# Programmentwicklung II

## Bachelor of Science

Prof. Dr. Rethmann / Prof. Dr. Brandt

Fachbereich Elektrotechnik und Informatik
Hochschule Niederrhein

Sommersemester 2021

*Erzeuger und Strukturen*

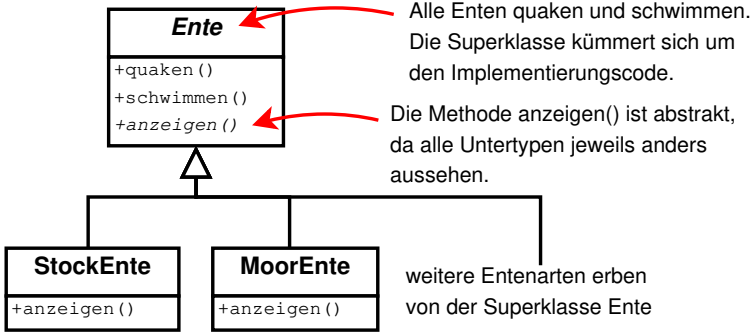
- Abstrakte Fabrik (Abstract Factory)
- Einzelstück (Singleton)
- Dekorierer (Decorator)
- Kompositum (Composite)

*Verhalten*

- Strategie (Strategy)
- Zustand (State)
- Beobachter (Observer)
- Model-View-Controller

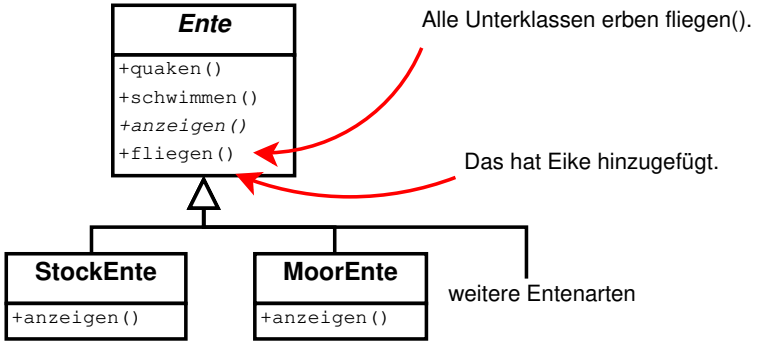
Eike arbeitet für ein Unternehmen, das das höchst erfolgreiche Ententeich-Simulationsspiel `SimEnte` erstellt[1].

- Das Spiel kann zeigen, wie eine Vielzahl von Entenarten umherschwimmt und Quak-Geräusche von sich gibt.
- Die Spiele-Entwickler haben klassische OO-Techniken verwendet:




[1] aus dem Buch *Entwurfsmuster von Kopf bis Fuß* von E. Freeman und E. Freeman

- Im letzten Quartal geriet das Unternehmen unter erheblichen Druck durch Konkurrenten.
- Die Unternehmensleitung beschließt beim Golfen, dass sie etwas richtig Beeindruckendes brauchen, was sie auf der Aktionärsversammlung nächste Woche präsentieren können.
- Eikes Manager sagt ihnen, dass fliegende Enten für Eike kein Problem sind, denn er ist ja OO-Programmierer, und so schwer kann das doch nicht sein.
- Eike überlegt: „Ich füge der Superklasse `Ente` einfach eine `fliegen`-Methode hinzu, die dann von allen Entenarten geerbt wird.“

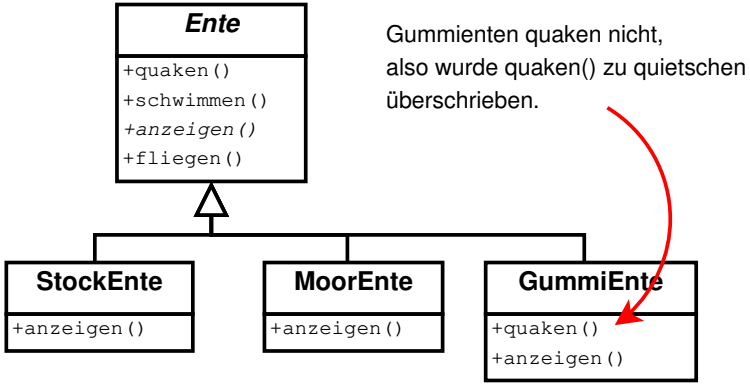



- Ein Anruf von der Aktionärsversammlung: „Eike, ich habe gerade eine Demo vorgeführt, auf der Gummienten über den Bildschirm flogen! Soll das ein Witz sein?“
- Was ist passiert? Was ging schief?

- Es war (wie immer) nicht genug Zeit, die Software ausreichend zu testen.
- Eike hat nicht daran gedacht, dass nicht alle Unterklassen von `Ente` fliegen dürfen.

→ Eine lokale Änderung des Codes führte zu einem globalen Nebeneffekt! (fliegende Gummiente)

- Eike denkt: „Na, gut. Mein Entwurf hat einen kleinen Haken. Warum nennen sie es nicht einfach Feature? Sieht doch süß aus.“




Eike denkt über Vererbung nach:

- „Ich könnte die Methode `fliegen` überschreiben, so wie ich das mit der Methode `quaken` gemacht habe.“
- „Aber was passiert, wenn wir hölzerne Lockenten hinzufügen, die auch nicht quaken oder fliegen?“

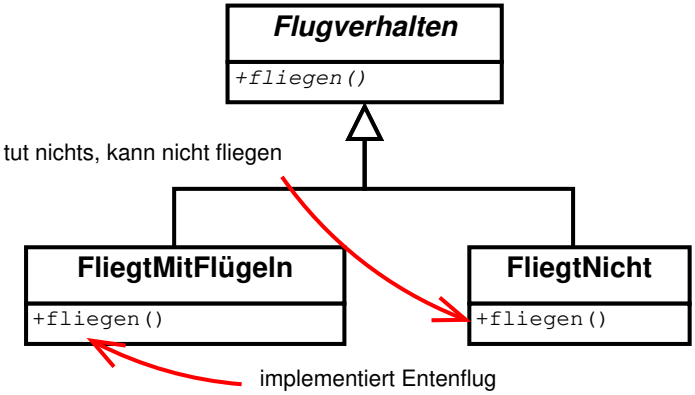
- Die Unternehmensleitung verkündet: „Das Produkt soll jetzt alle 6 Monate aktualisiert werden“. Genaueres weiß man noch nicht.
- Eike weiß jetzt, dass er jedes Mal, wenn dem Programm eine neue Entenklasse hinzugefügt wird, die Methoden `fliegen` und `quaken` unter die Lupe nehmen und ggf. überschreiben muss.
- Eine Konstante bei der Software-Entwicklung: Veränderung!

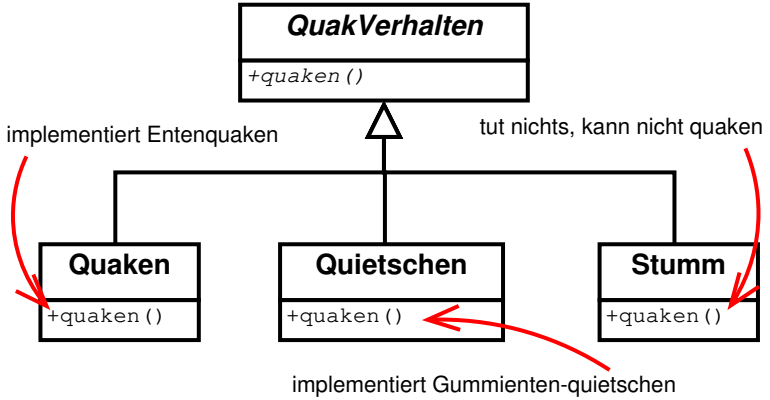
  Egal, wie gut Sie Ihre Anwendung entworfen haben, sie muss im Laufe der Zeit wachsen und sich wandeln, sonst ist sie tot.

Entwurfsentscheidung:

> Identifizieren Sie die Aspekte Ihrer Anwendung, die sich ändern können, und trennen Sie sie von denen, die konstant bleiben.

Die Methoden `fliegen` und `quaken` sind die Teile der Klasse Ente, die über die Enten hinweg variieren.






Bei diesem Entwurf können andere Ententypen die verschiedenen Flug- und Quakverhaltensweisen wiederverwenden.

Wir können weiteres Verhalten hinzufügen, ohne auch nur eine unserer bestehenden Verhaltensklassen zu ändern.

*Implementierung:* mittels Delegation!

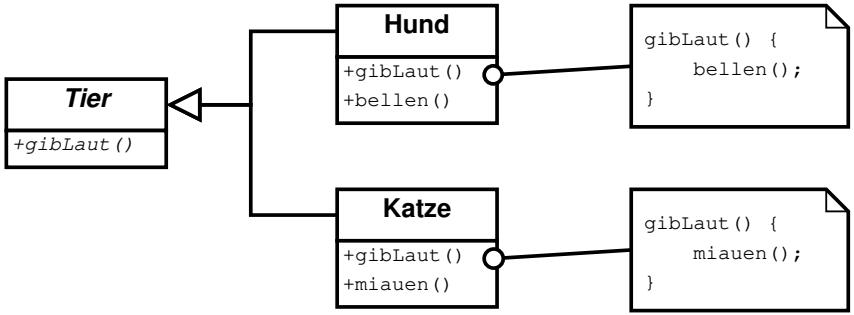
> Programmieren Sie auf eine Schnittstelle, nicht auf eine Implementierung.

Auf eine Implementierung programmieren wäre:

```
Hund *h = new Hund();
h->bellen();
```

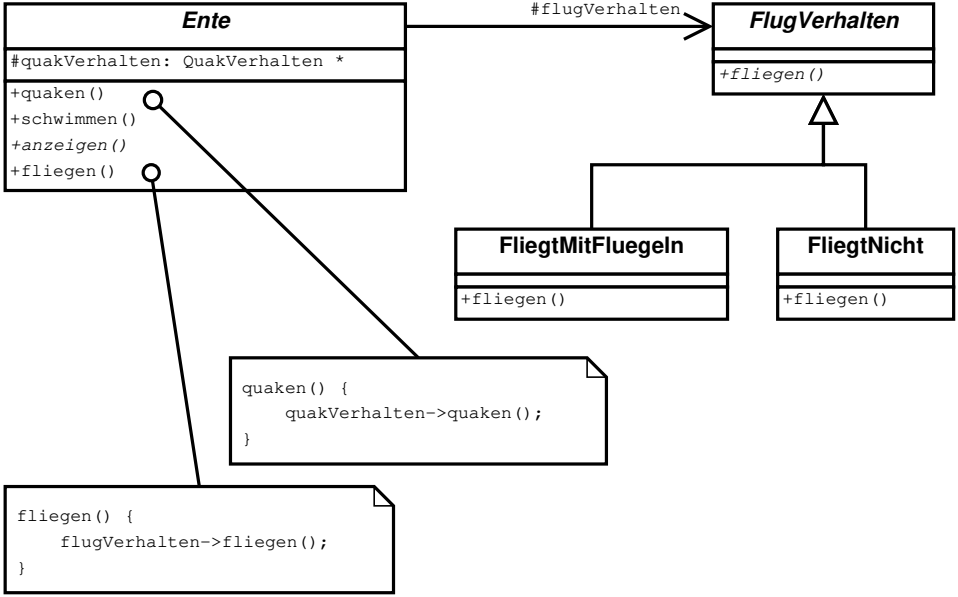
Auf eine Schnittstelle/Supertyp programmieren wäre:

```
Tier *t = new Hund();
t->gibLaut();
```




Noch besser wäre es, die Erzeugung des Untertyps nicht im Code festzuschreiben: (siehe abstrakte Fabrik)

```
Tier *t = holeTier();
t->gibLaut();
```


Ente
#quakVerhalten: QuakVerhalten *
+quaken()
+schwimmen()
+anzeigen()
+fliegen()
#flugVerhalten
FlugVerhalten
+fliegen()
FliegtMitFluegeln
+fliegen()
FliegtNicht
+fliegen()
quaken() {
quakVerhalten->quaken();
}
fliegen() {
flugVerhalten->fliegen();
}

```
class Ente {
protected:
    QuakVerhalten *quakVerhalten;
    ...
public:
    void quaken(void) {
        quakVerhalten->quaken();
    }
    ...
};
class StockEnte: public Ente {
public:
    StockEnte() {
        quakVerhalten = new Quaken();
        flugVerhalten = new FliegtMitFluegeln();
    }
    ...
};
```

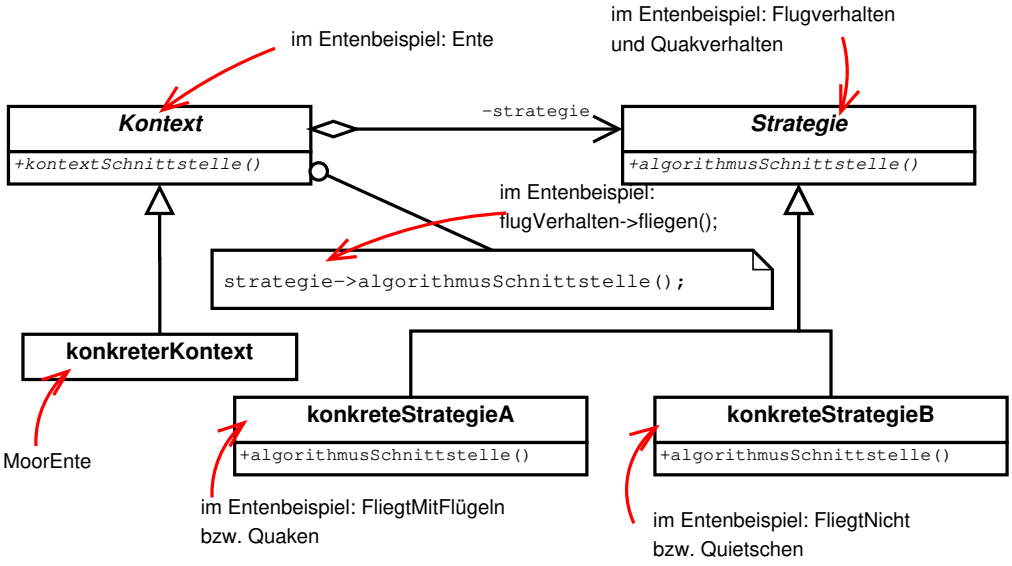
Was lernen wir daraus?

> Ziehen Sie Komposition der Vererbung vor.

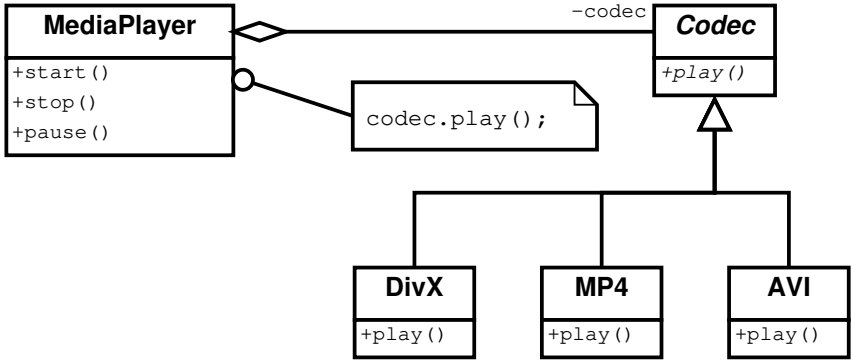
Herzlichen Glückwunsch zu Ihrem ersten Muster!

*Strategy:* Definiere eine Familie von Algorithmen, kapsele jeden einzelnen und mache sie austauschbar.

Das Strategiemuster ermöglicht es, den Algorithmus unabhängig von ihn nutzenden Klienten zu variieren.


im Entenbeispiel: Ente
im Entenbeispiel: Flugverhalten
und Quakverhalten
Kontext
+kontextSchnittstelle()
-strategie
Strategie
+algorithmusSchnittstelle()
im Entenbeispiel:
flugVerhalten->fliegen();
strategie->algorithmusSchnittstelle();
konkreterKontext
konkreteStrategieA
+algorithmusSchnittstelle()
konkreteStrategieB
+algorithmusSchnittstelle()
MoorEnte
im Entenbeispiel: FliegtMitFlügeln
bzw. Quaken
im Entenbeispiel: FliegtNicht
bzw. Quietschen

Hier eine Anwendung des Strategie-Musters für diejenigen, denen das Entenbeispiel nicht praktisch genug ist.



*Erzeuger und Strukturen*

- Abstrakte Fabrik (Abstract Factory)
- Einzelstück (Singleton)
- Dekorierer (Decorator)
- Kompositum (Composite)

*Verhalten*

- Strategie (Strategy)
- Zustand (State)
- Beobachter (Observer)
- Model-View-Controller

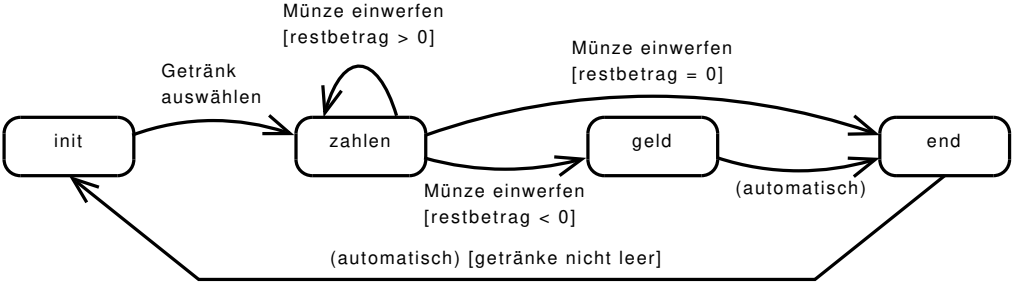
*Motivation:* Wir haben eine Methode innerhalb einer Klasse, die sich je nach Zustand anders verhalten muss.

*Beispiel:* Der Menü-Eintrag bearbeite in einem Zeichenprogramm ist abhängig vom ausgewählten Objekt.

- Punkt: `verschiebe`
- Linie: `verschiebe`, `rechts rotieren`, `Startpfeil einfügen`
- Rechteck: `verschiebe`, `rechts rotieren`, `ausfüllen`

*Beispiel:* Anzeige eines Getränkeautomaten

- Bitte Getränk auswählen (init)
- Noch zu zahlen: 62 Cent (zahlen)
- Bitte Rückgeld entnehmen (geld)
- Bitte Getränk entnehmen (end)



*Was wir nicht wollen:*

```
string anzeigen() {
    if (_status == INIT) {
        return "Bitte Getränk auswählen";
    }

    if (_status == ZAHLEN) {
        ostringstream os;
        os << "Noch zu zahlen: " << _restbetrag;
        return os.str();
    }

    if (_status == GELD)
        return "Bitte Rückgeld entnehmen";

    return "Bitte Getränk entnehmen";
}
```

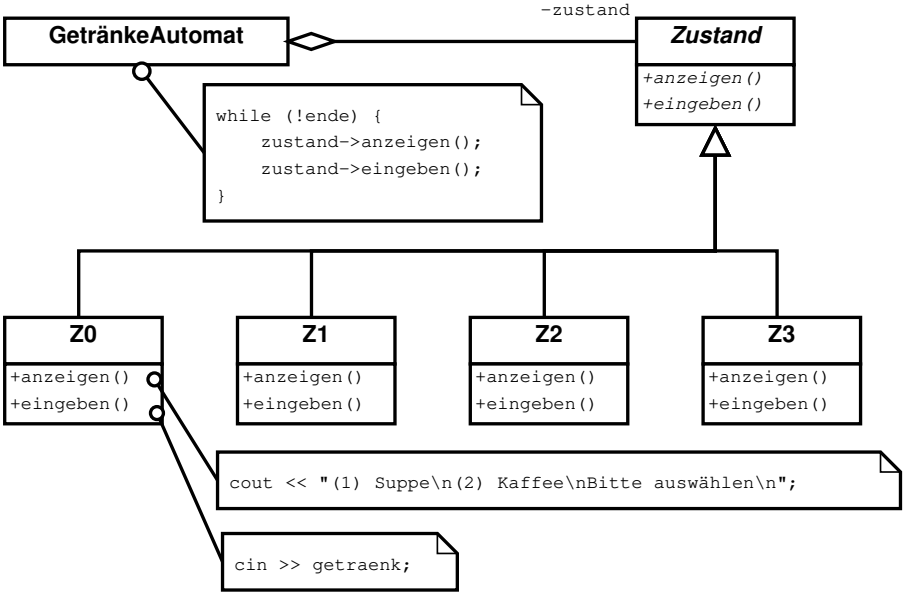
```
void eingeben() {
    if (_status == INIT) {
        cin >> auswahl;
        if (auswahl == 0)
            exit(0);
        _restbetrag = holePreis(auswahl);
        _status = ZAHLEN;
    } else if (_status == ZAHLEN) {
        cin >> einwurf;
        if (muenzeOk(einwurf))
            _restbetrag -= einwurf;
            if (_restbetrag == 0)
                _status = END;
            if (_restbetrag < 0)
                _status = GELD;
    } else if (_status == GELD) {
    .....
}
```

*Warum wollen wir das nicht?*

- Unübersichtlich, falls viele Zustände vorhanden.
- Zustandsübergänge sind nicht explizit; sie sind irgendwo in einem Haufen von Bedingungsanweisungen begraben.
- Wir haben nicht das gekapselt, was variiert!
- Wenn Code hinzugefügt wird, verursacht das mit großer Wahrscheinlichkeit Fehler.

*Idee:*

- Einführen einer abstrakten Klasse `Zustand`.
- Jede konkrete Unterklasse implementiert die Methoden `anzeigen` und `eingeben`.

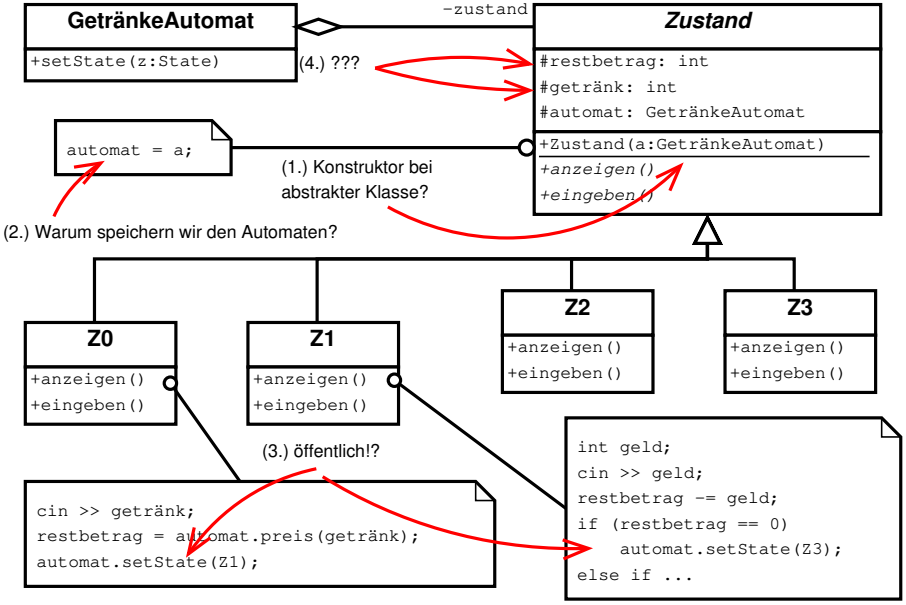

GetränkeAutomat
-zustand
Zustand
+anzeigen()
+eingeben()
while (!ende) {
zustand->anzeigen();
zustand->eingeben();
}
Z0
+anzeigen()
+eingeben()
Z1
+anzeigen()
+eingeben()
Z2
+anzeigen()
+eingeben()
Z3
+anzeigen()
+eingeben()
cout << "(1) Suppe\n(2) Kaffee\nBitte auswählen\n";
cin >> getraenk;

*Problem:* Wie wird der Zustandswechsel durchgeführt?

→ Jeder Zustand kennt seine Nachfolgezustände!

Fragen zur nächsten Folie:

- Kann eine abstrakte Klasse einen Konstruktor haben?
- Warum merken wir uns den Getränkeautomaten in dem Attribut `automat`?
- Durch die öffentliche Methode `setState` der Klasse `Automat` kann jeder von außen den inneren Ablauf des Automaten stören! Wie können wir das vermeiden?
- Wie können die Werte der Attribute `_restbetrag` und `_getraenk` beim Zustandswechsel erhalten bleiben?


GetränkeAutomat
+setState(z:State)
-zustand
Zustand
#restbetrag: int
#getränk: int
#automat: GetränkeAutomat
+Zustand(a:GetränkeAutomat)
+anzeigen()
+eingeben()
(4.) ???
automat = a;
(1.) Konstruktor bei abstrakter Klasse?
(2.) Warum speichern wir den Automaten?
Z0
+anzeigen()
+eingeben()
Z1
+anzeigen()
+eingeben()
Z2
+anzeigen()
+eingeben()
Z3
+anzeigen()
+eingeben()
(3.) öffentlich!?
cin >> getränk;
restbetrag = automat.preis(getränk);
automat.setState(Z1);
int geld;
cin >> geld;
restbetrag -= geld;
if (restbetrag == 0)
automat.setState(Z3);
else if ...

- Kann eine abstrakte Klasse einen Konstruktor haben?

→ Ja! Nur weil ein Konstruktor definiert ist, heißt das noch lange nicht, das auch ein Objekt erzeugt werden kann.

Der Konstruktur initialisiert nur die Attribute.

- Warum merken wir uns den Getränkeautomaten in dem Attribut `automat`?

→ Damit wir z.B. die Preise der Getränke erfragen können, denn dafür ist der Automat veranwortlich.

- Wie können wir eine öffentliche Methode `setState` in der Klasse `Automat` vermeiden?

→ Wir definieren `setState` als `private` und erklären die Klasse `Zustand` zum Freund der Klasse `Automat`.

Funktioniert das?

Nicht ohne weiteres!

Da Freunde nicht vererbt werden, können die Sub-Klassen `Z0`, `Z1` usw. die Methode `setState` nicht aufrufen. Es sei denn, wir definieren auch sie als Freunde von `Automat`.

Damit wir nicht alle Unterzustände als Freunde von `Automat` definieren müssen, definieren wir eine Methode

```
void Zustand::setState(Zustand *z);
```

in der Basisklasse `Zustand`. Die Methode `setState` der Basisklasse nimmt den Zustandswechsel beim Automaten vor.

Da alle abgeleiteten Zustände diese Methode erben, kann auf diesem Umweg ein Zustandswechsel durchgeführt werden.

Wie können die Werte der Attribute `_restbetrag` und `_getraenk` beim Zustandswechsel erhalten bleiben?

*Idee:* Wir setzen das Singleton-Entwurfsmuster ein. Dazu müssen wir aber Attribute aus der Klasse entfernen:

- Die Attribute `_restbetrag` und `_getraenk` können in die Klasse `Automat` ausgelagert werden.
- Das Attribut `automat` können wir nicht auslagern, daher übergeben wir diese Information an die Methoden `anzeigen` und `eingeben`.

```
class Zustand {     // Freund der Klasse Automat
protected:
    Zustand() = default;

    void setState(Automat *a, Zustand *z) {
        a->setState(z);
    }


public:
    virtual ~Zustand() = default;
    virtual void anzeigen(Automat *) = 0;
    virtual void eingeben(Automat *) = 0;

    .....
```

```
    int holeRestbetrag(Automat *a) {
        return a->_restgeld;
    }

    void setzeRestbetrag(Automat *a, int b) {
        a->_restgeld = b;
    }

    int holeGetraenk(Automat *a) {
        return a->_getraenk;
    }

    void setzeGetraenk(Automat *a, int g) {
        a->_getraenk = getr;
    }
};
```

```
class ZBereit: public Zustand {
    ZBereit() = default;
    static ZBereit *_exemplar;

public:
    void anzeigen(Automat *);
    void eingeben(Automat *);
    static ZBereit * exemplar();
};

// hier die Implementierung als Singleton
ZBereit * ZBereit::_exemplar = 0;

ZBereit * ZBereit::exemplar() {
    if (ZBereit::_exemplar == 0)
        ZBereit::_exemplar = new ZBereit();

    return ZBereit::_exemplar;
}
```

```
// hier die Implementierung der Methoden
void ZBereit::anzeigen(Automat *a) {
    string *auswahl = a->holeAuswahl();
    int *preise = a->holePreise();
    int n = a->holeAnzahl();

    cout << "\nZustand: Bereit\n\n";
    cout << " (0) " << auswahl[0] << endl;
    for (int i = 1; i < n; i++) {
        cout << " (" << i << ") " << auswahl[i];
        cout << "(" << preise[i] << ")" << endl;
    }
    cout << "--------------------------------\n";
    cout << "  Ihre Auswahl? ";
}
```

```
void ZBereit::eingeben(Automat *a) {
    int auswahl;

    cin >> auswahl;
    if (auswahl == 0) {
        setState(a, NULL);
    } else {
        int betrag = a->holePreis(auswahl);

        setzeGetraenk(a, auswahl);
        setzeRestbetrag(a, betrag);
        setState(a, ZBetragAnzeige::exemplar());
    }
}
```

Ist das übersichtlicher als unsere ursprüngliche `if/else`- bzw. `switch/case`-Anweisung?

Ja! Es ist übersichtlicher, weil

- die Zustandsübergänge explizit sind,
- das Verhalten jedes Zustands in seiner eigenen Klasse lokalisiert ist und
- die Klassenstruktur sich viel enger an das Zustandsdiagramm anlehnt, wodurch der Code besser lesbar und verständlich ist.

Dadurch, dass wir die Zustandsübergänge in den Zustandsklassen untergebracht haben, erzeugen wir Abhängigkeiten zwischen den Zustandsklassen.

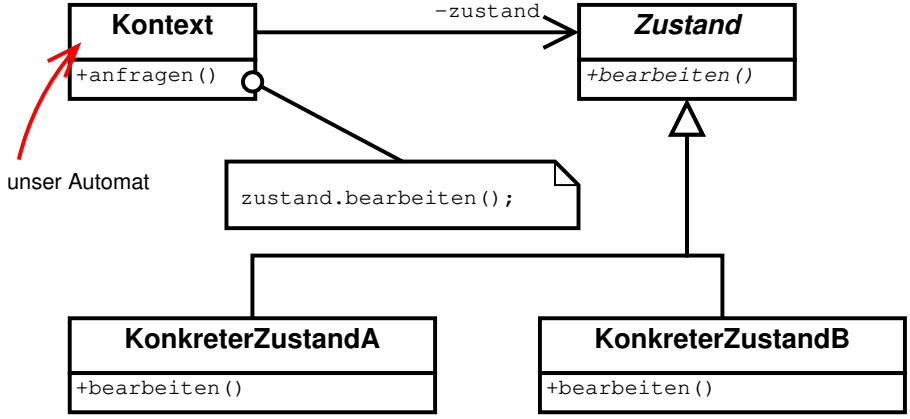
$\Rightarrow$ Kommt ein neuer Zustand hinzu, muss der bestehende Code geändert werden!

Ja, so ist das!

- Aber dadurch, dass dieser Entwurf besser lesbar und verständlich ist, sind diese Änderungen einfacher zu realisieren.
- Wir können auch den Automaten über den Fluss der Zustandsübergänge entscheiden lassen (siehe nächste Folie).

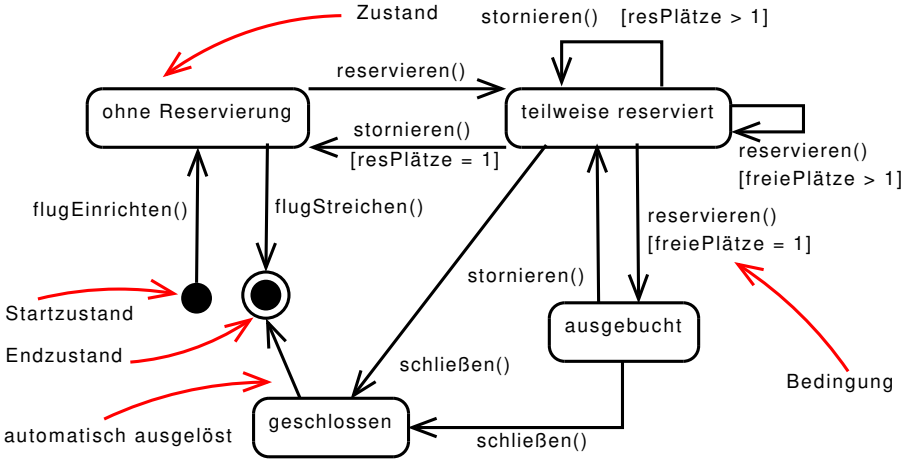
```
// Zustandswechsel durch den Automaten !!!
while (zust != NULL) {
    zust->anzeigen();
    zust->eingeben();

    string ident = typeid(*zust).name();
    if (ident == "ZBereit") {
        if (getraenk == 0)
            zust = NULL;
        else zust = ZBetragAnz::exemplar();
    } else if (ident == "ZBetragAnz") {
        if (restbetrag == 0)
            zust = ZAusgabe::exemplar();
        else if (restbetrag < 0)
            zust = ZRueckgeld::exemplar();
    } else if ...
}
```

Das State-Muster ermöglicht einem Objekt, sein Verhalten zu ändern, wenn sich sein innerer Zustand ändert.




Kommt Ihnen dieses Muster nicht sehr bekannt vor?

Ein *Zustandsdiagramm* zeigt eine Folge von Zuständen, die ein einzelnes Objekt im Laufe seines Lebens einnehmen kann und aufgrund welcher Stimuli Zustandsänderungen stattfinden.



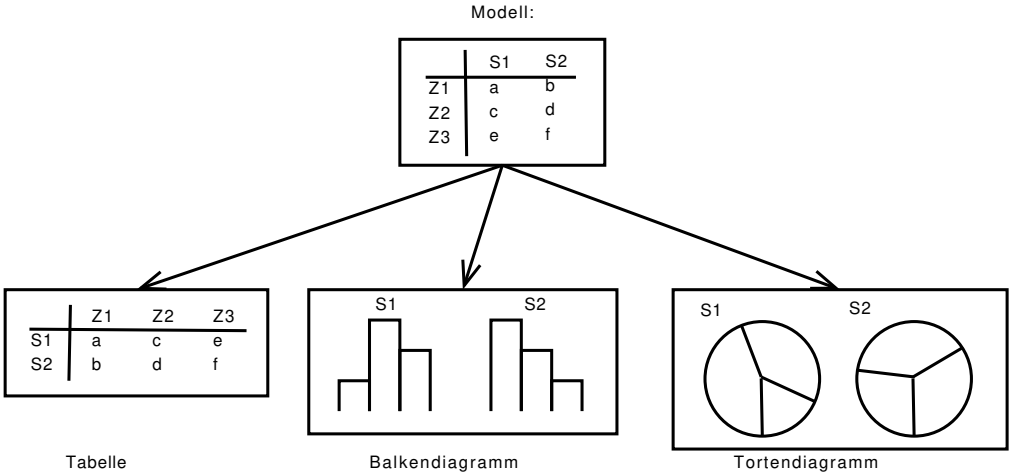
*Erzeuger und Strukturen*

- Abstrakte Fabrik (Abstract Factory)
- Einzelstück (Singleton)
- Dekorierer (Decorator)
- Kompositum (Composite)

*Verhalten*

- Strategie (Strategy)
- Zustand (State)
- Beobachter (Observer)
- Model-View-Controller

*Motivation:* Mehrere Komponenten stellen den Zustand eines einzigen Objekts dar, zum Beispiel als Tabelle, als Balken- und als Tortendiagramm.



Ändert sich der Zustand des Objekts, müssen alle Komponenten darüber informiert werden.

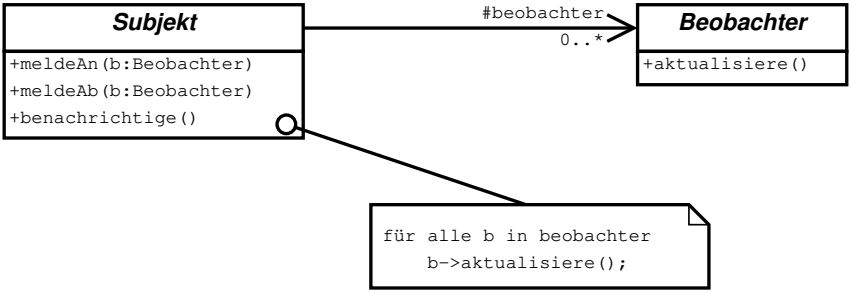
- Im Beispiel: Ändert sich das Tabellen-Modell, z.B. Spalte hinzu, Zeile löschen oder Eintrag ändern, dann müssen alle Diagramme und Views über diese Änderung informiert werden.
- Die Diagramme und Views können sich dann neu zeichnen und die Änderungen in der neuen Darstellung berücksichtigen.

Andererseits soll das Objekt von den Komponenten unabhängig sein, also die Schnittstelle nicht kennen. Warum? Damit weitere Komponenten einfach hinzugefügt werden können.

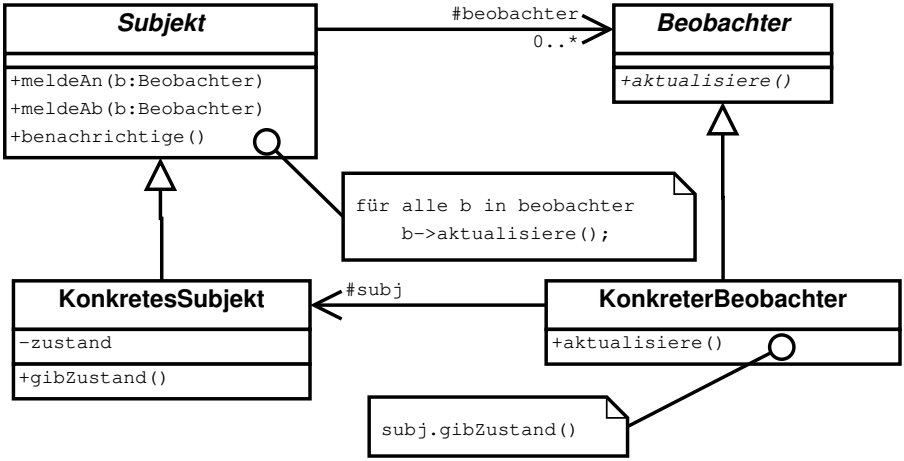
- Im Beispiel: Wir fügen einen Protokollierer hinzu, der alle Änderungen an dem Tabellen-Modell in einer Log-Datei speichert.

Das beobachtete Objekt, also das Subjekt, bietet einen Mechanismus, um Beobachter an- und abzumelden und diese über Änderungen zu informieren.

Das Subjekt kennt nur die Schnittstelle `Beobachter`.



Beobachter implementieren spezifische Methode, um auf Änderung zu reagieren.



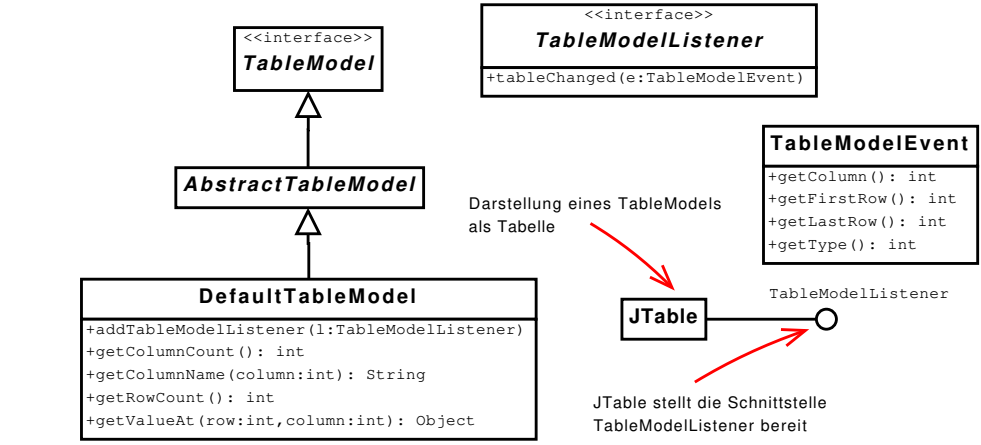

In der Regel werden die für eine Komponente relevanten Teile des Zustands abgefragt. Warum ist das schlecht?

Warum ist es schlecht, alle relevanten Teile des Zustands abfragen zu müssen?

- Die Schnittstelle von `KonkretesSubjekt` wird groß, es entsteht eine starke Kopplung zwischen Beobachter und Subjekt.
- Der Beobachter muss herausfinden, was sich geändert hat. Dies kann einige Zeit in Anspruch nehmen.

Anwendung in Java:

- `TableModel` als Subjekt
- `JTable` als Darstellungskomponente, also als Beobachter




In Java wird bei `tableChanged` eine zusätzliche Information über die Änderung mitgeteilt. Der Umfang dieser Information kann im Allgemeinen stark variieren.

- *Push-Modell:* Subjekt schickt detailierte Informationen über die Änderung mit
- *Pull-Modell:* nur minimale Informationen schicken; Beobachter fragt nach

*Erzeuger und Strukturen*

- Abstrakte Fabrik (Abstract Factory)
- Einzelstück (Singleton)
- Dekorierer (Decorator)
- Kompositum (Composite)

*Verhalten*

- Strategie (Strategy)
- Zustand (State)
- Beobachter (Observer)
- Model-View-Controller

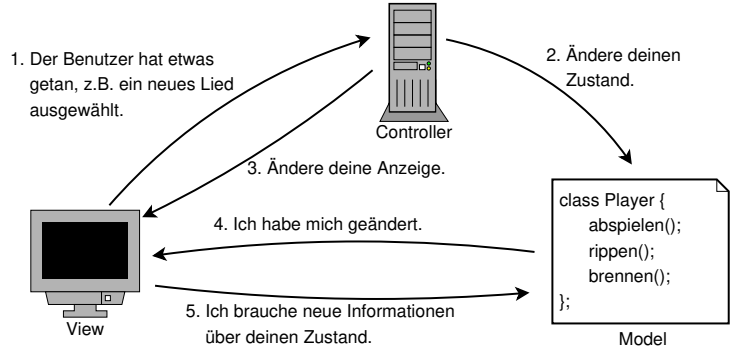
*Wie funktioniert Ihr Lieblings-MP3-Player?*

- Sie können über die Benutzeroberfläche neue Songs hinzufügen, Playlists verwalten und Titel umbenennen.
- Die Datenbank des Players enthält Ihre Songs mit Titel, Interpret, Abspieldauer, Genre und weiteren Informationen.
- Während des Abspielens der Stücke wird laufend die Benutzeroberfläche aktualisiert: Songtitel, Spieldauer usw.

Das Model-View-Controller-Entwurfsmuster ist kein neues Muster, sondern ein zusammengesetztes Muster. Die Trennung von Modell und Darstellung haben wir bereits beim *Beobachter* kennengelernt.

---

Das Beispiel ist dem Buch „Entwurfsmuster von Kopf bis Fuß“ von Eric Freeman, Elisabeth Freeman, Kathy Sierra und Bert Bates entnommen, erschienen im O'Reilly-Verlag. Ein wirklich sehr empfehlenswertes Buch.




- *View:* Präsentation des Modells. Erhält benötigte Daten i.Allg. direkt vom Modell oder fragt die Daten beim Modell ab.
- *Controller:* Stellt fest, was die Eingaben des Benutzers für das Modell bedeuten und ruft entsprechende Methoden auf.
- *Modell:* Enthält die gesamte Anwendungslogik, aber weiß nichts über View und Controller. Sein Zustand ist änder- und abrufbar. Es informiert Beobachter über Zustandsänderungen.

Der Benutzer wählt in der Anzeige ein neues Lied aus.

1. *View an Controller:* „Benutzer hat neues Lied X ausgewählt.“

   Der Controller muss nun herausfinden, was das Klicken eines Eintrags der Playlist bedeutet und welche Methoden vom Model aufzurufen sind.
2. *Controller an Model:* „Spiele neues Lied X.“
3. *Controller an View:* „Pause-Button aktivieren.“ und „Menüpunkt Foo deaktivieren.“
4. *Model an View:* „Mein Zustand hat sich geändert: Ich spiele neues Lied X.“
5. *View an Model:* „Gib mir Titel und Abspielzeit vom neuen Lied X.“

   Der View kann das Model auch dann nach dessen Zustand fragen, wenn er vom Controller aufgefordet wurde, die Ansicht zu verändern.

Das MVC-Muster setzt sich aus drei Mustern zusammen:

- *Beobachter (Observer):* Das Modell ist unabhängig von der Anzeige und der Steuerung. Für das gleiche Modell können verschiedene Anzeigen verwendet werden. Ändert sich das Modell, werden alle Anzeigen benachrichtigt, die sich dann neu zeichnen können.
- *Kompositum (Composite):* Die Darstellung besteht i.Allg. aus ineinander verschachtelten Komponenten. Jede Komponente ist entweder ein Kompositum wie ein Panel, oder es ist ein Blatt wie ein Label, Button oder TextField.

  Die Steuerung bewirkt eine Aktualisierung der Anzeige, indem die oberste Komponente informiert wird. Das Composite- Muster besorgt dann den Rest.

---

Anzeige = View; Steuerung = Controller; Modell = Model

Fortsetzung:

- *Strategie (Strategy):* Anzeige und Steuerung implementieren das Strategie-Muster. Die Anzeige kümmert sich nur um die Darstellung des Modells und delegiert die Verarbeitung der Benutzeraktionen an die Steuerung.

  Die Steuerung kümmert sich darum, dass die Eingaben des Benutzers in Aktionen auf dem Modell übersetzt werden. Die Steuerung ist also die Strategie für die Anzeige, die Steuerung ist das Objekt, das weiß, wie man mit den Aktionen des Benutzers umgeht.

  Wir können für die Anzeige ein anderes Verhalten wählen, indem wir die Steuerung austauschen.

# Entwurfsmuster: Noch einige Worte zum Abschluss

Halten Sie Ihren Entwurf so einfach wie möglich! Ihr Ziel sollte Einfachheit sein und nicht: “Wie kann ich ein Muster auf dieses Problem anwenden?“

Muster sind keine Wunderwaffe!

Sie können sie nicht einfach reinstöpseln, das Programm kompilieren und dann früh zum Mittagessen gehen. Sie müssen auch über die Folgen für den Rest Ihres Entwurfs nachdenken.

Kann es sein, dass Entwurfsmuster auch Nachteile haben?

### Wenn Sie es im Moment nicht brauchen, dann lassen Sie es!

Entwickler lieben es, schöne Architekturen zu erstellen, die man von allen Seiten verändern kann.

Wenn eine praktische Notwendigkeit dafür besteht, dass Ihr Entwurf Veränderungen unterstützt, dann nutzen Sie ein entsprechendes Muster. Ist der Grund rein hypothetisch, dann lassen Sie das Muster weg.

### Refactoring-Zeit ist Musterzeit!

Refactoring ist der Prozess, bei dem man den Code verändert, um ihn besser zu organisieren. Das Ziel ist es, seine Struktur zu verändern, nicht sein Verhalten.

Warum das alles? Um Änderungen einfacher in das Programm einbauen zu können.