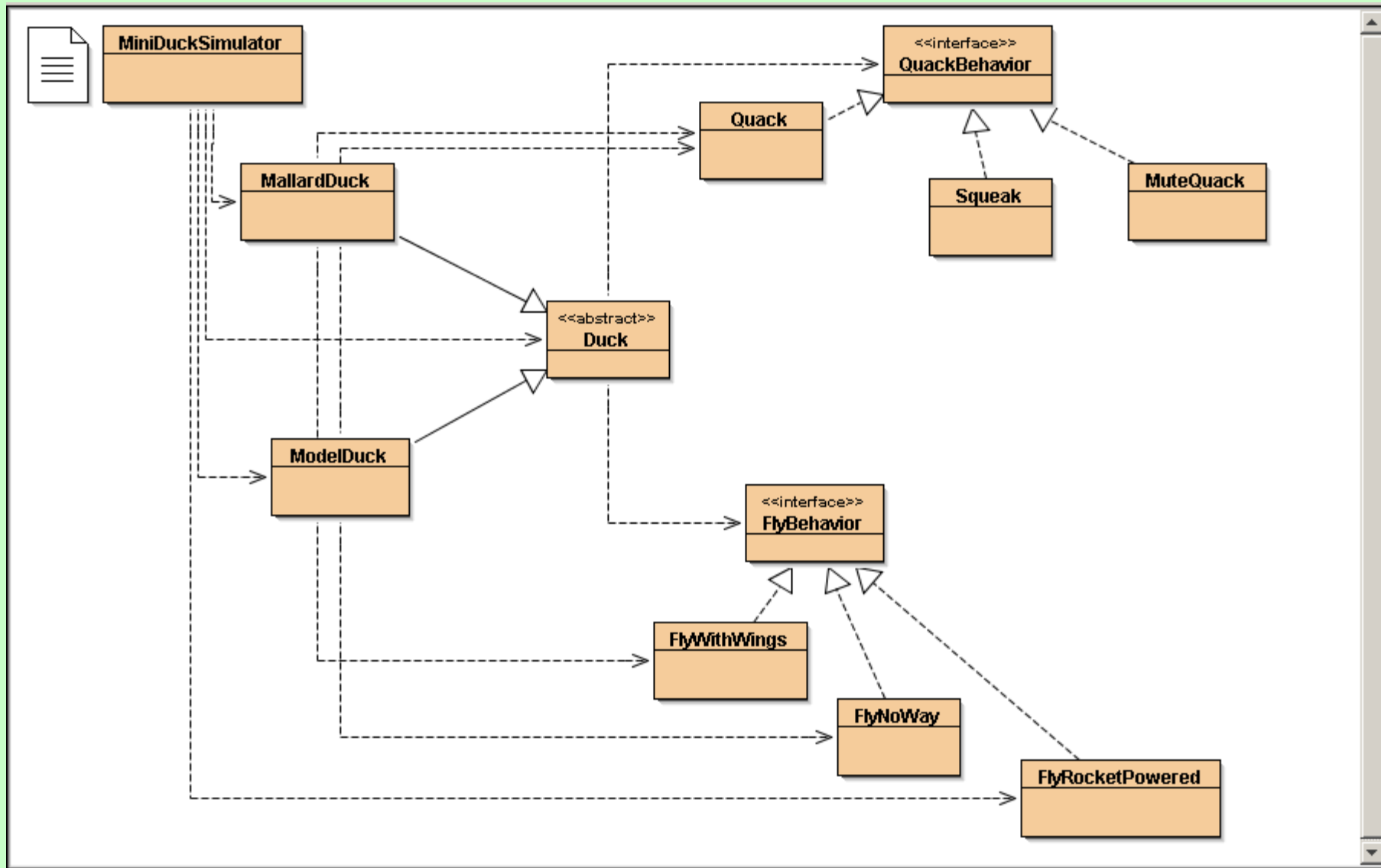
→

Compile



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();
```



Test again

```
BlueJ: Terminal Window - Strategy
Options
Quack
I'm flying!!
I can't fly
I'm flying with a rocket
```

From mallard

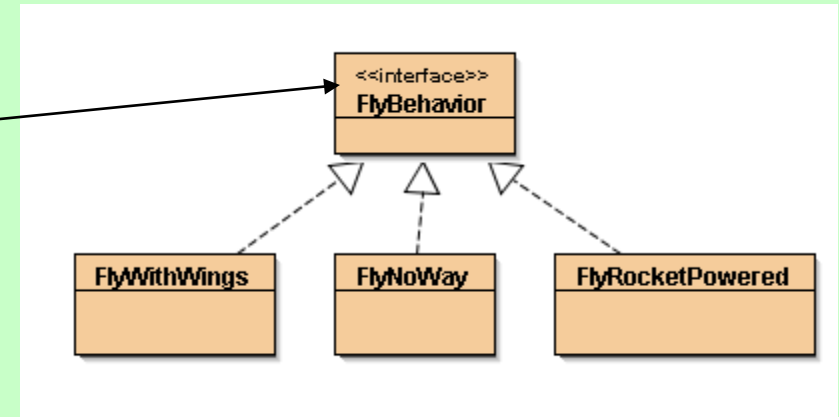
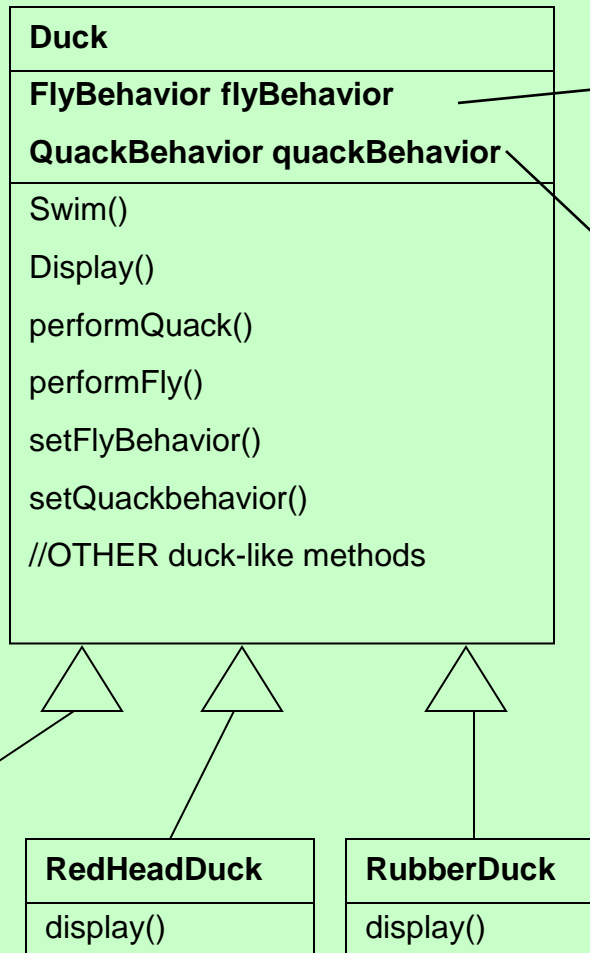
Due to setter method inherited from Duck but set in model

Due to
flyBehavior = new
FlyNoWay();
In constructor of
ModelDuck

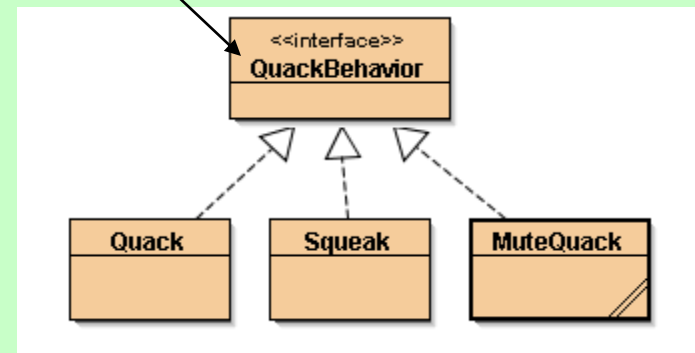
Summary

- Reworked class structure
 - ducks extending Duck
 - fly behaviors implementing FlyBehavior
 - quack behaviors implementing QuackBehavior
- Think of each set of behaviors as a family of algorithms
- Relationships: IS-A, HAS-A, IMPLEMENTS
- Exercise: mark up the following diagram with the relationships above

Encapsulated fly behavior



Encapsulated quack behavior



HAS-A can be better than IS-A

- Each duck has a FlyBehavior and a QuackBehavior to which it delegates flying and quacking
- **Composition** 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 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

References

- Design Patterns: Elements of Reusable Object-Oriented Software By Gamma, Erich; Richard Helm, Ralph Johnson, and John Vlissides (1995). Addison-Wesley. ISBN 0-201-63361-2.
- **Head First Design Patterns** By Eric Freeman, Elisabeth Freeman, Kathy Sierra, Bert Bates
First Edition October 2004
ISBN 10: 0-596-00712-4
- http://www.uwosh.edu/faculty_staff/huen/262/f09/slides/10_Strategy_Pattern.ppt