Head First Design Patterns

# FALLBEISPIEL: SimUDuck

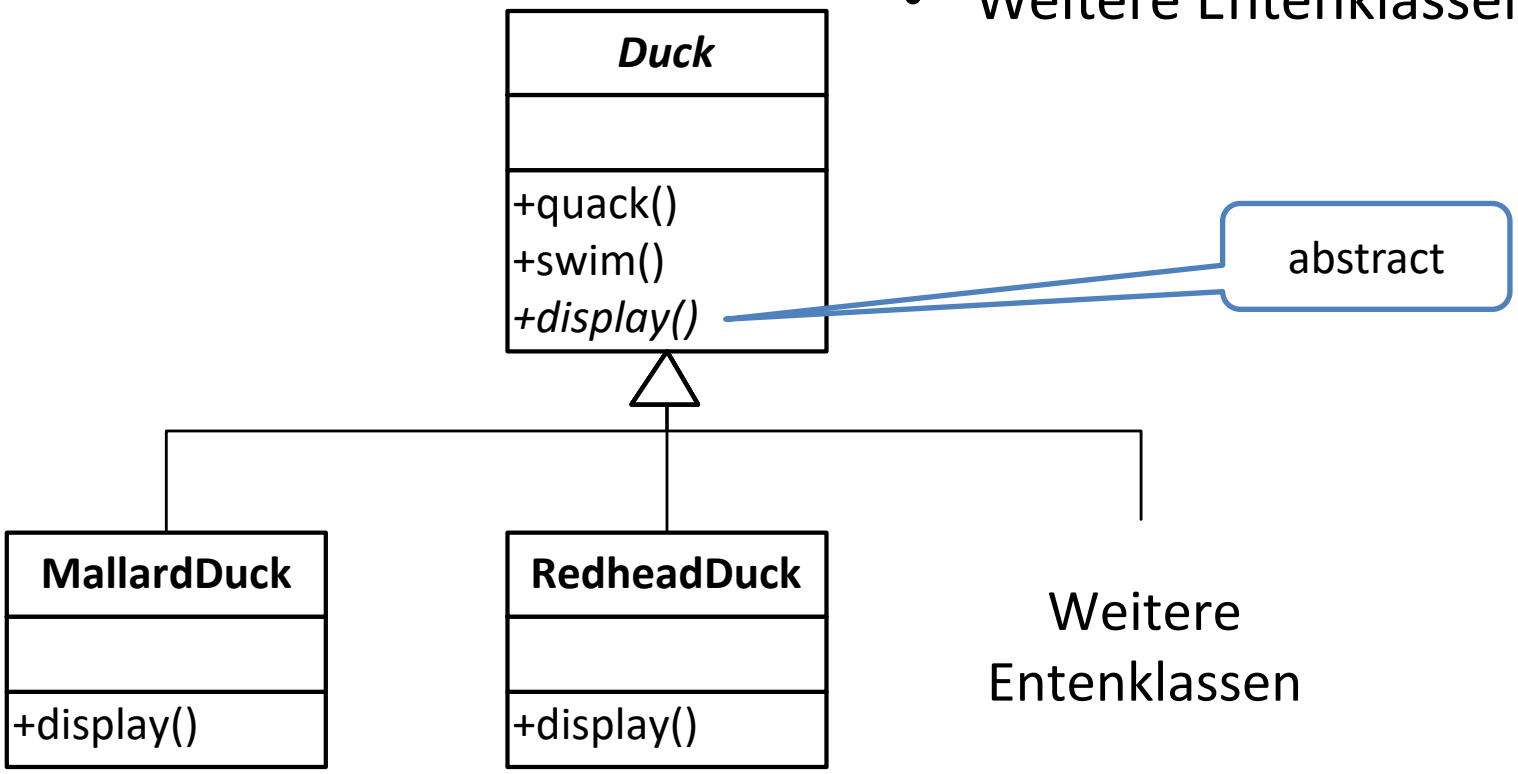
# SimUDuck

- Fallbeispiel aus *Head First Design Patterns [1]*
- SimUDuck:
  - Simulationsspiel wo verschiedenen Entenarten (Stockente, Rotschopfente, Schnatterente …) in einem Teich herumschwimmen und Quak-Geräusche von sich geben.
  - Spiel soll erweitert werden

[1] Eric Freeman et al.: Head First Design Patterns, O'Reilly, 2004

# Momentanes Design

- klassische OO-Technik
- Eine abstrakte Enten-Basisklasse
- Weitere Entenklassen leiten ab

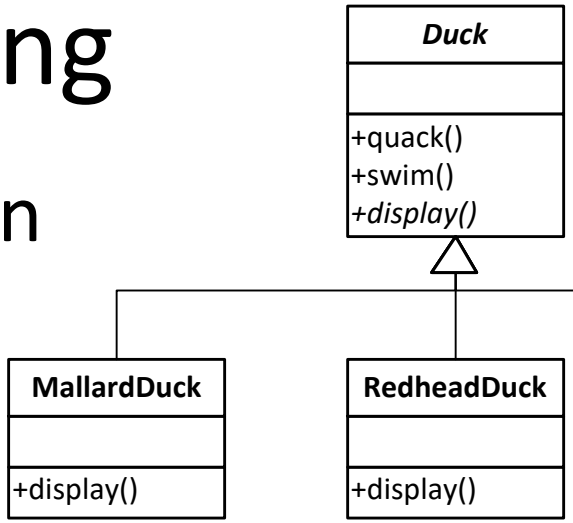


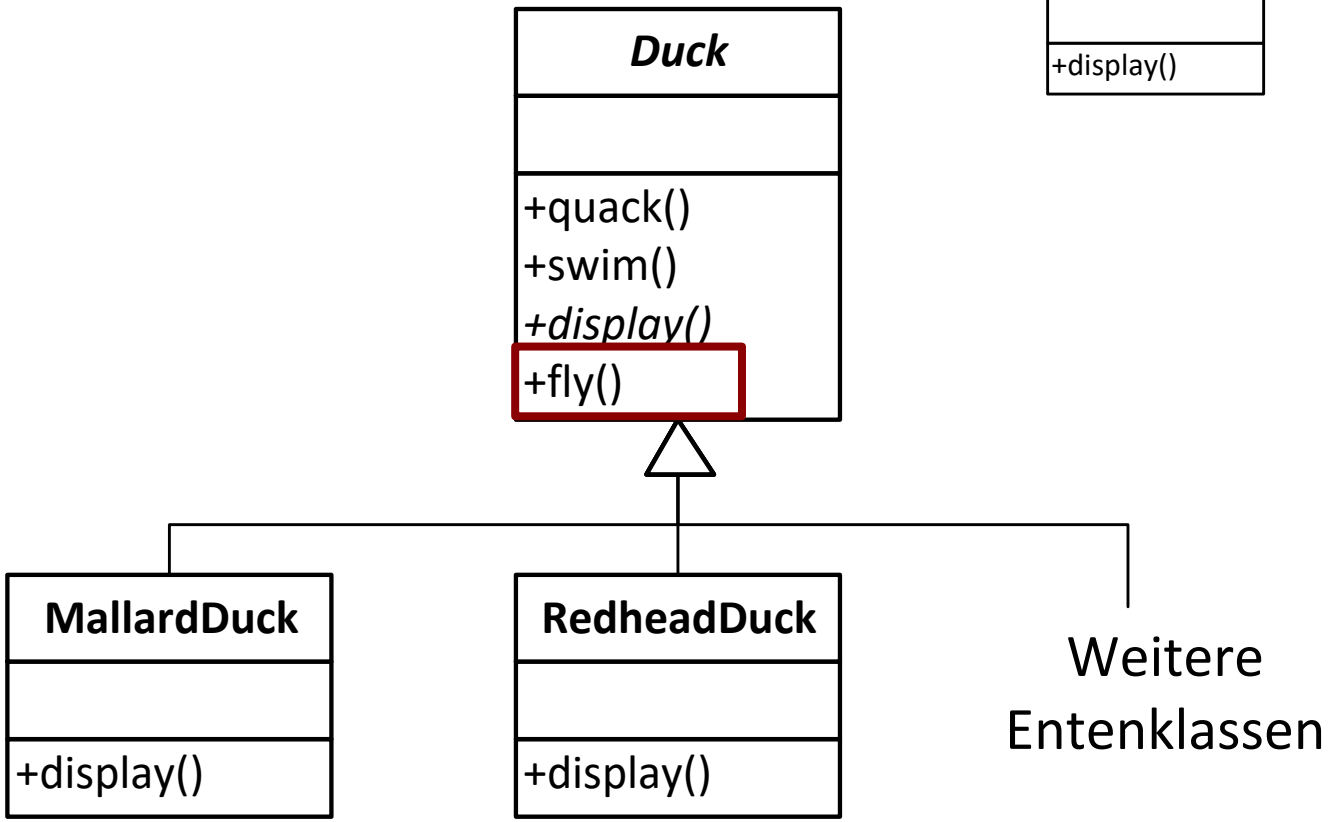
# Java code

- SimUDuck_initial

# Neue Anforderung

- Enten sollen auch fliegen können
- Lösungsvorschläge?



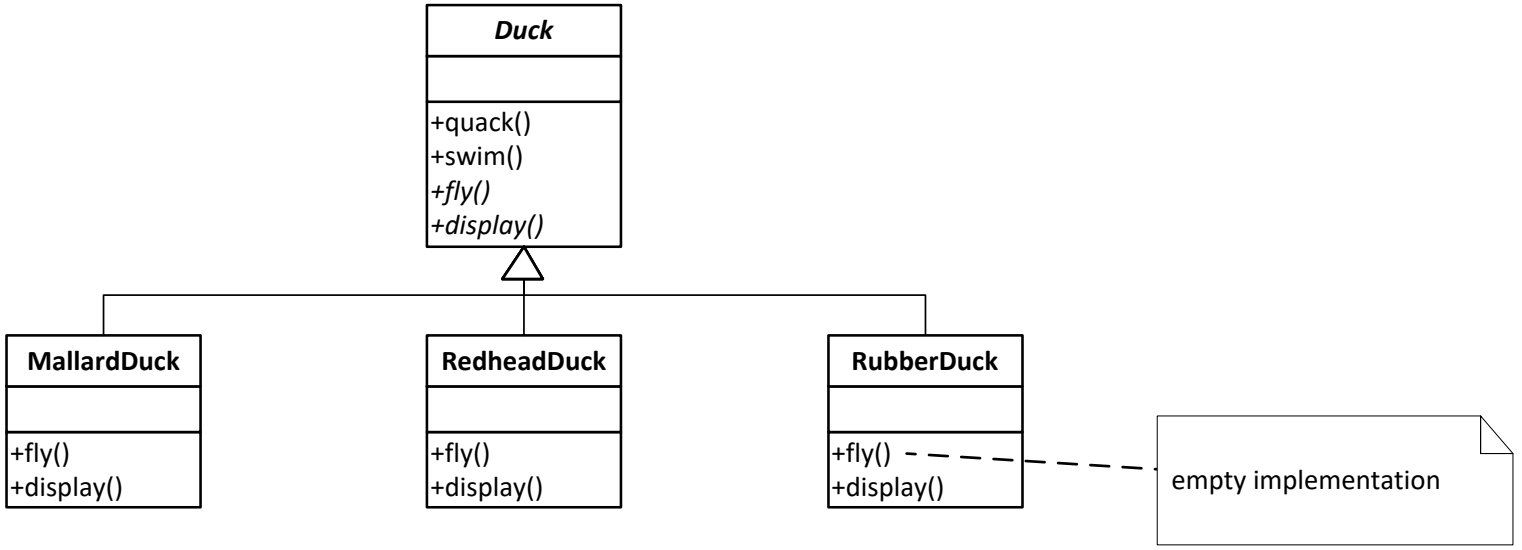
# Problem

- Nicht alle Enten können fliegen!

- Durch die Anpassung in der Superklasse wurde Verhalten hinzugefügt, welches nicht für alle Subklassen sinnvoll ist.

# Lösungsvorschläge? (1)

- Wir könnten unsere Methode fly() ebenso wie die Methode display() auf abstrakt setzen.
- Konsequenzen:
  - Alle Klassen, die von Ente ableiten müssen eine eigene Implementierung von fly() bereitstellen.
  - Enten die nicht fliegen können, bieten in diesem Fall einfach ein leere Implementierung von fliegen() an....



# Lösungsvorschläge? (2)

- Wir könnten unsere Methode fly() in RubberDuck überschreiben: Leere Implementierung.

- Weiteres Problem: Nicht alle Enten können quaken


# Konsequenz

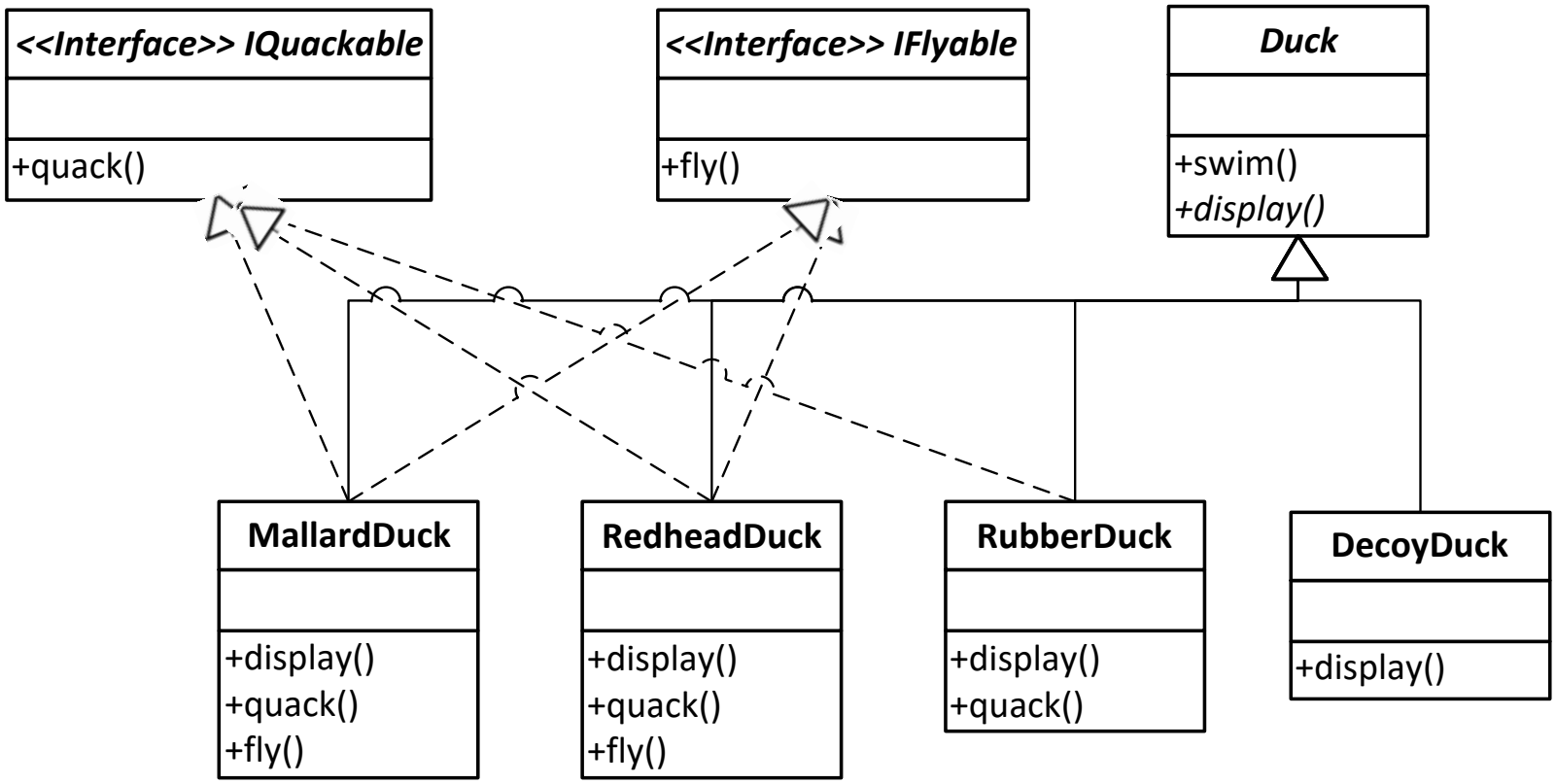
- Viele überschreibende Methoden mit leerer Implementierung.
  - → verdächtig!  Falsches Design!

- Ebenso verdächtig:
  - Viele Subklassen wo Methoden jeweils die gleiche Implementierung haben.
    - zB. 3 Entenarten quaken, 2 quietschen, 2 sind stumm.

- Weiters: Vielleicht möchte man wissen, ob eine Ente fliegen kann oder nicht (zB. Zusätzliches Symbol, Kontextmenu, …).  Man bräuchte zusätzlich ein canFly()  das false zurück gibt, wenn fly() leer ist.

# Neuer Vorschlag: Interfaces

- fliegen() und quaken() herausziehen und daraus zwei Interfaces machen



# Java code

- SimUDuck_if

# Konsequenzen

- Code Duplikation
  - Jede Entenklasse muss ihr Flug- und Quackverhalten separat implementieren.
  - Sehr viel arbeit wenn zB. am Flugverhalten etwas geändert werden muss
  - Fehleranfällig

- Vererbung ist also irgendwie nicht die richtige Antwort auf unser Problem….

# The one constant in software development

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

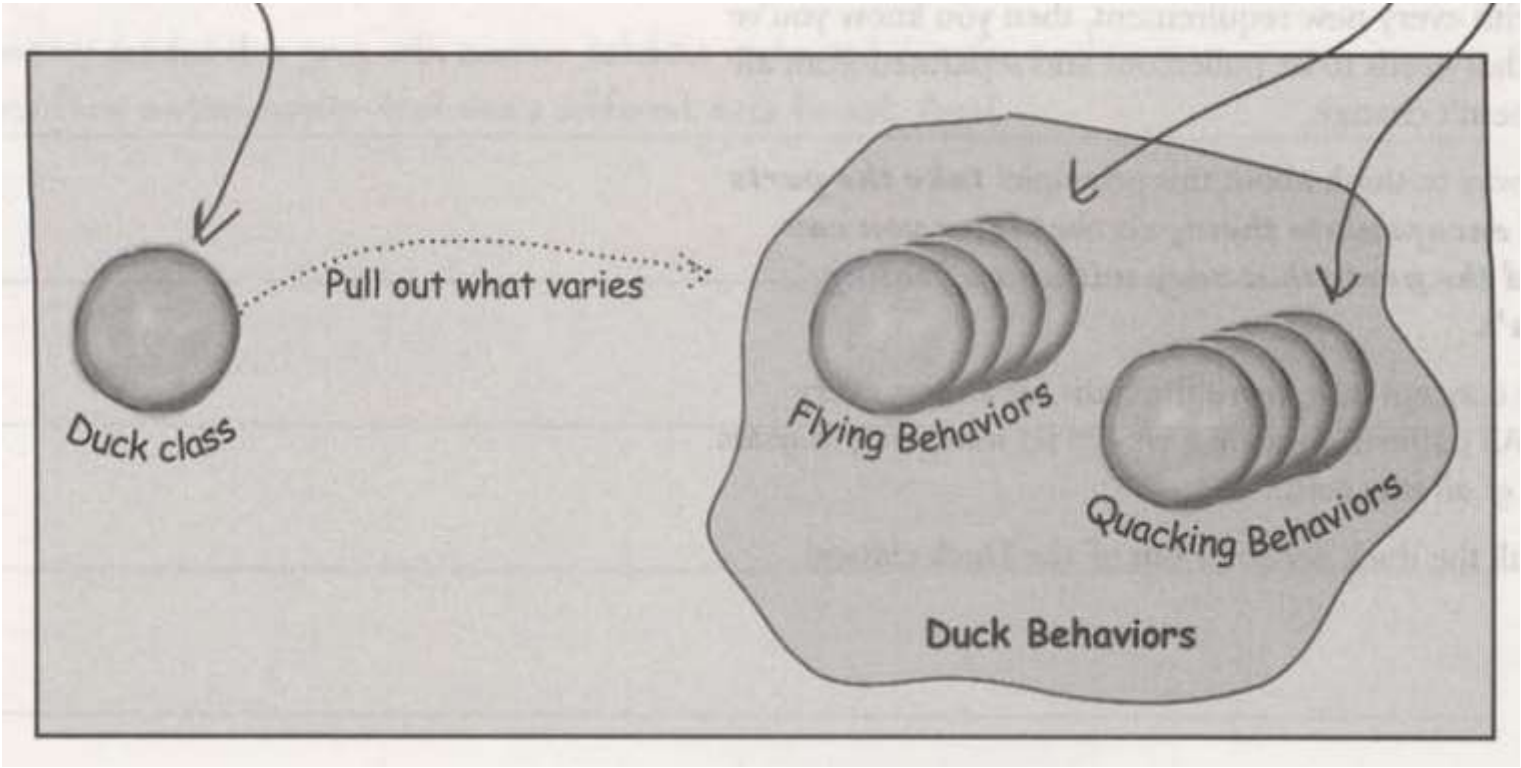
Designprinzip:

**Identifiziere jene Aspekte, die sich häufig ändern und trenne sie von jenen, die gleich bleiben.**

# Was ändert sich bei uns?

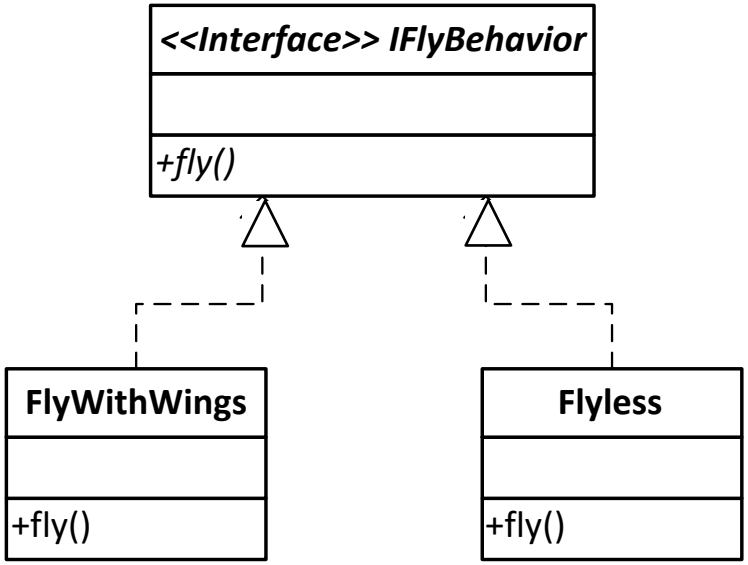
**Enten Grundverhalten**

**Flugverhalten Quakverhalten**

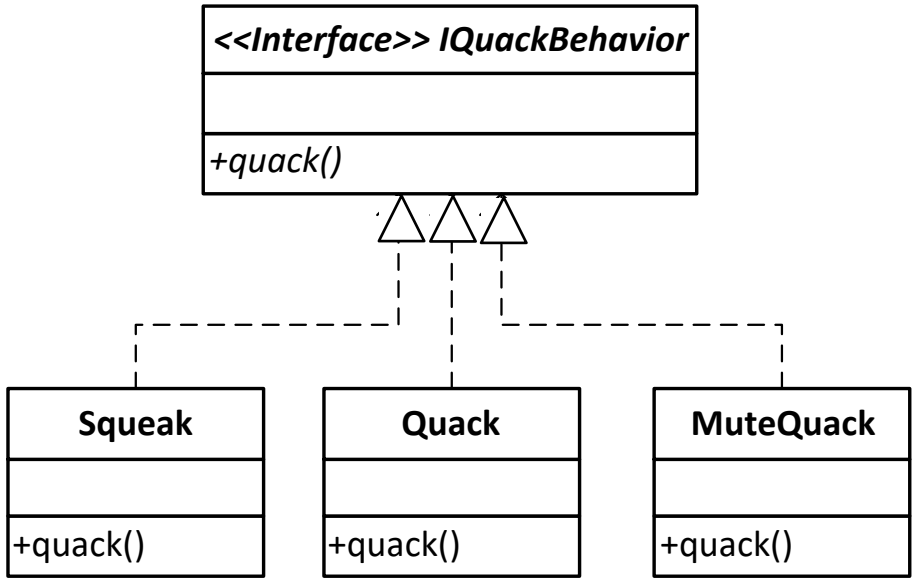


# Mögliche Implementierung

**Flugverhalten**




**Quakverhalten**



# Noch 2 verwendete Designprinzipien

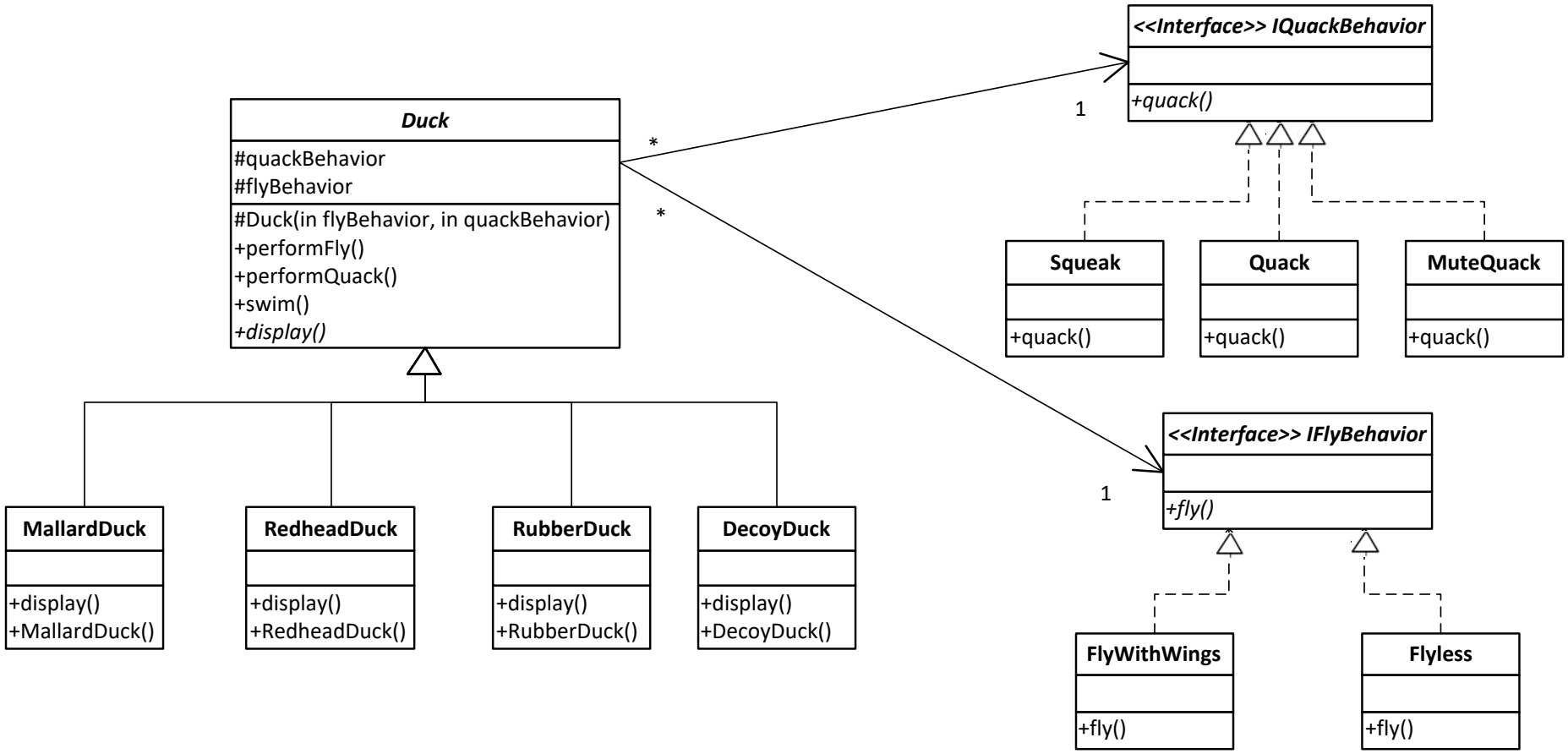
Designprinzip:

**Programmiere gegen ein Interface**
**anstatt gegen eine konkrete Implementierung.**

Designprinzip:

**Bevorzuge Komposition (has-a) gegenüber**
**Vererbung (is-a).**

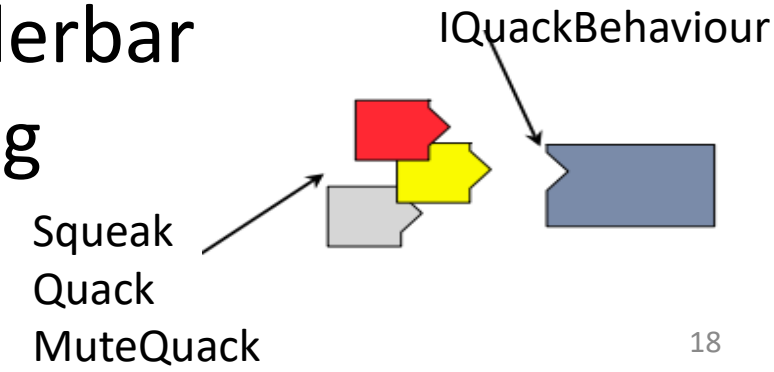
# SimUDuck Architektur



# Weitere Vorteile

- Neue Verhalten hinzufügbar, ohne bestehende Klassen ändern zu müssen
- Code reuse
  - Die Behaviour-Klassen könnten von anderen Klassen (nicht Enten) wiederverwendet werden.

- Verhalten zur Laufzeit änderbar durch dynamische Bindung

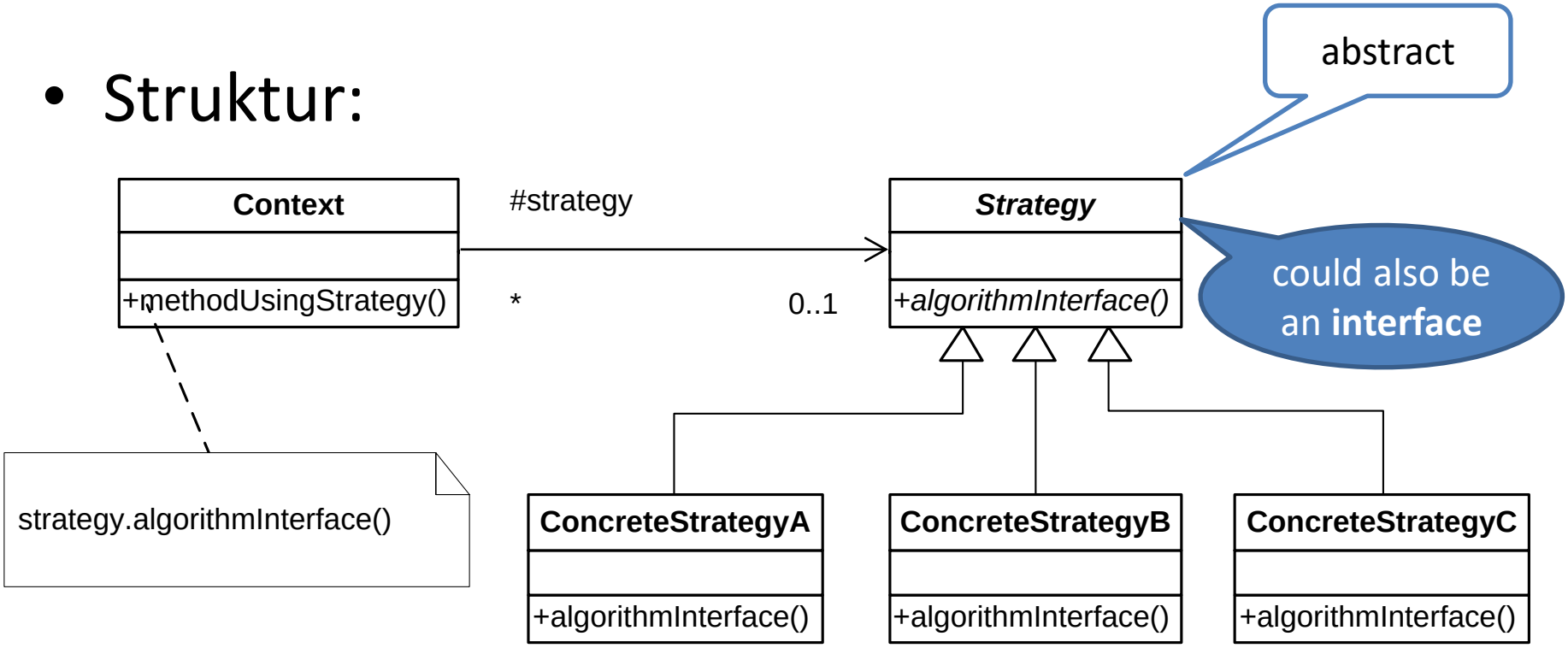


# Lessons Learned

- Drei Designprinzipien
  - Auslagern, was sich ändert.
  - Programmiere gegen ein Interface (Supertyp) anstatt gegen eine konkrete Implementierung.
  - Bevorzuge HAS-A (Komposition) gegenüber IS-A (Vererbung).
- Ein Design Pattern:
  - Strategy

# Design Pattern: Strategy

- Struktur:




***„Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.“***
*[GOF]*

# Design Pattern: Strategy

- Java Code:  see StrategyPattern

- Important
  - Do not stick to the names in the UML diagram.
    - Change class and method names according to your application.

# Design Pattern: Strategy

- Beteiligte Klassen:
  - Strategy
    - Definiert ein gemeinsames Interface für alle unterstützten Algorithmen.
  - ConcreteStrategy:
    - Konkrete Implementierung eines Algorithmus.
  - Context (Client):
    - Verwaltet eine Referenz mit statischem Typ *Strategy*

# Design Pattern: Strategy

- Anwendbarkeit
  - Viele Klassen unterscheiden sich nur in bestimmten Verhaltensweisen
  - Unterschiedliche Varianten eines bestimmten Algorithmus (z.B. Zeit- vs. Speicherverbrauch)
  - Vermeidet komplexe Datenstrukturen, die für einen Algorithmus benötigt werden, offenlegen zu müssen.

# Design Pattern: Strategy

- Konsequenzen (Vor-/Nachteile)
  - Vermeidet Codeduplikation
  - Switch Statements oder if-else-if Ketten werden vermieden
  - das Verhalten kann zur Laufzeit geändert werden
  - die Implementierung wird vor Clients versteckt

  - Unterschiedliche Strategien brauchen uU. Unterschiedliche Parameter
  - Erhöhte Anzahl von Objekten (für Verbesserung sh. Flyweight Pattern)