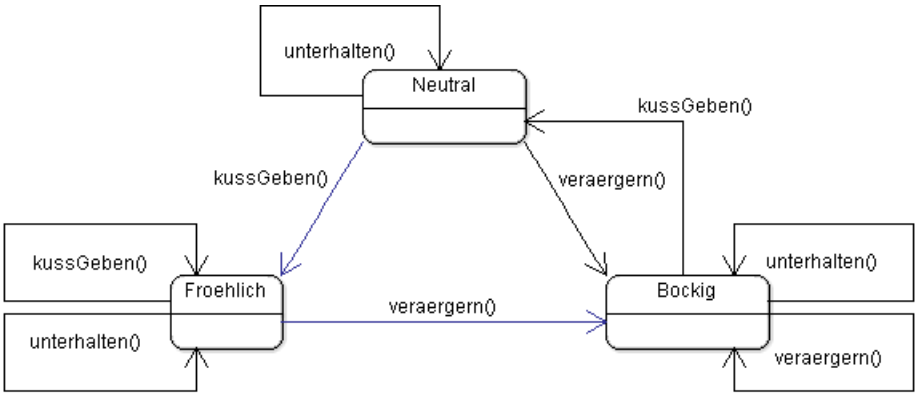
Informatik: State und Facade Pattern

- Verstehen wieso es sinnvoll ist, das State Pattern einzusetzen
- Das State Pattern konkret für ein Problem implementieren können
- Mit dem Facade Pattern komplexe Funktionalitäten einfach zur Verfügung stellen können

Einleitendes Beispiel State: Freundin

Freundin ändert je nach ausgeführter Aktion (`kussGeben()`, `unterhalten()`, `verärgern()` in Abhängigkeit des momentanen Zustands (`Neutral`, `Fröhlich`, `Bockig`) laut nachfolgendem Zustandsdiagramm diesen ab



Schlechte Lösung (strukturierte Programmierung)

```
public void kussGeben() {
  switch (aktuellerZustand) {
    case NEUTRAL: {
      System.out.println("Hihi :-)");
      aktuellerZustand = FROEHLICH;
      break;
    }
    case FROEHLICH: {
      System.out.println("Hihi");
      aktuellerZustand = FROEHLICH;
      break;
    }
    case BOCKIG: {
      System.out.println("Na gut! Hab dich wieder lieb :-|");
      aktuellerZustand = NEUTRAL;
      break;
    }
  }
}
```

Freundin
NEUTRAL : int FROEHLICH : int BOCKIG : int aktuellerZustand : int
unterhalten() : void kussGeben() : void veraergern() : void getZustand() : int

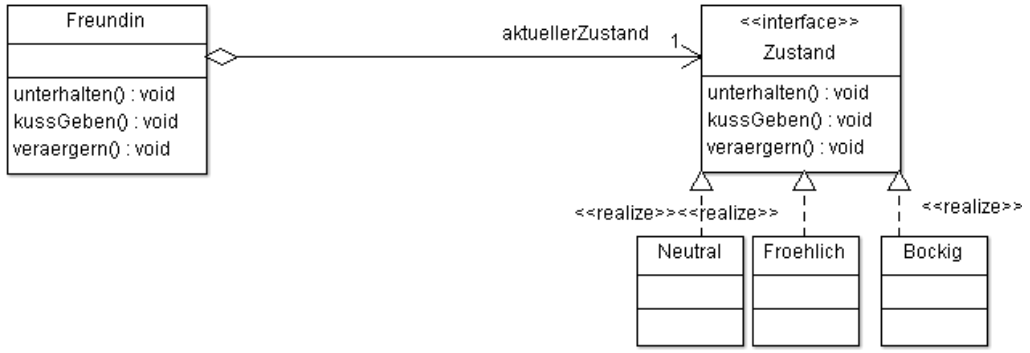
☺ Unübersichtlich denn Zustand ist über mehrere Methoden verteilt

☺ Code hat mit Zustandsautomaten nichts gemein

☺ Schlechte Wartbarkeit und Erweiterbarkeit, wenn neue Zustände dazu kommen ➔ Zustände in eigene Objekte auslagern

☺ Code der nur die Freundin betrifft muss angepasst werden, wenn Zustände dazukommen

Bessere Lösung




```
public class Freundin
{
  private Zustand aktuellerZustand = new Neutral();
  public void unterhalten() {
    this.aktuellerZustand.unterhalten();
  }
  public void kussGeben() {
    this.aktuellerZustand.kussGeben();
  }
  public void veraergern() {
    this.aktuellerZustand.veraergern();
  }
}
```

```
public class Neutral implements Zustand
{
  @Override
  public void unterhalten() {
    System.out.println("Hallo!");
  }
  @Override
  public void kussGeben() {
    System.out.println("Hihi :-)");
  }
  @Override
  public void veraergern() {
    System.out.println("Du spinnst wohl! …");
  }
}
```

☺ Bei neuen Zuständen muss Freundin nicht angepasst werden

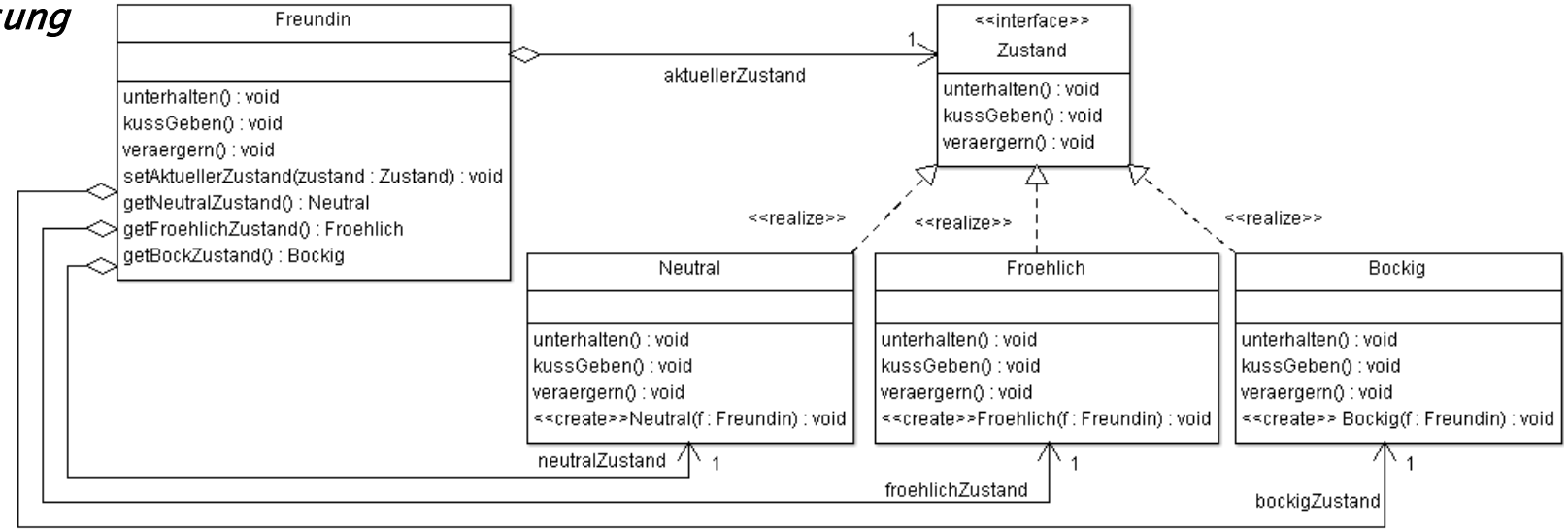
☺ Zustand kapselt sein Verhalten in einer Klasse

☺ Zustandsklassen haben dieselbe Schnittstelle wie die Freundin

☺ Freundin delegiert Aufrufe an Zustandsobjekt

☹ Zustandsobjekt kann seine Zustandsänderung nicht der Freundin mitteilen

Gute Lösung



Zustandsobjekte müssen selbständig Zustand der Freundin wechseln können. Dazu benötigen sie Referenz auf die Freundin und in Freundin die Setter-Methode (`setAktuellerZustand()`). Damit passendes Zustandsobjekt bei Änderung des Zustandes nicht immer neu angelegt wird, stellt die Freundin diese zur Verfügung

```
public class Neutral implements Zustand
{
  private Freundin freundin = null;

  public Neutral(Freundin freundin) {
    this.freundin = freundin;
  }

  …

  @Override
  public void veraergern() {
    System.out.println("Du spinnst wohl! Ich bin sauer! :-(");
    freundin.setAktuellerZustand(freundin.getBockigZustand());
  }
}
```

HINWEIS: Methode könnte auch Zustandsobjekt zurück liefern. Damit könnte auf beschriebenen Mechanismus verzichtet werden

State

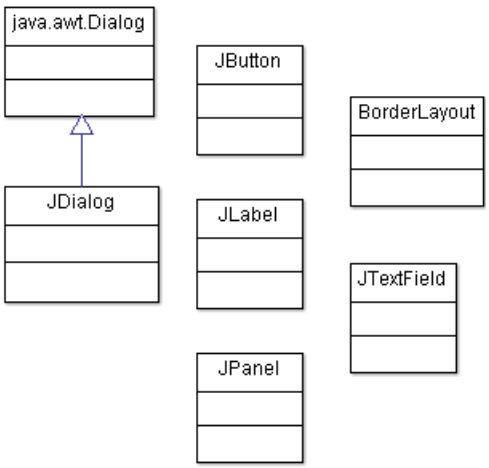
- "Ermögliche es einem Objekt, sein Verhalten zu ändern, wenn sein interner Zustand sich ändert. Es wird so aussehen als ob das Objekt seine Klasse gewechselt hat."

[1]

Unsere Klassenbezeichnung	Allgemeine Klassenbezeichnung
`Freundin`	`Context`
`Neutral, Bockig, Froehlich`	`Concrete States`

Einleitendes Beispiel Facade: Meldungsfenster in Java

Verschiedene Java-Programme (Clients) benötigen unterschiedlichste Meldungsfenster (OK, Ja/Nein, Speichern/Nicht Speichern/Abbrechen, usw.)



☹ Jeder Client muss benötigte Klassen und deren Zusammenspiel gut kennen

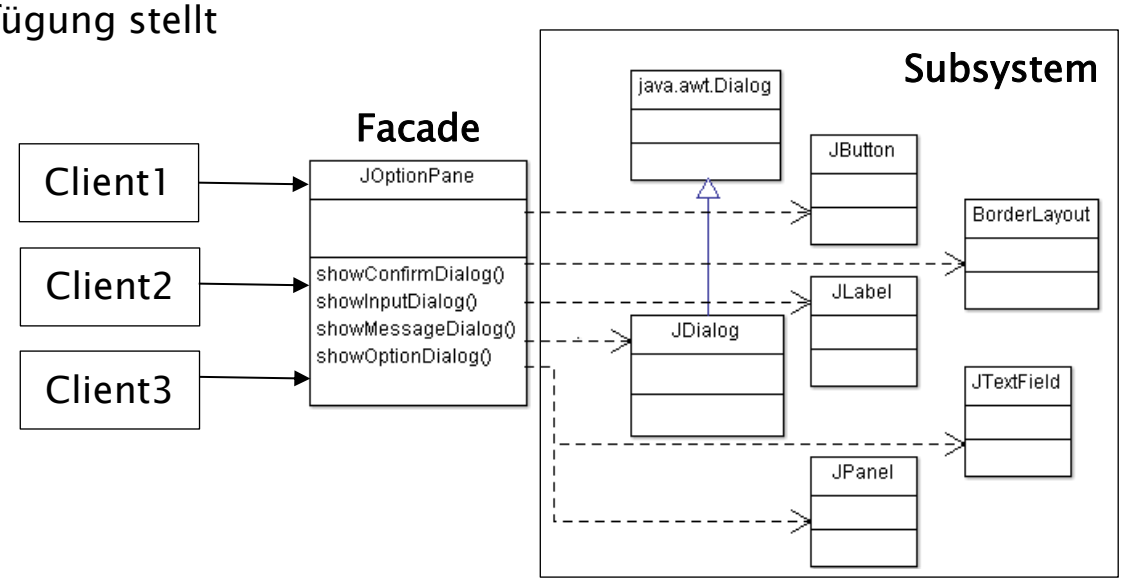
☹ Werden Klassen geändert, so hat Client großen Anpassungsaufwand

☹ Alle Clients schreiben immer den gleichen Code um Aufgabe zu lösen ➔ Redundanz und Inkonsistenz

[1] Nach Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: "Entwurfsmuster. Elemente wiederverwendbarer objektorientierter Software". Addison-Wesley. 1. Auflage 1996 (***Gang of Four***)

Gute Lösung

Einfügen einer Zwischenklasse (Facade) welche vereinfachte und einheitliche Schnittstelle (API) zur Darstellung von Meldungsfenstern zur Verfügung stellt



☺ Vereinfachte Benutzung, Clients benötigen kein Wissen über Subsystem. Facade kann beliebig erweitert werden

☺ Clients können Subsystem trotzdem noch direkt nutzen

☺ Client ist nur mehr von Facade abhängig. Änderungen am Code des Subsystems haben keinen direkten Einfluss auf Client

☺ Code ist nur einmal im Facade vorhanden

Facade

- "Biete eine einheitliche Schnittstelle zu einer Menge von Schnittstellen eines Subsystems. Die Fassadenklasse defniert eine abstrakte Schnittstelle, welche die Verwendung des Subsystems vereinfacht."

[2]

Unsere Klassenbezeichnung	Allgemeine Klassenbezeichnung
`JOptionPane`	`Facade`

[2] Nach Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: "Entwurfsmuster. Elemente wiederverwendbarer objektorientierter Software". Addison-Wesley. 1. Auflage 1996 (***Gang of Four***)