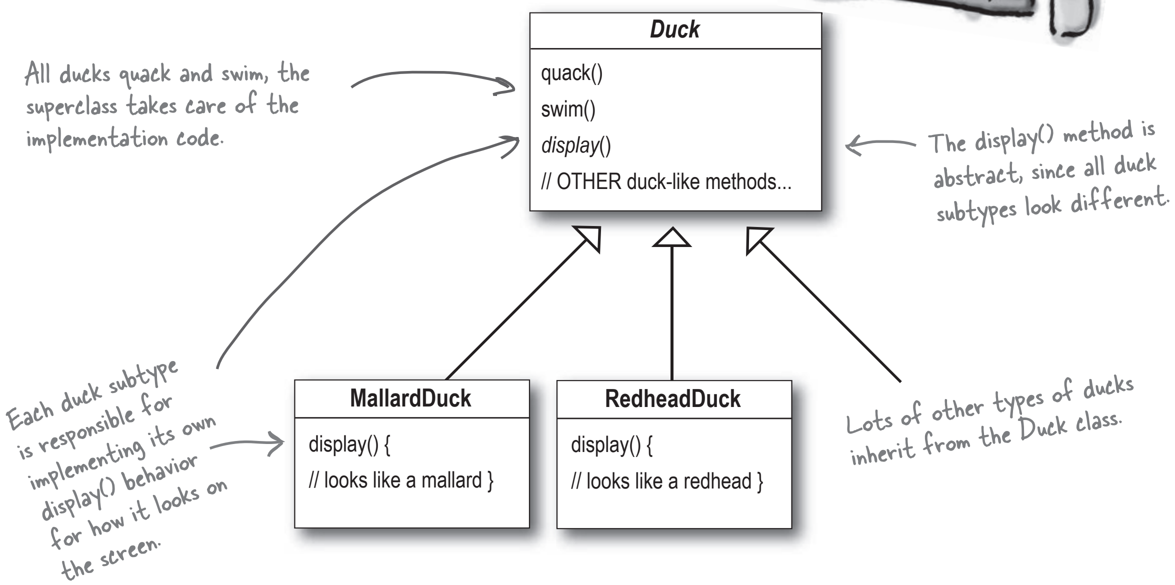
1 Intro to Design Patterns

Welcome to Design Patterns

Someone has already solved your problems. In this chapter, you'll learn why (and how) you can exploit the wisdom and lessons learned by other developers who've been down the same design problem road and survived the trip. Before we're done, we'll look at the use and benefits of design patterns, look at some key OO design principles, and walk through an example of how one pattern works. The best way to use patterns is to *load your brain* with them and then *recognize places* in your designs and existing applications where you can *apply them*. Instead of *code* reuse, with patterns you get *experience* reuse.

It started with a simple SimUDuck app

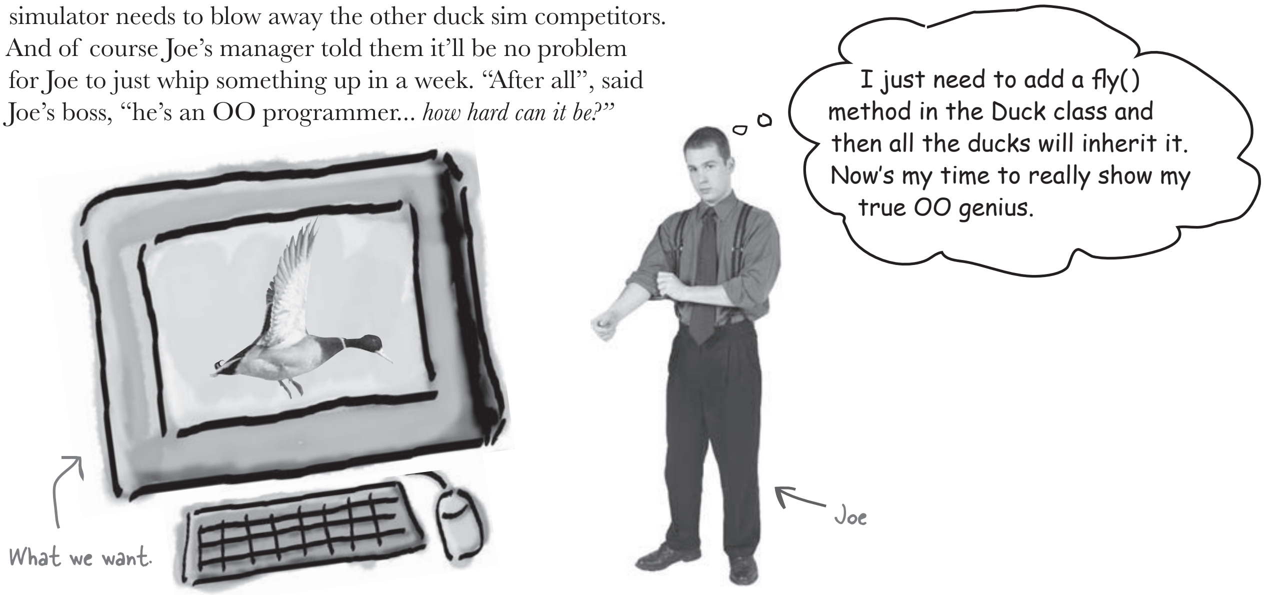
Joe works for a company that makes a highly successful duck pond simulation game, *SimUDuck*. The game can show a large variety of duck species swimming and making quacking sounds. The initial designers of the system used standard OO techniques and created one Duck superclass from which all other duck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

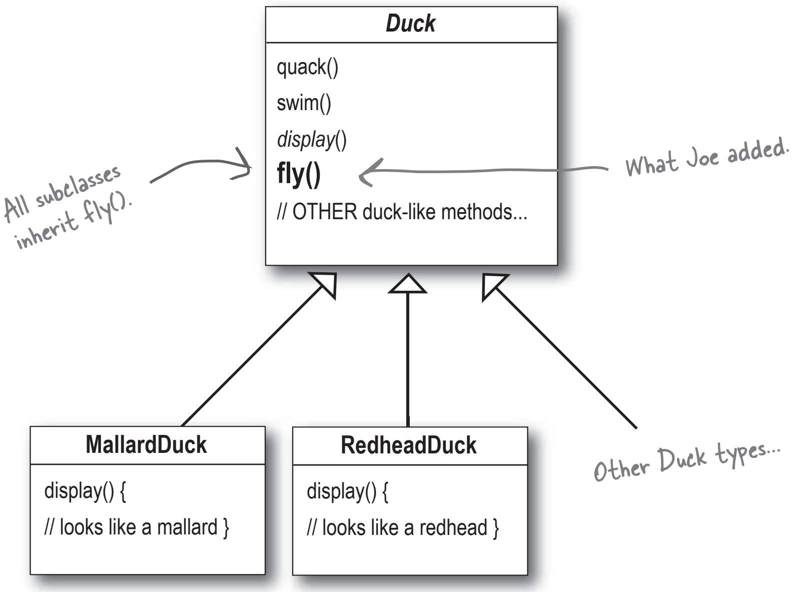


In the last year, the company has been under increasing pressure from competitors. After a week long off-site brainstorming session over golf, the company executives think it's time for a big innovation. They need something *really* impressive to show at the upcoming shareholders meeting in Maui *next week*.

But now we need the ducks to FLY

The executives decided that flying ducks is just what the simulator needs to blow away the other duck sim competitors. And of course Joe's manager told them it'll be no problem for Joe to just whip something up in a week. "After all", said Joe's boss, "he's an OO programmer... *how hard can it be?*"



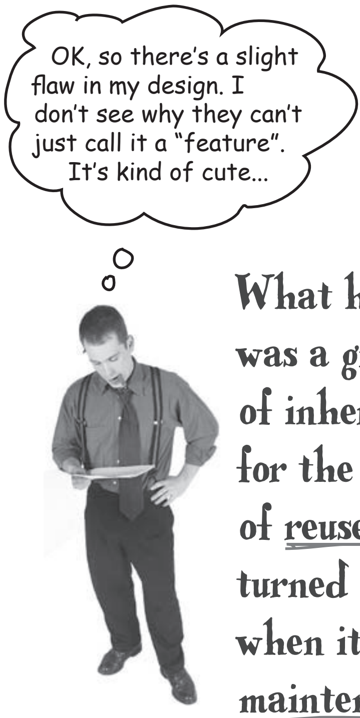


But something went horribly wrong...

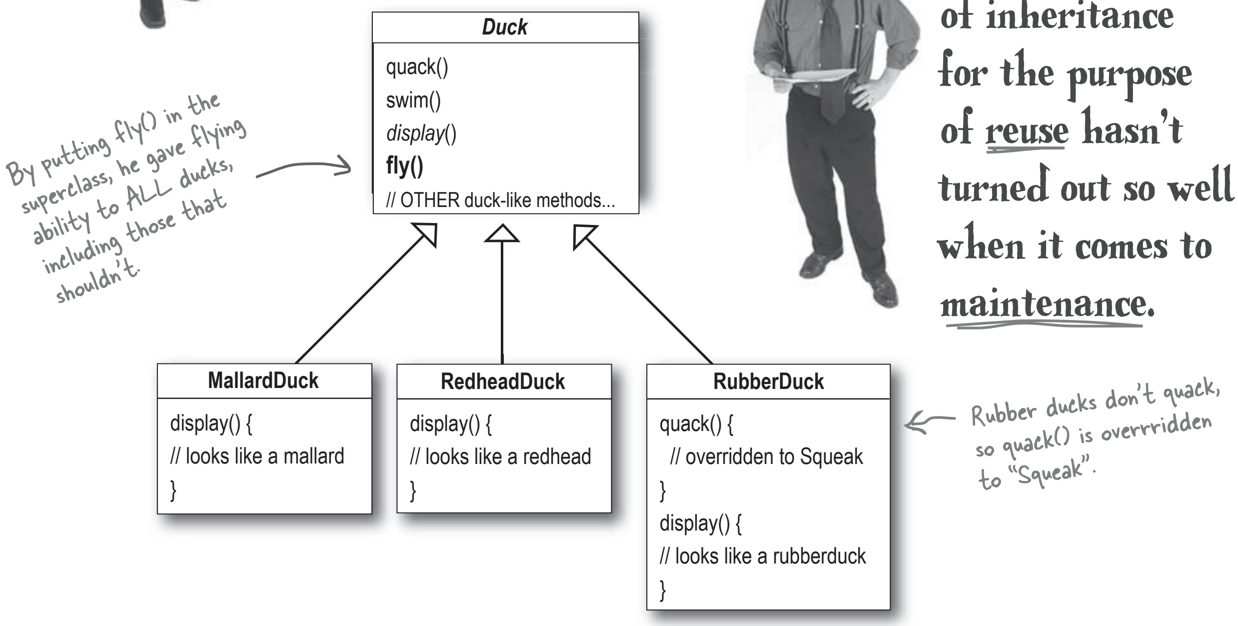
What happened?

Joe failed to notice that not *all* subclasses of Duck should *fly*. When Joe added new behavior to the Duck superclass, he was also adding behavior that was *not* appropriate for some Duck subclasses. He now has flying inanimate objects in the SimUDuck program.

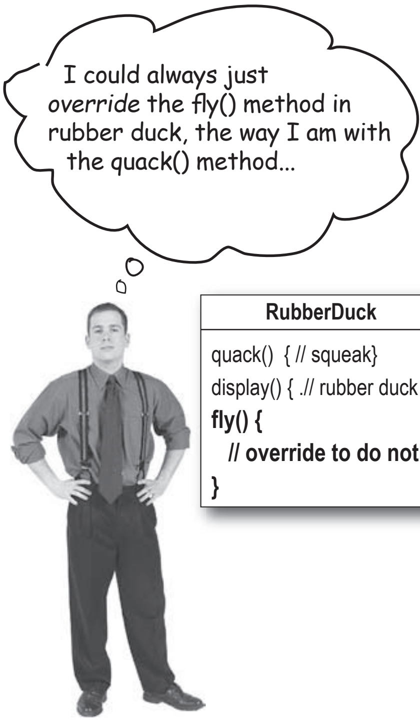
A localized update to the code caused a non-local side effect (flying rubber ducks)!



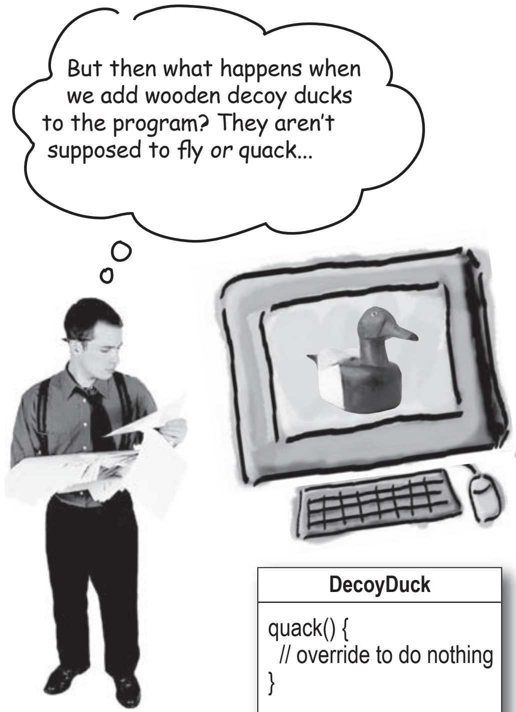
What he thought was a great use of inheritance for the purpose of <u>reuse</u> hasn't turned out so well when it comes to <u>maintenance</u>.



Joe thinks about inheritance...




```
RubberDuck
----------
quack()  { // squeak}
display() { .// rubber duck }
fly() {
   // override to do nothing
}
```




```
DecoyDuck
---------
quack() {
  // override to do nothing
}

display() { // decoy duck}

fly() {
  // override to do nothing
}
```

Here's another class in the hierarchy; notice that like RubberDuck, it doesn't fly, but it also doesn't quack.

Sharpen your pencil

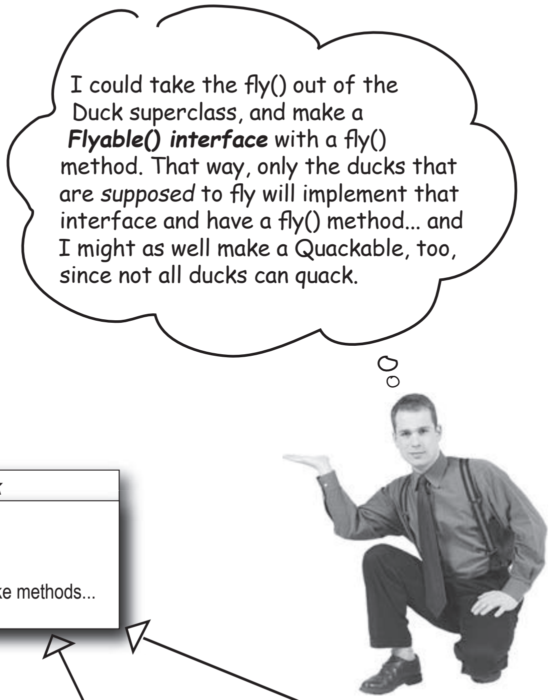
Which of the following are disadvantages of using *inheritance* to provide Duck behavior? (Choose all that apply.)

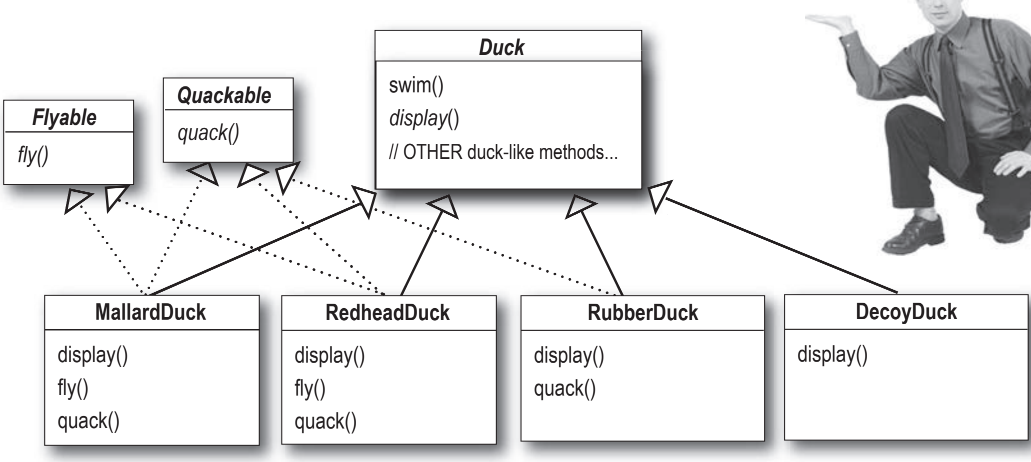
- ❑ A. Code is duplicated across subclasses.
- ❑ B. Runtime behavior changes are difficult.
- ❑ C. We can't make ducks dance.
- ❑ D. Hard to gain knowledge of all duck behaviors.
- ❑ E. Ducks can't fly and quack at the same time.
- ❑ F. Changes can unintentionally affect other ducks.

How about an 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.*

So, he needs a cleaner way to have only *some* (but not *all*) of the duck types fly or quack.





What do YOU think about this design?

That is, like, the dumbest idea you've come up with. **Can you say, "duplicate code"?** If you thought having to *override a few methods* was bad, how are you gonna feel when you need to make a little change to the flying behavior... *in all 48 of the flying Duck subclasses?!*

What would *you* do if you were Joe?

We know that not *all* of the subclasses should have flying or quacking behavior, so inheritance isn't the right answer. But while having the subclasses implement Flyable and/or Quackable solves *part* of the problem (no inappropriately flying rubber ducks), it completely destroys code reuse for those behaviors, so it just creates a *different* maintenance nightmare. And of course there might be more than one kind of flying behavior even among the ducks that *do* fly...

At this point you might be waiting for a Design Pattern to come riding in on a white horse and save the day. But what fun would that be? No, we're going to figure out a solution the old-fashioned way—*by applying good OO software design principles.*

Wouldn't it be dreamy if only there were a way to build software so that when we need to change it, we could do so with the least possible impact on the existing code? We could spend less time *reworking* code and more making the program do cooler things...

The one constant in software development

Okay, what's the one thing you can always count on in software development?

No matter where you work, what you're building, or what language you are programming in, what's the one true constant that will be with you always?

CHANGE

(use a mirror to see the answer)

No matter how well you design an application, over time an application must grow and change or it will *die*.

Sharpen your pencil

Lots of things can drive change. List some reasons you've had to change code in your applications (we put in a couple of our own to get you started).

My customers or users decide they want something else, or they want new functionality.

My company decided it is going with another database vendor and it is also purchasing its data from another supplier that uses a different data format. Argh!

Zeroing in on the problem...

So we know using inheritance hasn't worked out very well, since the duck behavior keeps changing across the subclasses, and it's not appropriate for *all* subclasses to have those behaviors. The Flyable and Quackable interface sounded promising at first—only ducks that really do fly will be Flyable, etc.—except Java interfaces have no implementation code, so no code reuse. And that means that whenever you need to modify a behavior, you're forced to track down and change it in all the different subclasses where that behavior is defined, probably introducing *new* bugs along the way!

Luckily, there's a design principle for just this situation.

Design Principle

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

Our first of many design principles. We'll spend more time on these thruoghout the book.

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!

In other words, if you've got some aspect of your code that is changing, say with every new requirement, then you know you've got a behavior that needs to be pulled out and separated from all the stuff that doesn't change.

Here's another way to think about this principle: ***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.***

As simple as this concept is, it forms the basis for almost every design pattern. All patterns provide a way to let *some part of a system vary independently of all other parts*.

Okay, time to pull the duck behavior out of the Duck classes!

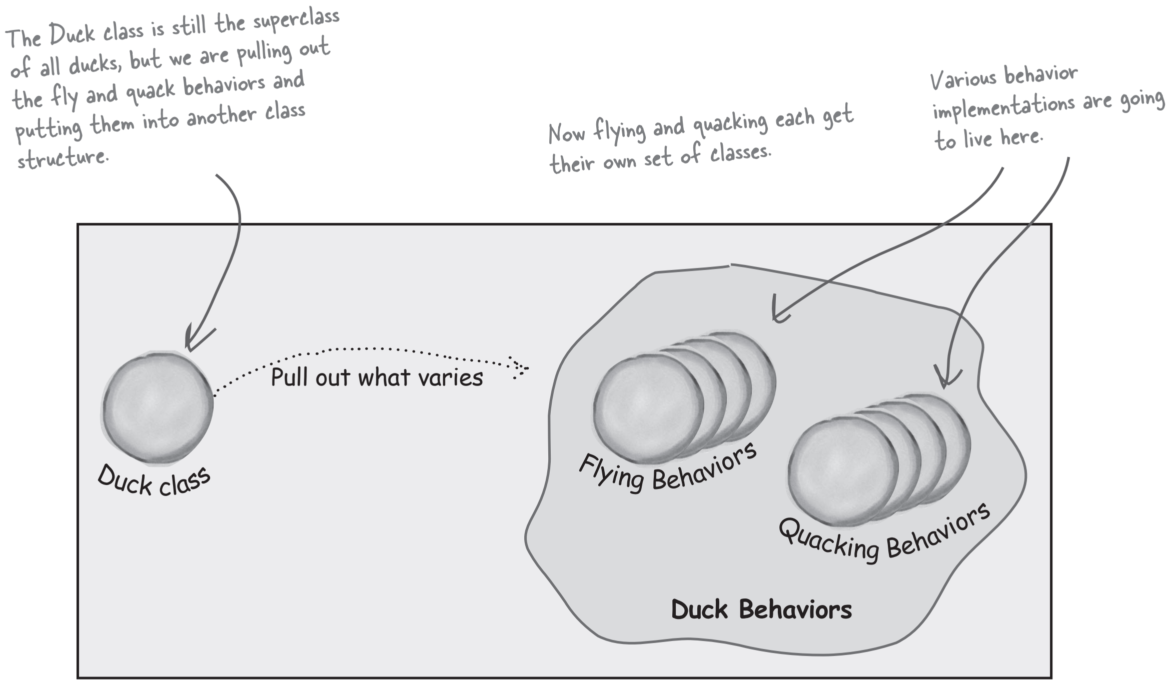
Separating what changes from what stays the same

Where do we start? As far as we can tell, other than the problems with fly() and quack(), the Duck class is working well and there are no other parts of it that appear to vary or change frequently. So, other than a few slight changes, we're going to pretty much leave the Duck class alone.

Now, to separate the "parts that change from those that stay the same", we are going to create two *sets* of classes (totally apart from Duck), one for *fly* and one for *quack*. Each set of classes will hold all the implementations of their respective behavior. For instance, we might have *one* class that implements *quacking*, *another* that implements *squeaking*, and *another* that implements *silence*.

We know that fly() and quack() are the parts of the Duck class that vary across ducks.

To separate these behaviors from the Duck class, we'll pull both methods *out* of the Duck class and create a new set of classes to represent each behavior.



Designing the Duck Behaviors

So how are we going to design the set of classes that implement the fly and quack behaviors?

We'd like to keep things flexible; after all, it was the inflexibility in the duck behaviors that got us into trouble in the first place. And we know that we want to *assign* behaviors to the instances of Duck. For example, we might want to instantiate a new MallardDuck instance and initialize it with a specific *type* of flying behavior. And while we're there, why not make sure that we can change the behavior of a duck dynamically? In other words, we should include behavior setter methods in the Duck classes so that we can, say, *change* the MallardDuck's flying behavior *at runtime*.

Given these goals, let's look at our second design principle:

Design Principle

Program to an interface, not an implementation.

We'll use an interface to represent each behavior – for instance, FlyBehavior and QuackBehavior – and each implementation of a *behavior* will implement one of those interfaces.

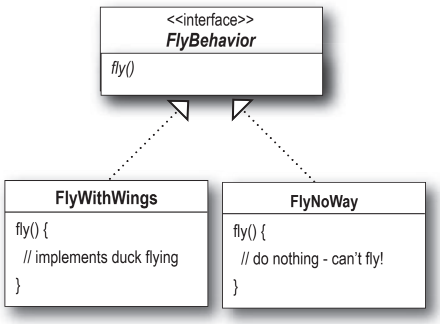
So this time it won't be the *Duck* classes that will implement the flying and quacking interfaces. Instead, we'll make a set of classes whose entire reason for living is to represent a behavior (for example, "squeaking"), and it's the *behavior* class, rather than the Duck class, that will implement the behavior interface.

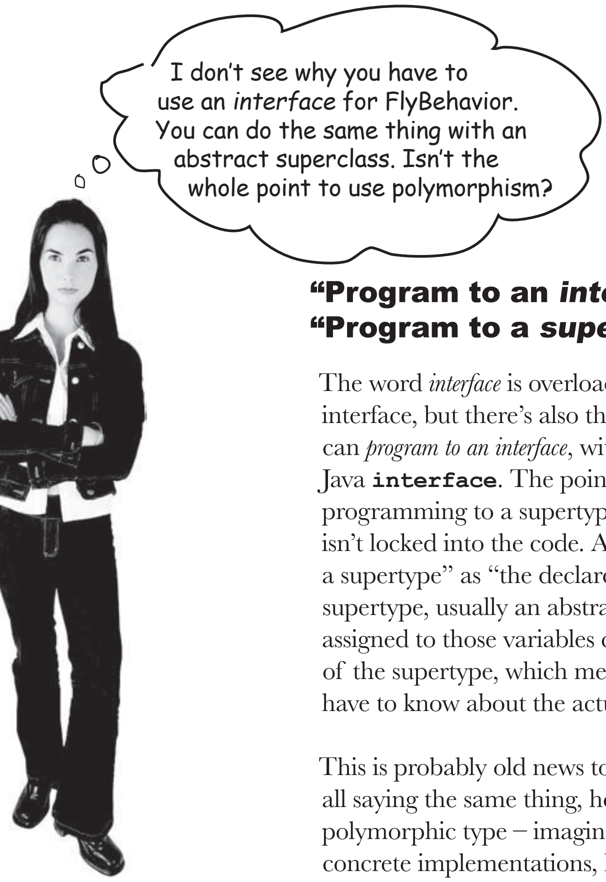
This is in contrast to the way we were doing things before, where a behavior either came from a concrete implementation in the superclass Duck, or by providing a specialized implementation in the subclass itself. In both cases we were relying on an *implementation*. We were locked into using that specific implementation and there was no room for changing out the behavior (other than writing more code).

With our new design, the Duck subclasses will use a behavior represented by an *interface* (FlyBehavior and QuackBehavior), so that the actual *implementation* of the behavior (in other words, the specific concrete behavior coded in the class that implements the FlyBehavior or QuackBehavior) won't be locked into the Duck subclass.

From now on, the Duck behaviors will live in a separate class—a class that implements a particular behavior interface.

That way, the Duck classes won't need to know any of the implementation details for their own behaviors.





"Program to an *interface*" really means "Program to a *supertype*."

The word *interface* is overloaded here. There's the *concept* of interface, but there's also the Java construct **`interface`**. You can *program to an interface*, without having to actually use a Java **`interface`**. The point is to exploit polymorphism by programming to a supertype so that the actual runtime object isn't locked into the code. And we could rephrase "program to a supertype" as "the declared type of the variables should be a supertype, usually an abstract class or interface, so that the objects assigned to those variables can be of any concrete implementation of the supertype, which means the class declaring them doesn't have to know about the actual object types!"

This is probably old news to you, but just to make sure we're all saying the same thing, here's a simple example of using a polymorphic type – imagine an abstract class Animal, with two concrete implementations, Dog and Cat.
Programming to an implementation would be:

```
Dog d = new Dog();
d.bark();
```

Declaring the variable "d" as type Dog (a concrete implementation of Animal) forces us to code to a concrete implementation.

But **programming to an interface/supertype** would be:

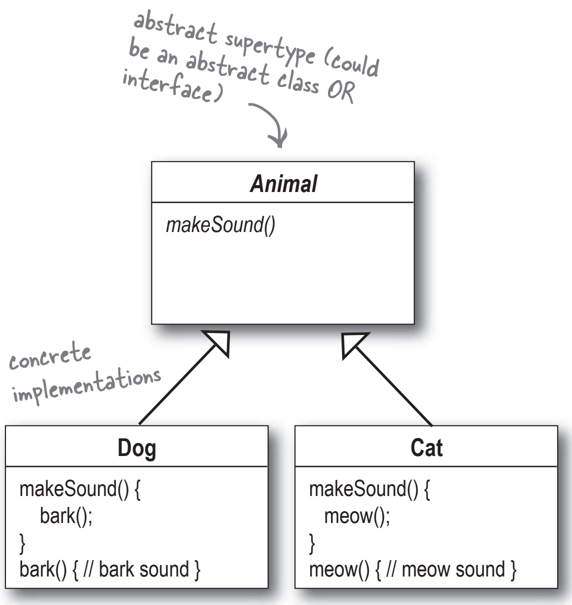
```
Animal animal = new Dog();
animal.makeSound();
```

We know it's a Dog, but we can now use the animal reference polymorphically.

Even better, rather than hard-coding the instantiation of the subtype (like new Dog()) into the code, **assign the concrete implementation object at runtime:**

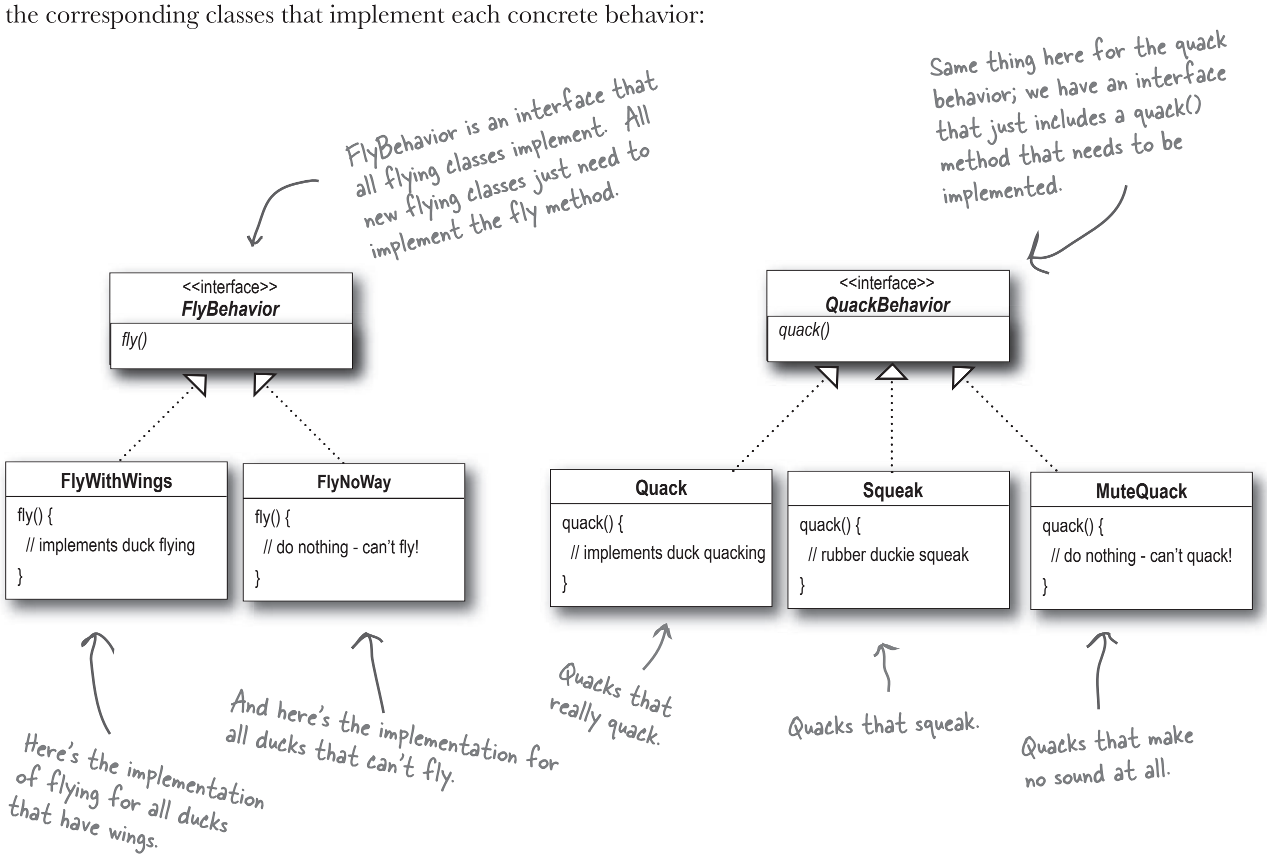
```
a = getAnimal();
a.makeSound();
```

We don't know WHAT the actual animal subtype is... all we care about is that it knows how to respond to makeSound().



Implementing the Duck Behaviors

Here we have the two interfaces, FlyBehavior and QuackBehavior along with the corresponding classes that implement each concrete behavior:



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

And we can add new behaviors without modifying any of our existing behavior classes or touching any of the Duck classes that *use* flying behaviors.

So we get the benefit of REUSE without all the baggage that comes along with inheritance.

there are no Dumb Questions

Q: Do I always have to implement my application first, see where things are changing, and then go back and separate & encapsulate those things?

A: Not always; often when you are designing an application, you anticipate those areas that are going to vary and then go ahead and build the flexibility to deal with it into your code. You'll find that the principles and patterns can be applied at any stage of the development lifecycle.

Q: Should we make Duck an interface too?

A: Not in this case. As you'll see once we've got everything hooked together, we do benefit by having Duck not be an interface and having specific ducks, like MallardDuck, inherit common properties and methods. Now that we've removed what varies from the Duck inheritance, we get the benefits of this structure without the problems.

Q: It feels a little weird to have a class that's just a behavior. Aren't classes supposed to represent *things? Aren't classes supposed to have both state AND behavior?*

A: In an OO system, yes, classes represent things that generally have both state (instance variables) and methods. And in this case, the *thing* happens to be a behavior. But even a behavior can still have state and methods; a flying behavior might have instance variables representing the attributes for the flying (wing beats per minute, max altitude and speed, etc.) behavior.

Sharpen your pencil

1. Using our new design, what would you do if you needed to add rocket-powered flying to the SimUDuck app?

2. Can you think of a class that might want to use the Quack behavior that isn't a duck?

Answers:
1) Create a FlyRocketPowered class that implements the FlyBehavior interface.
2) One example, a duck call (a device that makes duck sounds).

Integrating the Duck Behavior

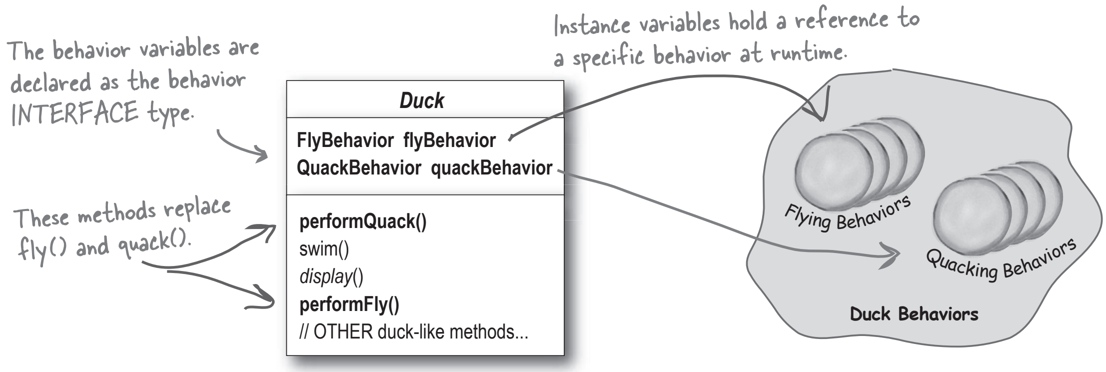
The key is that a Duck will now *delegate* its flying and quacking behavior, instead of using quacking and flying methods defined in the Duck class (or subclass).

Here's how:

1. **First we'll add two instance variables** to the Duck class called *flyBehavior* and *quackBehavior*, that are declared as the interface type (not a concrete class implementation type). Each duck object will set these variables polymorphically to reference the *specific* behavior type it would like at runtime (FlyWithWings, Squeak, etc.).

   We'll also remove the fly() and quack() methods from the Duck class (and any subclasses) because we've moved this behavior out into the FlyBehavior and QuackBehavior classes.

   We'll replace fly() and quack() in the Duck class with two similar methods, called performFly() and performQuack(); you'll see how they work next.



2. **Now we implement performQuack():**

```
public class Duck {
   QuackBehavior quackBehavior;
   // more

   public void performQuack() {
     quackBehavior.quack();
   }
}
```

Each Duck has a reference to something that implements the QuackBehavior interface.

Rather than handling the quack behavior itself, the Duck object delegates that behavior to the object referenced by quackBehavior.

Pretty simple, huh? To perform the quack, a Duck just allows the object that is referenced by quackBehavior to quack for it.
In this part of the code we don't care what kind of object it is, ***all we care about is that it knows how to quack()!***

More Integration...

3. Okay, time to worry about **how the flyBehavior and quackBehavior instance variables are set**. Let's take a look at the MallardDuck class:

```
public class MallardDuck extends Duck {

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

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.

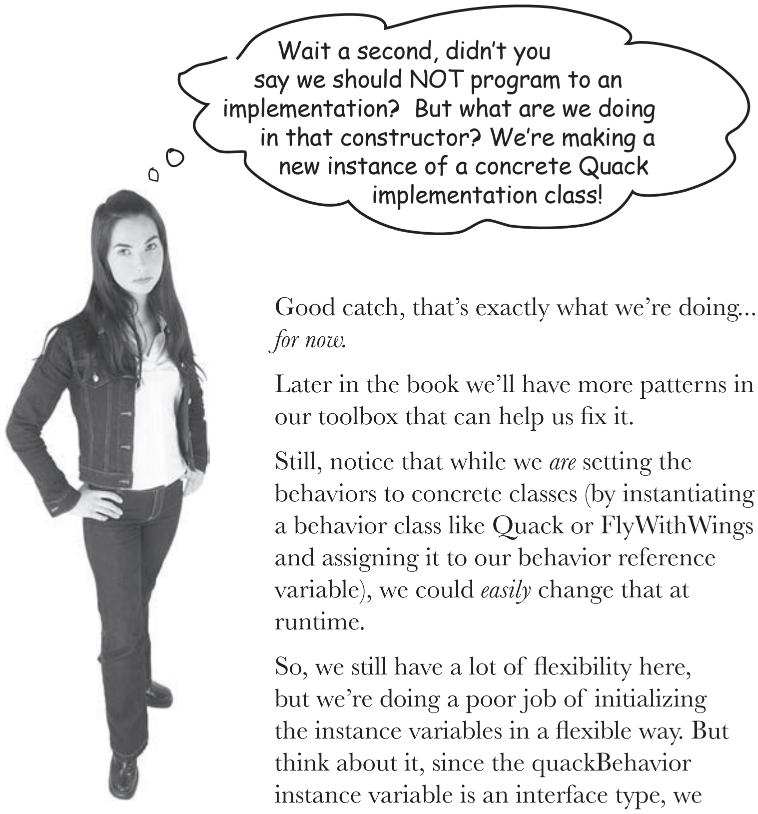
And it uses FlyWithWings as its FlyBehavior type.

Remember, MallardDuck inherits the quack-Behavior and flyBehavior instance variables from class Duck.

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

So MallardDuck's quack is a real live duck **quack**, not a **squeak** and not a **mute quack**. So what happens here? When a MallardDuck is instantiated, its constructor initializes the MallardDuck's inherited quackBehavior instance variable to a new instance of type Quack (a QuackBehavior concrete implementation class).

And the same is true for the duck's flying behavior—the MallardDuck's constructor initializes the flyBehavior instance variable with an instance of type FlyWithWings (a FlyBehavior concrete implementation class).



Good catch, that's exactly what we're doing... *for now.*

Later in the book we'll have more patterns in our toolbox that can help us fix it.

Still, notice that while we *are* setting the behaviors to concrete classes (by instantiating a behavior class like Quack or FlyWithWings and assigning it to our behavior reference variable), we could *easily* change that at runtime.

So, we still have a lot of flexibility here, but we're doing a poor job of initializing the instance variables in a flexible way. But think about it, since the quackBehavior instance variable is an interface type, we could (through the magic of polymorphism) dynamically assign a different QuackBehavior implementation class at runtime.

Take a moment and think about how you would implement a duck so that its behavior could change at runtime. (You'll see the code that does this a few pages from now.)

Testing the Duck code

1 **Type and compile the Duck class below (Duck.java), and the MallardDuck class from two pages back (MallardDuck.java).**

```
public abstract class Duck {

   FlyBehavior flyBehavior;
   QuackBehavior quackBehavior;
   public Duck() {
   }

   public abstract void display();

   public void performFly() {
      flyBehavior.fly();
   }

   public void performQuack() {
      quackBehavior.quack();
   }

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

Declare two reference variables for the behavior interface types. All duck subclasses (in the same package) inherit these.

Delegate to the behavior class.

2 **Type and compile the FlyBehavior interface (FlyBehavior.java) and the two behavior implementation classes (FlyWithWings.java and FlyNoWay.java).**

```
public interface FlyBehavior {
   public void fly();
}
```

The interface that all flying behavior classes implement.

```
public class FlyWithWings implements FlyBehavior {
   public void fly() {
       System.out.println("I'm flying!!");
   }
}
```

Flying behavior implementation for ducks that DO fly...

```
public class FlyNoWay implements FlyBehavior {
   public void fly() {
       System.out.println("I can't fly");
   }
}
```

Flying behavior implementation for ducks that do NOT fly (like rubber ducks and decoy ducks).

Testing the Duck code continued...

3 Type and compile the QuackBehavior interface (QuackBehavior.java) and the three behavior implementation classes (Quack.java, MuteQuack.java, and Sqeak.java).

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

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

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

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

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

```
public class MiniDuckSimulator {
   public static void main(String[] args) {
      Duck mallard = new MallardDuck();
      mallard.performQuack();
      mallard.performFly();
   }
}
```

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).

Then we do the same thing with MallardDuck's inherited performFly() method.

5 Run the code!

```
File Edit Window Help Yadayadayada
%java MiniDuckSimulator
Quack
I'm flying!!
```

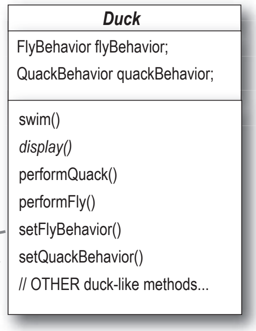
Setting behavior dynamically

What a shame to have all this dynamic talent built into our ducks and not be using it! Imagine you want to set the duck's behavior type through a setter method on the duck subclass, rather than by instantiating it in the duck's constructor.

1 Add two new methods to the Duck class:

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

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



We can call these methods anytime we want to change the behavior of a duck on the fly.

editor note: gratuitous pun - fix

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

```
public class ModelDuck extends Duck {
    public ModelDuck() {
       flyBehavior = new FlyNoWay();
       quackBehavior = new Quack();
    }

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

Our model duck begins life grounded... without a way to fly.

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

That's okay, we're creating a rocket powered flying behavior.

```
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.**

```
public class MiniDuckSimulator {
   public static void main(String[] args) {
      Duck mallard = new MallardDuck();
      mallard.performQuack();
      mallard.performFly();

      Duck model = new ModelDuck();
      model.performFly();
      model.setFlyBehavior(new FlyRocketPowered());
      model.performFly();
   }
}
```

The first call to performFly() delegates to the flyBehavior object set in the ModelDuck's constructor, which is a FlyNoWay instance.

This invokes the model's inherited behavior setter method, and...voila! The model suddenly has rocket-powered flying capability!

If it worked, the model duck dynamically changed its flying behavior! You can't do THAT if the implementation lives inside the duck class.

before

after

5 **Run it!**

```
File Edit Window Help Yabadabadoo
%java MiniDuckSimulator
Quack
I'm flying!!
I can't fly
I'm flying with a rocket
```

To change a duck's behavior at runtime, just call the duck's setter method for that behavior.

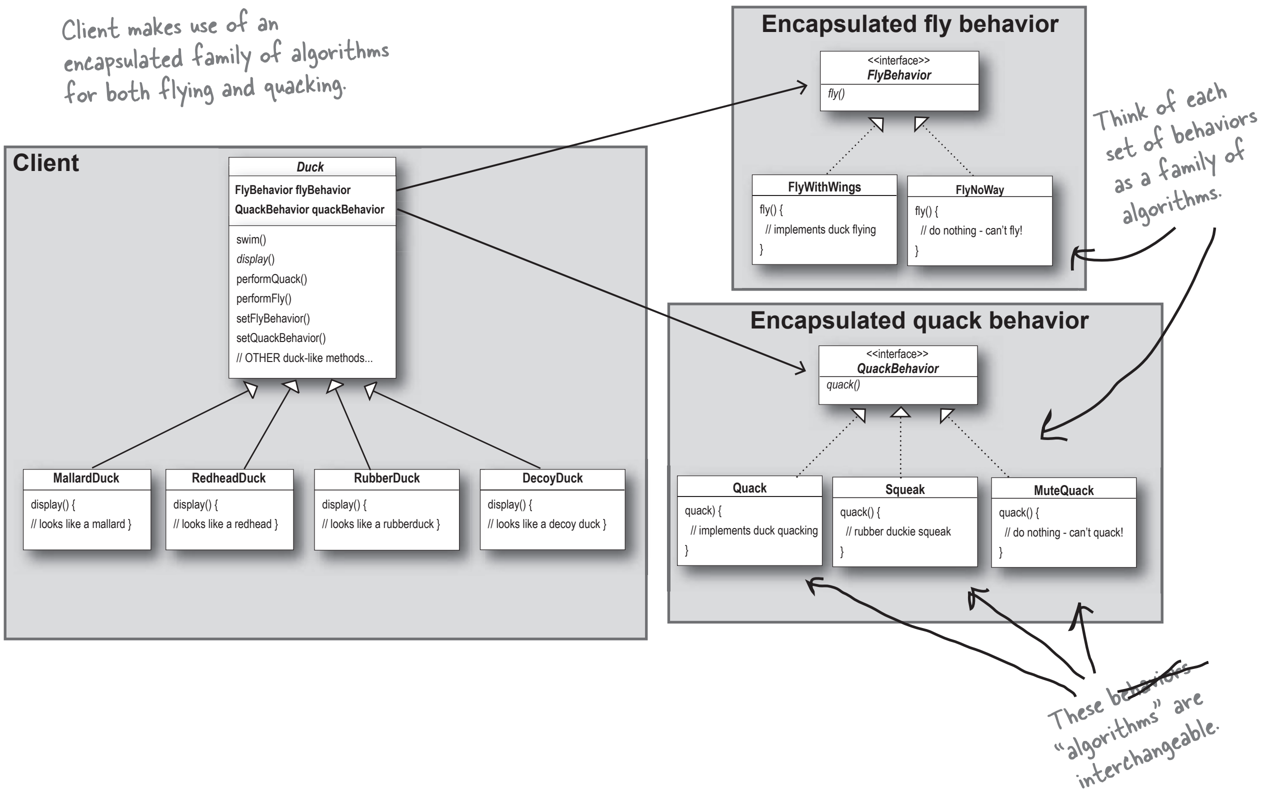
The Big Picture on encapsulated behaviors

Okay, now that we've done the deep dive on the duck simulator design, it's time to come back up for air and take a look at the big picture.

Below is the entire reworked class structure. We have everything you'd expect: ducks extending Duck, fly behaviors implementing FlyBehavior and quack behaviors implementing QuackBehavior.

Notice also that we've started to describe things a little differently. Instead of thinking of the duck behaviors as a *set of behaviors*, we'll start thinking of them as a *family of algorithms*. Think about it: in the SimUDuck design, the algorithms represent things a duck would do (different ways of quacking or flying), but we could just as easily use the same techniques for a set of classes that implement the ways to compute state sales tax by different states.

Pay careful attention to the *relationships* between the classes. In fact, grab your pen and write the appropriate relationship (IS-A, HAS-A and IMPLEMENTS) on each arrow in the class diagram.



HAS-A can be better than IS-A

The HAS-A relationship is an interesting one: each duck has a FlyBehavior and a QuackBehavior to which it delegates flying and quacking.

When you put two classes together like this you're using ***composition***. Instead of *inheriting* their behavior, the ducks get their behavior by being *composed* with the right behavior object.

This is an important technique; in fact, we've been using our third design principle:

> ***Design Principle***
>
> *Favor composition over inheritance.*

As you've seen, creating systems using composition gives you a lot more flexibility. Not only does it let you encapsulate a family of algorithms into their own set of classes, but it also lets you ***change behavior at runtime*** as long as the object you're composing with implements the correct behavior interface.

Composition is used in many design patterns and you'll see a lot more about its advantages and disadvantages throughout the book.

BRAIN POWER

A duck call is a device that hunters use to mimic the calls (quacks) of ducks. How would you implement your own duck call that does *not* inherit from the Duck class?

Master and Student...

Master: *Grasshopper, tell me what you have learned of the Object-Oriented ways.*

Student: *Master, I have learned that the promise of the object-oriented way is reuse.*

Master: *Grasshopper, continue...*

Student: *Master, through inheritance all good things may be reused and so we will come to drastically cut development time like we swiftly cut bamboo in the woods.*

Master: *Grasshopper, is more time spent on code* ***before*** *or* ***after*** *development is complete?*

Student: *The answer is* ***after****, Master. We always spend more time maintaining and changing software than initial development.*

Master: *So Grasshopper, should effort go into reuse* ***above*** *maintaintability and extensibility?*

Student: *Master, I believe that there is truth in this.*

Master: *I can see that you still have much to learn. I would like for you to go and meditate on inheritance further. As you've seen, inheritance has its problems, and there are other ways of achieving reuse.*

Speaking of Design Patterns...

Congratulations on your first pattern!

You just applied your first design pattern—the STRATEGY pattern. That's right, you used the Strategy Pattern to rework the SimUDuck app. Thanks to this pattern, the simulator is ready for any changes those execs might cook up on their next business trip to Vegas.

Now that we've made you take the long road to apply it, here's the formal definition of this pattern:

> **The Strategy Pattern** defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Use THIS definition when you need to impress friends and influence key executives.

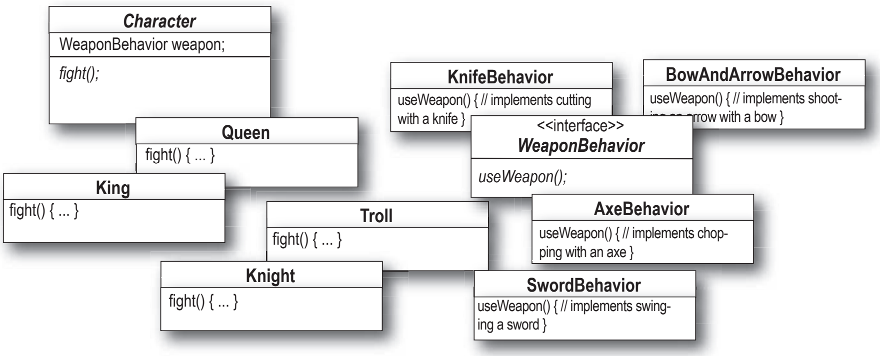
Design Puzzle

Below you'll find a mess of classes and interfaces for an action adventure game. You'll find classes for game characters along with classes for weapon behaviors the characters can use in the game. Each character can make use of one weapon at a time, but can change weapons at any time during the game. Your job is to sort it all out...

(Answers are at the end of the chapter.)

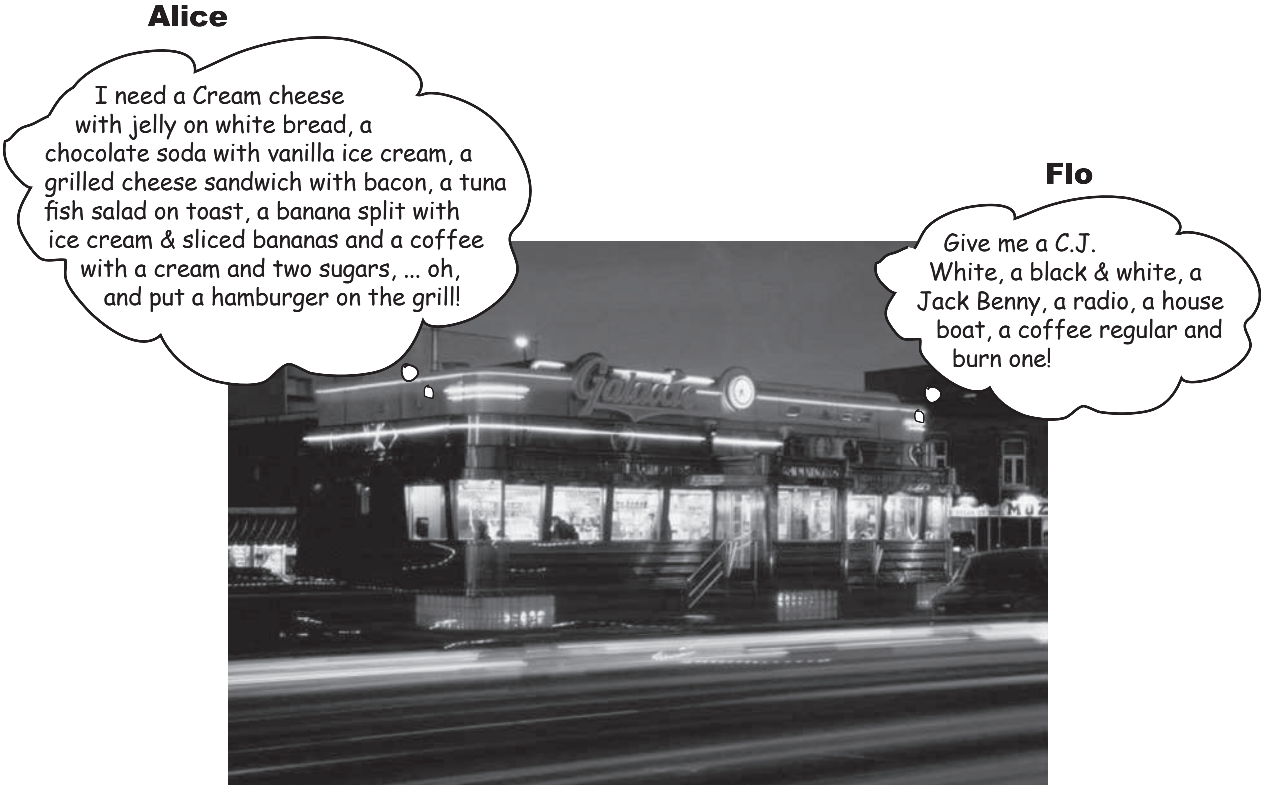
Your task:

1. Arrange the classes.
2. Identify one abstract class, one interface and eight classes.
3. Draw arrows between classes.
   a. Draw this kind of arrow for inheritance ("extends").
   b. Draw this kind of arrow for interface ("implements").
   c. Draw this kind of arrow for "HAS-A".
4. Put the method setWeapon() into the right class.




```
setWeapon(WeaponBehavior w) {
    this.weapon = w;
}
```

Overheard at the local diner...

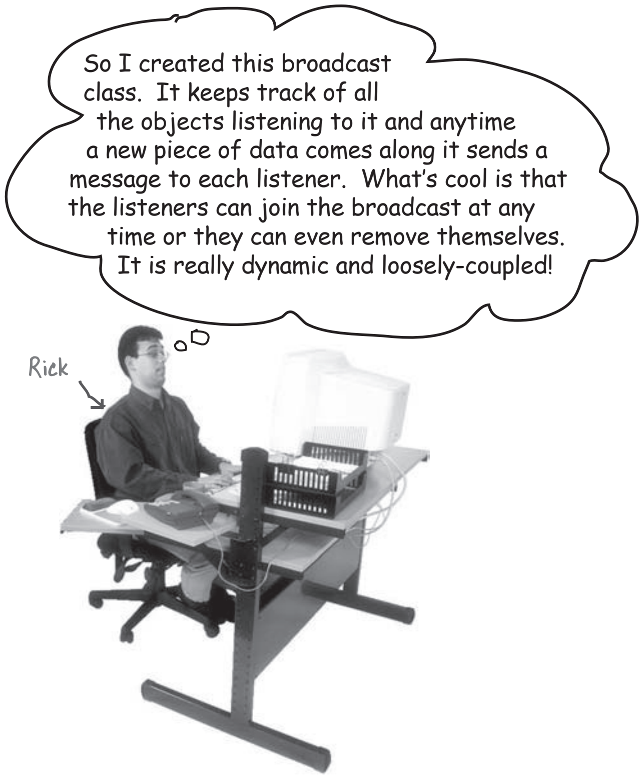


What's the difference between these two orders? Not a thing! They're both the same order, except Alice is using twice the number of words and trying the patience of a grumpy short order cook.

What's Flo got that Alice doesn't? **A shared vocabulary** with the short order cook. Not only is it easier to communicate with the cook, but it gives the cook less to remember because he's got all the diner patterns in his head.

Design Patterns give you a shared vocabulary with other developers. Once you've got the vocabulary you can more easily communicate with other developers and inspire those who don't know patterns to start learning them. It also elevates your thinking about architectures by letting you **think at the *pattern* level**, not the nitty gritty *object* level.

Overheard in the next cubicle...



BRAIN POWER

Can you think of other shared vocabularies that are used beyond OO design and diner talk? (Hint: how about auto mechanics, carpenters, gourmet chefs, air traffic control) What qualities are communicated along with the lingo?

Can you think of aspects of OO design that get communicated along with pattern names? What qualities get communicated along with the name "Strategy Pattern"?

The power of a shared pattern vocabulary

When you communicate using patterns you are doing more than just sharing LINGO.

Shared pattern vocabularies are POWERFUL. When you communicate with another developer or your team using patterns, you are communicating not just a pattern name but a whole set of qualities, characteristics and constraints that the pattern represents.

"We're using the strategy pattern to implement the various behaviors of our ducks." This tells you the duck behavior has been encapsulated into its own set of classes that can be easily expanded and changed, even at runtime if needed.

Patterns allow you to say more with less. When you use a pattern in a description, other developers quickly know precisely the design you have in mind.

Talking at the pattern level allows you to stay "in the design" longer. Talking about software systems using patterns allows you to keep the discussion at the design level, without having to dive down to the nitty gritty details of implementing objects and classes.

How many design meetings have you been in that quickly degrade into implementation details?

Shared vocabularies can turbo charge your development team. A team well versed in design patterns can move more quickly with less room for misunderstanding.

As your team begins to share design ideas and experience in terms of patterns, you will build a community of patterns users.

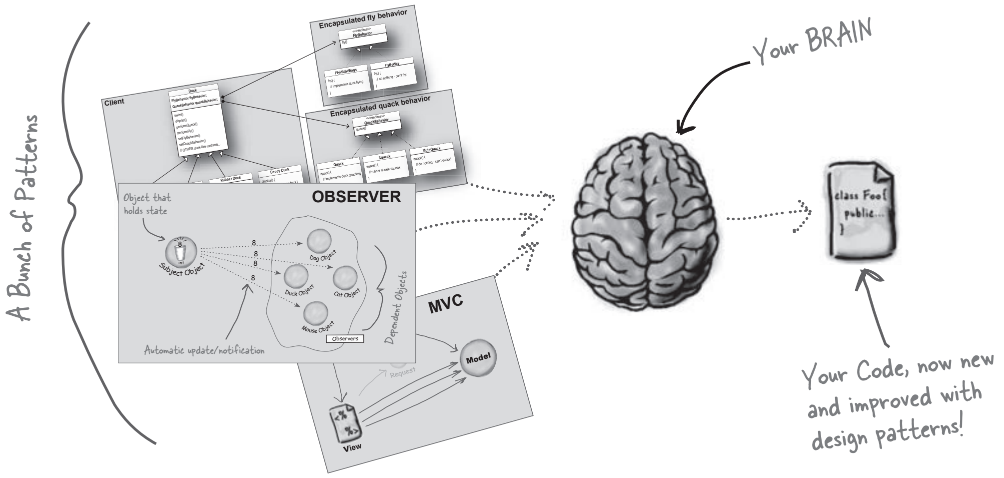
Shared vocabularies encourage more junior developers to get up to speed. Junior developers look up to experienced developers. When senior developers make use of design patterns, junior developers also become motivated to learn them. Build a community of pattern users at your organization.

Think about starting a patterns study group at your organization, maybe you can even get paid while you're learning... ;)

How do I use Design Patterns?

We've all used off-the-shelf libraries and frameworks. We take them, write some code against their APIs, compile them into our programs, and benefit from a lot of code someone else has written. Think about the Java APIs and all the functionality they give you: network, GUI, IO, etc. Libraries and frameworks go a long way towards a development model where we can just pick and choose components and plug them right in. But... they don't help us structure our own applications in ways that are easier to understand, more maintainable and flexible. That's where Design Patterns come in.

Design patterns don't go directly into your code, they first go into your BRAIN. Once you've loaded your brain with a good working knowledge of patterns, you can then start to apply them to your new designs, and rework your old code when you find it's degrading into an inflexible mess of jungle spaghetti code.



there are no Dumb Questions

Q: If design patterns are so great, why can't someone build a library of them so I don't have to?

A: Design patterns are higher level than libraries. Design patterns tell us how to structure classes and objects to solve certain problems and it is our job to adapt those designs to fit our particular application.

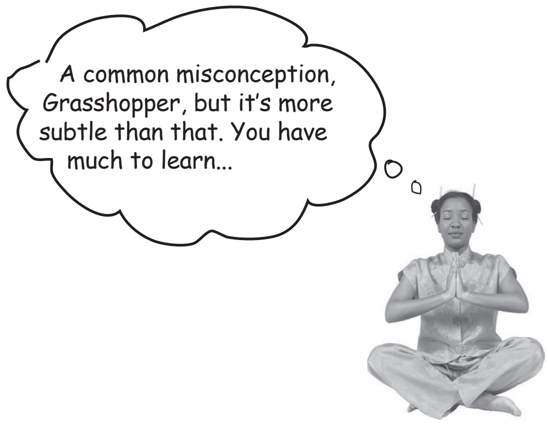
Q: Aren't libraries and frameworks also design patterns?

A: Frameworks and libraries are not design patterns; they provide specific implementations that we link into our code. Sometimes, however, libraries and frameworks make use of design patterns in their implementations. That's great, because once you understand design patterns, you'll more quickly understand APIs that are structured around design patterns.

Q: So, there are no libraries of design patterns?

A: No, but you will learn later about pattern catalogs with lists of patterns that you can apply to your applications.

Skeptical Developer



Friendly Patterns Guru

Developer: Okay, hmm, but isn't this all just good object-oriented design; I mean as long as I follow encapsulation and I know about abstraction, inheritance, and polymorphism, do I really need to think about Design Patterns? Isn't it pretty straightforward? Isn't this why I took all those OO courses? I think Design Patterns are useful for people who don't know good OO design.

Guru: Ah, this is one of the true misunderstandings of object-oriented development: that by knowing the OO basics we are automatically going to be good at building flexible, reusable, and maintainable systems.

Developer: No?

Guru: No. As it turns out, constructing OO systems that have these properties is not always obvious and has been discovered only through hard work.

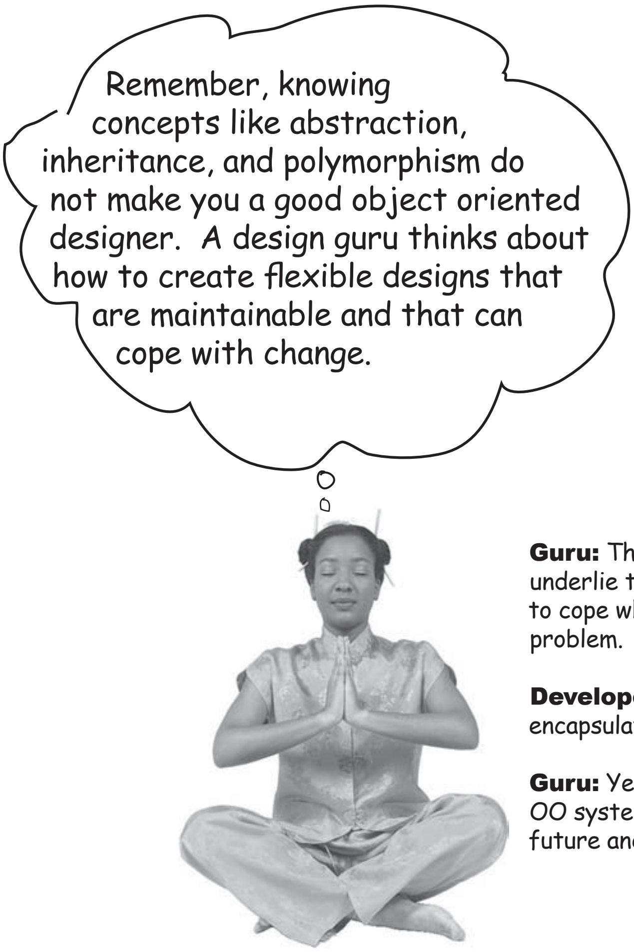
Developer: I think I'm starting to get it. These, sometimes non-obvious, ways of constructing object-oriented systems have been collected...

Guru: ...yes, into a set of patterns called Design Patterns.

Developer: So, by knowing patterns, I can skip the hard work and jump straight to designs that always work?

Guru: Yes, to an extent, but remember, design is an art. There will always be tradeoffs. But, if you follow well thought-out and time-tested design patterns, you'll be way ahead.

Developer: What do I do if I can't find a pattern?



Guru: There are some object oriented-principles that underlie the patterns, and knowing these will help you to cope when you can't find a pattern that matches your problem.

Developer: Principles? You mean beyond abstraction, encapsulation, and...

Guru: Yes, one of the secrets to creating maintainable OO systems is thinking about how they might change in the future and these principles address those issues.

Tools for your Design Toolbox

You've nearly made it through the first chapter! You've already put a few tools in your OO toolbox; let's make a list of them before we move on to Chapter 2.

OO Basics

Abstraction

Encapsulation

Polymorphism

Inheritance

We assume you know the OO basics of using classes polymorphically, how inheritance is like design by contract, and how encapsulation works. If you are a little rusty on these, pull out your Head First Java and review, then skim this chapter again.

OO Principles

Encapsulate what varies.

Favor composition over inheritence.

Program to interfaces, not implementations.

We'll be taking a closer look at these down the road and also adding a few more to the list

Throughout the book think about how patterns rely on OO basics and principles.

OO Patterns

Strategy - defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

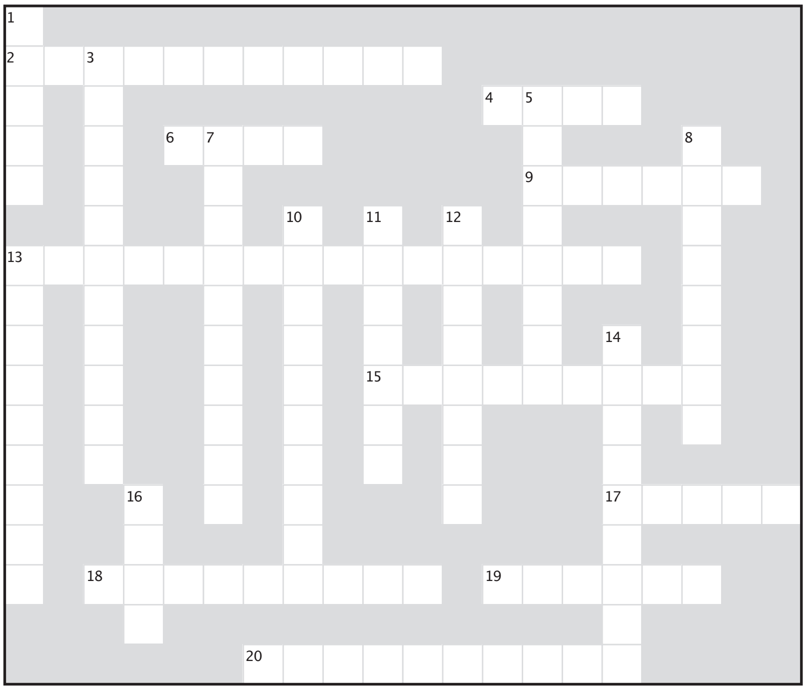
One down, many to go!

BULLET POINTS

- Knowing the OO basics does not make you a good OO designer.
- Good OO designs are reusable, extensible and maintainable.
- Patterns show you how to build systems with good OO design qualities.
- Patterns are proven object-oriented experience.
- Patterns don't give you code, they give you general solutions to design problems. You apply them to your specific application.
- Patterns aren't *invented*, they are *discovered*.
- Most patterns and principles address issues of *change* in software.
- Most patterns allow some part of a system to vary independently of all other parts.
- We often try to take what varies in a system and encapsulate it.
- Patterns provide a shared language that can maximize the value of your communication with other developers.

Let’s give your right brain something to do.

It’s your standard crossword; all of the solution words are from this chapter.



Across
2. ________ what varies
4. Design patterns ______
6. Java IO, Networking, Sound
9. Rubberducks make a
13. Bartender thought they were called
15. Program to this, not an implementation
17. Patterns go into your ________
18. Learn from the other guy's
19. Development constant
20. Patterns give us a shared ________

Down
1. Patterns _____ in many applications
3. Favor over inheritance
5. Dan was thrilled with this pattern
7. Most patterns follow from OO ________
8. Not your own
10. High level libraries
11. Joe's favorite drink
12. Pattern that fixed the simulator
13. Duck that can't quack
14. Grilled cheese with bacon
16. Duck demo was located where