Software Design Patterns

Zusammensetzung

Daniel Gerber

UNIVERSITÄT LEIPZIG

Gliederung

- Einführung
- Iterator
- Composite
- Flyweight
- Zusammenfassung

So wird's werden

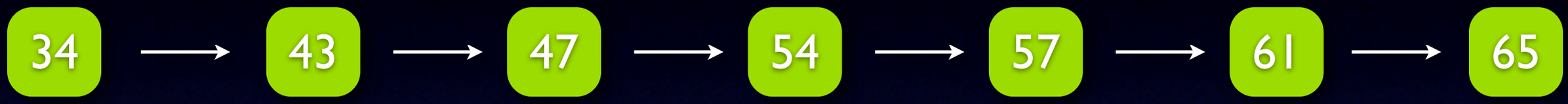
- Problem und Kontext an einem Beispiel vorstellen
- Lösung des Problems durch Design Pattern
- Beschreibung des Design Pattern
- Beispielhafte Implementierung
- Vor- und Nachteile

Warum Design Patterns?

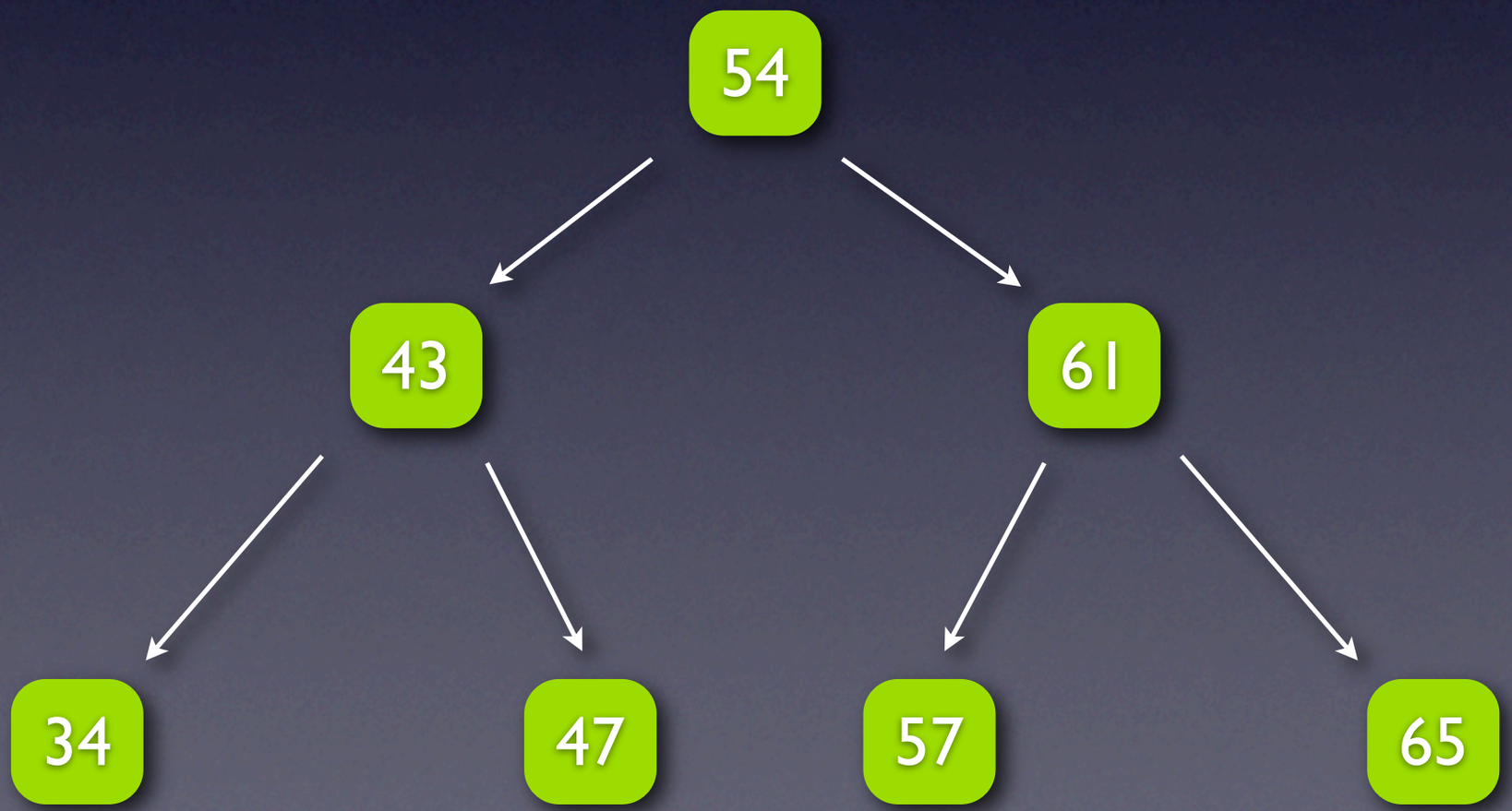
- schnellere Entwicklung
- einheitliche Software
- geringere Einarbeitungszeit
- geringere Fehleranfälligkeit

Iterator

Warum?

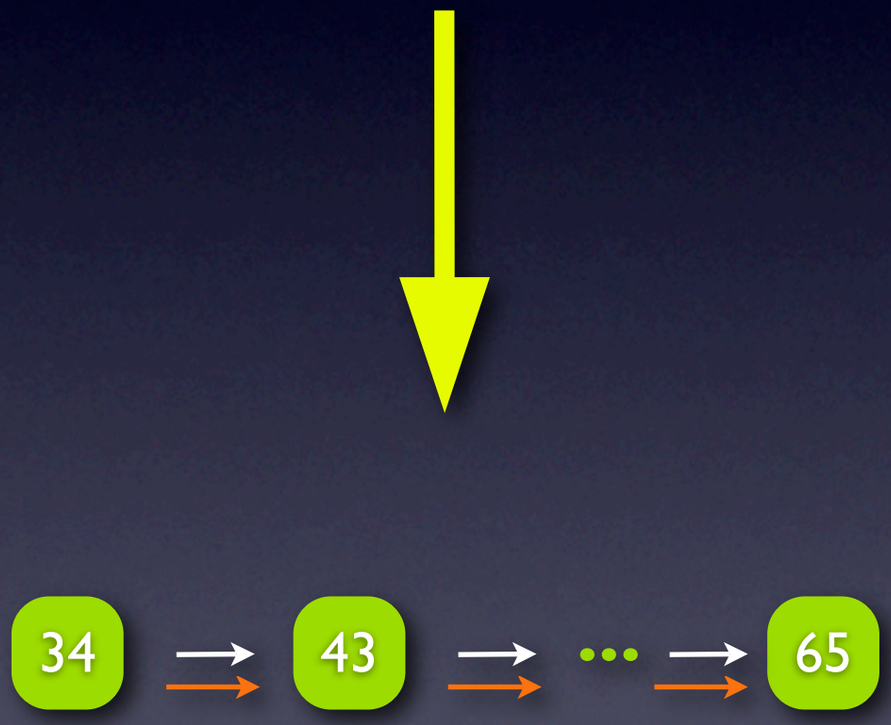


versus



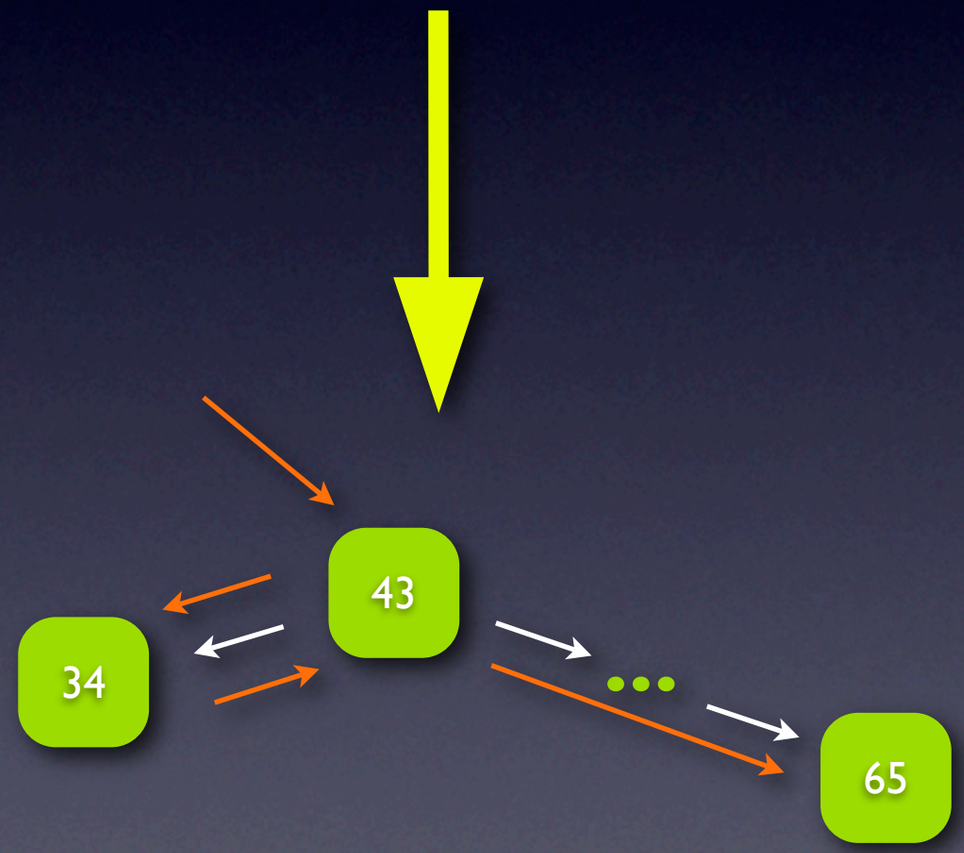
Problem

Wenn das Objekt eine Liste ist, dann



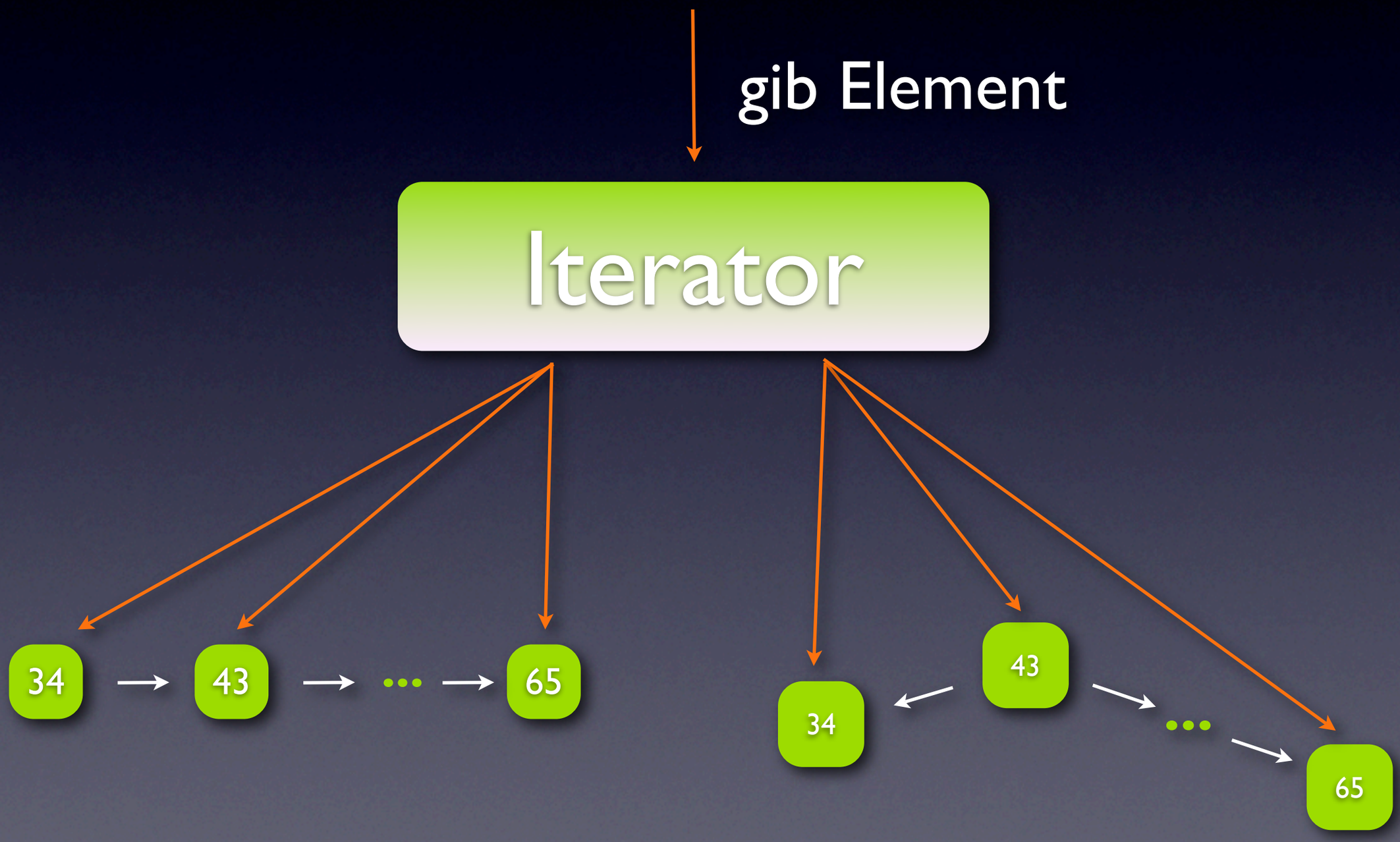
gehe sequentiell durch die Liste.

Wenn das Objekt ein Baum ist, dann



gehe in Preorder durch den Baum.

Lösung



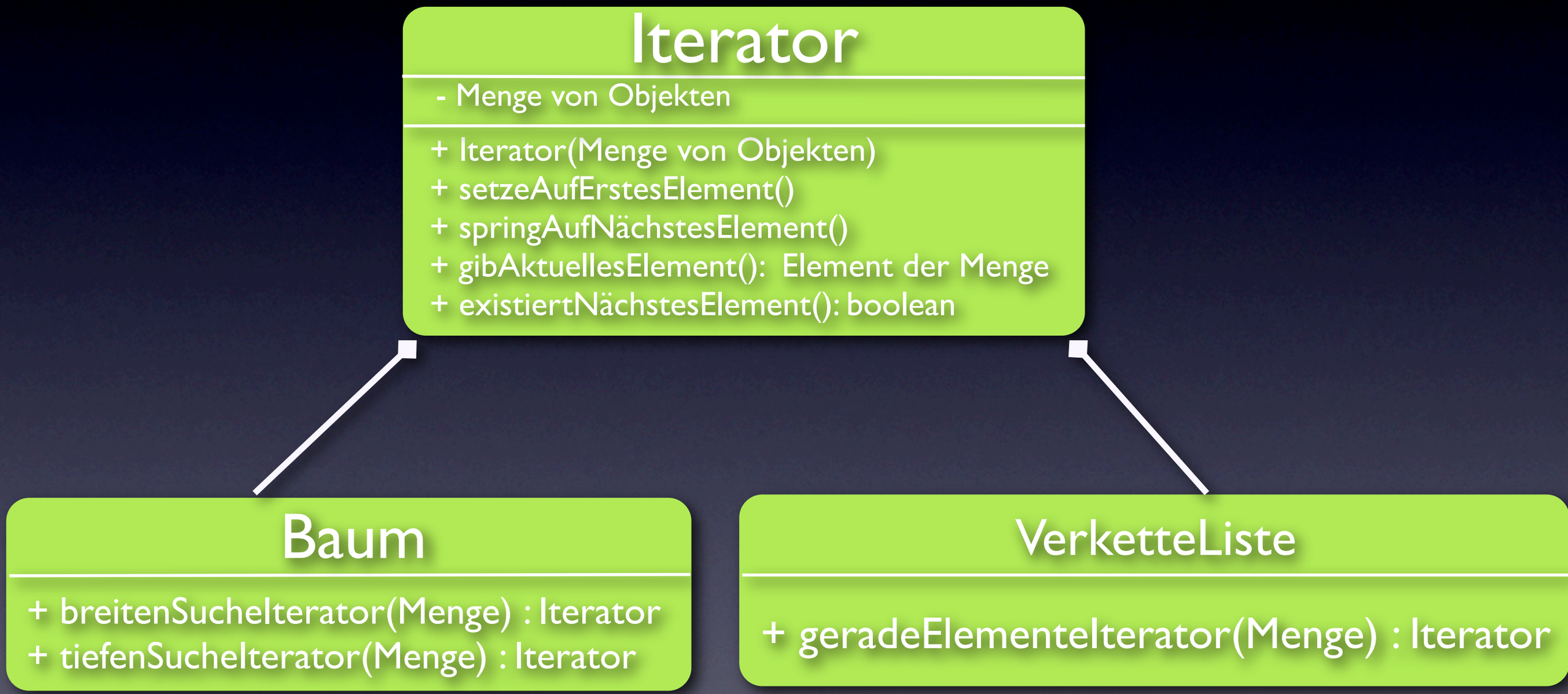
Etwas genauer

Iterator

- Menge von Objekten

+ Iterator(Menge von Objekten)
+ setzeAufErstesElement()
+ springAufNächstesElement()
+ gibAktuellesElement(): Element der Menge
+ existiertNächstesElement(): boolean

Noch etwas genauer



Vorbereitung

```
public void springeAufNächstesElement() {

    anzahlBesuchterKnoten++;
    besuchteKnoten.hinzufügen(aktuellerKnoten);

    if ( aktuellerKnoten.hatLinkenNachfolger() )
        this.aktuellerKnoten = aktuellerKnoten.holeLinkenNachfolger();
    else
        this.aktuellerKnoten = rechtsAbzweigen(aktuellerKnoten);
}

public boolean existiertNächstesElement() {

    return anzahlBesuchterKnoten
                    <= anzahlAllerKnoten;
}

public Object gibAktuellesElement() {

    return this.aktuellesElement;
}

public void setzteAufErstesElement() { ... }
```

```
// solange nach oben bewegen bis eine "Abzweigung" nach rechts möglich ist
private Knoten rechtsAbzweigen(Knoten knoten) {

    if ( knoten.hatRechtenNachfolger() )
        return knoten.holeRechtenNachfolger();

    else {
        while ( knoten.binIchRechterNachfolger() )
            knoten = knoten.holeVaterKnoten();

        return rechtsAbzweigen(knoten.holeVaterKnoten());
    }
}
```

Ab durch den Baum

```
Baum baum = new Baum();
baum.setzeWurzel(new Integer(54));
baum.setzeKnotenUndBlätter();

Iterator tiefenSucheIterator =
baum.tiefenSucheIterator(baum);

while ( tiefenSucheIterator. existiertNächstesElement() ) {

    tiefenSucheIterator.springeAufNächstesElement();
    gibAus( tiefenSucheIterator. gibAktuellesElement() );
}
```

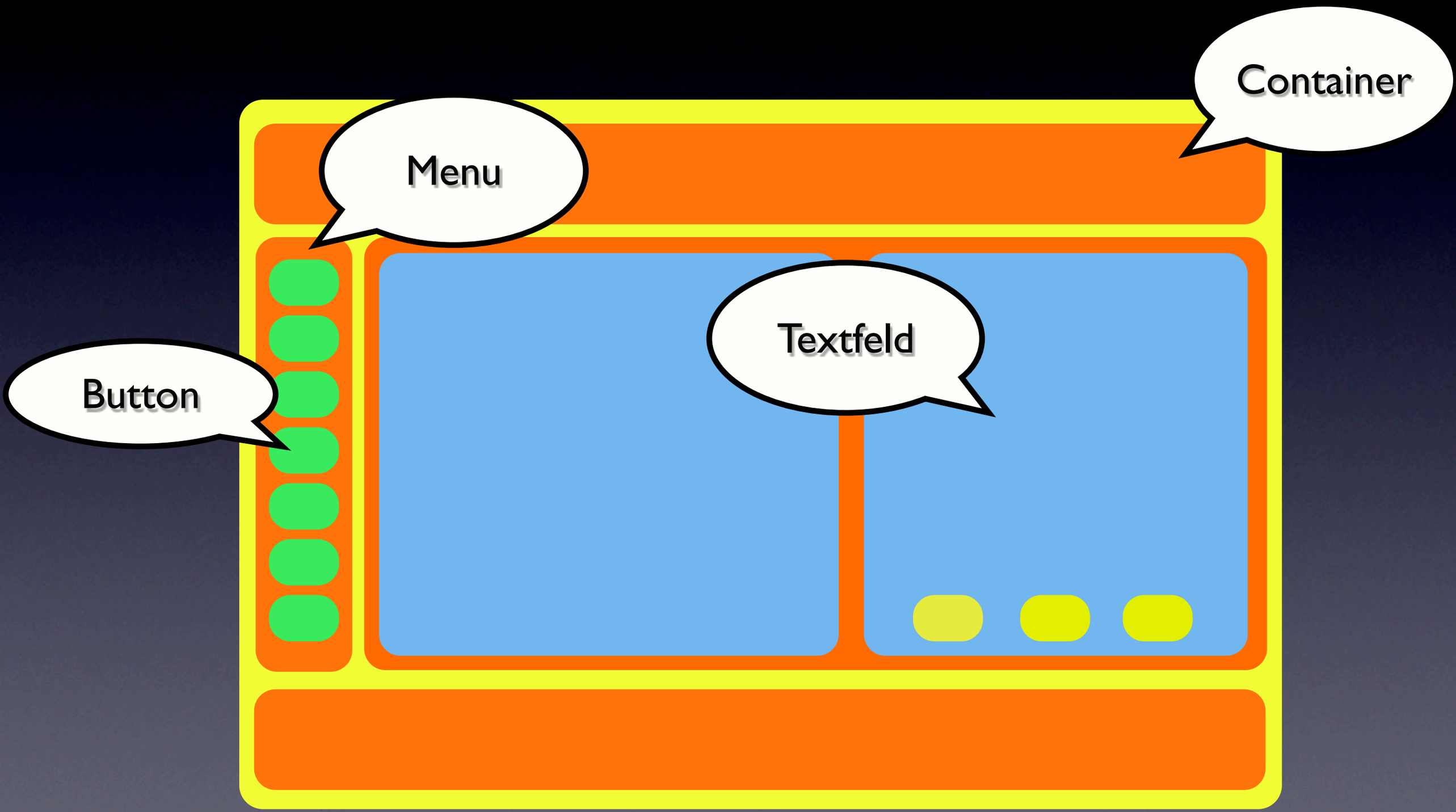
ja, nein, vielleicht?

- \+ durch eine Sammlung laufen
- \+ interner Aufbau der Sammlung uninteressant
- \+ paralleler Durchlauf der gleichen Sammlung
- \+ ändern der Sammlung während des Durchlaufs
- − Speicherkosten für neues Objekt
- − Sammlung verdoppelt sich mit Durchlauf
- − Abfrage ob nächstes Element existiert

Composite

a.k.a. Kompositum

Warum?



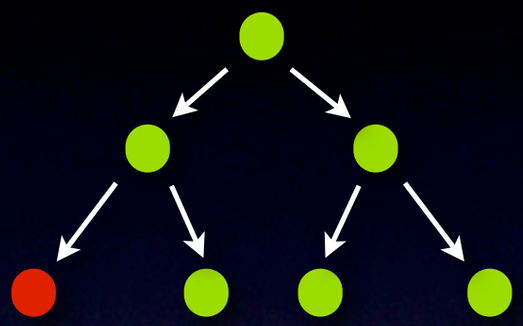
igittigitt

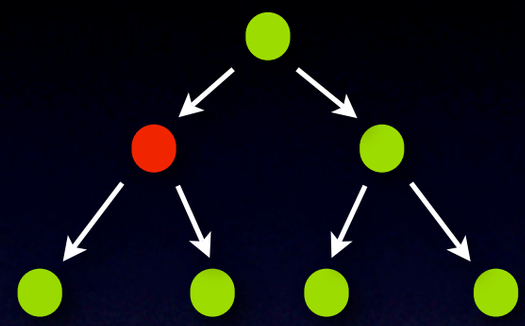
```
public class Fenster {

    List<Knopf> knöpfe;
    List<Menu> menus;
    List<Behälter> behälterListe;

    public void aktualisieren() {
        for ( Knopf knopf : knöpfe )
                button. aktualisieren();
        for ( Menu menu : menus )
                menu. aktualisieren();
        for ( Behälter behälter : behälterListe )
                behälter. aktualisieren();
    }
}
```

Entweder oder





Aktualisiere Blatt

Aktualisiere alle Unterknoten dieses Behälters

schwer erweiterbar, fehleranfällig, aufwendig

Rettung naht ...

Quelle: http://www.mattbrownart.com/

Alle sind gleich!

JA

Knöpfe

=

Datei

- Öffnen
- Speichern
- Schließen
- Importieren
- Exportieren
- Drucken

Menüs

=

JA
Nein
JA

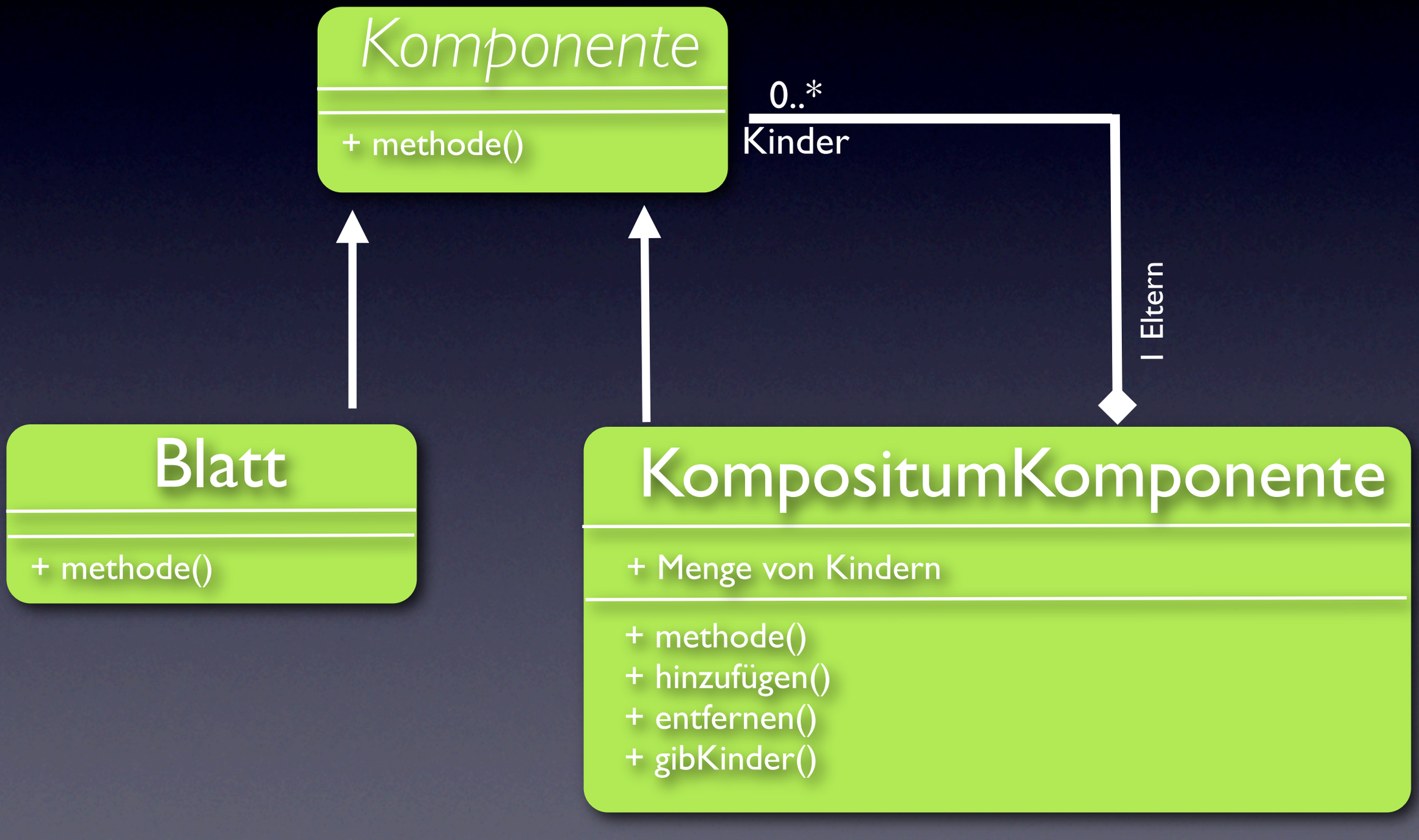
Nein
JA
JA

Datei

- Öffnen
- Speichern
- Schließen
- Importieren
- Exportieren
- Drucken

Behälter

en détail


Komponente
+ methode()
0..*
Kinder
1 Eltern
Blatt
+ methode()
KompositumKomponente
+ Menge von Kindern
+ methode()
+ hinzufügen()
+ entfernen()
+ gibKinder()

Spielraum

- ➡ Komposita kennen ihre enthaltenen Komponenten.
  - Sollten Sie auch ihre Eltern kennen?
- ➡ Es müssen Methoden existieren um Kinder zu verwalten.
  - Wo sollen diese Methoden definiert werden?

In abstrakter Klasse	Im Kompositum
◉ Transparenz	◉ Sicherheit
◉ Verlust an Sicherheit	◉ Verlust an Transparenz

jetzt aber ...

```
public abstract class Komponente {
    public void aktualisere();     public void tueEtwas(){schreibe(”Hallo”);}
}
```

```
public class Knopf extends Komponente{
    public void aktualisere() {
        tueEtwas();
    }
}
```

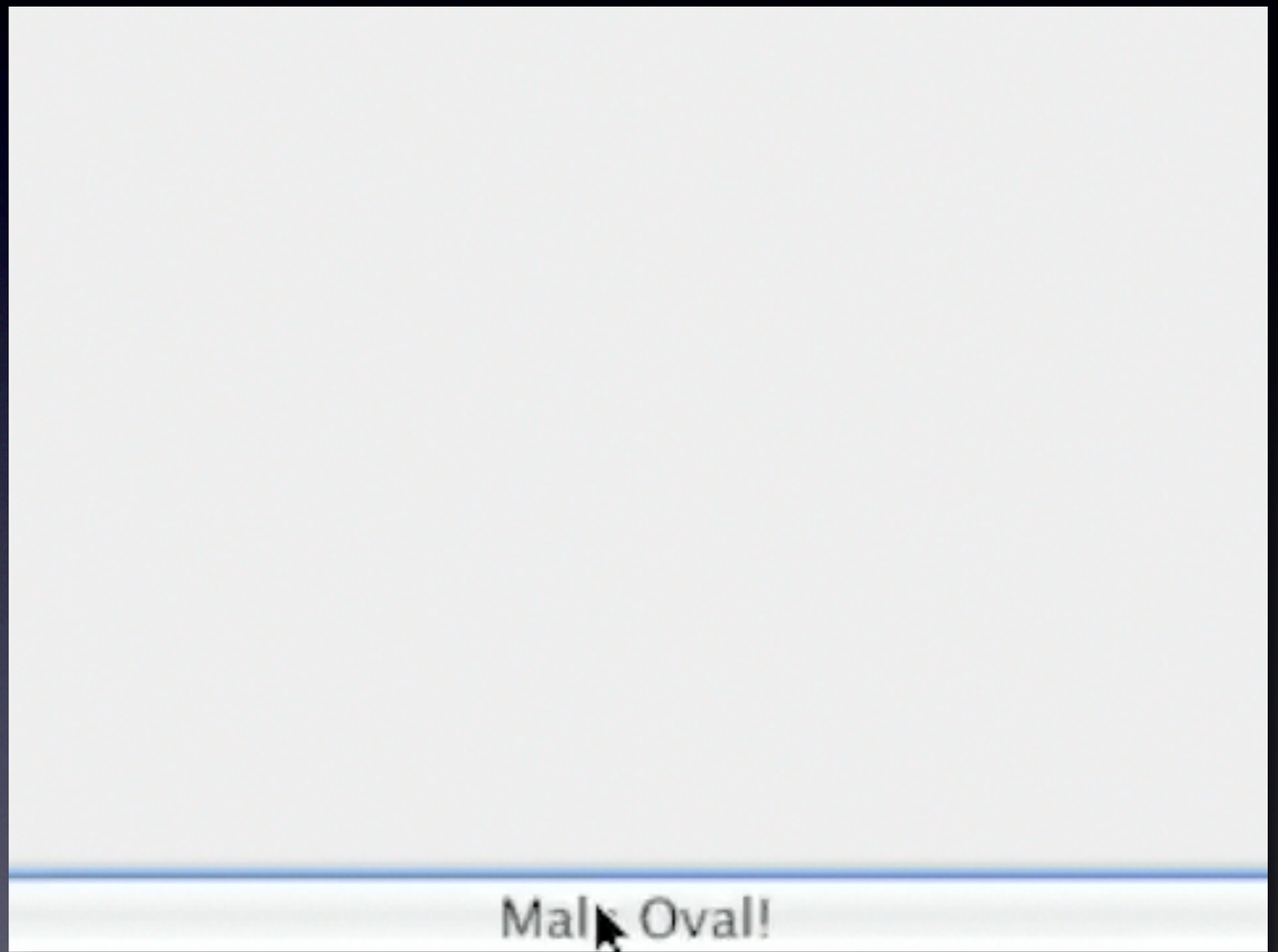
```
public class KomponentenKompositum extends Komponente {
    List<Komponenten> komponenten;
    public void aktualisere() {
        for (Komponente komponente : komponenten )
          komponente.aktualisiere();
        tueEtwas();
    }
}
```

hui oder pfui?

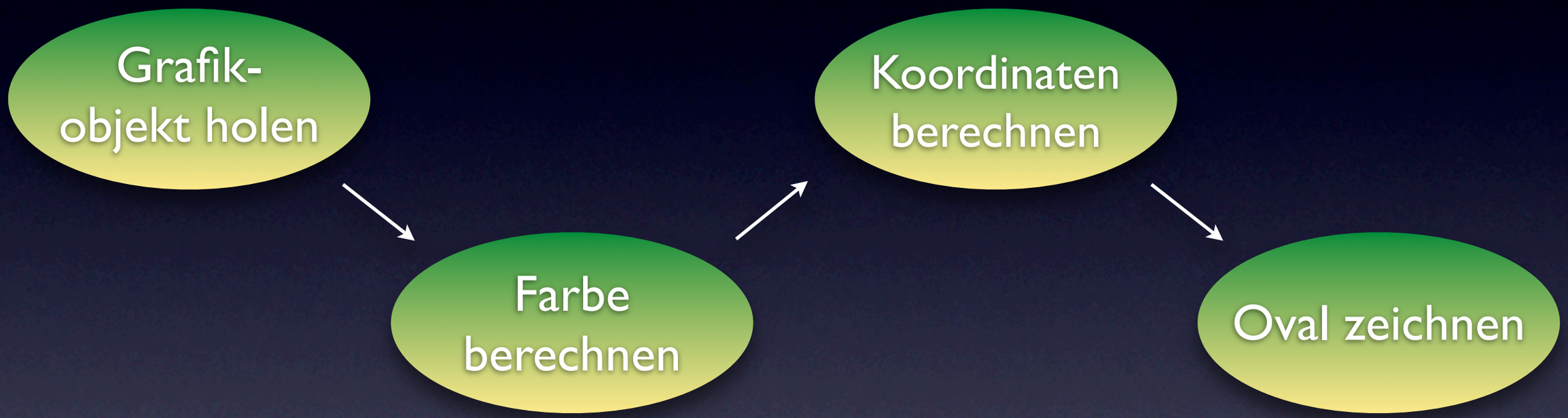
- \+ behandle primitive genauso wie komplexe Objekte
- \+ einfach neue Komponenten hinzuzufügen
- \+ Klienten werden simpler, da keine Unterscheidung notwendig ist
- – Typen der Komponenten der Komposita schlecht einzuschränken
- – nicht zwangsläufig Bäume
- – nicht garantiert zyklenfrei

Flyweight

Warum?



Etwas naiv



➡ nicht objektorientiert

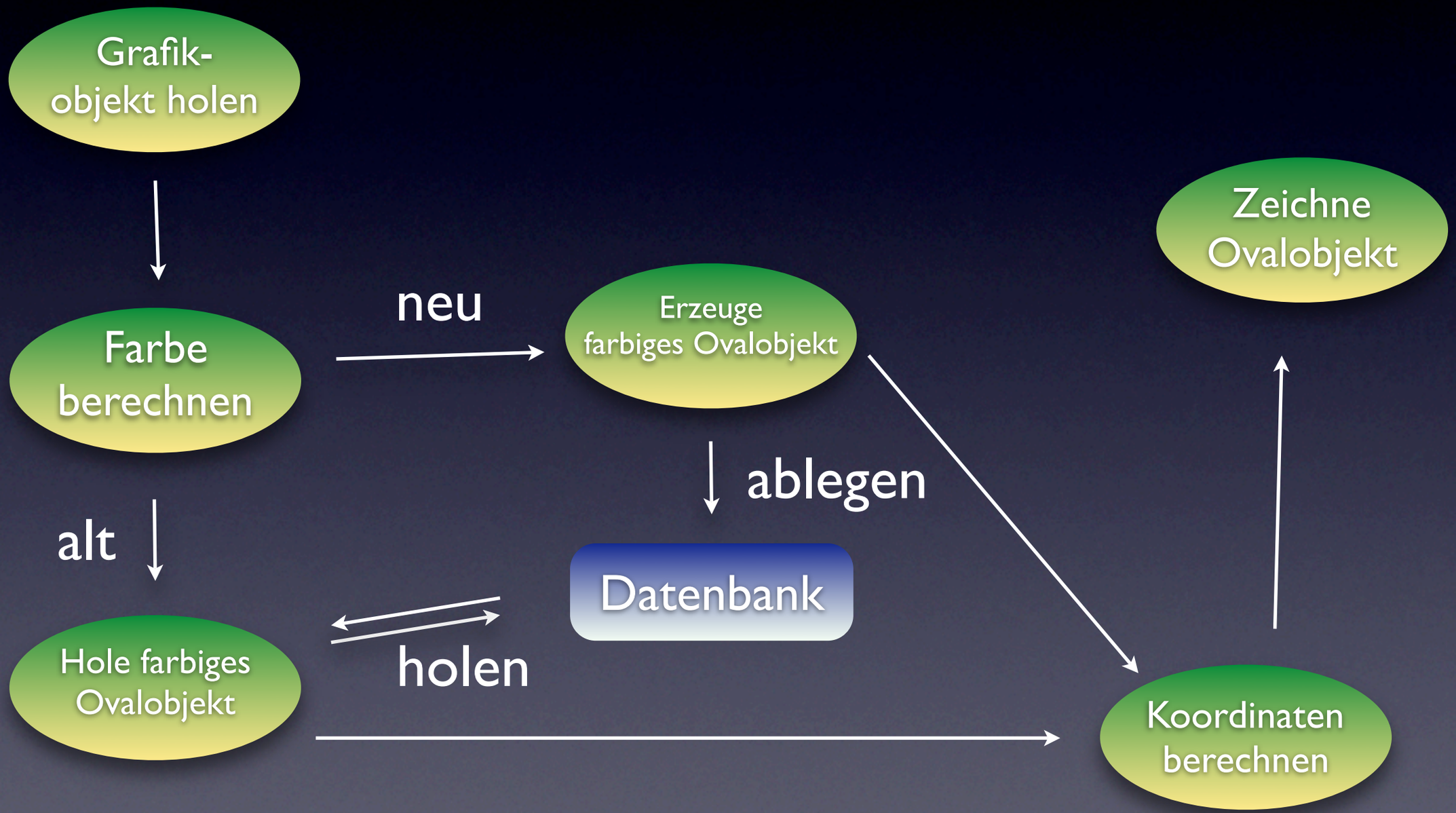
Ein bisschen besser

objektorientiert

\+ pro zu zeichnendem Oval ein Objekt

jede Menge Objekte

Fliegengewichte



Innen vs. Außen I

Intrinsischer Zustand:

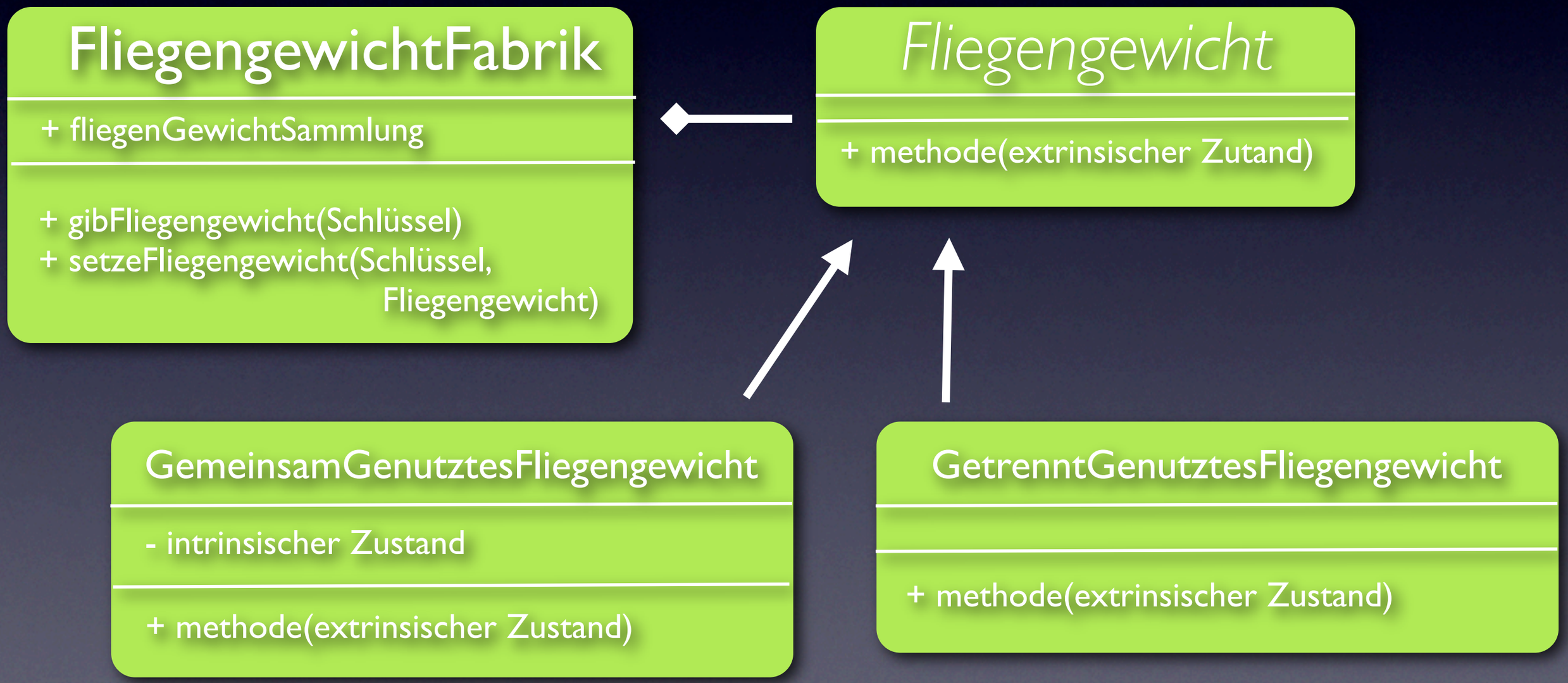
- bestimmt durch alle Eigenschaften die dem Objekt selbst gehören und es zu dem machen was es ist
- wird ein mal berechnet und verändert sich nie
- die Farbe oder die innere Motivation

Innen vs. Außen II

Extrinsischer Zustand:

- bestimmt durch alle Einflüsse die von außen auf das Objekt einwirken
- steht nicht fest und wird zur Laufzeit übergeben
- die Koordinaten oder eine Gehalts-(erhöhung|kürzung)

So sieht's aus

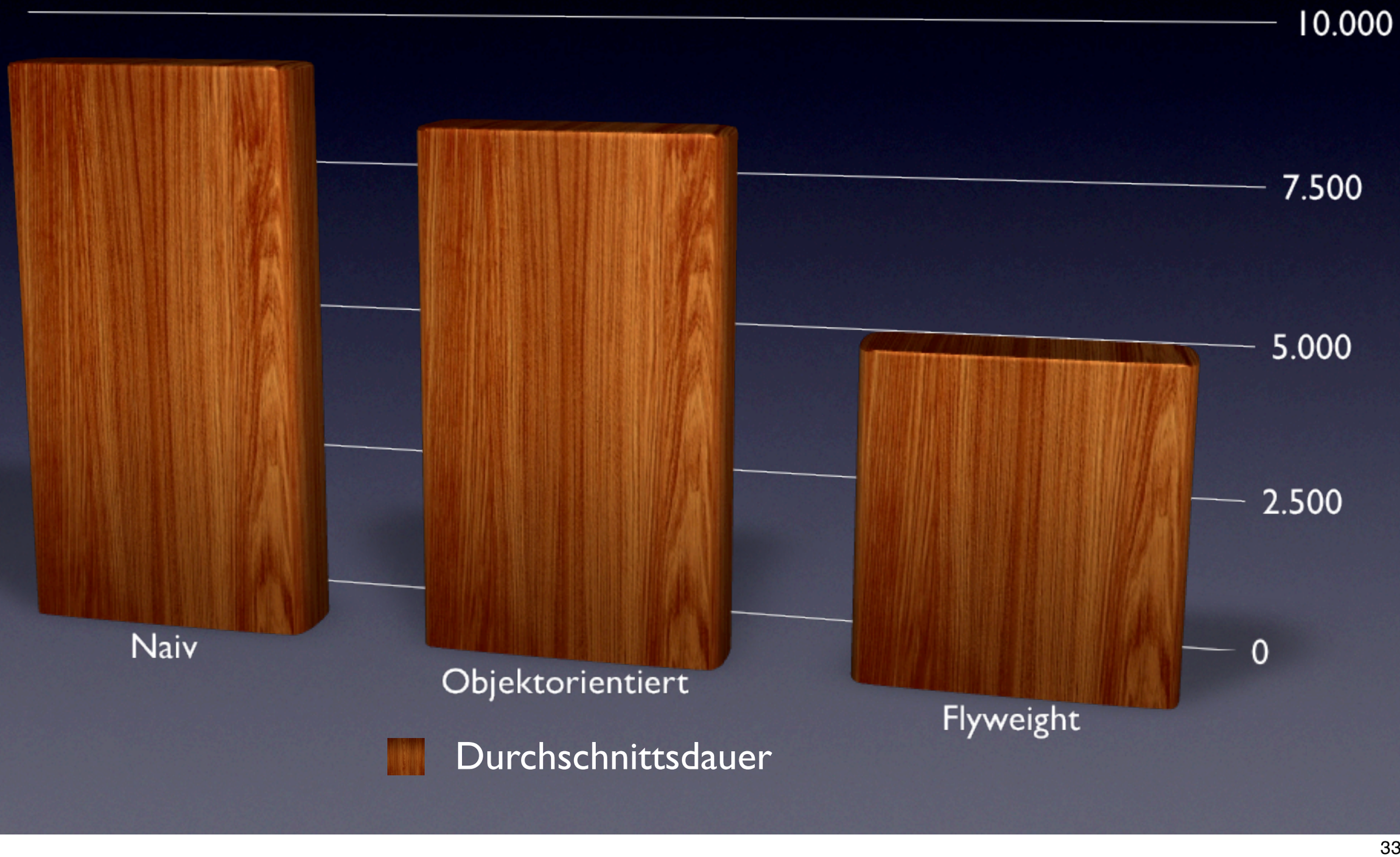


Let's fly!

```
...
for( int i = 0 ; i < ANZAHL_OVALE ; ++i ) {
   Oval oval = fliegenGewichtFabrik.holeOval(zufallsFarbe());
   oval.zeichne(grafik, zufallsX(), zufallsY(), zufallsHöhe(), zufallsBreiteY());
}
...
```

```
public class FliegenGewichtFabrik {
   private HashMap<Farbe, Oval> ovaleNachFraben = new HashMap<Farbe, Oval>();
   public Oval holeOval(Farbe farbe) {
        Oval oval = ovaleNachFarben.hole(farbe);
        if( oval == null ) {
           oval = new Oval(farbe);
           ovaleNachFarben.fügeHinzu(farbe, oval);
        }
        return oval;
   }
}
```

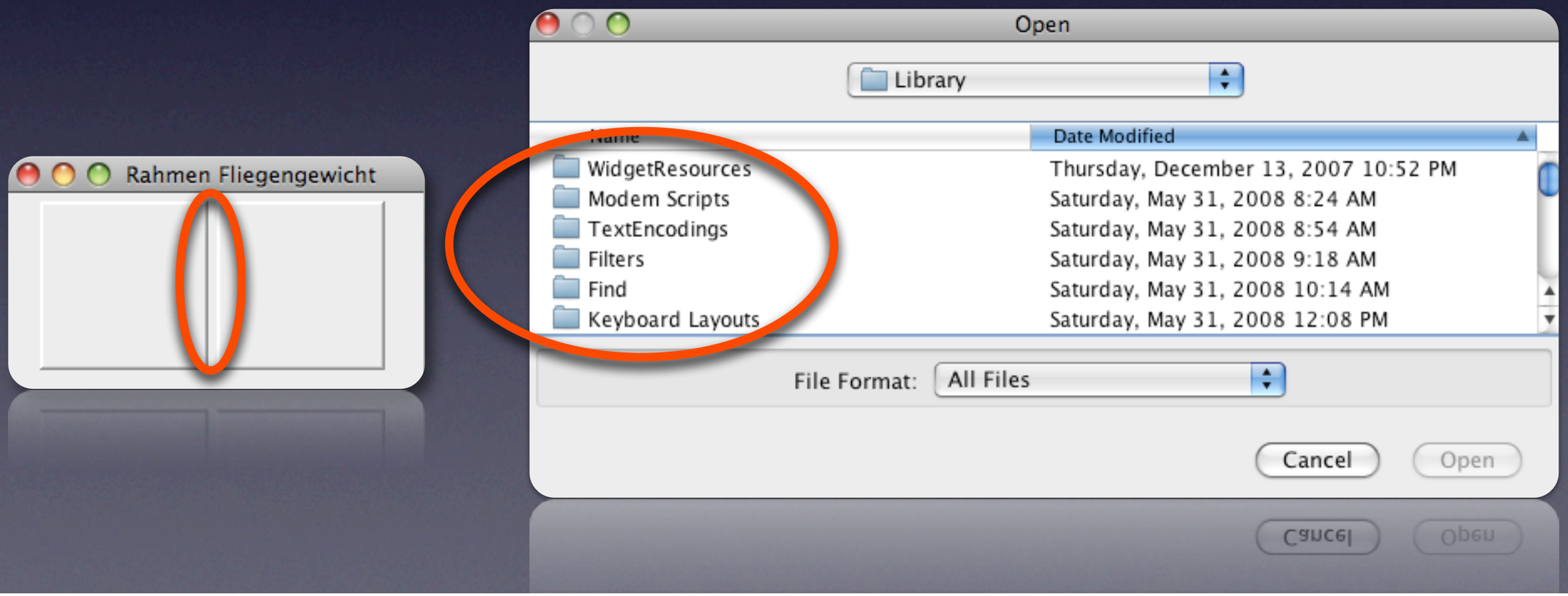
Nachgemessen



Gibt's das auch noch wo anders?

String:

```
String fliegenGewicht1 = "fliegenGewicht";
String fliegenGewicht2 = "fliegenGewicht";
System.out.println(fliegenGewicht1 == fliegenGewicht2);
```



Top oder Flop

\+ verringert Anzahl erzeugter Objekte

\+ Senkung der Speicherkosten

\+ Performancegewinn

\- Objekte müssen gefunden werden

\- ausgelagerter Zustand muss gefunden werden

\- eventuell sogar berechnet

Doppelt hält besser I

Iterator

- Durchlaufen einer aggregierten Struktur ohne Struktur zu enthüllen

+ Implementierung der Datenstruktur bleibt verborgen

– erhöhte Speicher- und Laufzeitkosten

Doppelt hält besser II

Composite

- Repräsentation von Teil-Ganzes Beziehungen

+ gleiche Behandlung von primitiven Objekten und deren Behältern

– eventuell Typüberprüfungen zur Laufzeit

Doppelt hält besser III

Flyweight

- Reduktion der Anzahl an Objekten durch teilen der Eigenschaften

\+ Speicherkosten werden minimiert

– Wiederauffinden des ausgelagerten Zustands

Q & A

da hab ich's her

Alle:

Gamma, Erich; Richard Helm, Ralph Johnson, John M. Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. pp. 395. ISBN 0201633612

Iterator:

- http://en.wikipedia.org/wiki/Iterator_pattern
- http://de.wikibooks.org/wiki/Muster:_Iterator
- http://de.wikibooks.org/wiki/Java_Standard:_Muster_Iterator

Composite:

- http://en.wikipedia.org/wiki/Composite_pattern
- http://userpages.umbc.edu/~tarr/dp/lectures/Composite.pdf

Flyweight:

- http://en.wikipedia.org/wiki/Flyweight_pattern
- http://www.javaworld.com/javaworld/jw-07-2003/jw-0725-designpatterns.html
- http://www.javabeat.net/advanced/design-pattern/flyweight-pattern.php