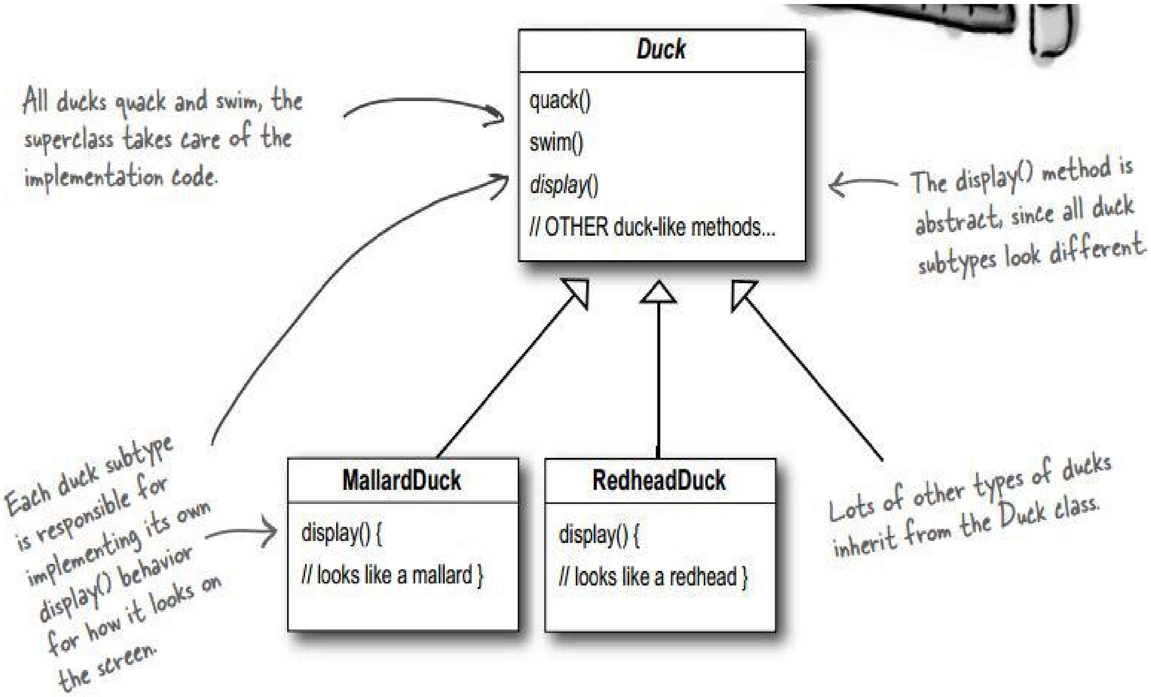
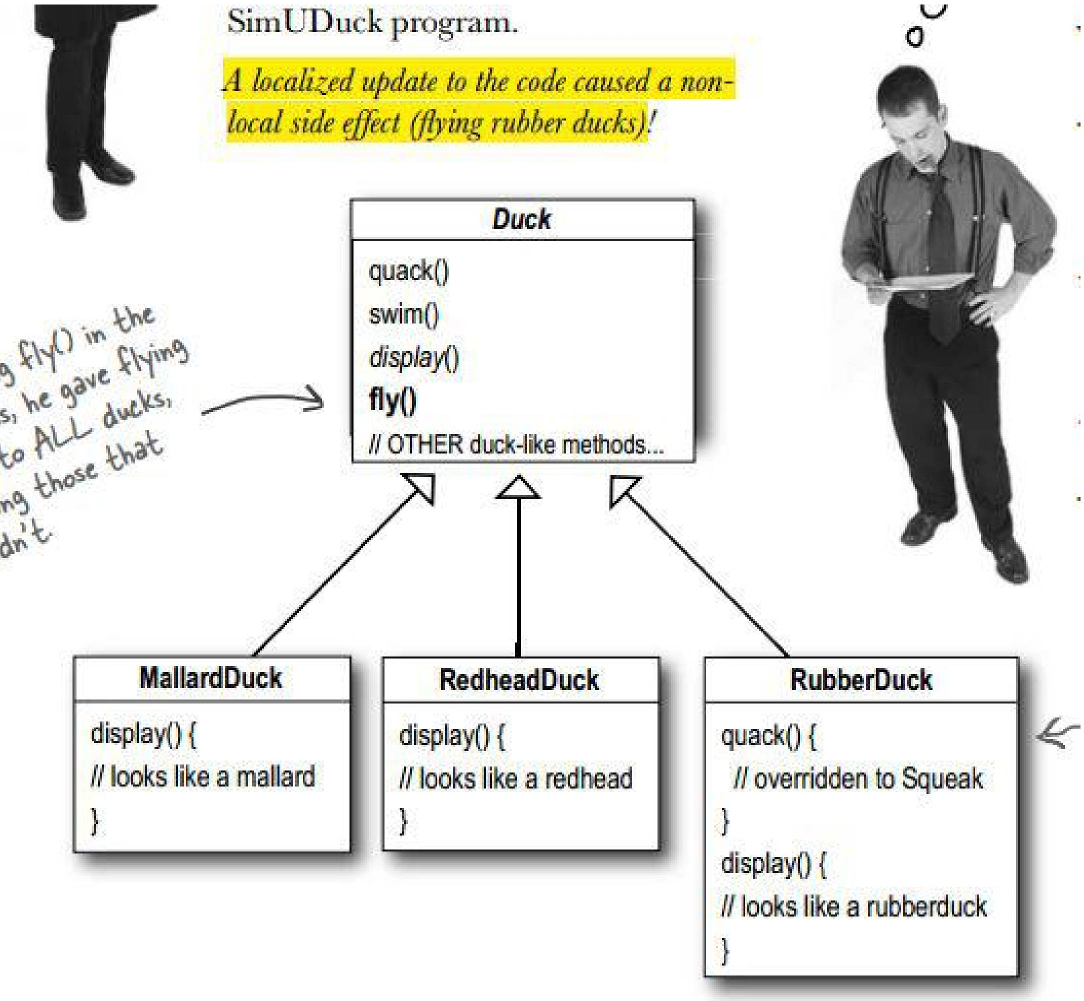
Strategy Patterns


Duck
quack()
swim()
display()
// OTHER duck-like methods...
MallardDuck
display() {
// looks like a mallard }
RedheadDuck
display() {
// looks like a redhead }
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() behavior for how it looks on the screen.
Lots of other types of ducks inherit from the Duck class.

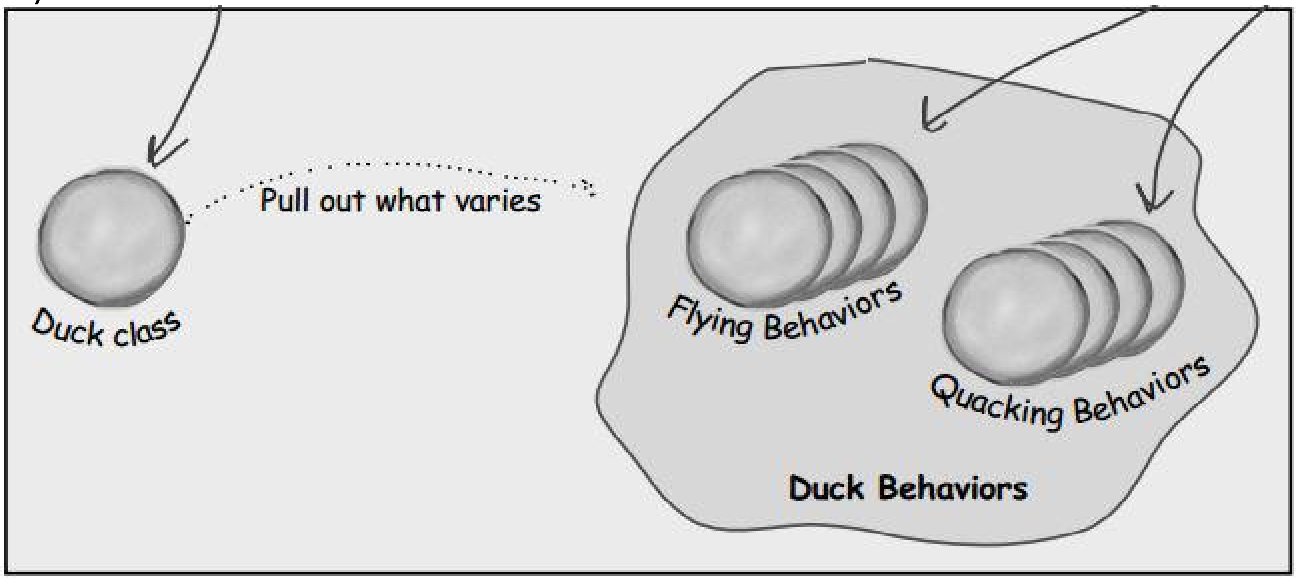
But now we need the ducks to FLY:

Strategy Patterns


SimUDuck program.
A localized update to the code caused a non-local side effect (flying rubber ducks)!
By putting fly() in the superclass, he gave flying ability to ALL ducks, including those that shouldn't.
Duck
quack()
swim()
display()
fly()
// OTHER duck-like methods...
MallardDuck
display() {
// looks like a mallard
}
RedheadDuck
display() {
// looks like a redhead
}
RubberDuck
quack() {
// overridden to Squeak
}
display() {
// looks like a rubberduck
}

Design Principle:

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

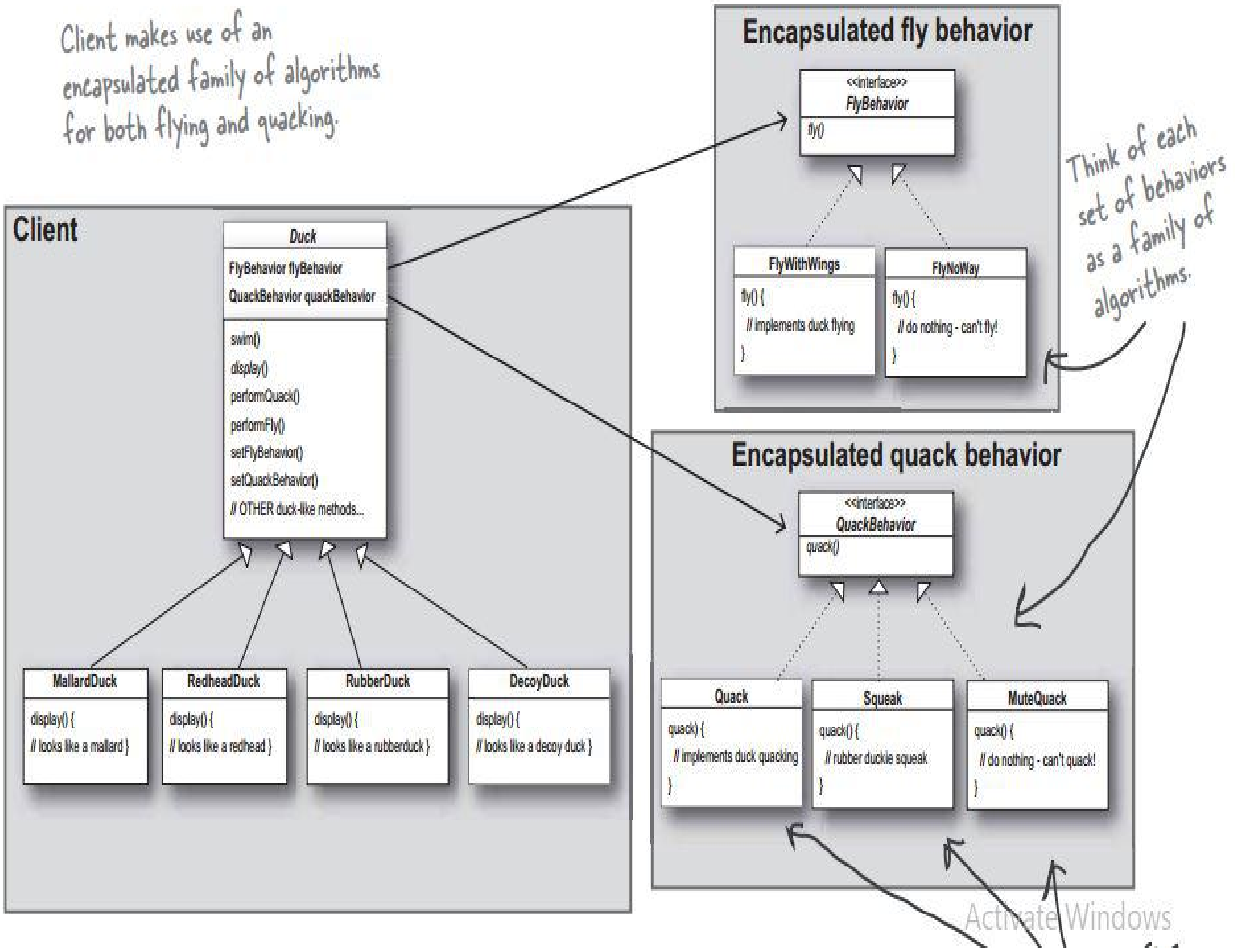


Strategy Patterns

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.


Client makes use of an encapsulated family of algorithms for both flying and quacking.
Client
Duck
FlyBehavior flyBehavior
QuackBehavior quackBehavior
swim()
display()
performQuack()
performFly()
setFlyBehavior()
setQuackBehavior()
// OTHER duck-like methods...
MallardDuck
display() {
// looks like a mallard }
RedheadDuck
display() {
// looks like a redhead }
RubberDuck
display() {
// looks like a rubberduck }
DecoyDuck
display() {
// looks like a decoy duck }
Encapsulated fly behavior
<<interface>>
FlyBehavior
fly()
FlyWithWings
fly() {
// implements duck flying
}
FlyNoWay
fly() {
// do nothing - can't fly!
}
Think of each set of behaviors as a family of algorithms.
Encapsulated quack behavior
<<interface>>
QuackBehavior
quack()
Quack
quack) {
// implements duck quacking
}
Squeak
quack() {
// rubber duckie squeak
}
MuteQuack
quack() {
// do nothing - can't quack!
}

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 useflying behaviors.

Strategy Patterns

HAS-A can be better than IS-A:

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.

Design Principle:

Favor composition over inheritance

it also lets you change behavior at runtime.

Strategy 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.