

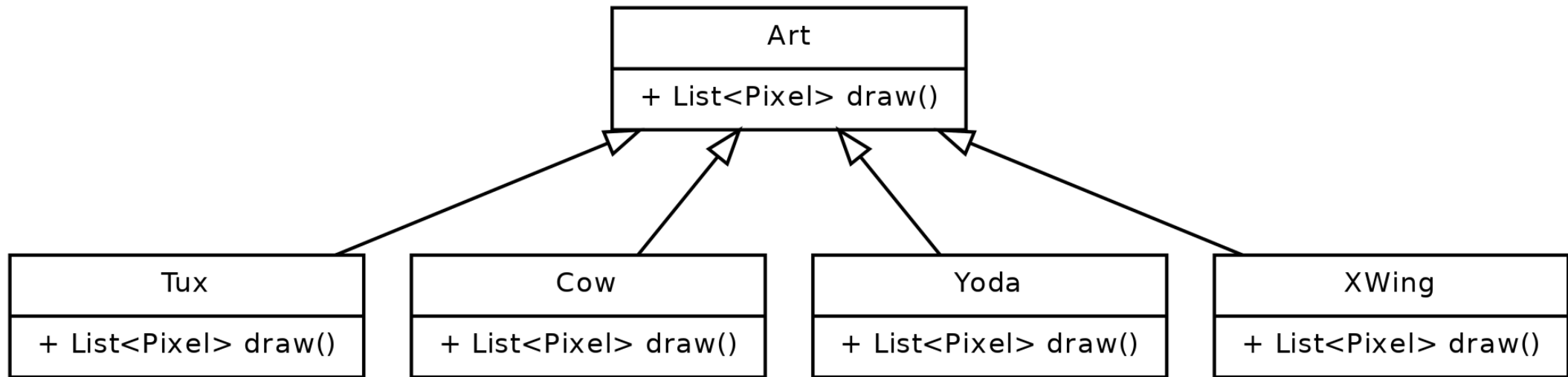
Eine ASCIIShop- Story

EINE NEUE HOFFUNG DURCH
ENTWURFSMUSTER


ASCIIShop

- ASCII-Art Bearbeitungssoftware
- Objekt-orientierte Java-basiert Architektur

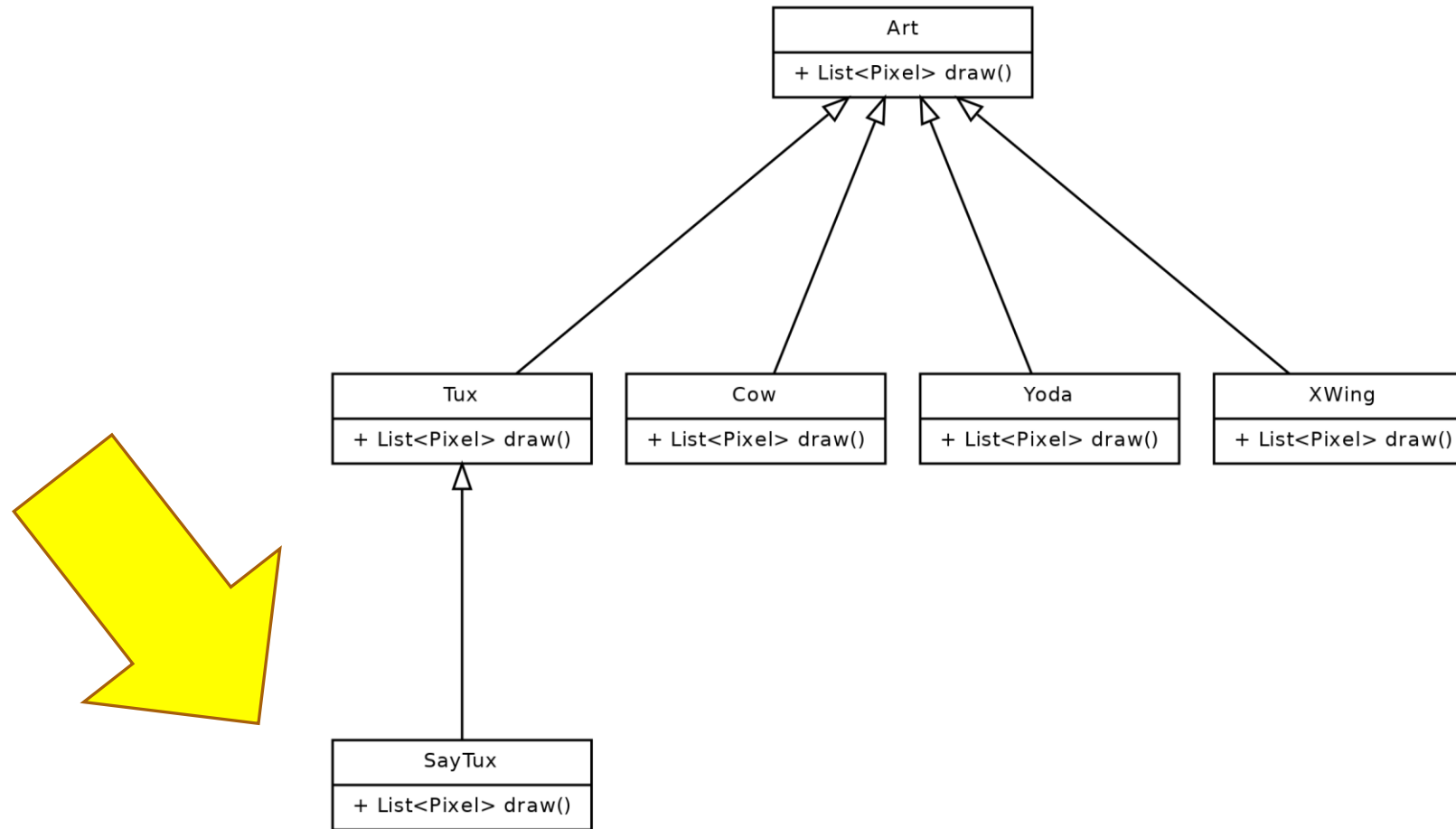
Klassen-Diagramm ASCII-Shop



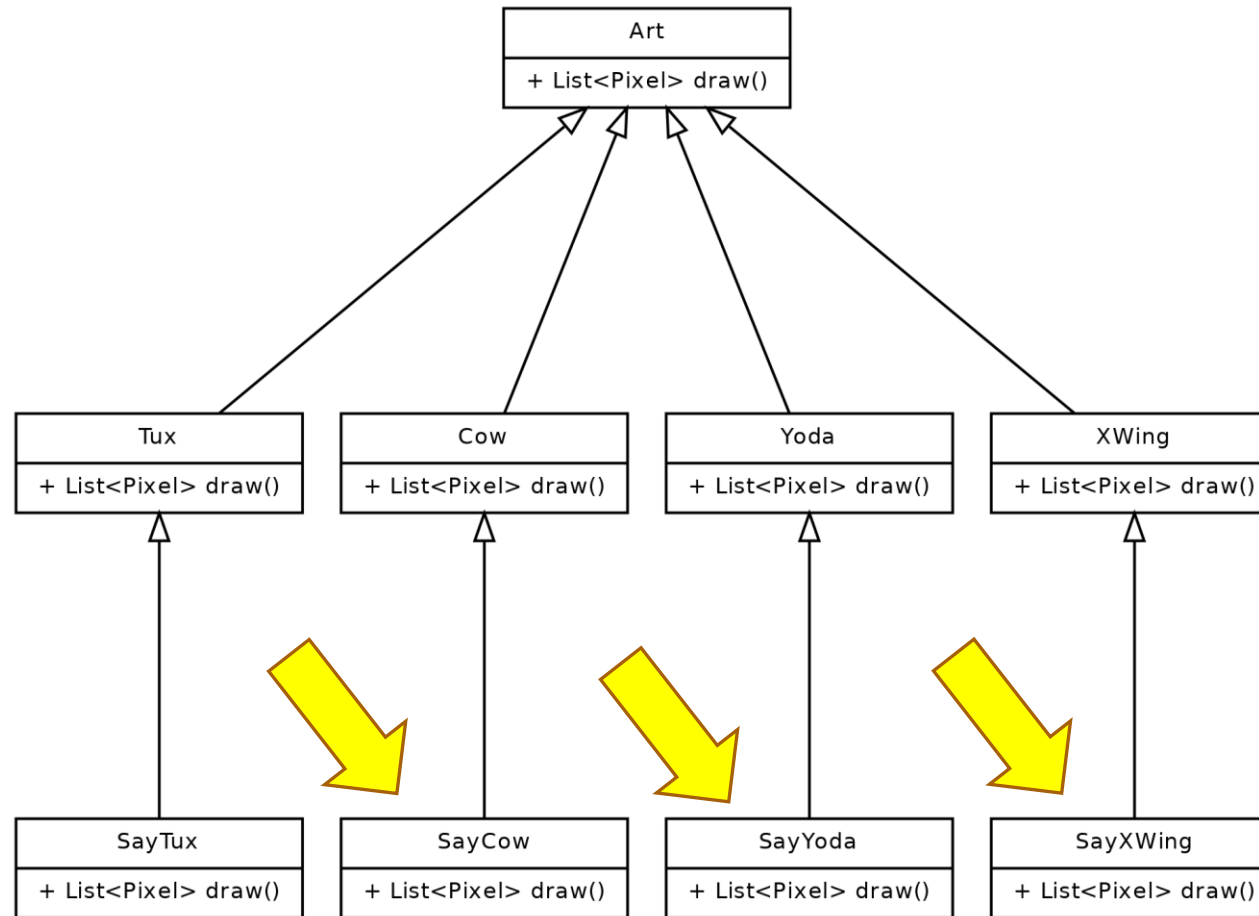
ASCIIShop

- ASCII-Art Bearbeitungssoftware
- Objekt-orientierte Java-basiert Architektur
- Erfolgreich 
- Die Kunden wollen mehr Variationen
 - Mehr Klassen & Vererbung

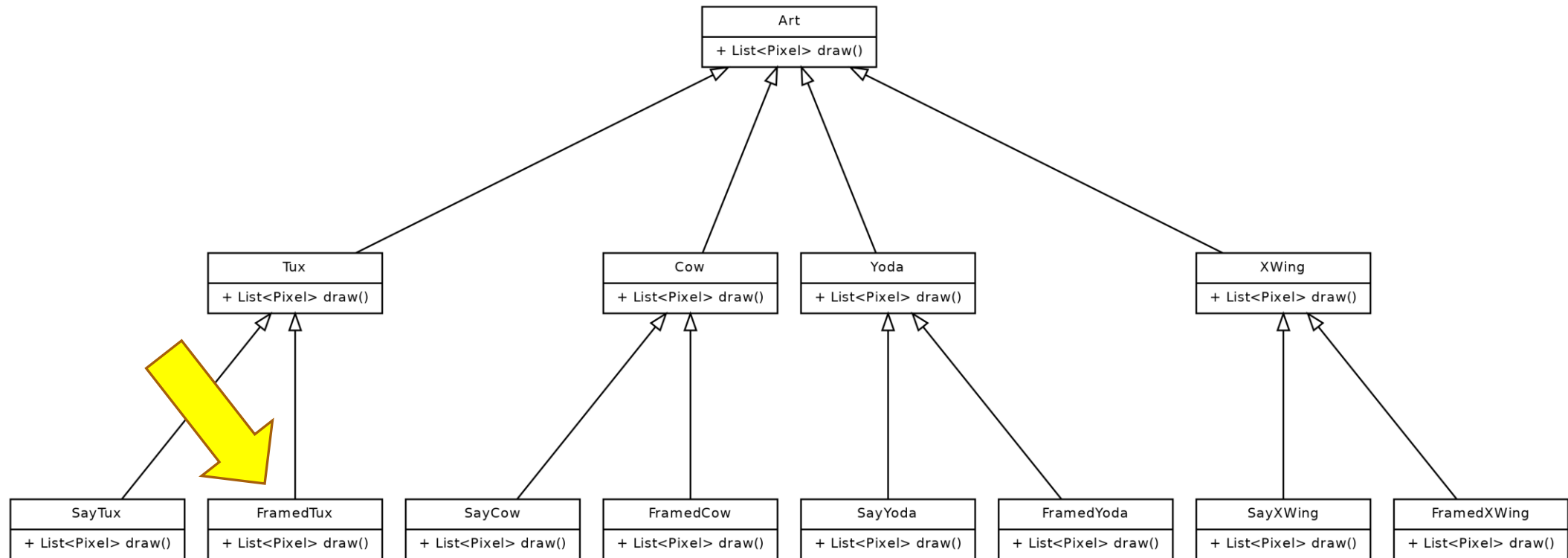
Mehr Klassen



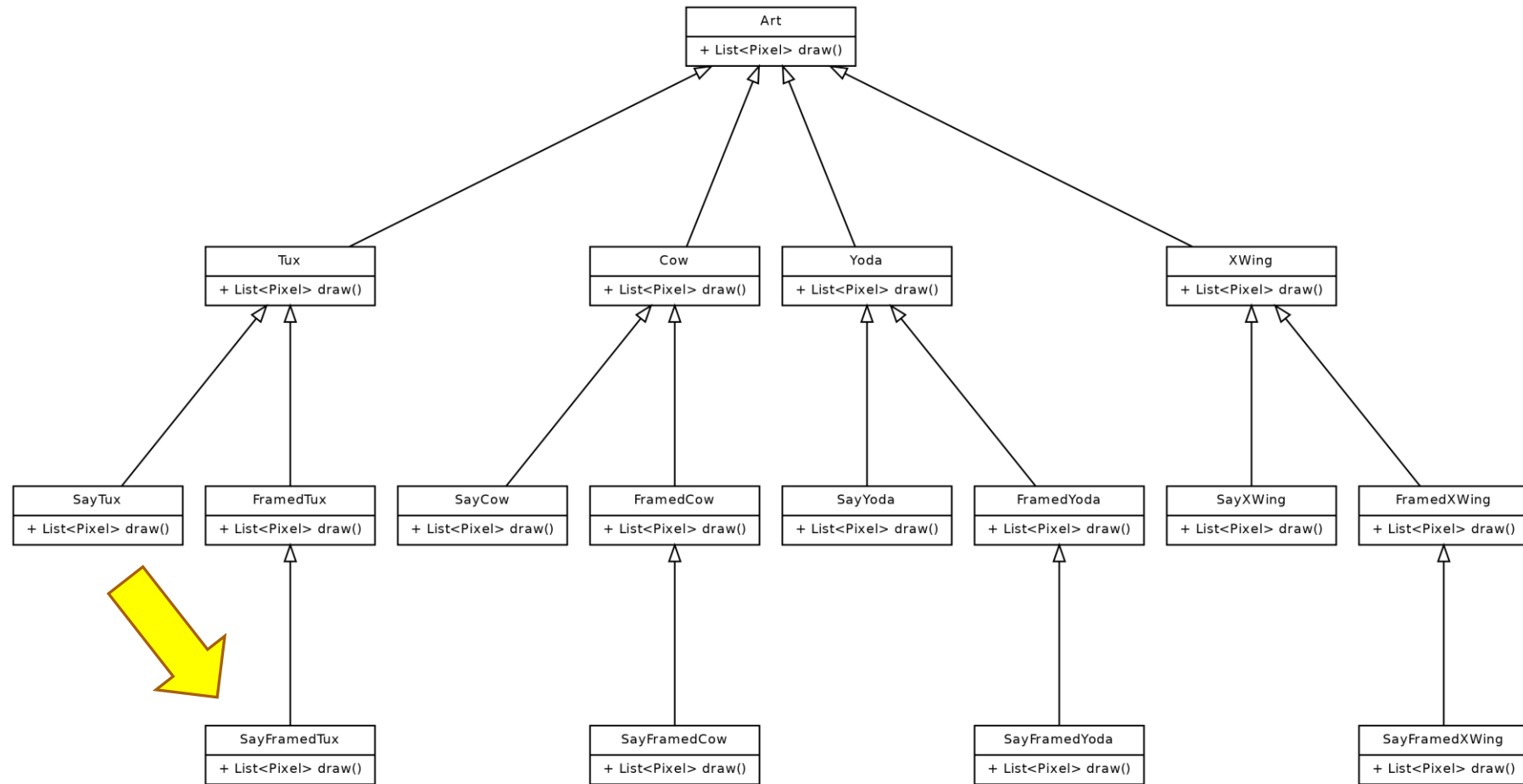
Gleiches Recht für alle



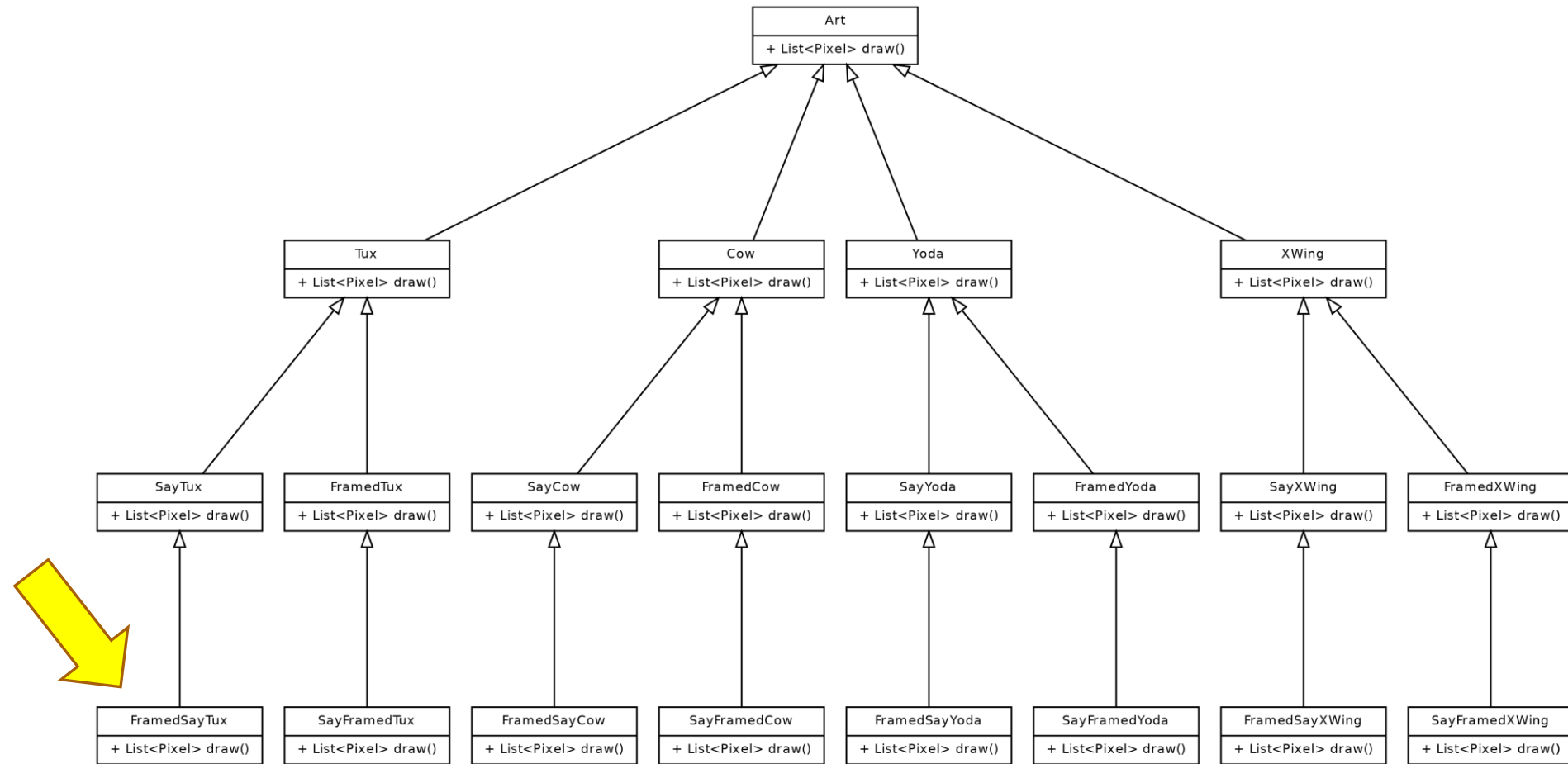
Framing them



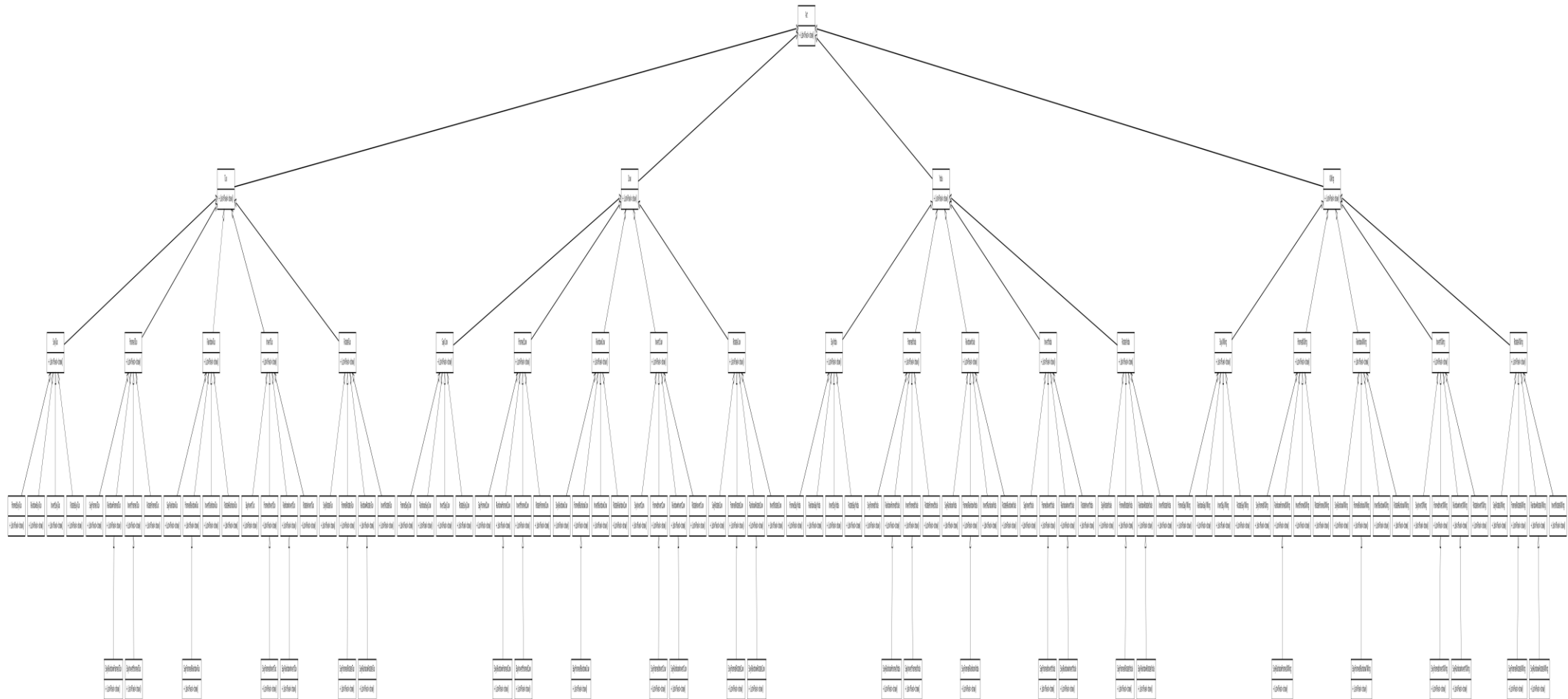
Say who framed you



Andersrum



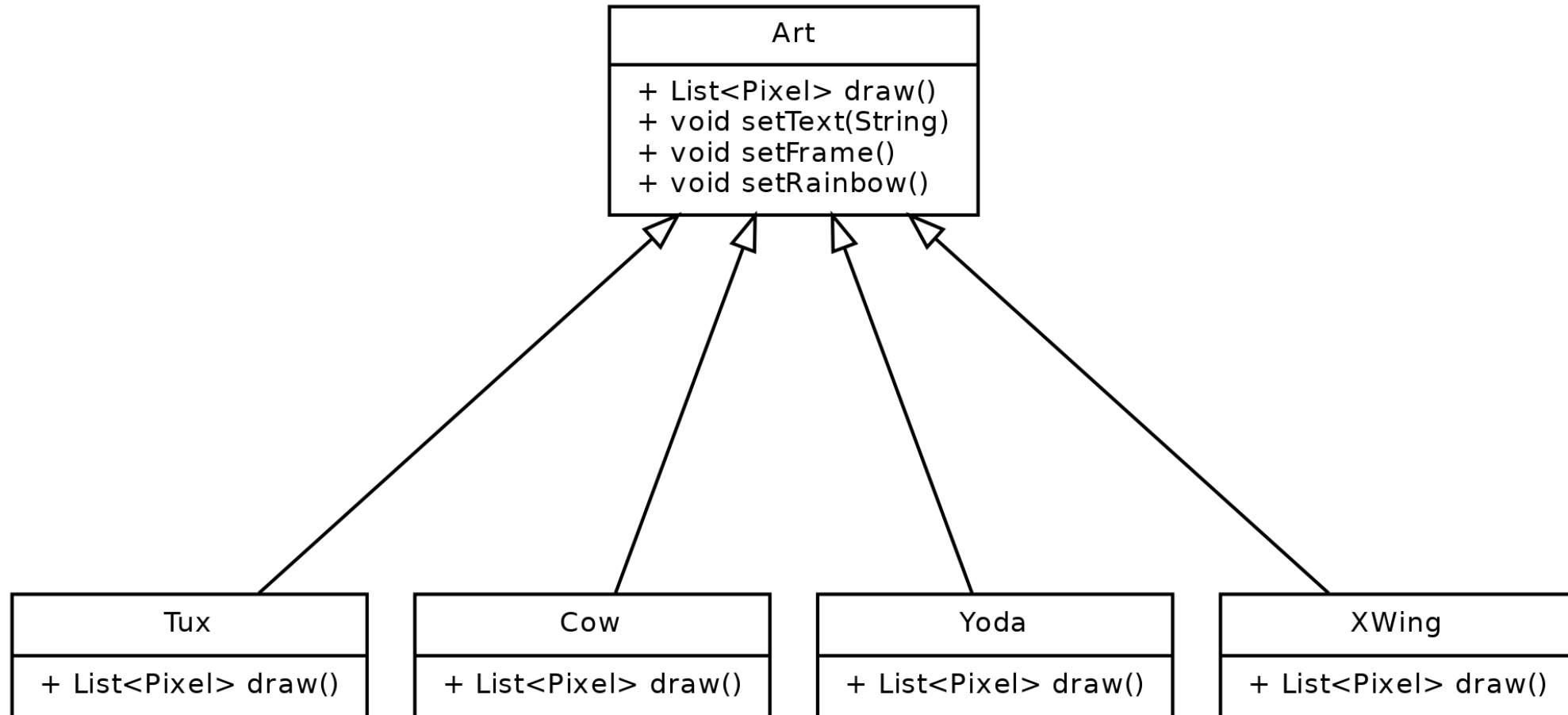
20000 Klassen unter dem Meer



Okay wir haben ein Problem!

- Viele sehr ähnliche Klassen
 - Redundanter Code
 - Erweiterung nur zu Compilezeit möglich
- Einfach alles in einer Klasse?

My big fat artsy Class



Art is everything

- Open-Closed Prinzip missachtet
 - Open for extension
 - Closed for modification
- Eine große Klasse
 - Schwer zu warten
 - Unabhängige Modifikation schwer

Decorator

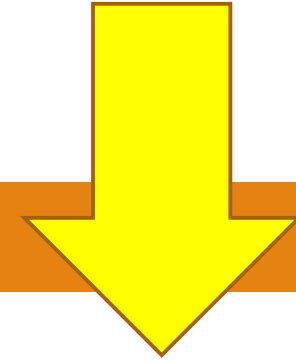
ENTWURFSMUSTER

Agenda

1. Was ist ein Decorator?
2. Wieso und wann Decorators verwenden?
3. Alternativen
4. ASCIIShop

Einleitung

Entwurfsmuster



Erzeugungsmuster (Creational Patterns)

- Objektinstanziierung

Verhaltensmuster (Behavioral Patterns)

- Algorithmen
- Kommunikation

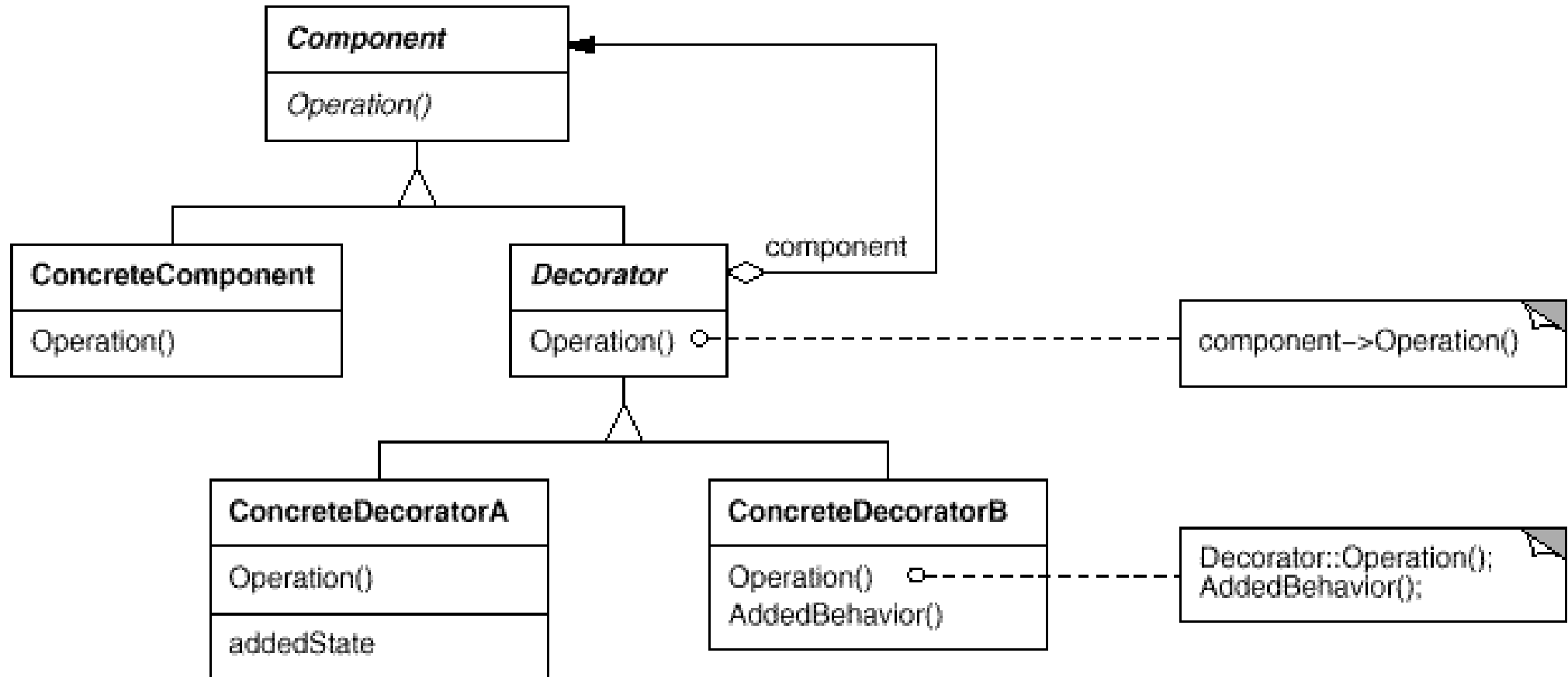
Strukturmuster (Structural Patterns)

- Komposition

Was ist ein Decorator?

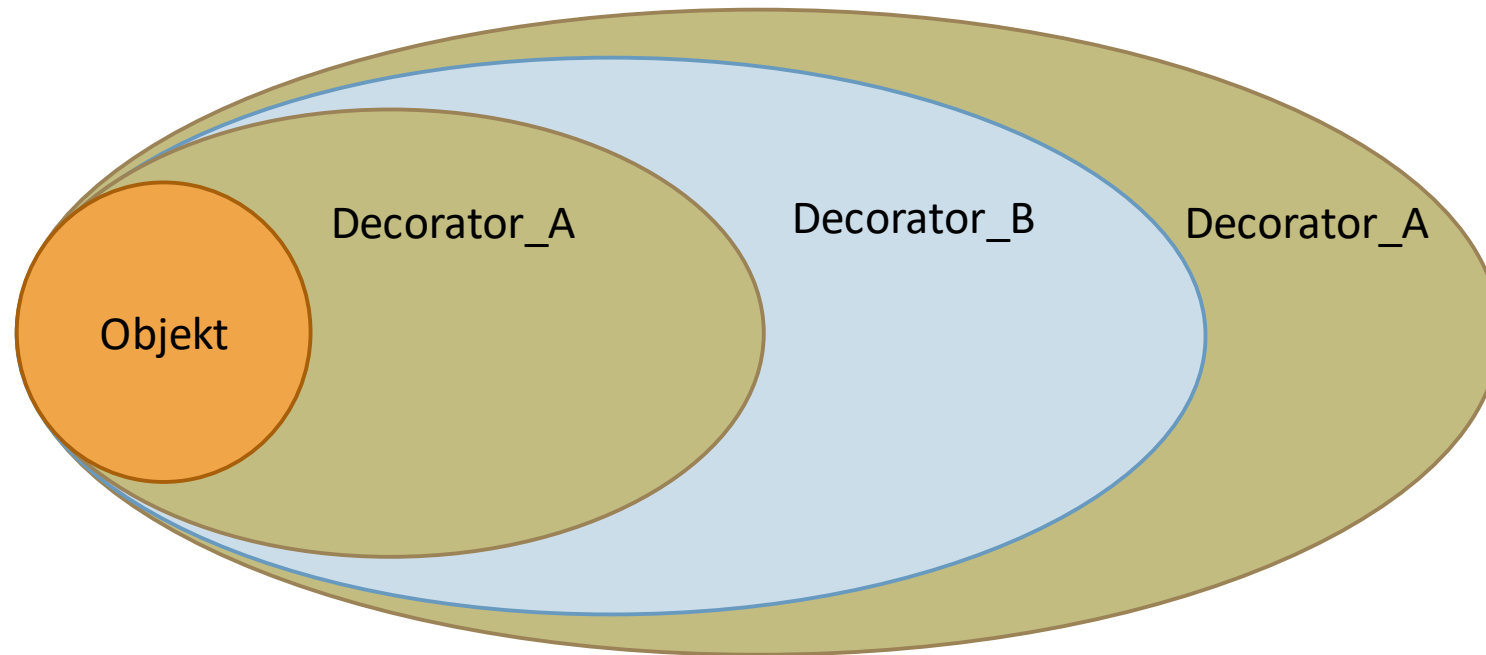
„Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.“ [GoF Buch]

Was ist ein Decorator?



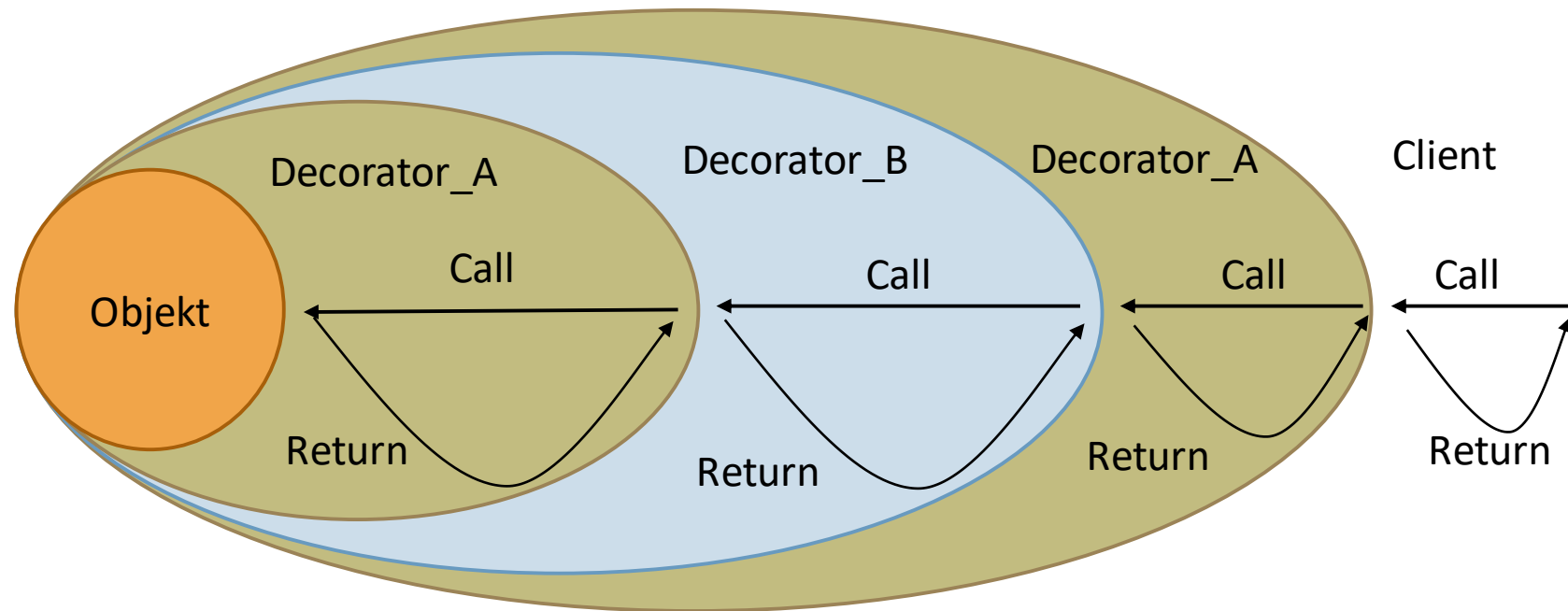
Was ist ein Decorator?

Dekorieren = Wrappen



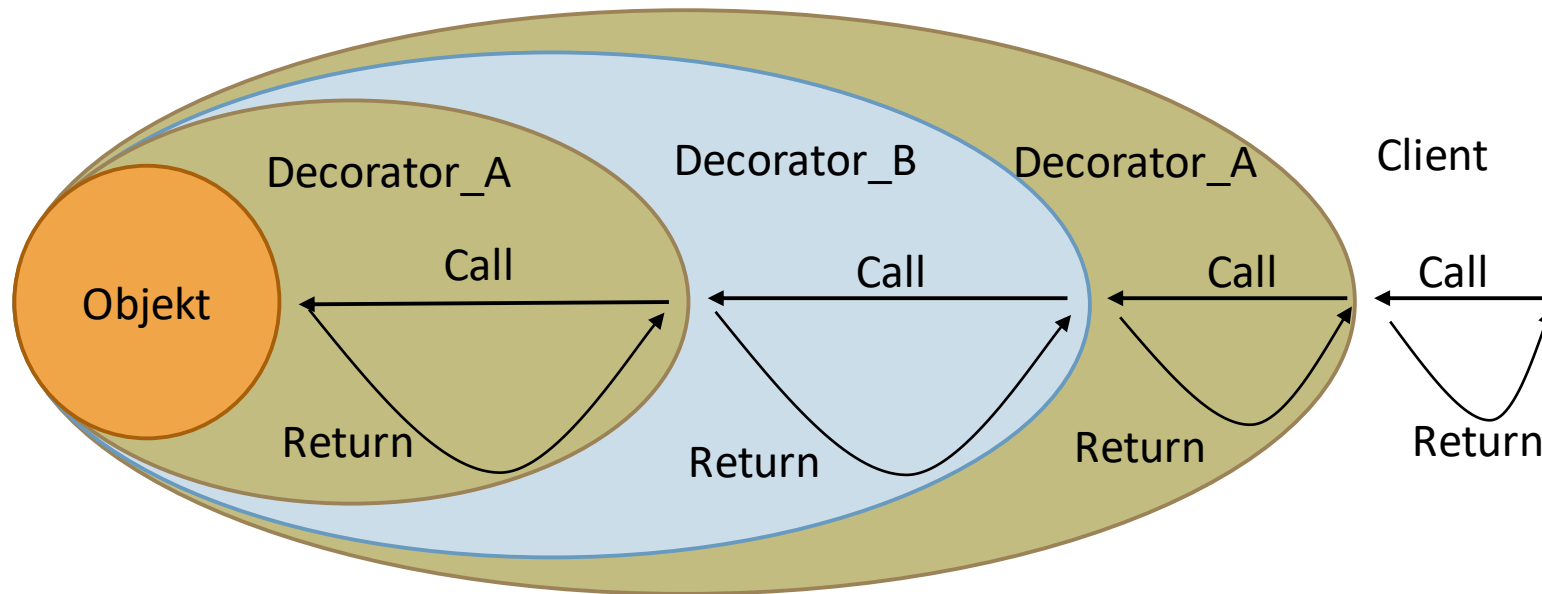
Was ist ein Decorator?

Dekorieren = Wrappen



Was ist ein Decorator?

Dekorieren = Wrappen




Zusätzliches Verhalten

- Vor dem Call
- Nach dem Call

→ Transparenz durch Interface-Konformität

Was ist ein Decorator?

```
public interface Component {  
    void operation();  
}
```



```
public class ConcreteComponent implements Component{  
    public void operation(){  
        System.out.println("Operation");  
    }  
}
```

```
public class Decorator implements Component {  
    private Component component;  
    Decorator(Component component){  
        this.component = component;  
    }  
    public void operation(){  
        System.out.print("Decorated ");  
        this.component.operation();  
    }  
}
```

Was ist ein Decorator?

```
public static void main(String[] args) {  
    Component component = new ConcreteComponent();  
    component.operation();  
    component = new Decorator(component);  
    component.operation();  
}
```



Operation
Decorated Operation

Wieso und wann Decorators?

- Modellierung von vielen Variationen und Kombinationen
 - z.B. Java FileInputStream (BufferedInputStream, GzipInputStream, ObjectInputStream, ...)
- Mehrfache Dekorierung
- Hinzufügen von Funktionen / Verantwortlichkeiten (im Vergleich zu Vererbung)
 - Zu einem Objekt, nicht zu einer Klasse
 - Zur Laufzeit, nicht zur Compilezeit
 - Ohne die Klasse selbst zu verändern (Open-Closed-Principle)

Wieso und wann Decorators?

- Vermeiden von zu viel Vererbung bzw. zu vielen Klassen
- Vermeiden von zu komplexen Klassen → Performance, Lesbarkeit , Wartbarkeit
- Kohäsion → Single Responsibility Principle

Nachteile

- Instanziierung etwas komplizierter

```
Component component = new ConcreteComponent();  
component = new Decorator_1(component);  
component = new Decorator_2(component);  
component = new Decorator_3(component);
```

als bei Vererbung

```
Component component = new MySpecialComponent();
```

oder einem Monolithen

```
Component component = new ConcreteComponent(true, true, true);
```

➔ Factory zur Instanziierung häufiger Kombinationen

Nachteile

- Viele kleine Objekte
 - Performance-Einbuße durch object overhead
- Zusätzliche Funktionalität nicht in der IDE sichtbar
- Interfacekonformität
 - Alle Methoden müssen im gemeinsamen Interface vorgegeben sein
 - Keine zusätzliche Funktionalität durch erweitertes Interface möglich

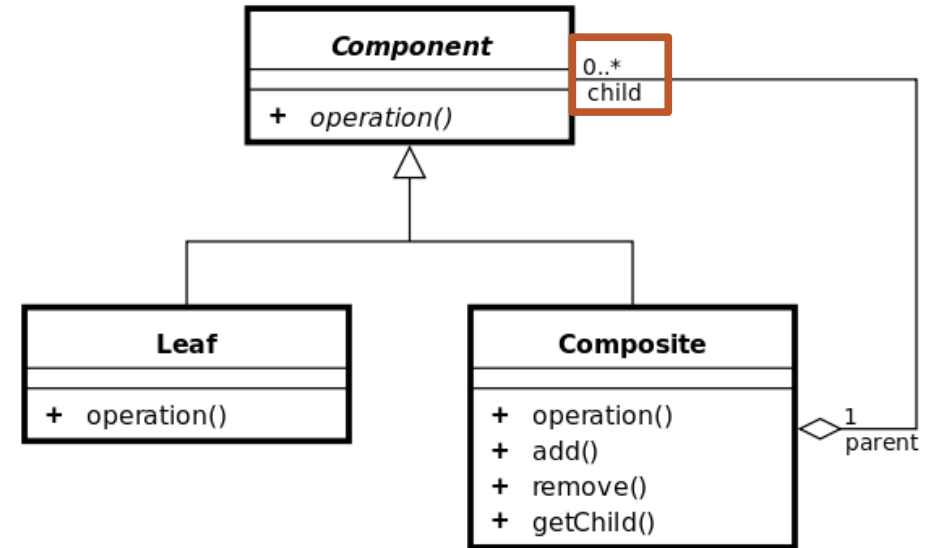
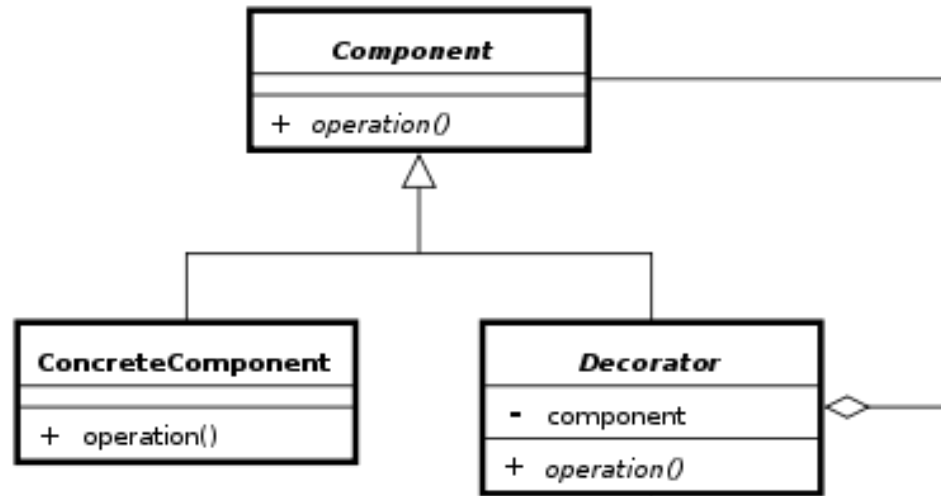
Alternativen - Strategy

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

- Dynamisches Austauschen statt Hinzufügen von Verhalten
„Changing the skin of an object versus changing its guts.” [GoF Buch]
- Delegation von Funktionen an Strategy Objekte
- Ähnlicher Intent
- Andere Herangehensweise
- Bei großen Komponenten weniger aufwendig
- Komponente muss angepasst werden

Alternativen - Composite

“Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly. “ [GoF Buch]



- Sehr ähnliche Struktur
- Andere Intention → Objekt Aggregation

Decorator in anderen Paradigmen

- Für das Decorator Pattern ist keine Vererbung notwendig, Interfaces sind ausreichend
 - Rust, Go
- Wenn Funktionen First-Class Citizen sind, können Decorators durch Funktionskomposition

```
tux :: String  
tux = "Tux"
```

```
rainbow :: String → String  
rainbow = (++) "Rainbow"
```

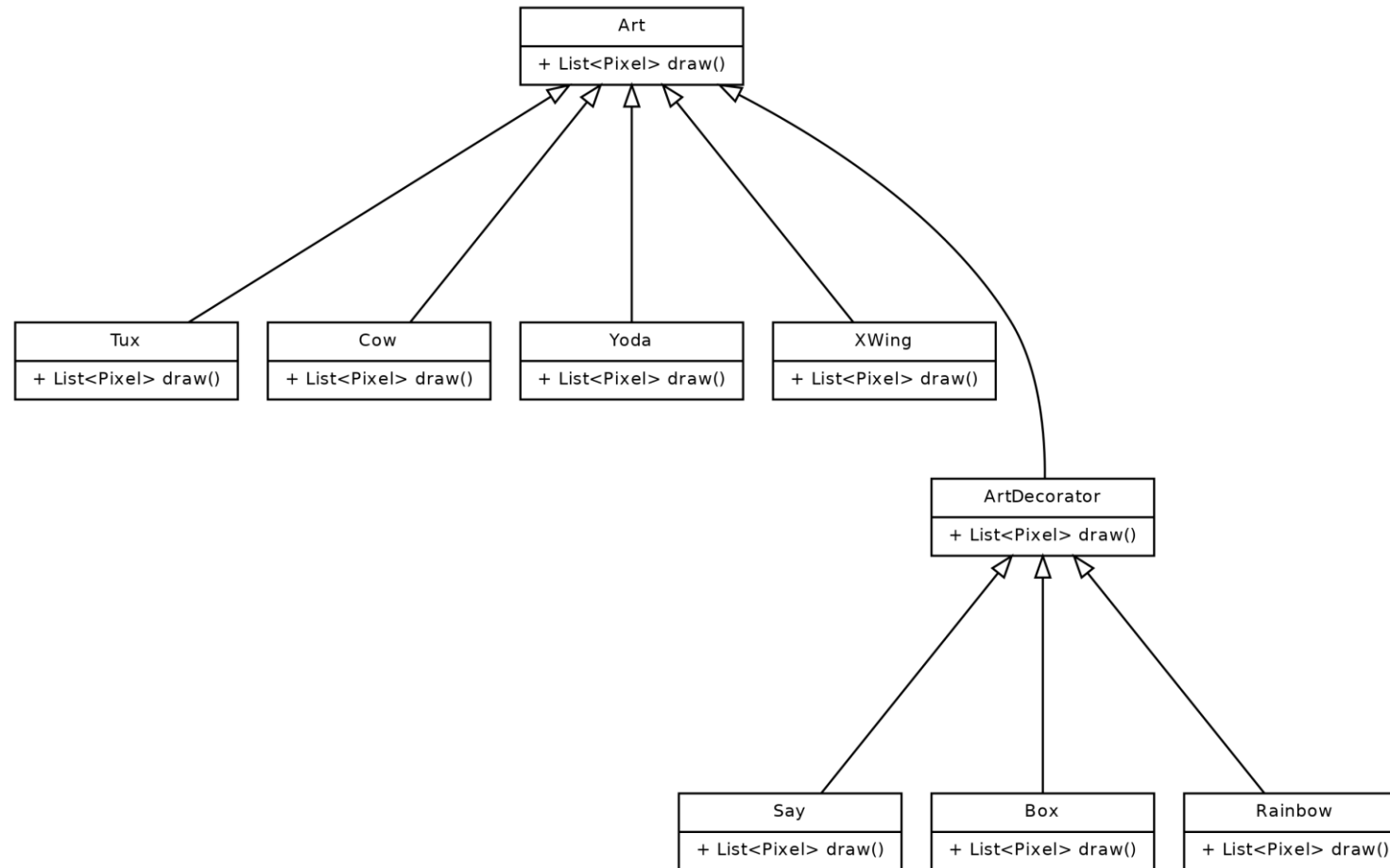
```
frame :: String → String  
frame = (++) "Frame"
```

```
main :: IO ()  
main = print $ (rainbow . frame) tux
```

- Decorator als Workaround für Sprachen ohne Funktionen als First-Class Citizen

Back to ASCIIShop

ASCIIShop



Demo

Quellen

Design Patterns: Elements of reusable object-oriented Software; Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides; 1994

Head First Design Patterns; Eric Freeman, Elisabeth Freeman; 2004

<https://www.philippbauer.de/study/se/design-pattern/decorator.php>

<http://www-home.htwg-konstanz.de/~haase/lehre/thisterm/pare/slides/Decorator.pdf>

TODO

- Nested inheritance, big steps
- Color scheme code
- Alternativen
- Wieso Decorators umbenennen
- Nachteile design
- Decorator combo function -> Factory
- Allgemeine meinung
- Wieso Decorators, nicht obj orient sprachen, fktnl prgrm
- Quellen