Einführung in Entwurfsmuster

Entwurfsmuster von Kopf bis Fuß

Warum Design Patterns

- Sourcecode
  - leichter zu implementieren
  - einfacher zu erweitern
  - Wartbarkeit

- Programmieren
  - Steigerung der Effizienz
  - Software-Design
  - Qualität der Projekte

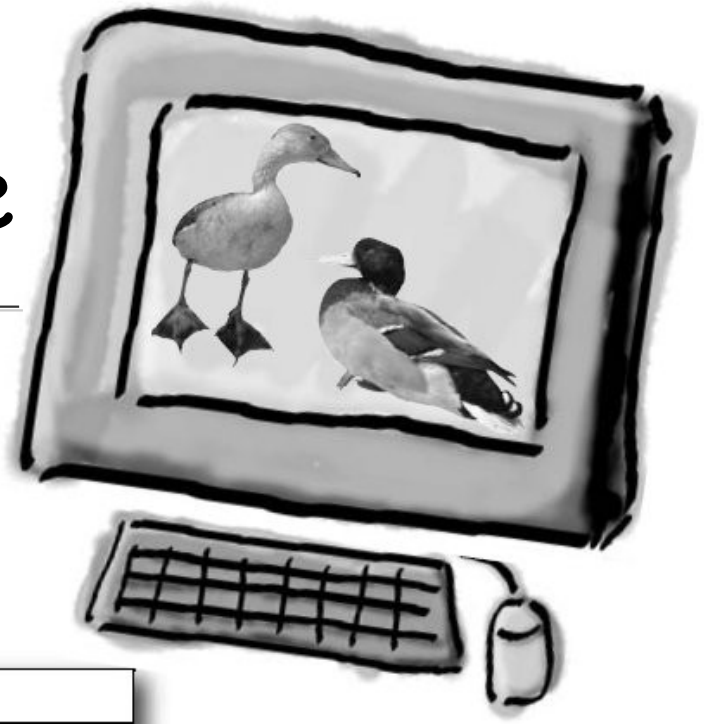
Basis von Design Patterns

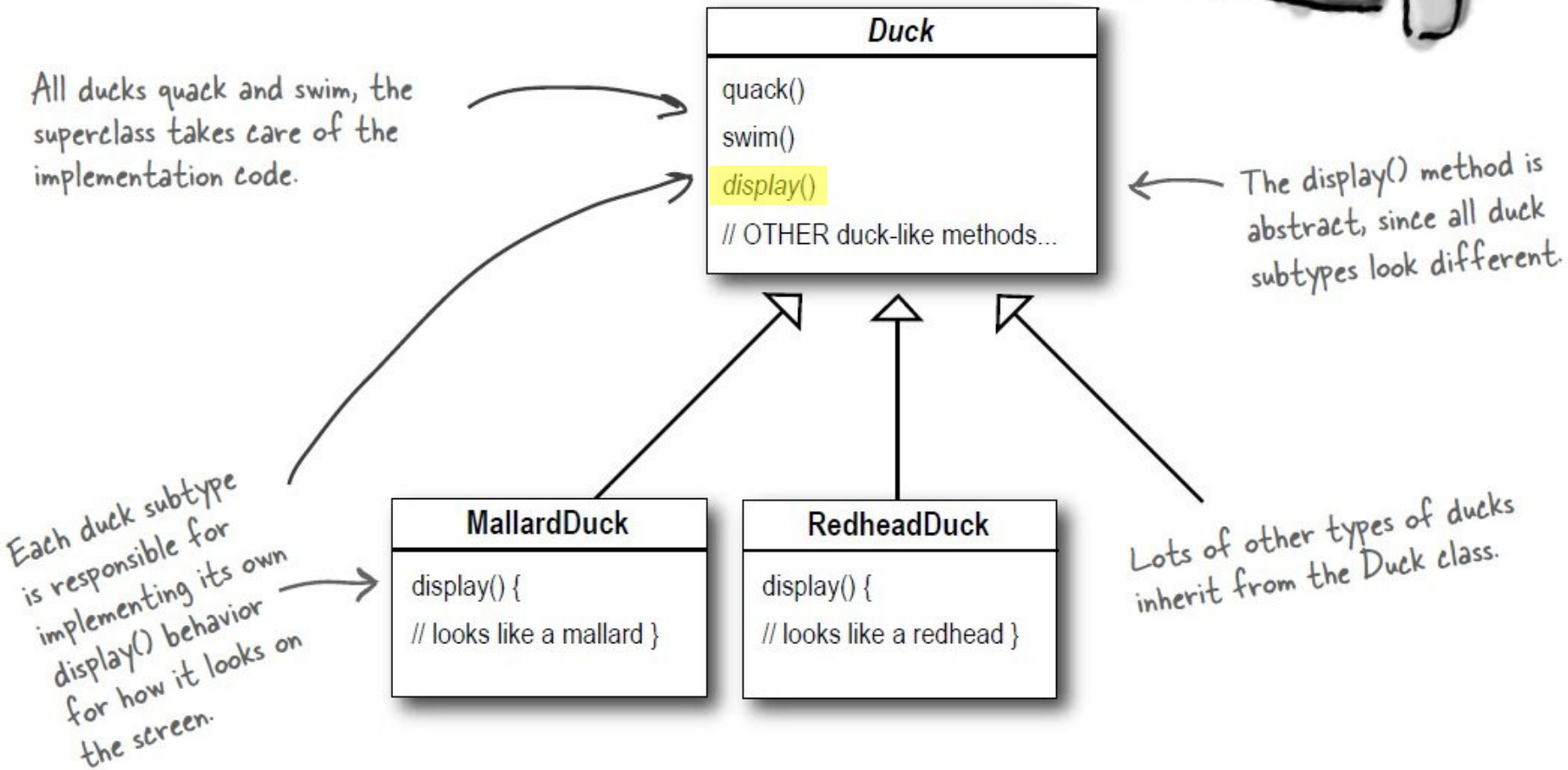
- Objekt-orientiertes Paradigma
  - Abstraktion
  - Kapselung
  - Polymorphie
  - Vererbung

Was sind Design Patterns?

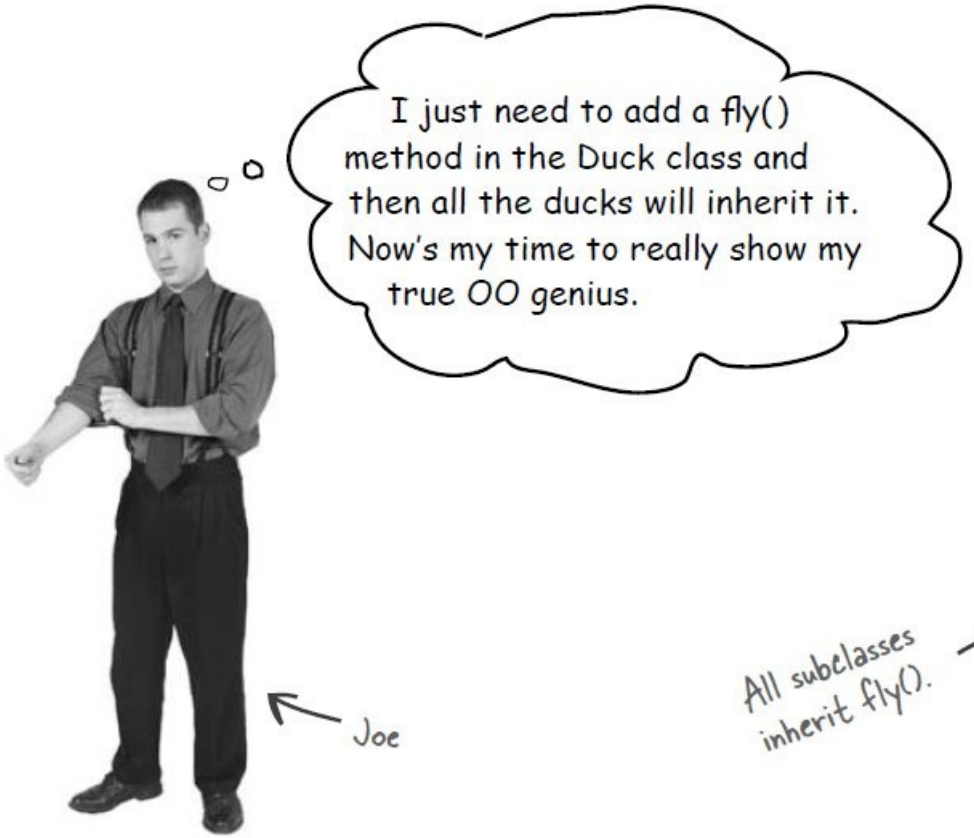
- Lösungen zu Programmierproblemen
- Gute Design-Techniken (best practice)
- Erste Veröffentlichung von der **G**ang **o**f **F**our
  - Erich Gamma
  - Richard Helm
  - Ralph Johnson
  - John Vlissides

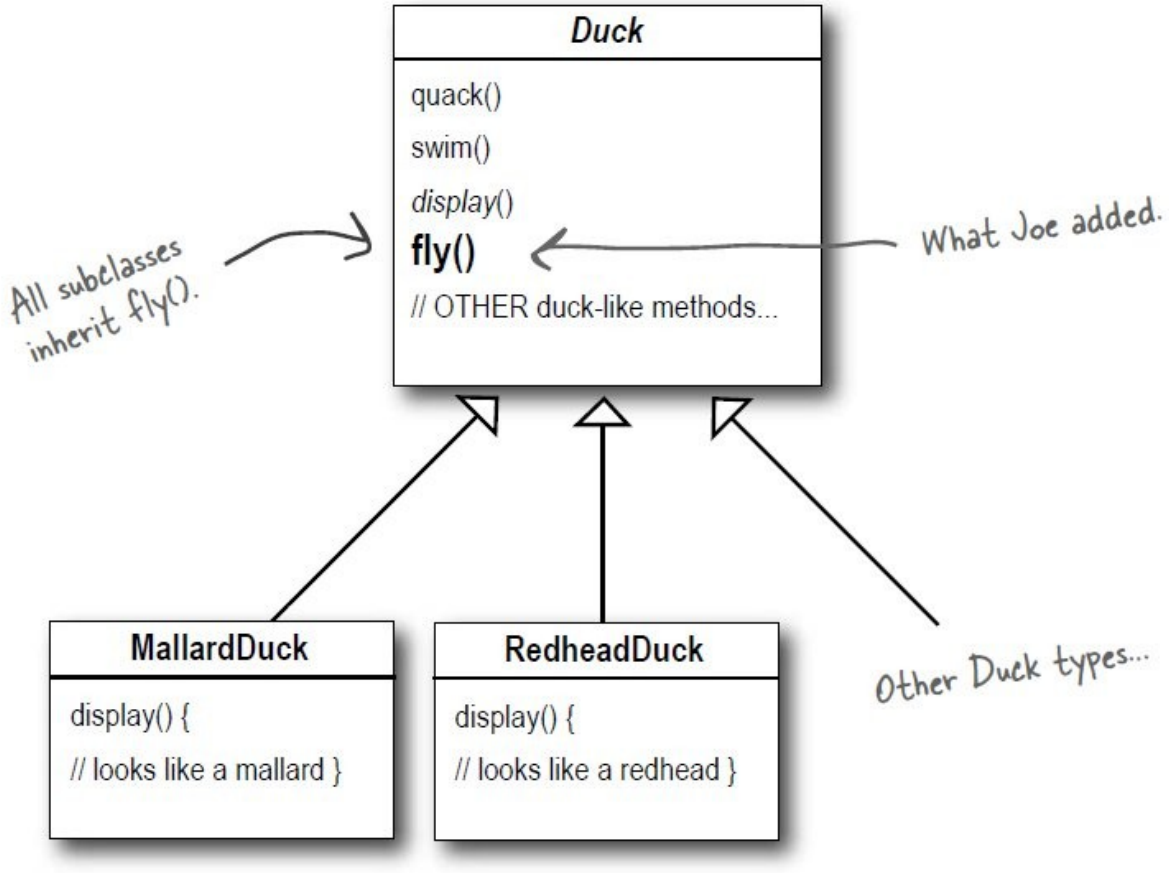
Alles begann mit SimEnte





Enten müssen fliegen können



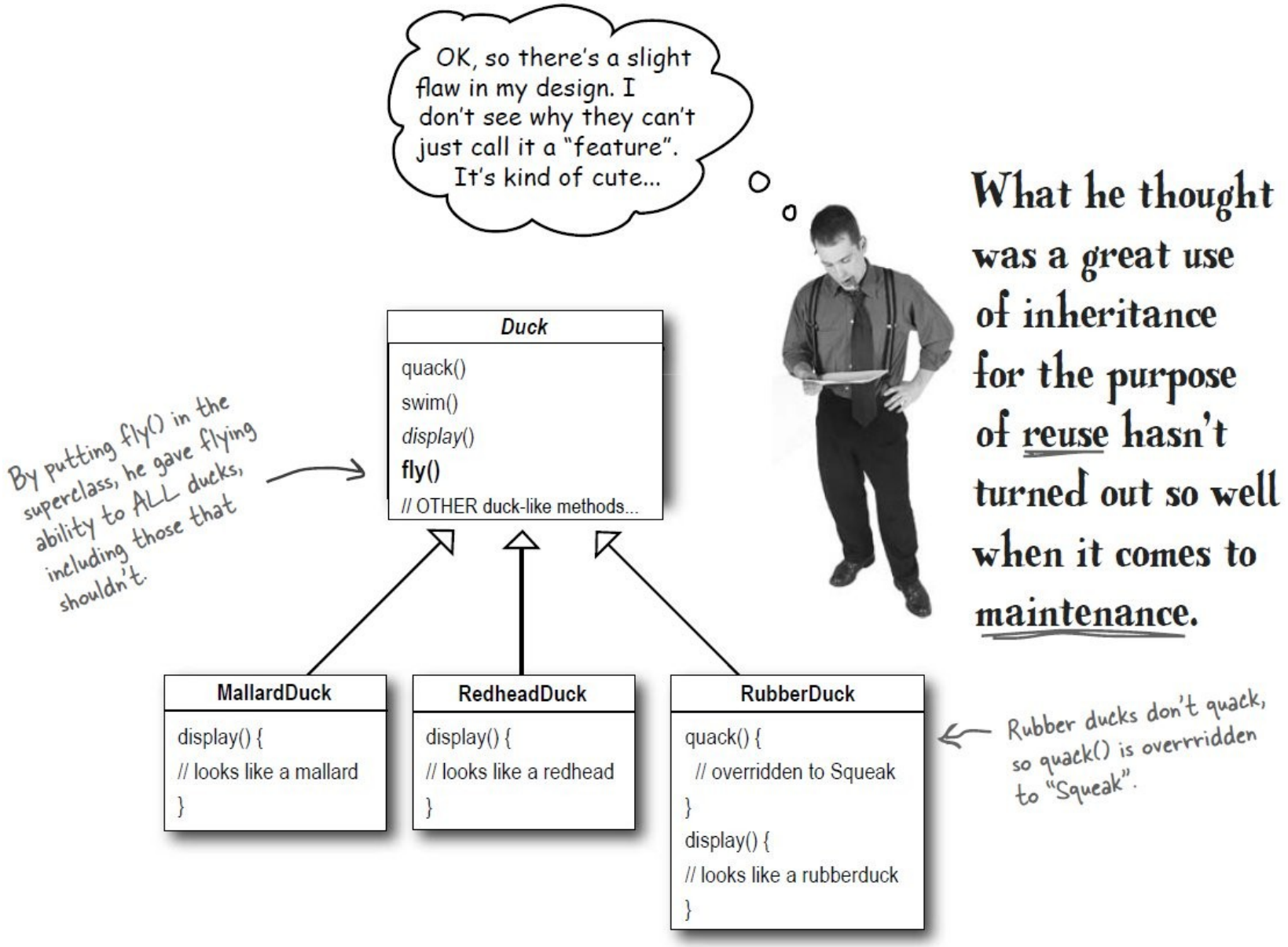


Achtung!

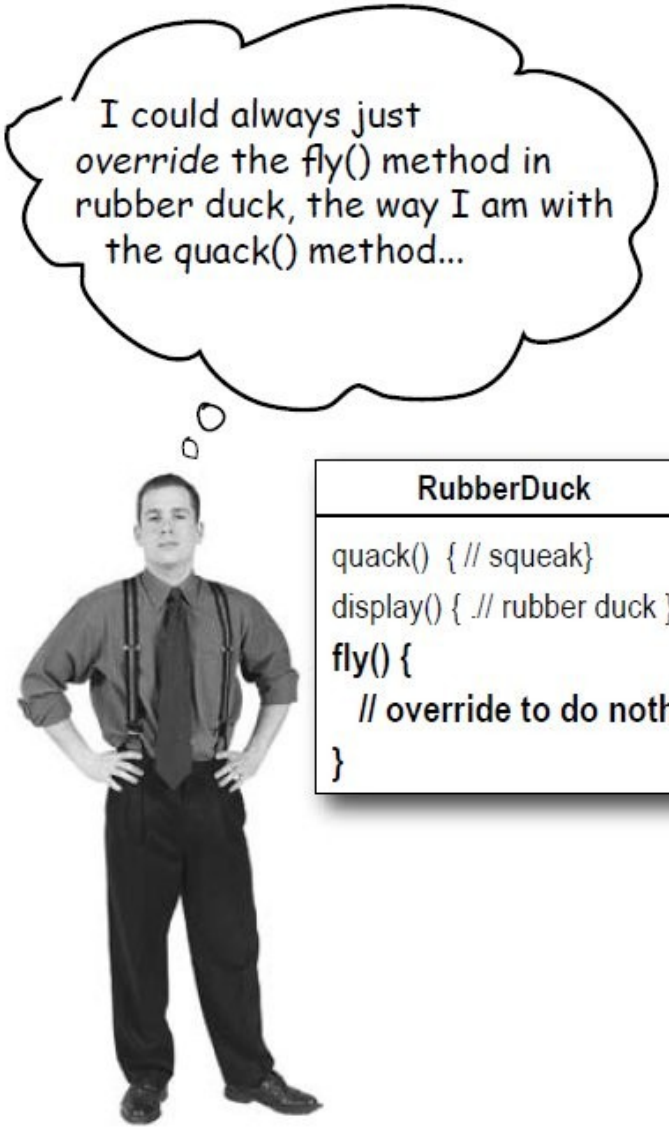
Lokale Änderung des Codes,

führte zu einem nicht lokalen Nebeneffekt

Fliegende Gummienten???



Lösungsweg?



RubberDuck

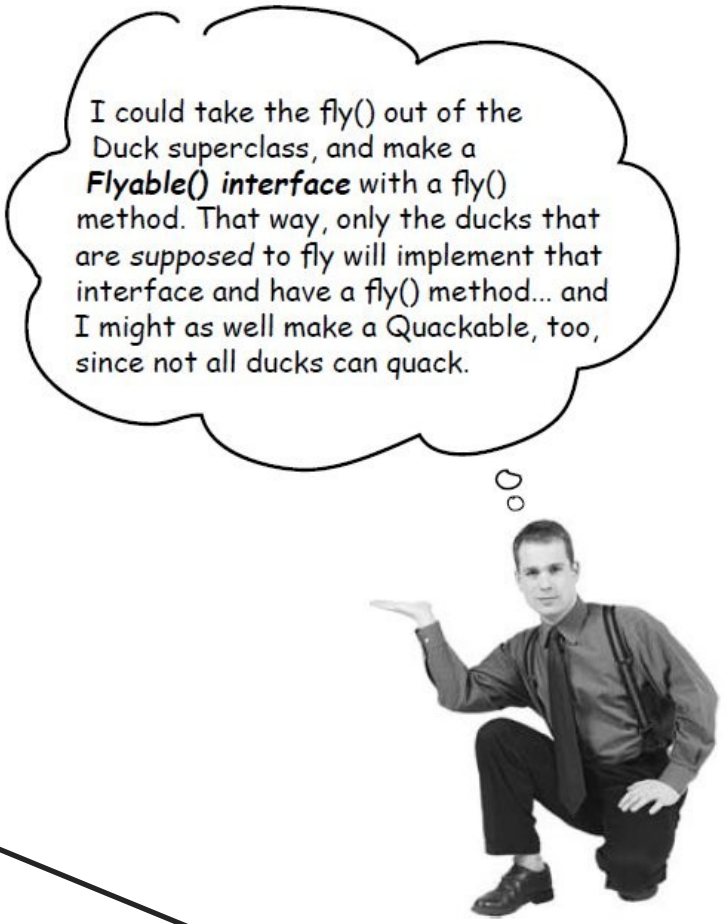
```
quack()  { // squeak}
display() { .// rubber duck }
fly() {
  // override to do nothing
}
```

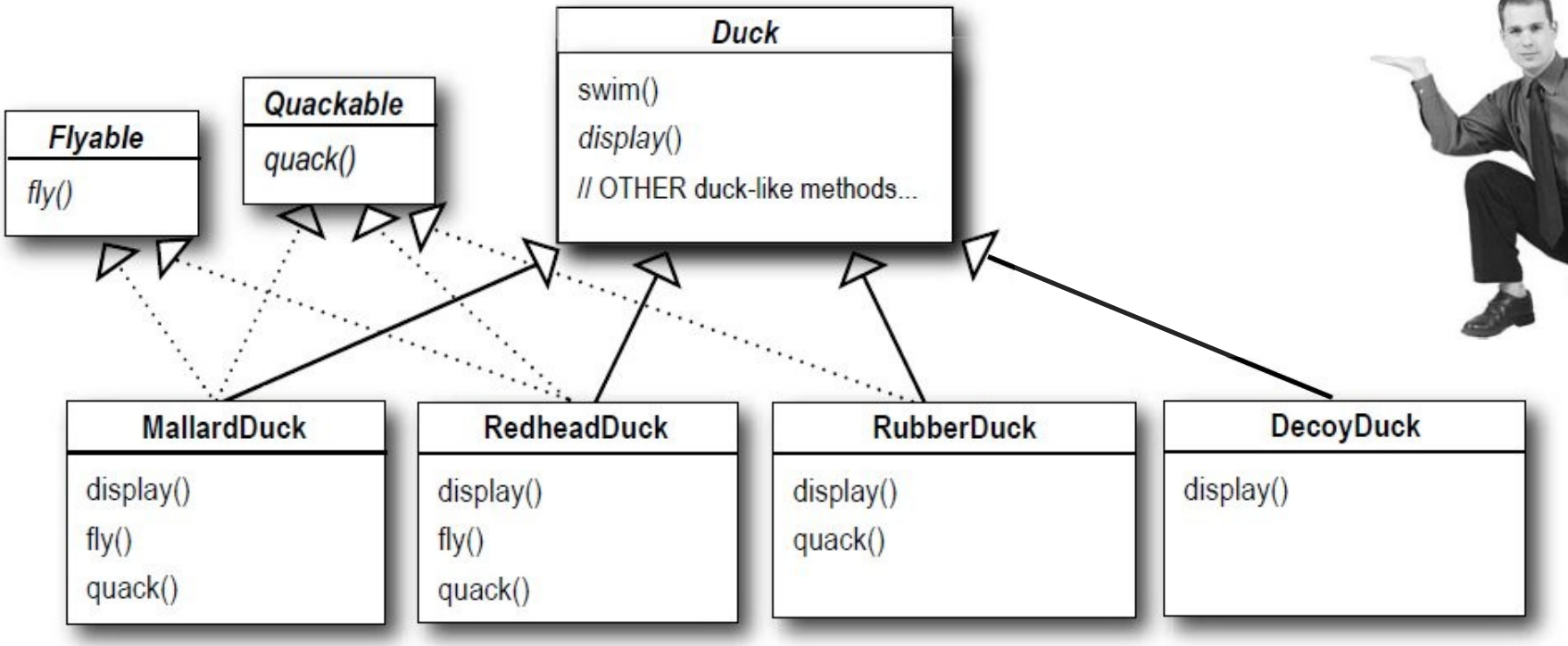
DecoyDuck

```
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.

Lösungsweg?


I could take the fly() out of the Duck superclass, and make a Flyable() interface with a fly() method. That way, only the ducks that are supposed to fly will implement that interface and have a fly() method... and I might as well make a Quackable, too, since not all ducks can quack.


Duck
swim()
display()
// OTHER duck-like methods...
Flyable
fly()
Quackable
quack()
MallardDuck
display()
fly()
quack()
RedheadDuck
display()
fly()
quack()
RubberDuck
display()
quack()
DecoyDuck
display()

Problem: Codeverdopplung

Ein typisches Beispiel

- Es gibt eine Konstante im Bereich SW-Engineering:

Das Problem einkreisen

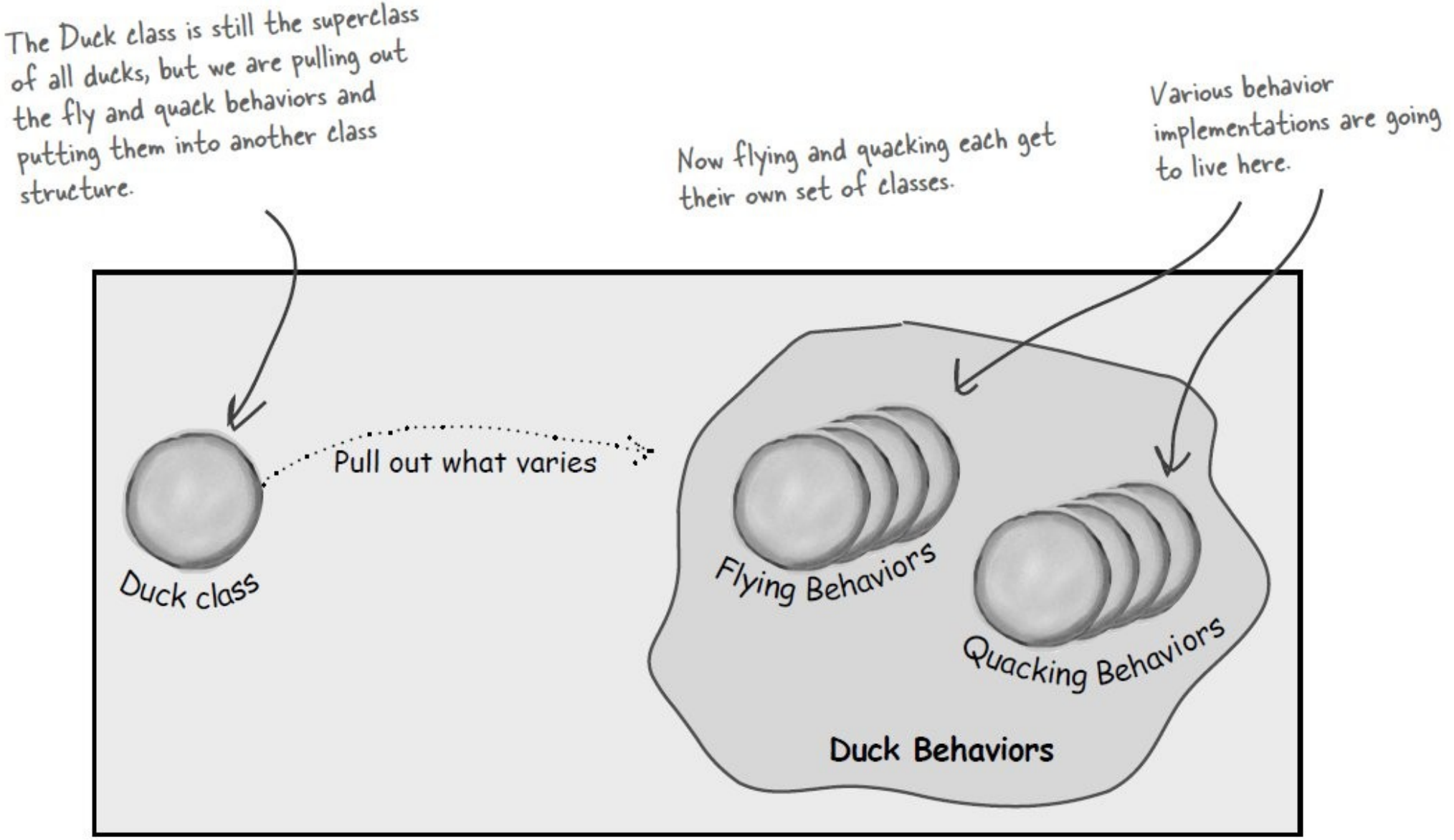
- Design Prinzip:

Design Principle

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

Ist Teil aller Entwurfsmuster!!!

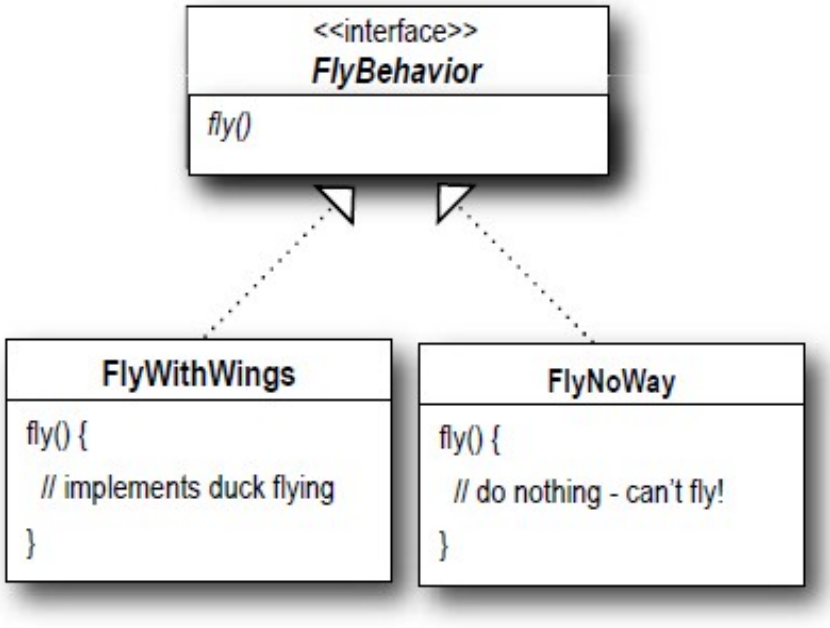
Verhalten extrahieren



Entenverhalten entwerfen

- Design Prinzip:

- Schnittstelle :=
  - Interface
  - Abstrakte Klasse



Auf eine Schnittstelle implementieren

= auf einen Supertyp implementieren!

- Auf eine Implementierung programmieren:

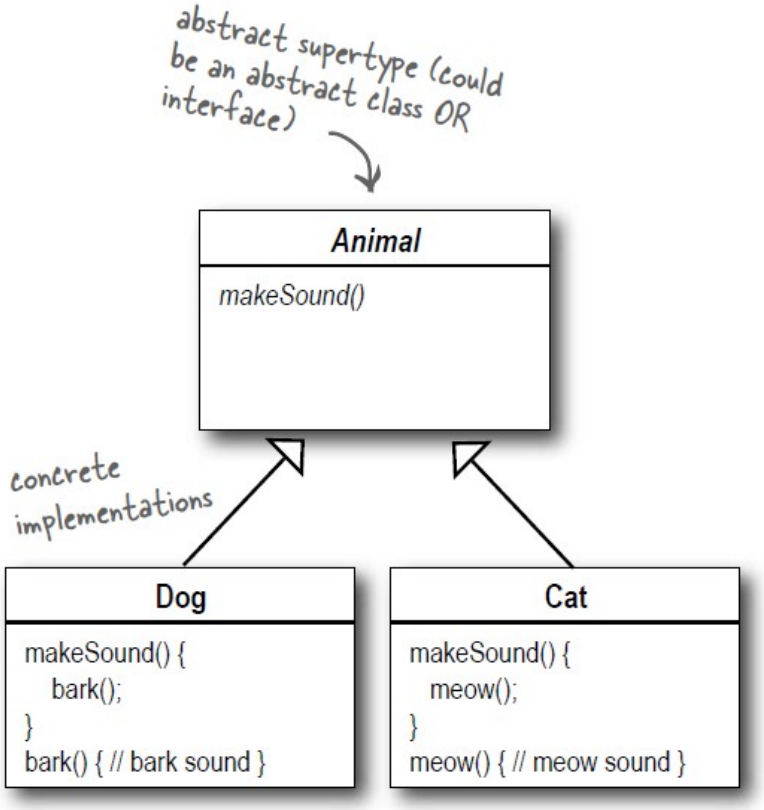
```
Dog d = new Dog();
d.bark();
```

- Auf eine Schnittstelle/Supertyp programmieren:

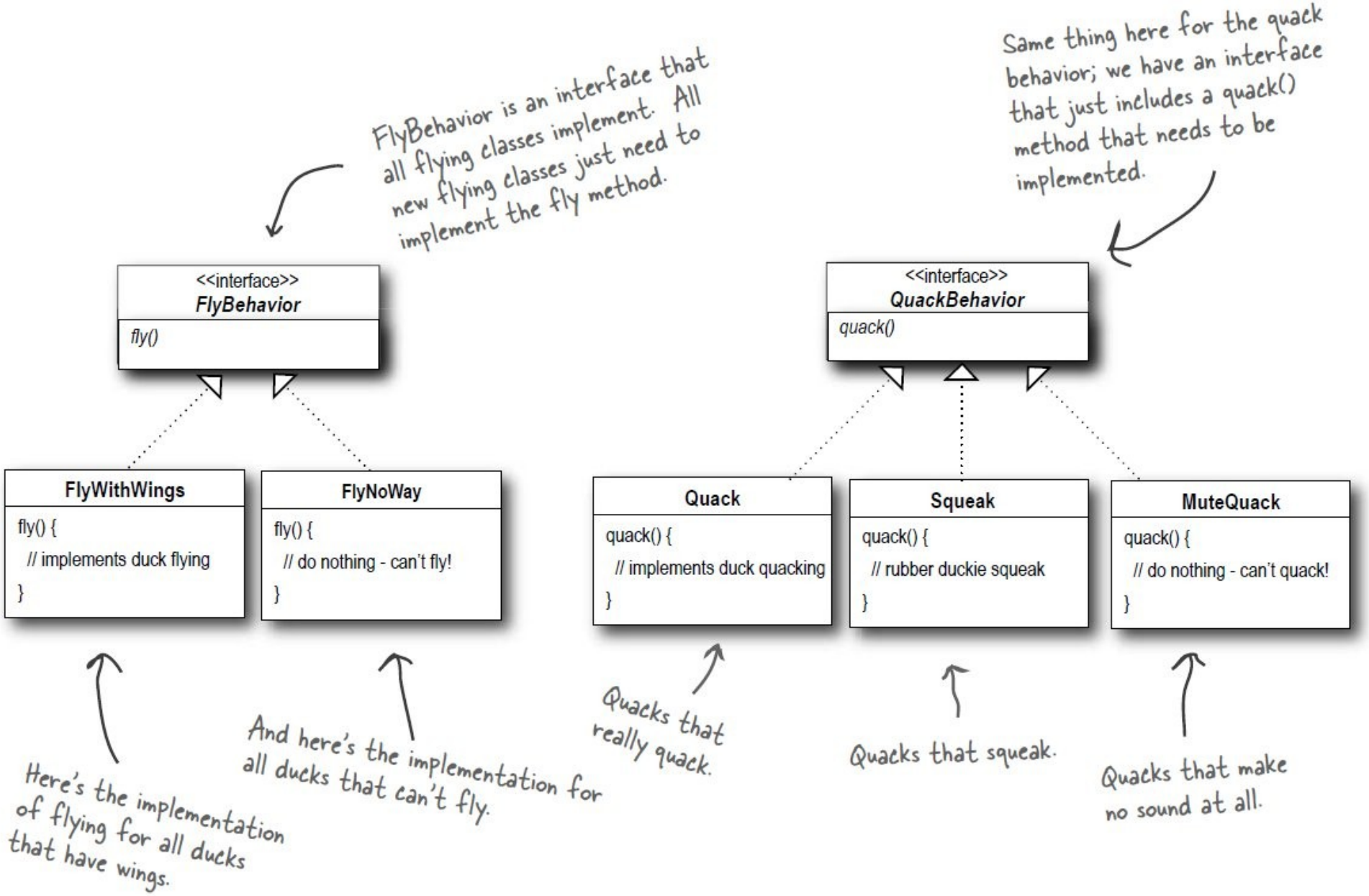
```
Animal animal = new Dog();
animal.makeSound();
```

- Noch besser → Implementierung zur Laufzeit zuweisen:

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

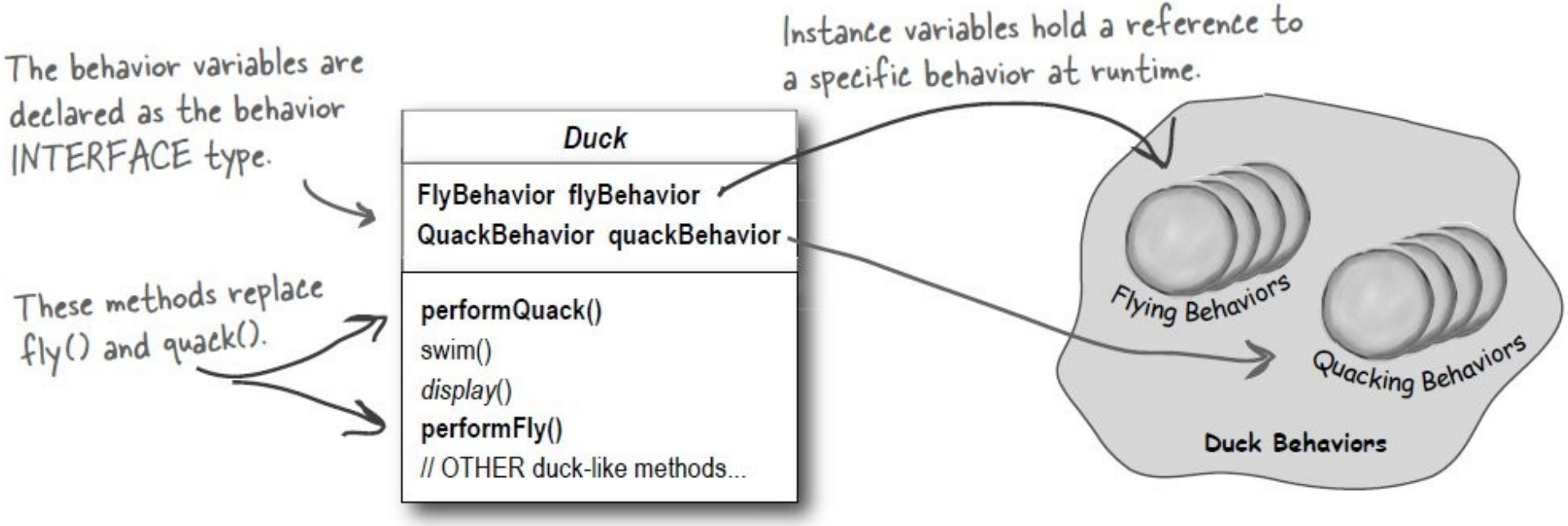


Flug- und Quakverhalten



Integration

Zwei Instanzvariablen zur Klasse Ente hinzufügen:



Integration

- `performQuack()` und `performFly()` implementieren, z.B. `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.

Integration

- Setzen der Instanzvariablen in den Unterklassen:

```
public class MallardDuck extends Duck {

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

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

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 quackBehavior and flyBehavior instance variables from class Duck.

Entenverhalten testen

- Testklasse:

```
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.

Run the code!

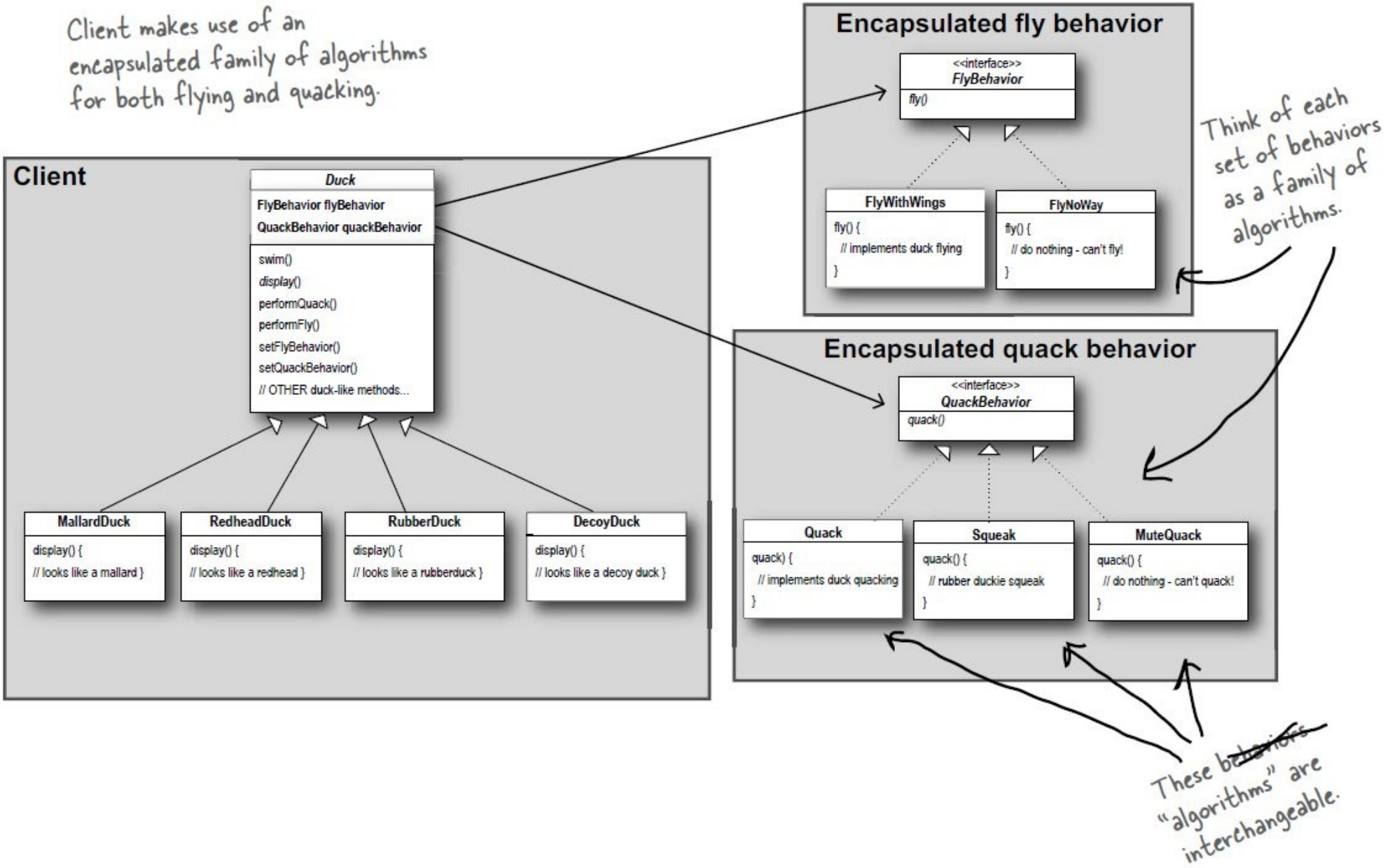
```
File Edit Window Help Yadayadayada
%java MiniDuckSimulator
Quack
I'm flying!!
```

Verhalten dynamisch setzen

- Setter-Methoden in der Entenklasse für Quak- und Flugverhalten schreiben
- Dynamisch zuweisen:

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

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

Das große Ganze

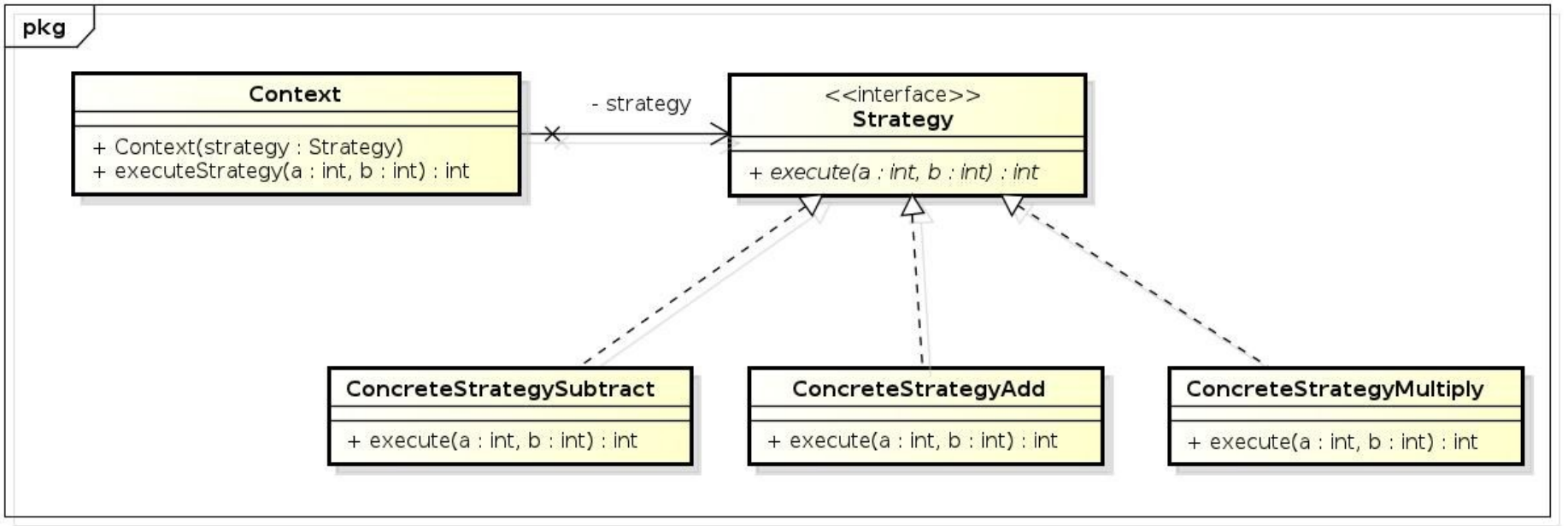


„Hat-Ein“ kann „Ist-Ein“ überlegen sein

- Design Prinzip:

Das erste Entwurfsmuster

- Das **Strategy-Pattern** definiert eine Familie von Algorithmen, kapselt sie einzeln und macht sie austauschbar.
- Algorithmus unabhängig von Clients variierbar



to be continued ...