TECHNISCHE UNIVERSITÄT DRESDEN

PROCESS CONTROL SYSTEMS

PROCESS SYSTEMS ENGINEERING

DRESDEN concept

Objektorientierter Entwurf (OOD) … mit UML 2.5 am Beispiel des Ampelautomaten

Vorlesung Mikrorechentechnik 2

Sommersemester 2020

Timetable

Einstieg in C++ an einem Beispiel: Rechnen mit komplexen Zahlen

Object Oriented Analysis: Anforderungsdefinition und Entwurf eines Ampelsystems

Object Oriented Design: Entwurf mit UML 2.5 Zustands- und Klassendiagrammen

Object Oriented Design: Code-Reusability und Automaten-Bibliothek, Automaten als Zustandsbeschreibung formaler Sprachen und Grammatiken

STL: Standard Template Library

STL: Standard Template Library Übung

-- Ostermontag --

IoTS-Ampelsteuerung: Umsetzung von der Faehigkeiten-Fachklasssen & E/A-Modell

IoTS-Amplesteuerung: Umsetzung von der Peripherie-Fachklasssen und E/A-Schnittstellen am RaspberryPI

-- Maifeiertag --

IoTS-Ampelsteuerung: OPC UA

-- VL findet nicht statt --

IoTS-Ampelsteuerung: 180°-Feedback

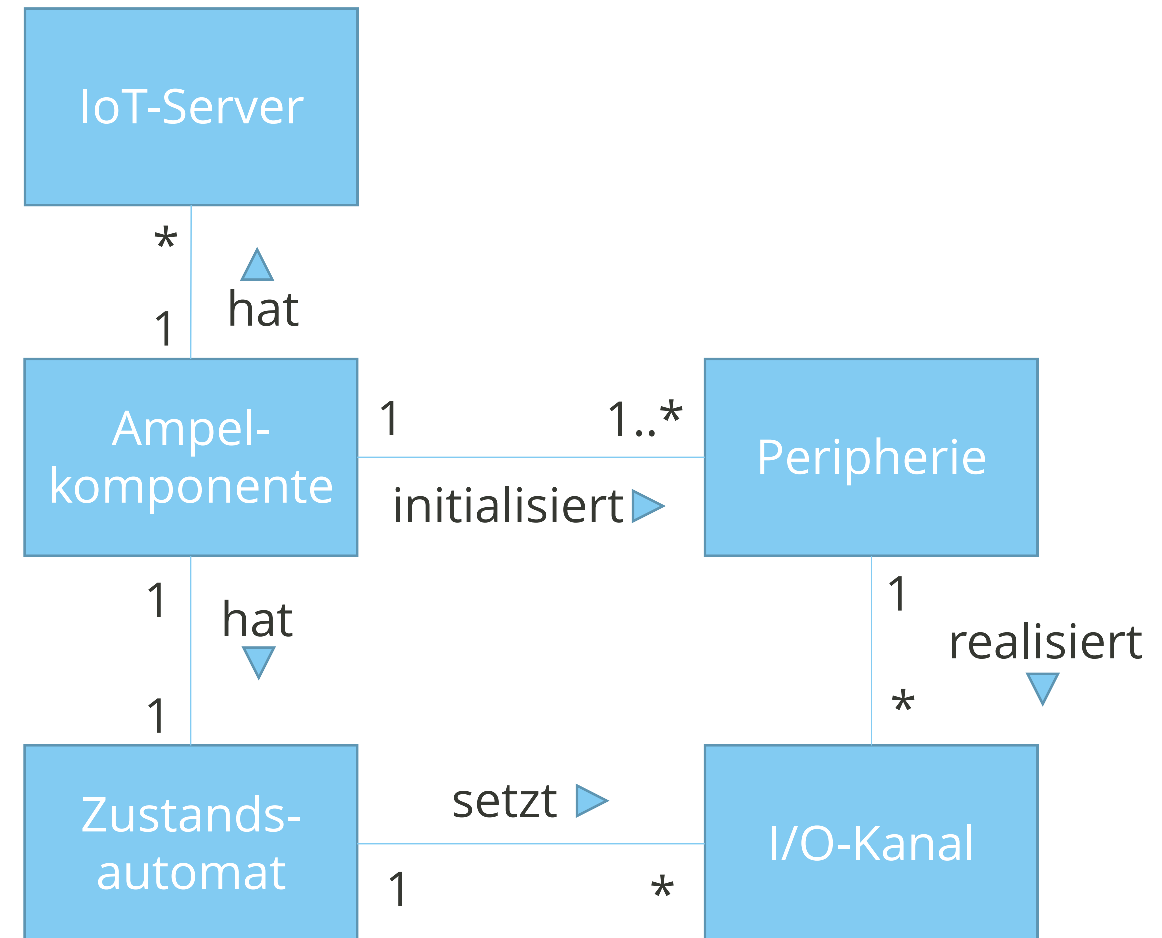
Puffer

Rückblende
Letzter Schritt der OOA: Fachklassenmodell der Ampelkomponente

Fachklassenmodell
- Keine Details und Feinheiten
- Werden während Design häufig weiter zerlegt

Assoziationen mit **Multiziplitäts**angaben
- Zulässige Mengenangaben, spiegelt Fachwissen der Interessenhalter wieder
- 1 genau 1
- 0..1 0 oder eins
- 0..4 0 bis vier
- * größer oder gleich 0 (default)
- 1..* größer oder gleich eins

Leserichtung durch Dreiecke
- Genau eine (=Jede) Ampelkomponente hat genau einen Zustandsautomat
- Jede Ampelkomponente hat beliebig viele IoT-Server
- Jede Ampelkomponente initialisiert 1 oder mehr Peripherie
- Jeder Zustandsautomat setzt beliebig viele I/O-Kanäle
- Jede Peripherie realisiert beliebig viele I/O-Kanäle



Agenda ObjektorientierterEntwurf (OOD)

OOOD mit UML 2.5:

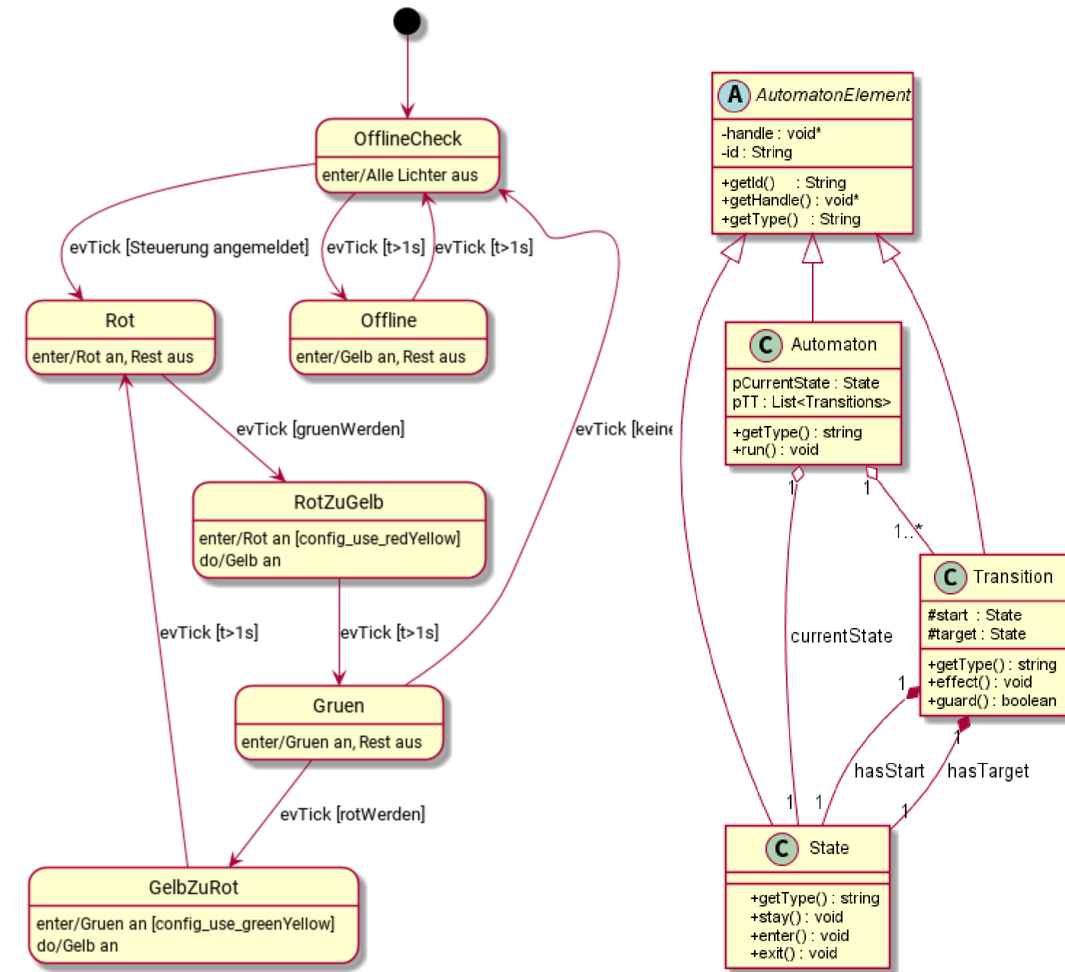
- Herleitung des Ampel-Steuerautomaten
  - UML Zustandsdiagram
- Herleitung der Steuerautomat-Klassen
  - UML Klassendiagram

C++ Konzepte

- Grundlagen Klassen, Sichbarkeit, Vererbung
- Vertiefung const
- Freundkonzept
- Spezialisierung von Klassen
- virtuelle Methoden und abstrakte Klassen

Entwurfsmuster

- Strategiemuster



Unified Modelling Language Version 2.5

(nach Kecher, Salvanos, Hoffmann-Elbern 2018)

Unified Modeling Language

Unified Modeling Language definiert eine allgemein verwendbare **Modellierungssprache** (auch Notation).

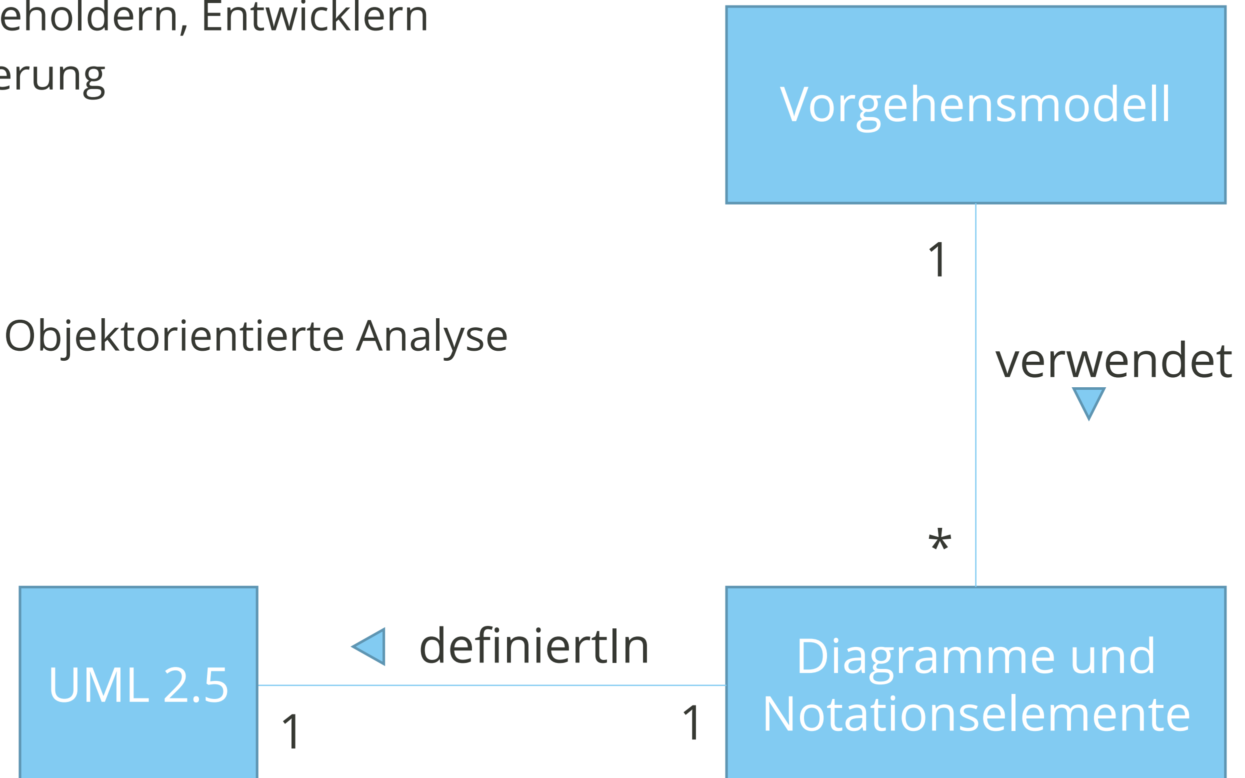
→ Anwendungsbereich ist nicht auf Softwareentwicklung beschränkt

- Eindeutig: Präzise Semantik der Notationselemente
- Verständlich: Einfach gehaltene Visualisierung des modellierten Systems
- Ausdrucksstark: Beschreibung nahezu aller Kompositions- und Laufzeitaspekte des Systems
- Standardisierung & Akzeptanz: Offener Standard der Object Management Group
- Plattform- und Sprachunabhängig: Objektorientiert, Prozedural
- Unabhängig von Vorgehensmodellen: UML ist ein Werkzeug zur Modellierung

Warum modellieren wir Software?

Modellierung (von Software):
- Kommunikation zwischen Planern, Stakeholdern, Entwicklern
- Grundlage der schrittweisen Konkretisierung
- Dokumentation

Was Sie schon kennen:
- Strukturiertes Vorgehensmodell für die Objektorientierte Analyse
- UML Klassendiagramm



Übersicht Unified Modeling Language

UML stellt **Diagramme** und **Notationselemente zur Modellierung** von statischen und dynamische Aspekte beliebiger Anwendungsgebiete zur Verfügung.

https://www.oose.de/wp-content/uploads/2012/05/UML-Notations%C3%BCbersicht-2.5.pdf

Strukturdiagramme

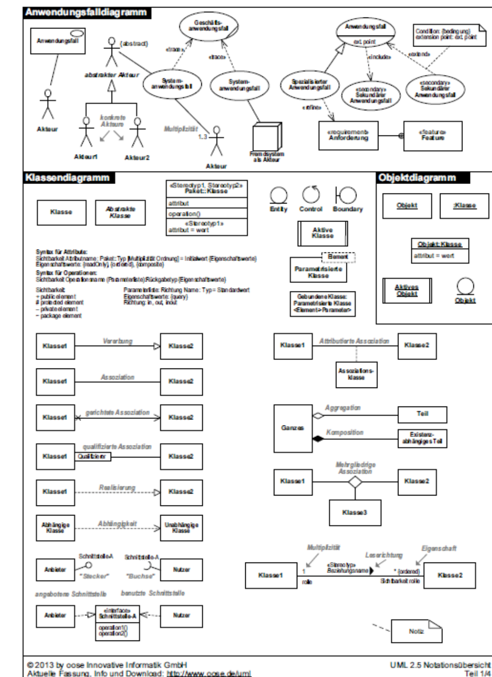
- Klassendiagramme
- Objektdiagramme
- Kompositionsstrukturdiagramme
- Komponentendiagramme
- Verteilungsdiagramme
- Paketdiagramme

Verhaltensdiagramme

- Anwendungsfalldiagramme
- Aktivitätsdiagramme

Interaktionsdiagramme

- **Zustandsdiagramme**
- Sequenzdiagramme
- Kommunikationsdiagramme
- Timing-Diagramme
- Interaktionsübersichtsdiagramme
- Profildiagramme



4+1-Sichtenmodell
(Kruchten 1995, c.f. Miles, Hamilton 2006)

PCS PSE

Logische Sicht

- Abstrakte Beschreibung von Bestandteilen. Modelliert woraus ein System besteht und wie diese Bestandteile interagieren.

Prozesssicht

- Beschreibt Abläufe in einem System. Visualisiert, was/wann innerhalb eines Systems wie zu erfolgen hat.

Entwicklersicht

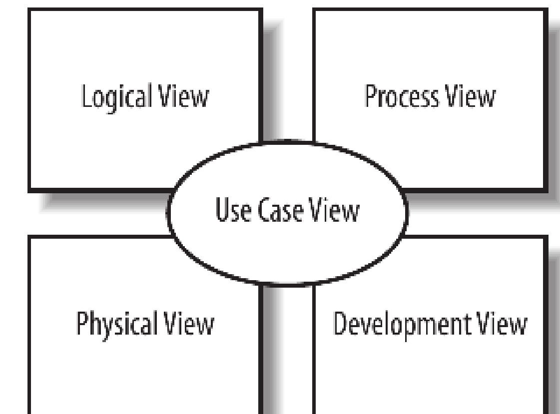
- Beschreibt, wie die Bestandteile eines Systems in Kategorien wie Module oder Komponenten organisiert werden.

Physische Sicht

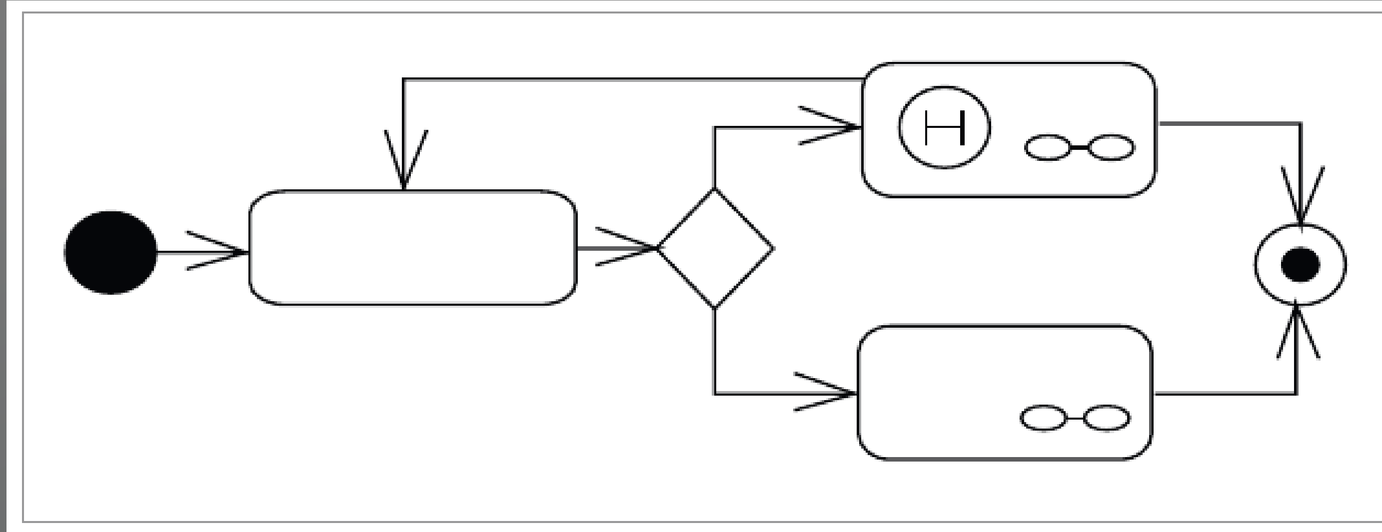
- Beschreibt, wie die in anderen Ansichten beschriebenen, abstrakten Bestandteile/Module/Komponenten eines Systems sich in der physischen Welt manifestieren.

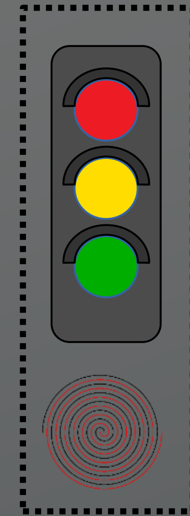
Use-case-Ansicht (das +1 in Kruchtens Sichtenmodell)

- Beschreibt die Funktionalität des modellierten Systems, wie es in der realen Welt ("von außen") wahrgenommen wird. Beschreibt, was das System macht und legt Interaktionsszenarien mit der realen Welt fest.



(Miles, Hamilton 2006)





UML 2.5 Zustandsdiagramm

Objektorientierter Entwurf Finite State Machine

Auswahl einer Steuerlogik/Steuermechanik für unsere Ampel:

- **Endliche Automaten** (engl. Finite State Machine, FSM)

Anwendung von Automaten im Studiumsverlauf

- **Vorlesung Prof. Janschek**: Steuerung diskreter Systeme
- **Hauptseminar AMR**: Einparkhilfe

Vorgehen:

- UML Zustandsdiagram als Grundlage und Modellierungswerkzeug
- Vereinfachungen für unsere Ampel
- Herleitung der Klassenstruktur für unsere Ampel

UML Zustandsdiagramm

UML 2.5 bietet mit Zustandsdiagrammen die Möglichkeit

- Zustände
- Zustandsübergänge
- Ereignisse und
- Aktionen

in einem System zu modellieren.

Zustandsdiagramme basieren auf dem Konzept **endlicher deterministischer Automaten**

Perfekt geeignet, um unsere Ampel zu beschreiben

Elemente eines Zustandsdiagramms

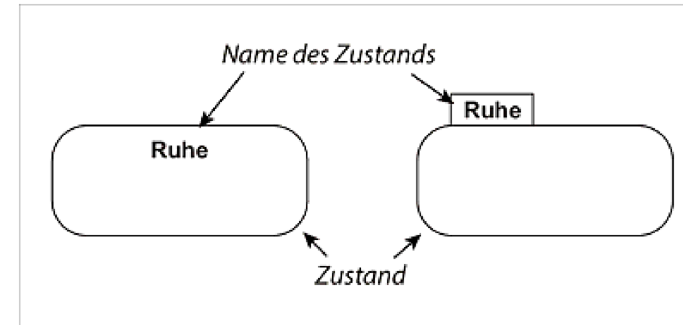
Ein **Zustand** modelliert eine Situation des Systems, in dem genau definierte Beziehungen gelten

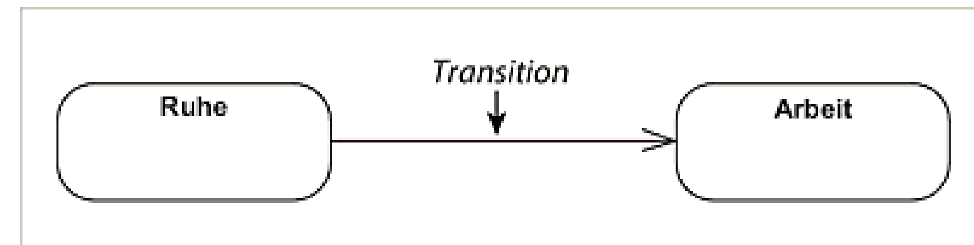
Transitionen Stellen als gerichtete Beziehungen Zustandsübergänge vom Quell- zum Zielzustand dar

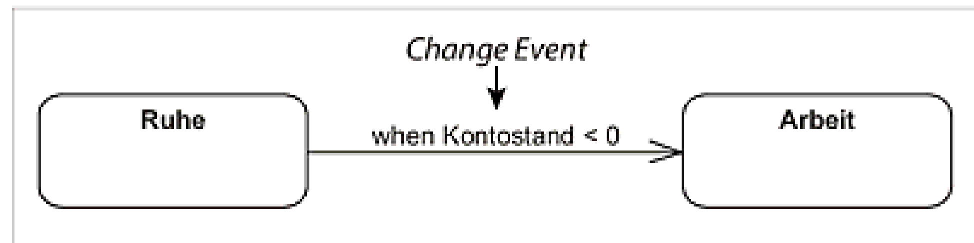
- Man spricht vom "nehmen" oder "feuern" einer Transition

Ereignisse lösen Transitionen aus

- Signal Event
- Time Event
- Change Event







(Bildquelle: Kecher, Salvanos, Hoffmann-Elbern 2018)

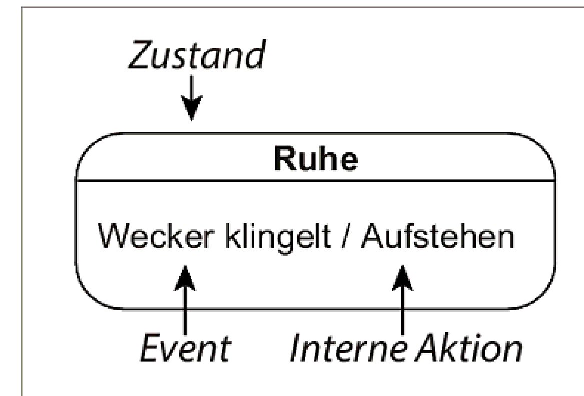
Handeln mit Effekten und Aktionen

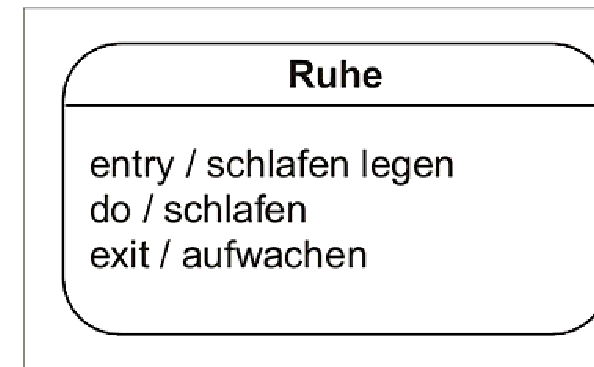
Effekte sind Handlungen, die ausgeführt werden, wenn ein System über eine Transition vom Quell- in den Zielzustand wechselt. (engl.: "**Behavior"**)

Ereignisse dürfen auch in Zuständen auftreten. Deren Effekte heißen in Zuständen **Interne Aktionen** (engl.: "**internal Behavior"**).

UML definiert die nachfolgenden **speziellen Aktionen** in jedem Zustand:

- Entry
- Do (auch als "stay" bezeichnet)
- Exit





(Bildquelle: Kecher, Salvanos, Hoffmann-Elbern 2018)

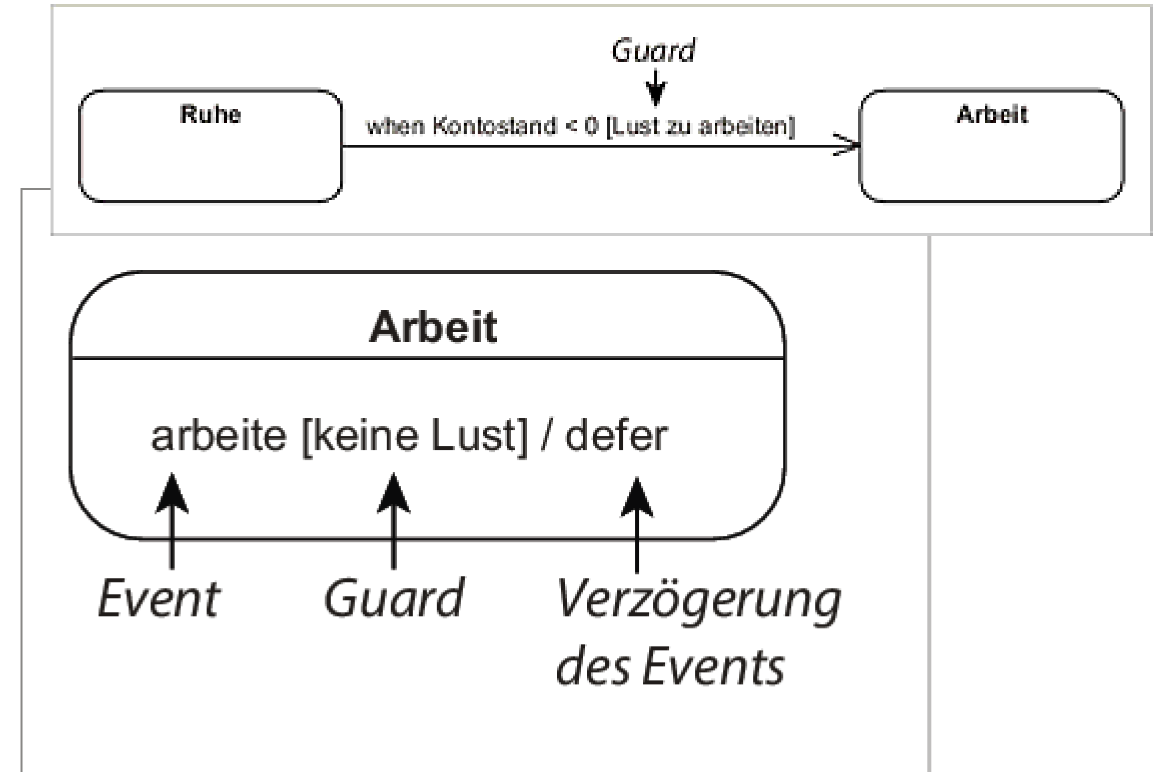
Wächter

Transitionen oder Aktionen werden durch Ereignisse "ausgelöst"

- Was passiert, wenn zusätzliche Bedingungen berücksichtigt werden müssen?

Wächter (**Guards**) sind Bedingungen, die auf **wahr oder falsch** auflösbar sind

- Guards werden evaluiert, wenn die Transition/die Aktion "feuert"
- Ist der Guard "falsch" wird das Ereignis ignoriert



(Bildquelle: Kecher, Salvanos, Hoffmann-Elbern 2018)

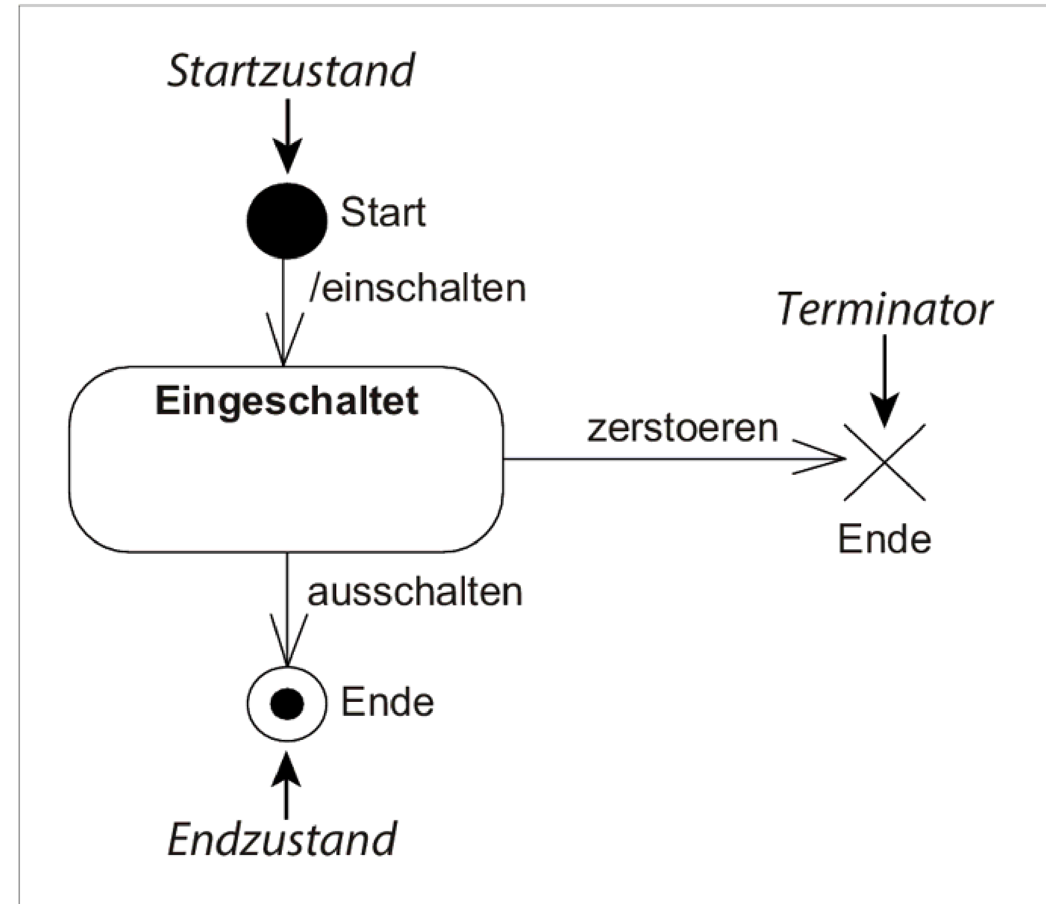
Start- und Endzustand

In einem Zustandsautomaten darf maximal ein Startzustand definiert werden.

- Wird nach der Erstellung des Objektes automatisch aktiviert und wird sofort verlassen.
- Transition aus Startzustand darf keine Guards/Ereignisse haben

Ein Endzustand kann nicht mehr verlassen werden, der Automat bleibt jedoch weiterhin erhalten und kann ggf. neu gestartet werden.

Ein Endzustand, bei dessen Erreichen der Automat zerstört wird, heißt Terminator

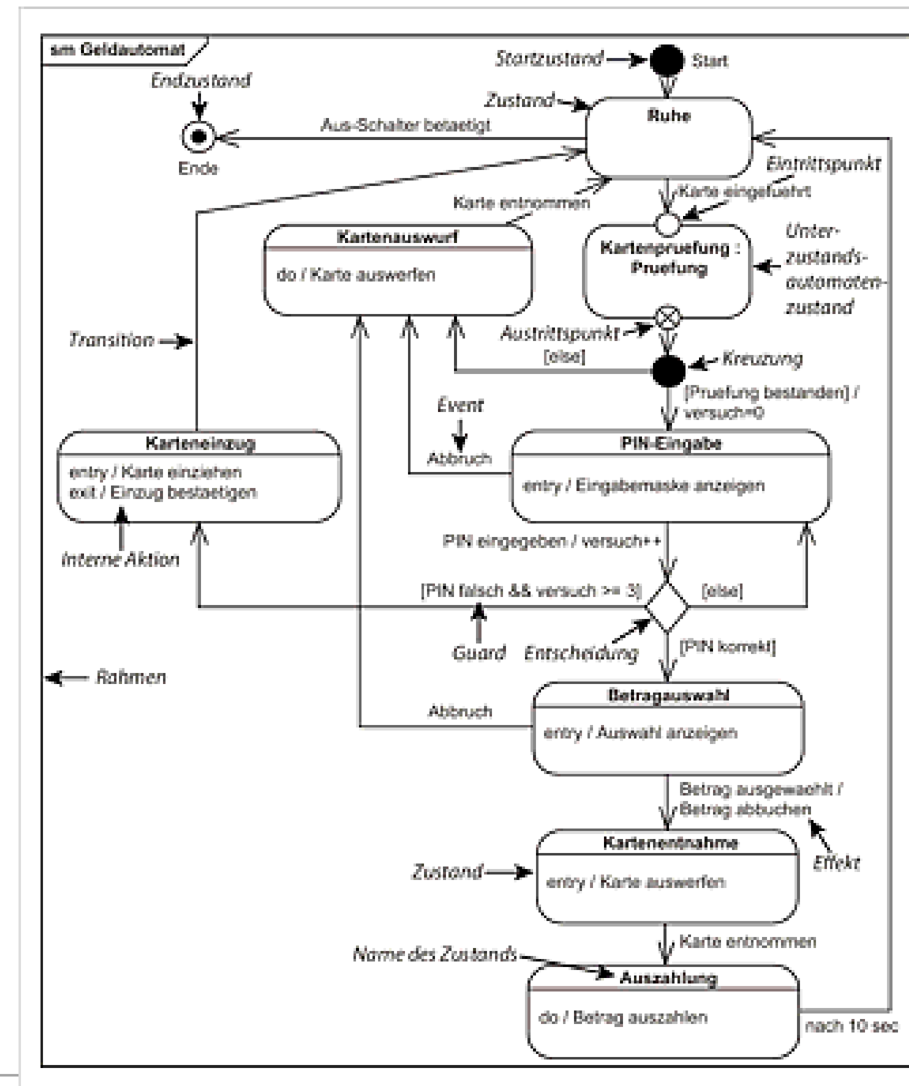


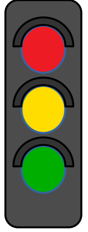
(Bildquelle: Kecher, Salvanos, Hoffmann-Elbern 2018)

Weiterführende Modellierungsaspekte

Nachfolgende Aspekte der UML Zustandsdiagramme wurden nicht näher betrachtet:

- Aktionssequenzen
- Zusammengesetzte Zustände
- Regionen, Parallele Zustandsausführung
- Rahmen, Ein- und Austrittspunkte
- Historische und Tief-historische Zustandsführung
- Signal-Sendung und Signal-Empfang
- Kreuzungen, Dynamische Verzweigung
- Generalisierung von Zuständen





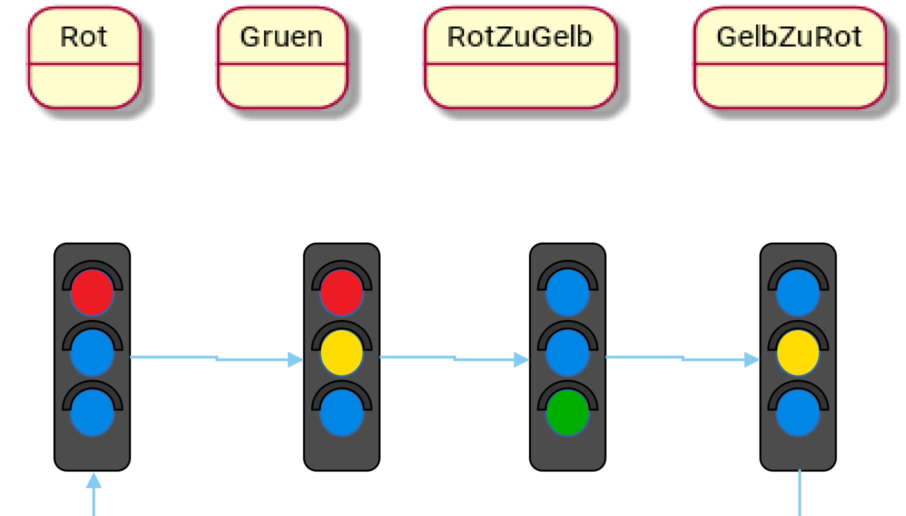
Ampel als UML Zustandsautomat

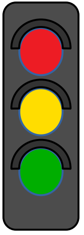
Welche Zustände der Ampel kennen wir?

- Vorschlag: Verwendung des Ampel-Ausgabeverhaltens als Zustandsbeschreibung

Genaue Abfolge der Lichter variiert nach Land und Baujahr…

- Schlüsselunterscheidung: Wird Gelb beim Wechsel RG oder GR in Verbindung mit einer anderen Farbe aktiviert
- StVO §37: “Wechsellichtzeichen haben die Farbfolge Grün - Gelb - Rot - Rot und Gelb (gleichzeitig) - Grün. Rot ist oben, Gelb in der Mitte und Grün unten.“



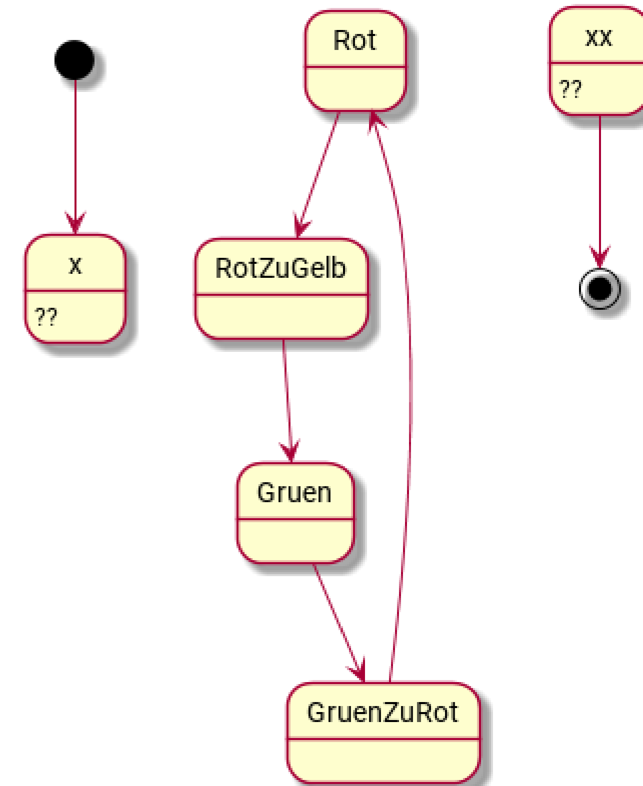


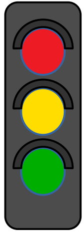
Zustandsabfolgen

Welche Zustandsabfolgen kennen wir?

Achtung:
Immer in diesem Schritt auch folgende Fragen stellen

- **Welcher Zustand ist der Startzustand?**
- **Gibt es einen odere mehrere Endzustände?**
- **Gibt es einen Terminator?**





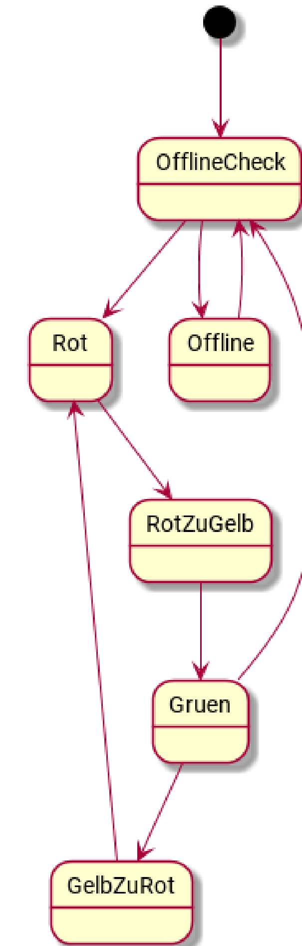
Finale Zustände

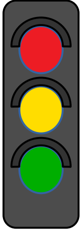
Festlegung

- Unsere Ampel braucht eine übergeordnete, koordinierende IoT-Steuerung und darf nicht selbstständig den Verkehr regeln

Weitere Anforderungen an den Zustandsautomat

- Bei Abwesenheit einer Steuerung muss die Ampel gelb blinken
- Wir brauchen einen Anmelde-Mechanismus (später)
- Alle blockierenden Zustände müssen die Anwesenheit der Steuerung prüfen





Zustandsübergänge

Wann und wie schalten die Zustände?

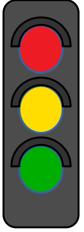
- Festlegung der Events
- Festlegung der Guards

Vorschlag zur Vereinfachung

- Nur ein zentrales „Zeitschritt"-Event (evTick), bei dem wir die Ampel aktualisieren
- Komplizierte Ereignis- und Signalkommunikation entfällt

Was enthalten die Guards?

OfflineCheck
evTick [Steuerung angemeldet]
evTick [t>1s]
evTick [t>1s]
Rot
Offline
evTick [gruenWerden]
evTick [keine Steuerung angemeldet]
RotZuGelb
evTick [t>1s]
evTick [t>1s]
Gruen
evTick [rotWerden]
GelbZuRot

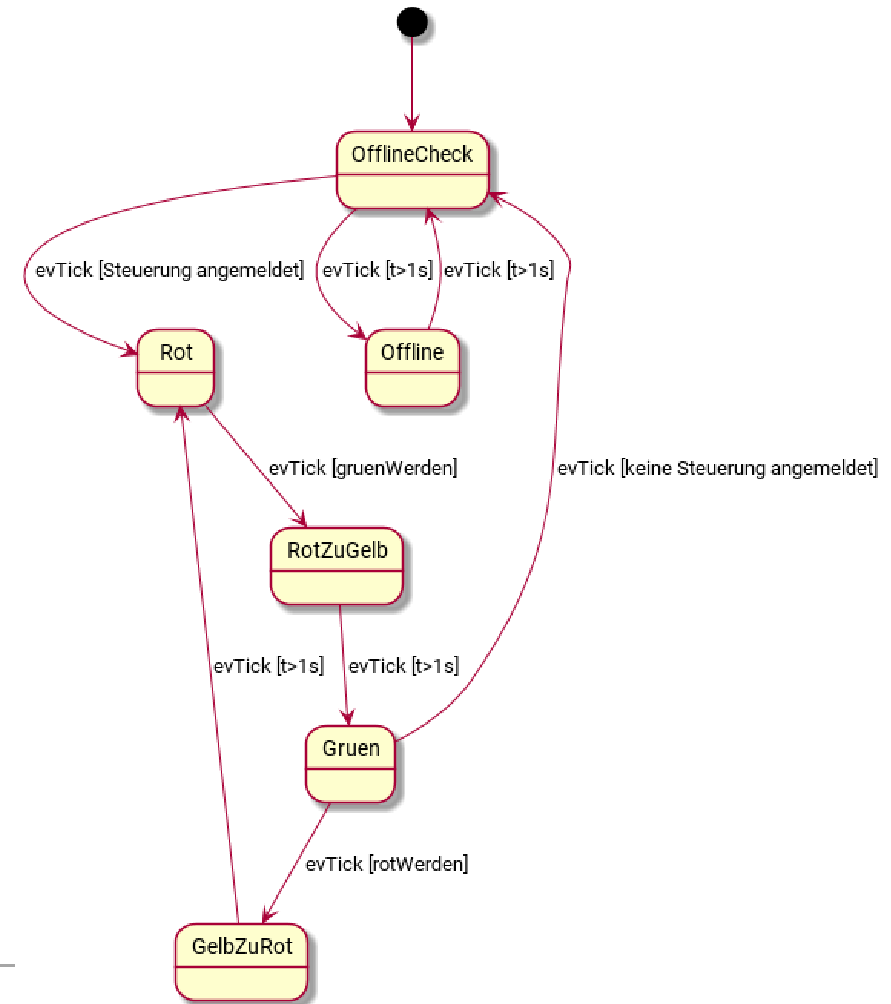


Aktionen und Effekte

Welche Aktionen und Effekte werden ausgelöst?

Grundsätzliche Herangehensweisen:

- Dauer-Aktionen in allen Zuständen (Moore-Automat)
- Einmal-Effekte an allen Transitionen (Mealy-Automat)
- Erweitern/Mischen, wo notwendig und sinnvoll (keep it simple)



Prüfen von Entwürfen

Ein Entwurf ist falsch

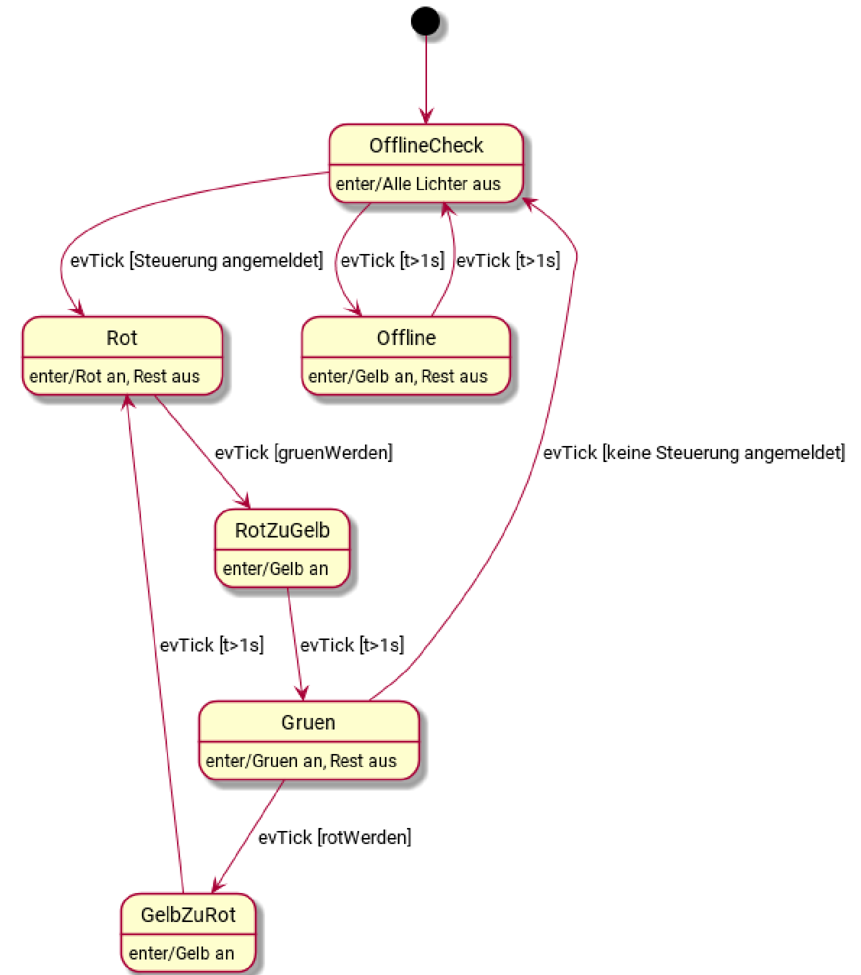
- Wenn er die Regeln der Modelierungssprachen nicht einhält. (Bsp.: zwei Startzustände bei UML)
- Wenn er nicht den Anforderungen und Inhalten der Analysephase entspricht

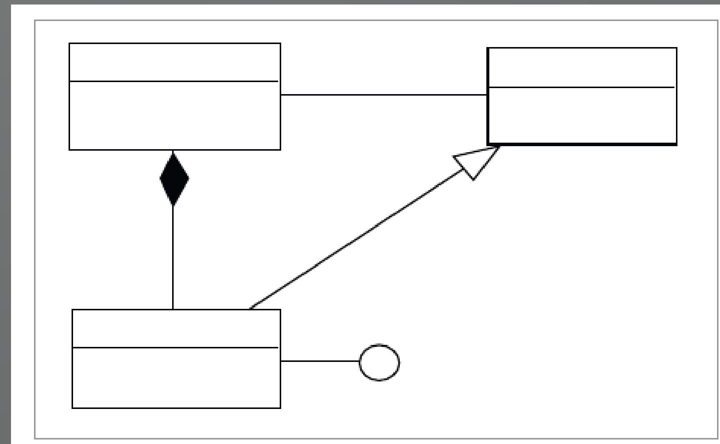
Wenn er sich anders Verhält als das Model

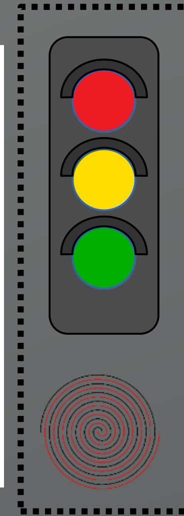
Es gibt selten eine einzige Lösung auf eine Problemstellung

Es gibt gute und schlechte Lösungen

- Wartbarkeit der Implementation
- Dokumentationsqualität der Implementation
- Verständlichkeit der Lösung
- Testbarkeit und Testabdeckung der Implementation
- Wiederverwendbarkeit der Entwurfsentscheidungen in anderen Problemen







UML 2.5 Klassendiagramme

Übersicht

Klassendiagramme modellieren (statische) Elemente eines Systems, ihre Eigenschaften und Beziehungen zueinander.

→ Dynamische Objekte (Instanzen) werden durch Objektdiagramme modelliert

Klassendiagramme in der **Analyse**

- Bestandteile des Systems aus der Sicht der Anwender und Auftraggeber
- Bewusst einfach gehalten

Klassendiagramme im **Entwurf**

- Logische Klassendiagramme zur Repräsentation interner Systemstrukturen
- Dargestellte Klassen sollen auch realisiert werden

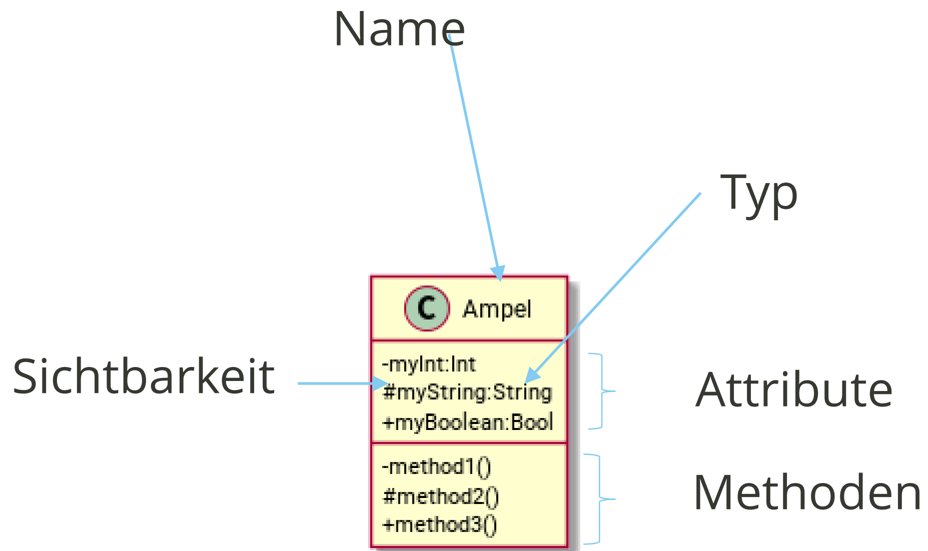
Klassendiagramme in der **Implementierung**

- Umsetzung der Entwurfs-Klassendiagramme in Programmcode

Fachklassen-Notation

Verfeinerung Und Präzisierung

Klassen, Attribute, Methoden



Entprechende Klassendeklaration in C++:

```
class Ampel {
private:
        int myInt;
        void method1();
protected:
        std::string &myString;
        void method2();
public:
        bool myBoolean;
        void method3();
}
```

Sichtbarkeit

-: private (privat):

Private Attribute sind nur in der Klasse selbst sichtbar.

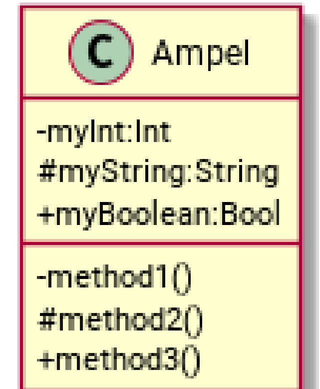
#: protected (geschützt):

Geschützte Attribute sind nur für Klassen sichtbar, die von dieser Klasse abgeleitet sind (Spezialisierung)

+: public (öffentlich):

Ein öffentliches Attribut ist für alle sichtbar.

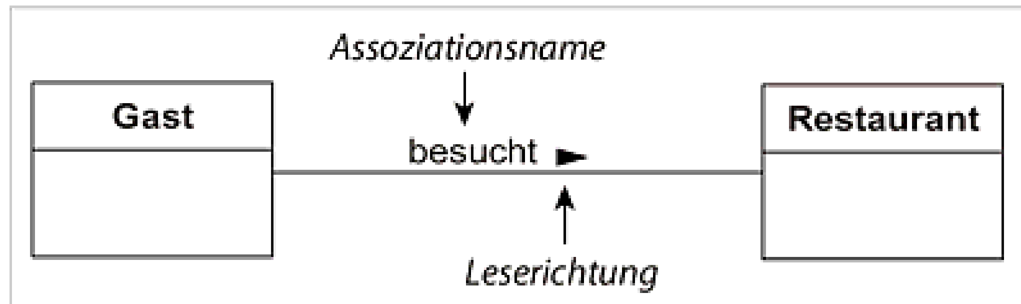
```
class Ampel
{
private:
        int myInt;
protected:
        std::string &myString;
public:
        bool myBoolean;
}
```

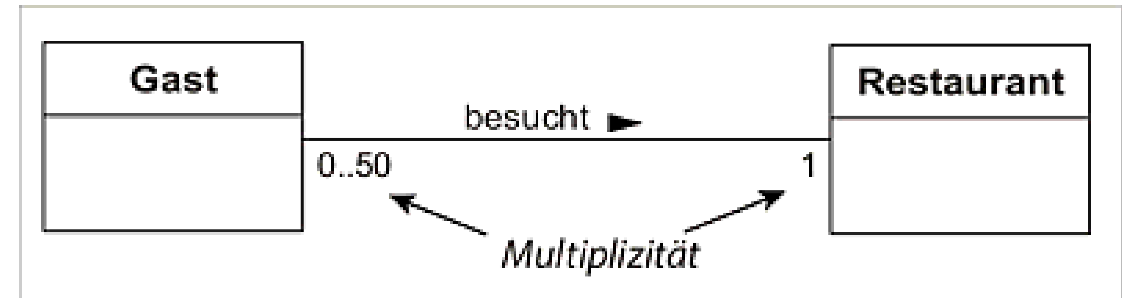


Beziehungen, Multiplizitäten, Rollen

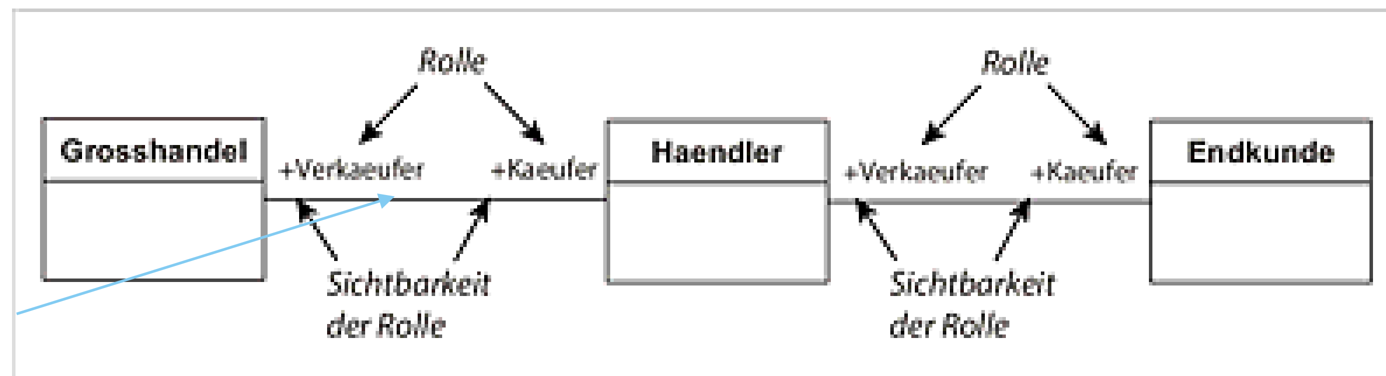
Teilweise bereits aus der Analysephase bekannt

- Wurde verwendet, um die Fachklassen und deren Zusammenhänge zu beschreiben





Rollen mit explizitem Bezug und Sichtbarkeit der Klassen-Attribute

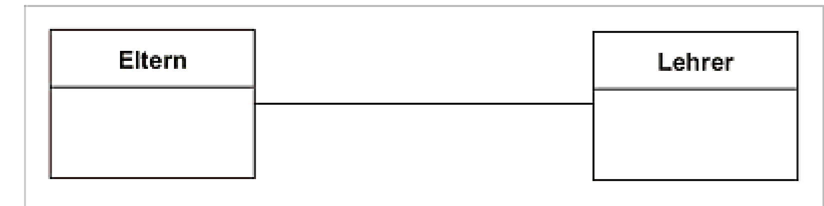


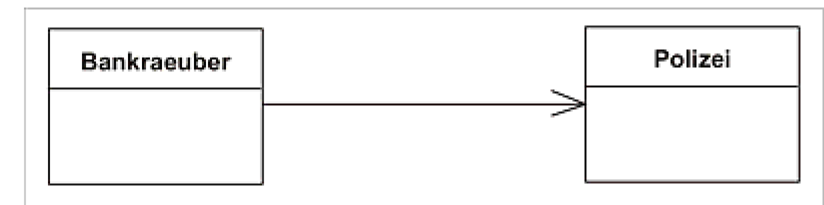
(Bildquelle: Kecher, Salvanos, Hoffmann-Elbern 2018)

Navigierbarkeit von Beziehungen

Angabe, wie sich Klassen gegenseitig “kennen” dürfen

- Unspezifiziert
- Gerichtet →
- Unidirektional →
- Bidirektional ←→
- Navigationsverbot X
  - Es besteht eine Beziehung, kennen dürfen sich die Klassen aber nicht





Wichtig, um möglichst früh Einschränkungen oder Anforderungen der Beziehung festzulegen

(Bildquelle: Kecher, Salvanos, Hoffmann-Elbern 2018)

Aggregation und Komposition

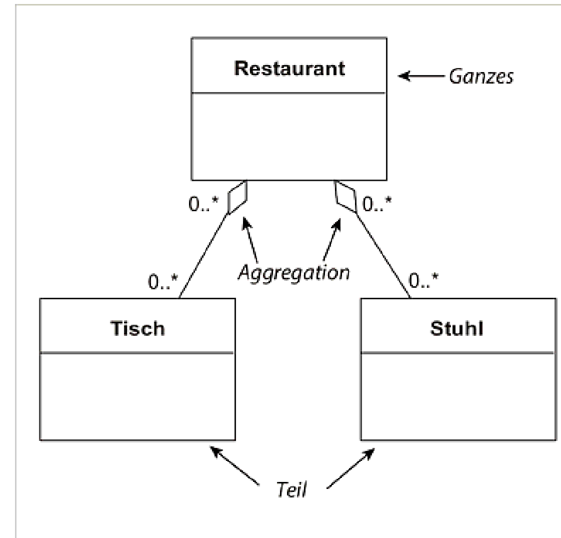
Stärkere Form der allgemeinen Beziehungen zwischen Objekten

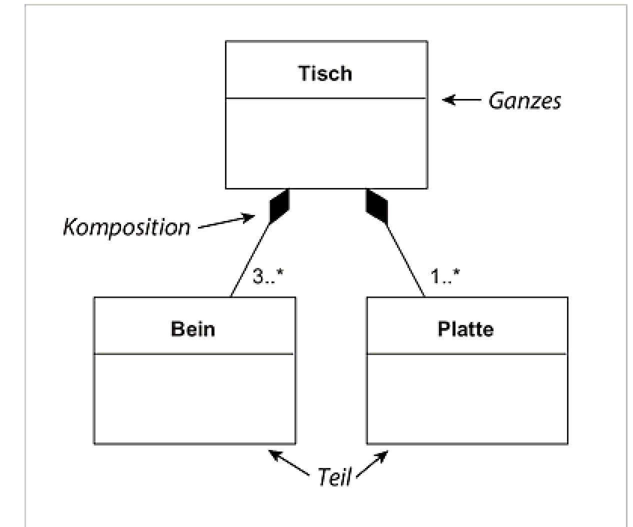
Aggregation beschreibt eine Ganzes-Teile-Beziehung.

- A kennt nicht nur B, sie bilden eine Einheit
- Aber: Eine 0..*-Multiplizität wäre möglich…

Komposition ist eine stärkere Form der Aggregation – sie beschreibt eine untrennbare Ganzes-Teile-Beziehung.

- Eine 0..*-Multiplizität ist unmöglich
- Wird ein Objekt zerstört, dann müssen auch alle komponierten Objekte zerstört werden.





(Bildquelle: Kecher, Salvanos, Hoffmann-Elbern 2018)

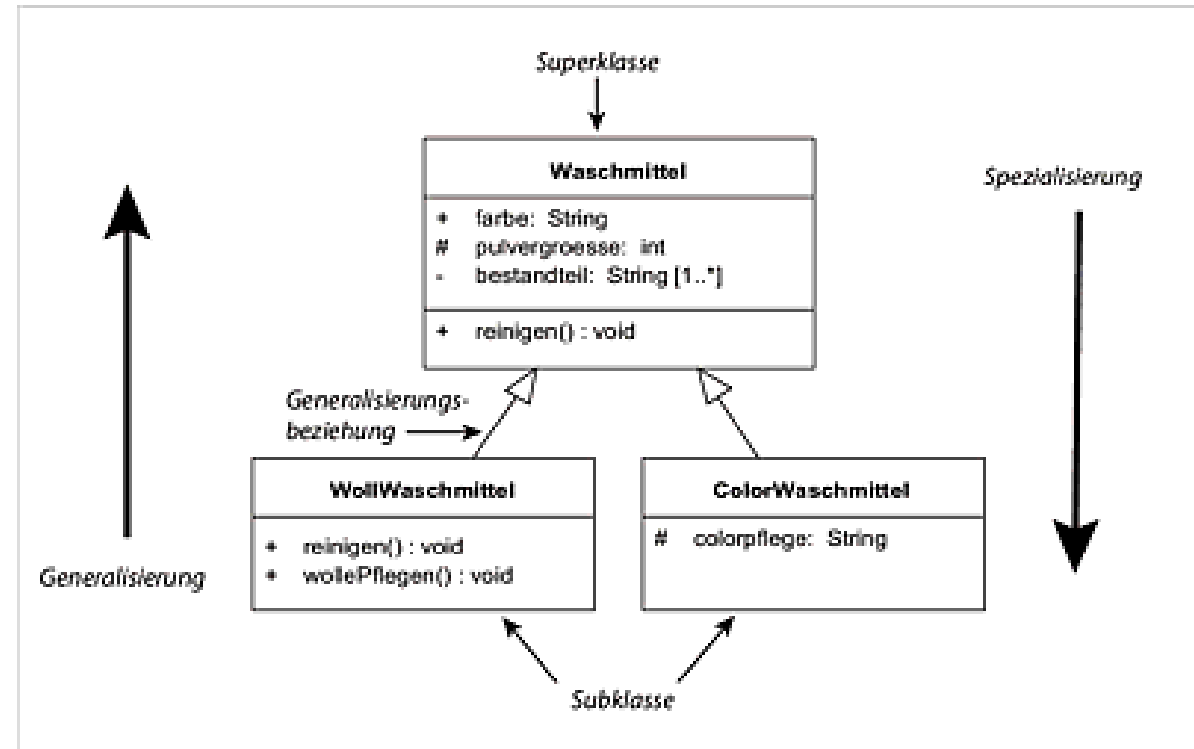
Generalisierung und Spezialisierung

Modelliert Beziehung einer Superklasse zu einer Subklasse

- Auch bekannt als "Vererbung" bei objektorientierter Programmierung

Grundidee

- Subklasse ist genau wie Elternklasse und hat alle Attribute und Methoden der Superklasse
  - Kann nur auf Protected und Public zugreifen!
- Subklasse kann
  - Methoden & Attribute hinzufügen
  - Methoden überschreiben ?

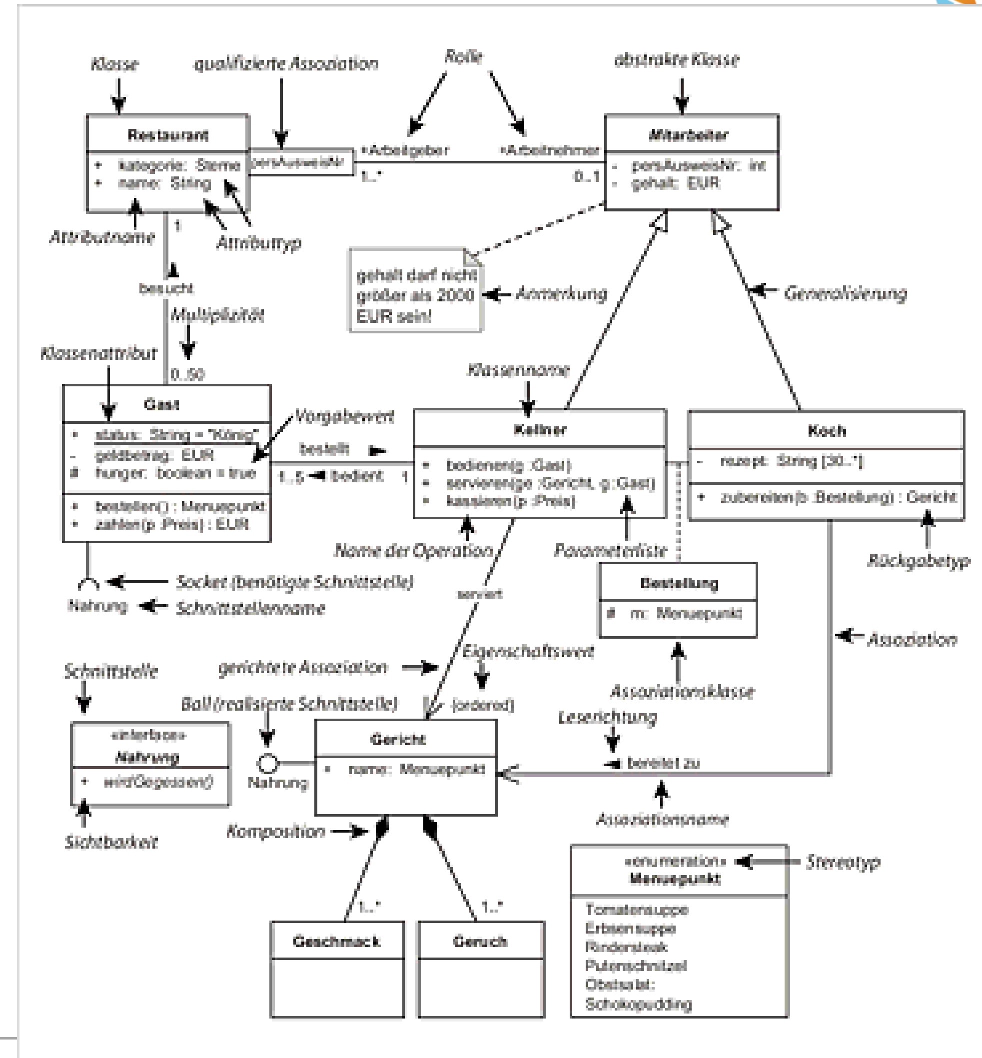


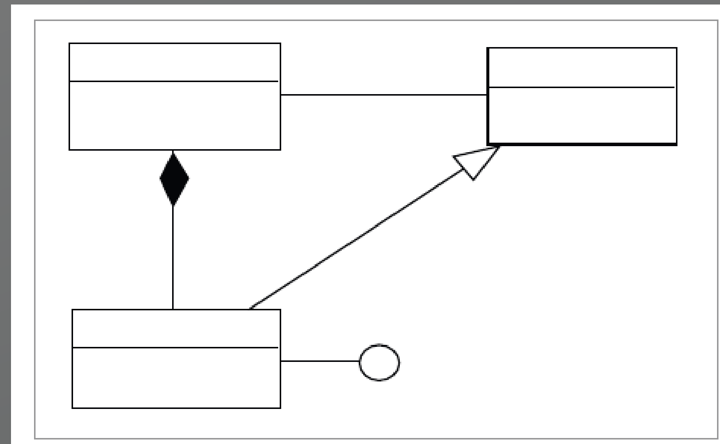
(Bildquelle: Kecher, Salvanos, Hoffmann-Elbern 2018)

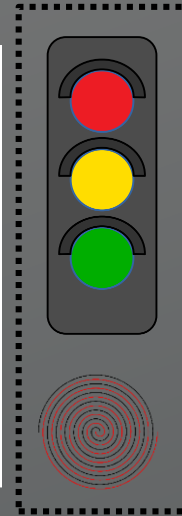
Weiterführende Aspekte

Nachfolgende Aspekte der UML Klassendiagrammen wurden nicht näher betrachtet:

- Genauer Aufbau von Methoden-Beschreibungen
  - Pro Parameter: `[Übergabemodus] Name :Typ[Multiplizität][=Vorgabewert][{Eigenschaft}]`
- Besitz,
- n-äre Assoziationen, qualifizierte Assoziation,
- Stereotypen
- Templates (kommt später)







Implementierung Zustandsautomat

Klassenstruktur Zustandsautomat

Automat

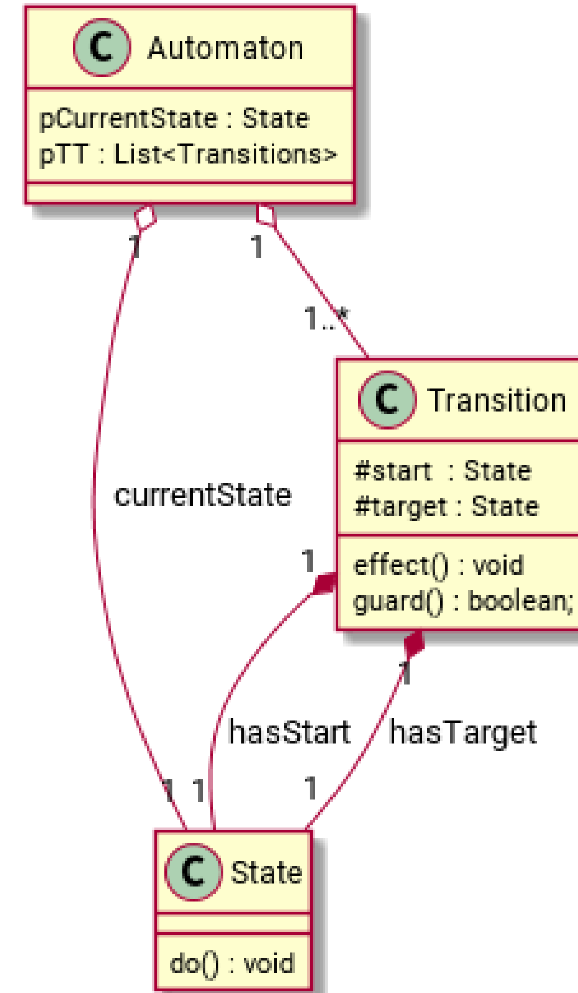
- Aktueller Zustand (= Startzustand bei t=0)
- Methode zum Einreichen von Ereignissen (evTick)
- Liste aller Transitionen
  - Methoden zum Erstellen und Löschen
- Liste aller Zustände
  - Methoden zum Erstellen und Löschen

Zustand

- Methoden für do, (enter, exit)

Transitionen

- Methoden für Wächterfunktion (guard) und Übergangsverhalten
- Zielzustand der Transition



Alle Elemente des Automaten als Spezialisierung?

PCS PSE

Alle Elemente des Automaten sollen

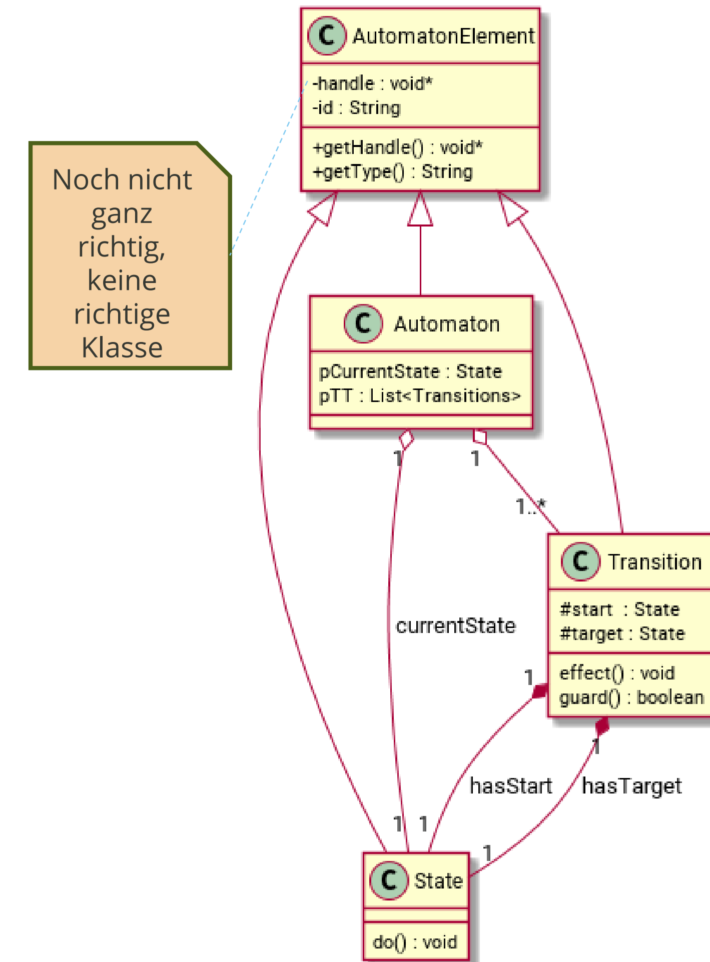
- Eine String-basierte ID („Namen“) bekommen
- Transitionen, States werden zur Interaktion mit der Peripherie unserer Ampel auf eine gemeinsame „Schnittstellen-Klasse“ zugreifen müssen

  getHandle();
- Ihren Typ zurückgeben können

  getType();

Idee:

- Alle AutomatenKlassen „erben“ Attribute und Methoden von einer Superklasse?



Spezialisierung & Überschreiben von Methoden in C++

PCS PSE

Elternklasse sei AutomatonElement

- AutomatonElement hat Methode getType()

Spezialisierung eines AutomatonElement:

- State, Transition, Automaton
- Wollen getType() überschreiben

Absicht:
Die Methode getType() der Basisklasse wird überschrieben und wir erhalten immer den Typen der Instanz zur Laufzeit ??

- Leider funktioniert das in unserem Kontext so nicht zuverlässig!

```
class AutomatonElement {
private :
  const string& id;
public:
 AutomatonElement(const string& id) : id(id) {  }
  std::string getType() { return „AutomatonElement“; }
};

class State : public AutomatonElement {
public:
 std::string getType() { return „State“; }
};

// In einem Code weit weit weg
deleteItem(AutomatonElement &ae) {
  std::string type = ae.getType();
}
```

Spezialisierung & Überschreiben: Early und Late Binding

Frühes (statisches) Binden

- Der Compiler bestimmt bereits zur Übersetzungszeit die konkret aufzurufende Methode
- Effizienter Code, Optimierung, ...

Spätes (dynamisches) Binden

- Der Compiler erzeugt Code, der zur Laufzeit den Typ des Objekts und die konkret aufzurufende Methode bestimmt

Regeln

- Frühe Bindung: Typ des Zeigers → AutomationElement::getType()
- Späte Bindung: Typ des Objects → State::getType()

Late Binding durch “Virtual”

PCS PSE

Schlüsselwort “virtual”

- Methoden werden spät (zur Laufzeit) gebunden/ermittelt

Immer dann notwendig, wenn die Generalisierung in einer Sammlung gehalten wird und auf die Methoden der Spezialisierung zugegriffen werden soll

- Zur Kompilierzeit kann nur die Adresse der generalisierten Klasse direkt bestimmt werden

Benutzt man virtual in einer Klasse ist fast immer auch richtig:

- virtueller Destruktor!

```
class AutomatonElement {
private:
  const string& id;
public:
 AutomatonElement (const string& id) : id(id) {  }
 virtual ~AutomatonElement() { }
 virtual std::string getType() { return „AutomatonElement“; }
};
class State : public AutomatonElement {
public:
 std::string getType() { return „State“; }
};
// In einem Code weit weit weg
deleteItem(AutomatonElement &ae) { std::string type = ae.getType(); }
```

Abstrakte Klassen und Überschreiben

Update unseres Klassendiagrammes

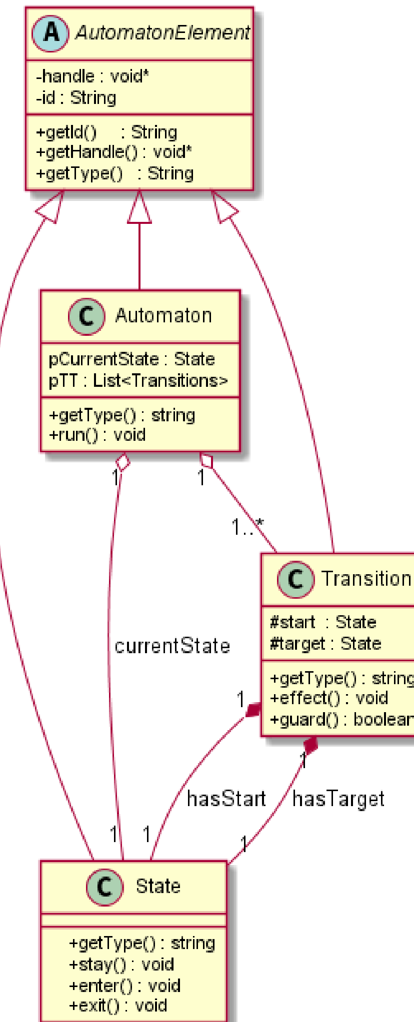
- Wir überschreiben Funktionen
  - getType()
- Diese Funktion muss in Spezialisierungen von AutomationElement sichtbar sein
  - Gleiche Sichtbarkeit wie Generalisierung!

AutomatonElement ist eine Klasse, die nicht direkt zur „Nutzung“ gedacht ist

- „Abstrakte Klasse“ (engl. abstract class)

Abstrakte Klassen, die nur „gemeinsame Methoden“ spezifizieren?

- „Schnittstellen“ (engl. interfaces)



Deklaration der Klasse State
(der Einfachheit halber ohne Vererbung von automatonElement)

```
#include <string>  // Zeichenketten-template
using namespace std;

class State {
         const string id; // Identifier
public:
         State(const string& id) : id(id) { }; // Konstruktor
         void enter() const; // Verhaltensspezifikation Eintritt
         void do() const;   // Verhaltensspezifikation Aufenthalt
         void exit() const; // Verhaltensspezifikation Ausgang
         const string& getID(); // Zugriff auf Identifier
};
```

Nicht veränderlicher Objekt string id…

Soll beim Anlegen der Klasse gesetzt werden!?

Element-Initialisierungsliste

Liste von Konstruktoren für die Elemente zwischen Kopf und Körper der Konstruktormethode

```
State::State(const string& id) : id(id) { };
```

Anweisungen der Elementinitialisierungsliste werden ausgeführt bevor das Objekt „lebt“

- Klassentypen ohne Standardkonstruktoren und const Elemente müssen in der Elementinitialisierungsliste definiert werden!

Die Reihenfolge der Elementinitialisierung wird durch die Reihenfolge der Definition der Elemente bestimmt.

- Moderne Compiler warnen, wenn Sie eine Diskrepanz zwischen Elementinitialisierungsliste und Definitionsreihenfolge finden

Kein Überdecken bei Namensgleichheit von Argument und Element

Konstanten

PCS PSE

Anweisung an den Compiler Konstrukte zu erkennen und zurückzuweisen, die potentiell zu einer (unbeabsichtigten) Änderung führen können.

Fall 1: Konstante Objektinstanzen

- das Objekt und seine Elemente werden nach Initialisierung nicht mehr verändert!

```
const string id;
```

Fall 2: Nichtmodifizierende Methoden

- Die Methode soll die Objektinstanz nicht ändern

```
void exit() const;
```

Fall 3: Referenz auf ein konstantes Objekt

- Das Objekt soll auch an anderer Stelle nicht geändert werden

```
const string& getID();
```

Unveränderliche Zeiger auf unveränderliche Objekte

Fall 4: Objektinstanz

- das Objekt dessen Adresse im Zeiger verwaltet wird darf nicht geändert werden
- const links von * bezieht sich auf Objekt (wie const double)

```
const State* pCurrentState;
State const * pCurrentState;
```

Fall 5: Adresse

- Adresse soll nicht geändert werden
- const rechts von * bezieht sich auf Adresse

```
State* const pCurrentState;
```

Fall 6: Adresse und Objektinstanz(en) unveränderlich

```
const Transition* const pTT;
```

Private, Protected & Friends

„private“:

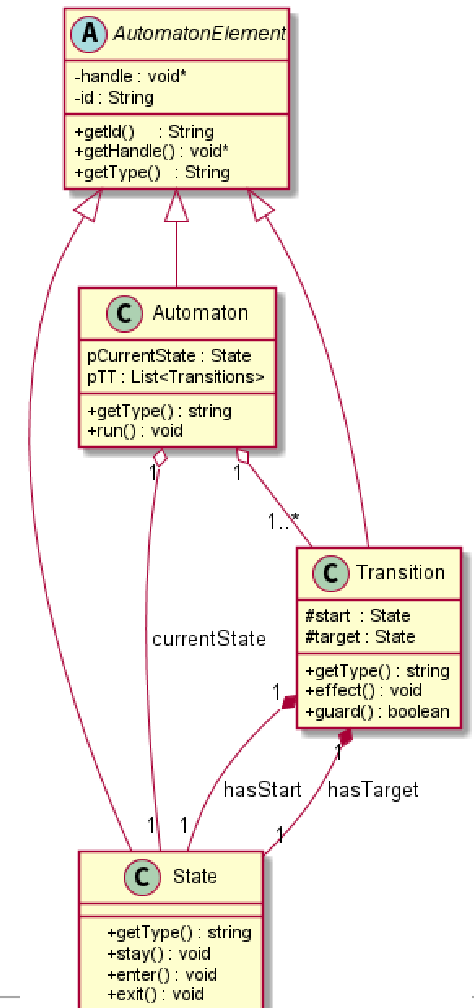
- Nur die Methoden der Klasse dürfen auf Attribut/Methode zugreifen.

„protected“:

- Nur Zugriff für Klassen unter der Vererbungshierarchie

Gelegentlich sinnvoll: Ganz bestimmte Klassen dürfen/sollen diese Regel brechen dürfen

Bsp:
Wir wollen `ostream (cout <<)` den Zugriff auf die ID der State-Klasse erlauben.

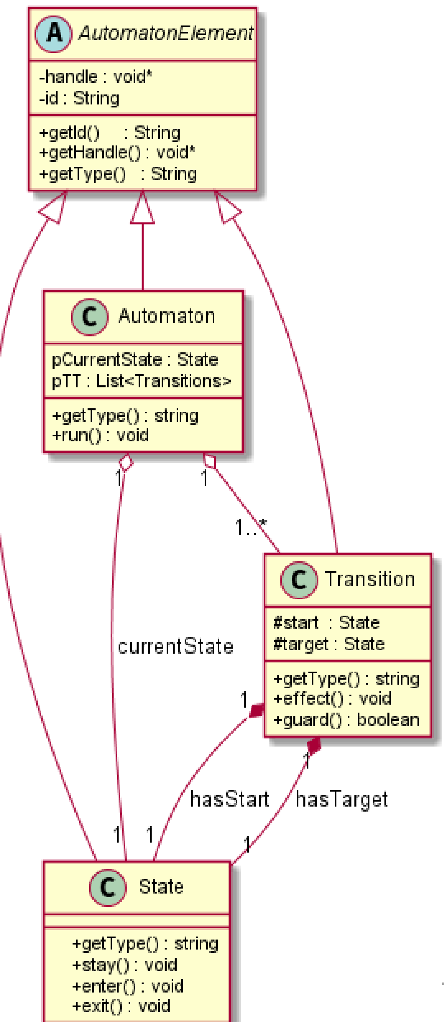


Private, Protected & Friends

Mit friend deklarierte Methoden dürfen auf private Daten zugreifen

```
class State {
        friend ostream& operator<<(ostream& lhs, State& rhs);
        friend ostream& operator<<(ostream& lhs, const State& rhs);
private:
        const string id; // Identifier
public:
        State(const string& id) : id(id) { }
[...]
};
inline ostream& operator<<(ostream& lhs, State& rhs) { return lhs << rhs.id; }
inline ostream& operator<<(ostream& lhs, const State& rhs) { return lhs << rhs.id; }
```

Keine Doppelung der Methode! Unterschiedliche Signaturen



ImplementPrivate, Protected & Friends

Ein-/Ausgabeoperatoren und andere überladene Operatoren sollen häufig auf geschützte (private, protected) Attribute zugreifen, dürfen aber nicht Elementfunktionen sein.

Der Freundmechanismus erlaubt **festgelegten Funktionen** oder **Klassen** Zugang zu den nichtöffentlichen Elementen

Jede überladene Nichtelementfunktion, die Zugang zu den Elementen haben soll, muss als Freund deklariert sein. Die Signatur **muss exakt** übereinstimmen!

```
friend ostream& operator<<(ostream& lhs, State& rhs);
friend ostream& operator<<(ostream& lhs, const State& rhs);
```

Transition

```
#include <iostream>
using namespace std;
#include "State.h"

class Transition {
public:
        const State& start; // Ausgangszustand
        const State& target; // Endzustand
        Transition(const State& start, const State& target) : start(start), target(target) { }
        bool guard() const; // Wächterfunktion
        void effect() const; // Übergangsverhalten
};
```

Deklaration des Automaten

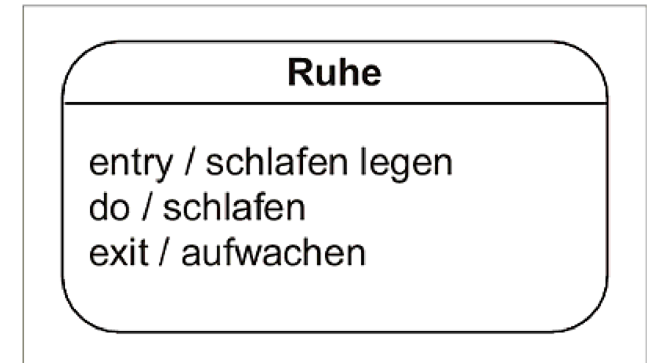
```
#include <list>
#include <iostream>
using namespace std;
#include "State.h„
#include "Transition.h"

class Automaton {

private:

  const State* pCurrentState;
  const list<Transition*> pTT;
  const int TTlength;
  int currentStateTicks;
  bool first;
public:
  Automaton(State& startState, const list<Transition*> pTT);
  void run();
};
```

const x*:
Veränderliche Adresse eines unveränderlichen Objekts

const x:
Unveränderliche Liste veränderlicher Transitionen

Definition des Automatens: Kern der Zustandsmaschine

```
void Automaton::run() {
  currentStateTicks++;
  if (first) { pCurrentState->enter(); first=false; }
  for(std::list<Transition *>::const_iterator pTTi = pTT.begin(); pTTi != pTT.end(); pTTi++) {
    Transition *t = *(pTTi);
    if (&t->start == pCurrentState) {
      if (t->guard()) {
        pCurrentState->exit();      // Verlassen Zustand
        t->effect();                // Übergang
        pCurrentState = &t->target;
        currentStateTicks = 0;
        pCurrentState->enter();     // Eintritt Zustand
        return;
      }
    }
  }
  pCurrentState->stay(); // Verharren Zustand
}
```



(Bildquelle: Kecher, Salvanos, Hoffmann-Elbern 2018)

Kern der Zustandsmaschine in Pseudo-Code

// Prüfe ausgehende Transitionen

ÜBER ALLE ausgehenden Transitionen des aktuellen Zustands

WENN Transition durch Wächter freigegeben ist DANN

- Ausgangsverhalten des aktuellen Zustands
- Übergangsverhalten der Transition
- Eingangsverhalten des Zielzustand der Transition
- aktueller Zustand = Zielzustand der Transition
- RÜCKSPRUNG

ENDE WENN

ENDE ÜBER ALLE

// Wenn keine Transition geschaltet hat

- Beharrungsverhalten des aktuellen Zustands

Vom allgemeinen Automaten zur Ampel

Wie können die Wächter, Aktionen und Effekte in Transitionen und Zuständen angepasst werden?

Lösungsansatz Spezialisierung:

- Mit Hilfe von Vererbung, virtuellen Methoden und Überschreiben derselben könnte für jeden Zustand der Ampel eine eigene Zustandsklasse angelegt werden

```
class State {
        // ..
        virtual void do() const;   // Verhaltensspezifikation Aufenthalt
        // …
};

class State_GelbZuRot : public State {
public:
        State_GelbZuRot ( string& id ) : State(id) { }
        void do() const { … }
};
class State_Rot : public State { // … }

// usw.
```

Spezialisierung

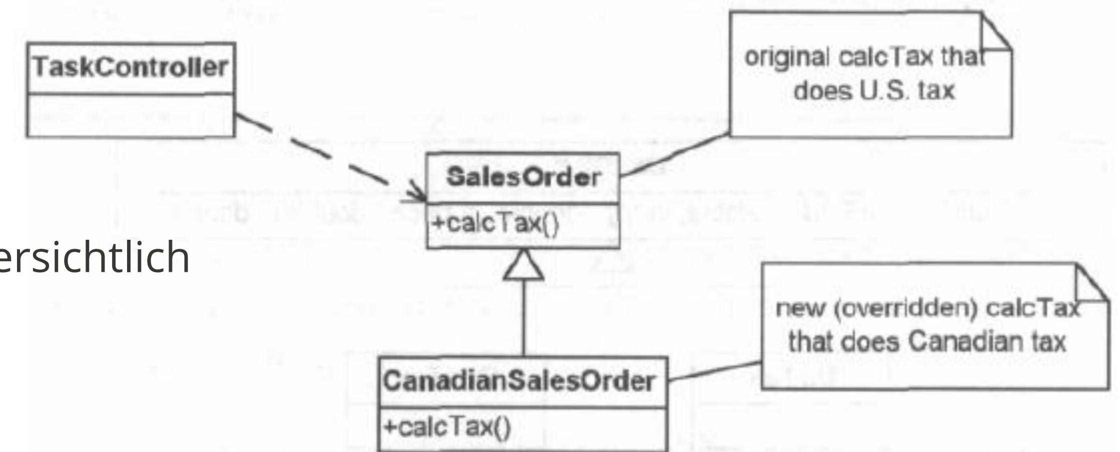
Spezialisierung wäre machbar

Vorteil:

- Functionen hätten Zugriff auf Methoden der Klassen
- Friend/Protected würde gültig bleiben

Nachteil:

- Jeder Zustand braucht eigene Klasse
- Initialisierung des Automaten ist unwartbar und unübersichtlich
  - Unelegante Lösung



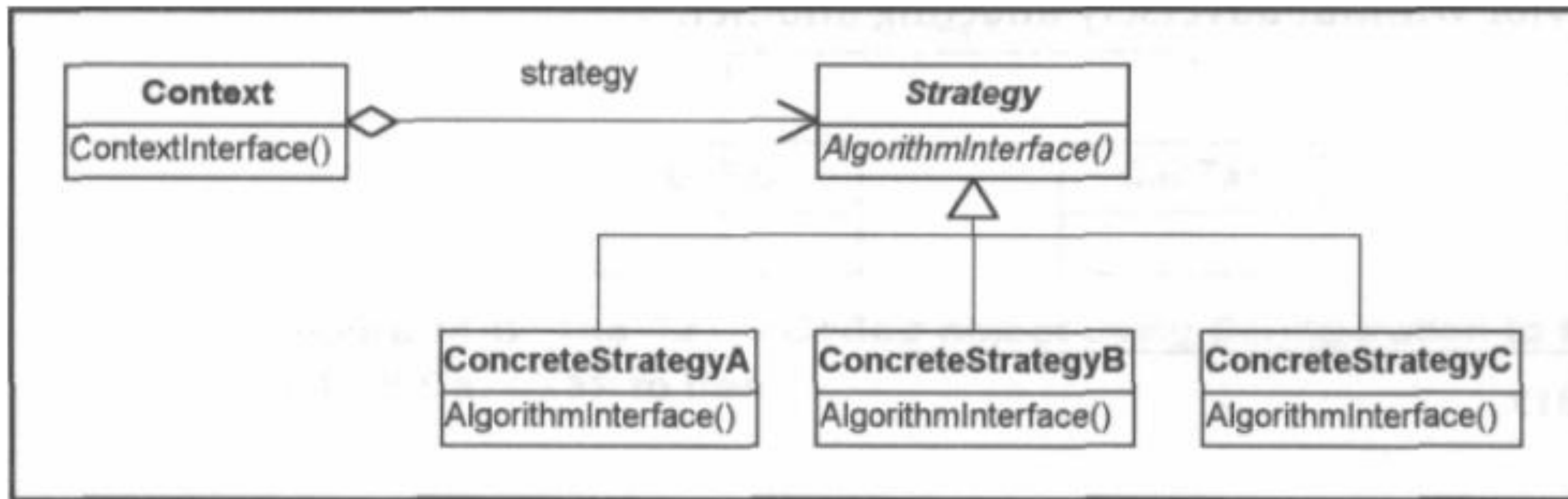
Dieses Problem ist nicht neu

(Bildquelle: Shalloway, Trott 2014)

Entwurfsmuster Strategy-Pattern

Strategy-Pattern:

- Trennen der Auswahl/Festlegung des individuellen Lösungsalgorithmus von dessen Implementierung
  - Bei uns: Die Logik der Ausführung einer Methode (enter, stay, exit) wird an die Methode einer Strategieklasse ausgelagert
- Austausch des Algorithmus unter Beibehalt des Kontextes



(Bildquelle: Shalloway, Trott 2014)

Implementierung Strategy-Pattern mit Funktionszeigern

Hier: Nutzung von Funktionszeigern (siehe MRT1)

- Das Ausführen einer Methode werden bestimmte extern definierte Funktionen ausgeführt

Vorteile:

- Kein Überladen von virtuellen Methoden notwendig
- Statisches Binden möglich
- Objekte von unterschiedlichen Typ können gleiche Strategien haben

Nachteil:

- Externe Methoden haben keinen Zugriff auf die Klasse,
- Aufweichen der Kapselung über public bzw. freund Konzepte nicht sinnvoll

Typsichere Funktionszeiger

PCS PSE

- Definition in Behavior.h

```
class AutomatonElement; // forward declaration

typedef void (*Behavior)(const AutomatonElement&, const string&);
void defaultBehavior(const AutomatonElement&, const string&);

typedef bool (*Guard)(const AutomatonElement&, const string&);
bool defaultGuard(const AutomatonElement&, const string&);
```

- Nutzung in Klasse State

```
private:
        Behavior enterFunc, stayFunc, exitFunc;
public:
        explicit State(const string& id,
                Behavior enterFunc=defaultBehavior,
                Behavior stayFunc= defaultBehavior,
                Behavior exitFunc=defaultBehavior);
        inline void enter() const { enterFunc(*this, "enter");
}
```

„kleiner" zirkulärer Bezug, deshalb forward declaration notwendig

Zum Komfort: Typ für Funktionszeiger

Funktionsprototyp, Implementierung an anderer Stelle

Defaultargumente – wenn nicht angegeben „nichts tun"

Umgang mit zirkulären Abhängigkeit

State.hpp

```
#include "Transition.h"
class State { ... }
```

Transition.hpp

```
#include "State.h"
class State; // forward declaration
class Transition { ... }
```

Zirkuläre Abhängigkeiten deuten oft auf ein Modellierungsproblem hin!

- Speicheranalyse:
  Ist es möglich, dass Klasse A gelöscht wird und B mit Referenz A-B weiter existiert?
- Kann das Design so verändert werden, dass A B nicht kennen braucht?
  → Stellvertreter-Muster

Voraussetzung:

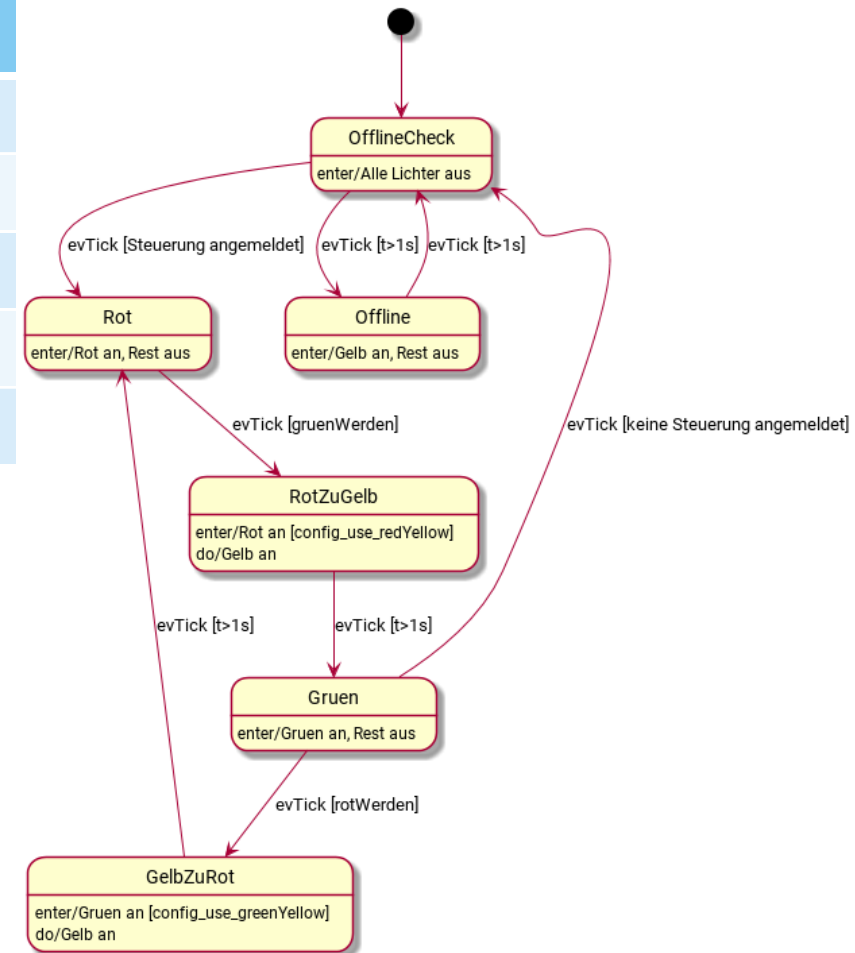
Compiler muss in Transition.hpp **nur Referenzen** auf States verwalten.

Compiler kennt noch nicht Offsets der Attribute !!!

Zustandstabelle

ID	enter	stay	Exit
OfflineCheck	allOff	defaultEffect	defaultEffect
Rot	redOnly	defaultEffect	defaultEffect
RotZuGelb	yellowOnly	defaultEffect	defaultEffect
Grün	greenOnly	defaultEffect	defaultEffect
GrünZuRot	yellowOnly	defaultEffect	defaultEffect

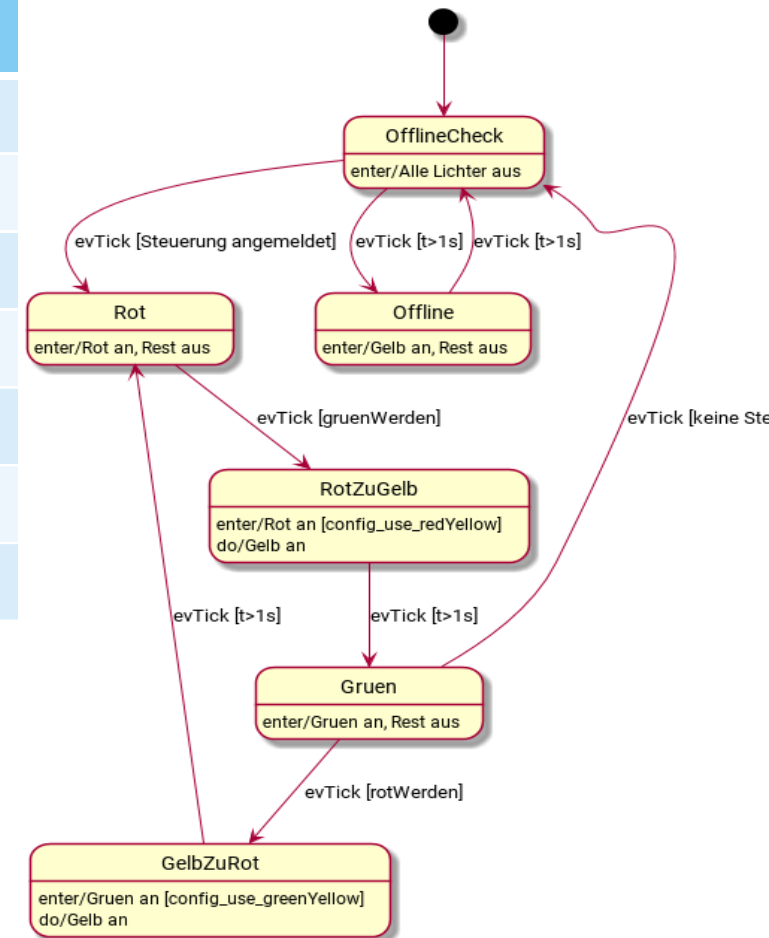
```
State(const string& id,
        Behavior enterFunc=defaultBehavior,
        Behavior stayFunc=defaultBehavior,
        Behavior exitFunc=defaultBehavior,
        void *handle=nullptr
);
```



Transitionstabelle

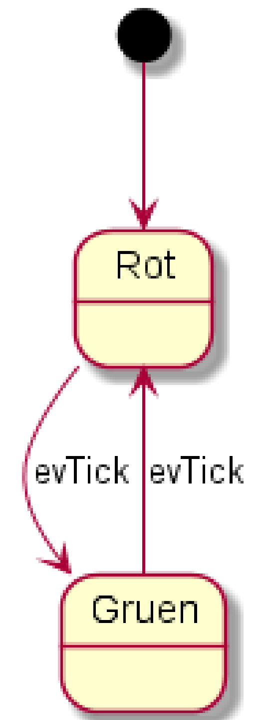
Start	Target	Guard	Effect
OfflineCheck	Offline	controllerPresent	defaultEffect
OfflineCheck	Rot	DefaultGuard	defaultEffect
Rot	RotZuGelb	cmd_goGreen	defaultEffect
RotZuGelb	Grün	DefaultGuard	defaultEffect
Grün	GrünZuRot	cmd_goRed	defaultEffect
GrünZuRot	Rot	DefaultGuard	defaultEffect
Grün	OfflineCheck	DefaultGuard	defaultEffect

```
Transition(const State& start,
         const State& target,
         Guard guardFunc=defaultGuard,
         Behavior effectFunc=defaultBehavior,
         void *handle=nullptr
);
```



Einfaches Beispiel: rot - grün

```
int main() {
  std::map<std::string, State*> states;
  states["red"]   = new State("red",    behavior_red,    defaultBehavior,  defaultBehavior, nullptr);
  states["green"] = new State("green",  behavior_green,  defaultBehavior,  defaultBehavior, nullptr);
  std::list<Transition*> transitions;
  transitions.push_back(new Transition(*states["red"],  *states["green"], defaultGuard, defaultBehavior,
nullptr));
  transitions.push_back(new Transition(*states["green"],*states["red"],   defaultGuard, defaultBehavior,
nullptr));
  Automaton stateMachine(*states["red"], transitions);
  for (int i=0; i<100; i++ ) {
    stateMachine.run();
    usleep(1000*500);
  }
  cout << „If you enjoyed this demo, buy our real traffic light for additional fun." << endl;
  return 0;
}
```



Übungsaufgabe Build-Your-Own-Automaton

Verwenden Sie das Beispiel, um die Konstruktion des Automaten nachzuvollziehen

- im github.com Repository → gleiches Repo wie in MRT1, verlinkt auf OPAL
  - Github: PLT_MRT_ARM-RPi2\Vorlesungsbeispiele\MRT2_VL-3_Automaton
  - Eclipse Projektname: Automaton_Example
- Sofern in Eclipse eingerichtet
  - Per „pull" das Repository aktualiasieren
  - Das Kurzbeispiel als Projekt importieren

Implementieren Sie Ihre Lösung für die Ampelsteuerung mithilfe des Automaton-Kurzbeispiels.

Machen Sie sich wieder mit dem RPi vertraut

Typische Fallstricke

- Ist die Werkzeugkette richtig konfiguriert
  - Toolchain: MinGW / Cross GCC
  - Dialekt: -std=c++0x
  - Libraries: lib/libbcm2385.a
- Ist das Target richtig konfiguriert?
  - PC = C/C++ Application
  - RPi = C/C++ Remote Application

Zusammenfassung

OOD mit UML 2.5

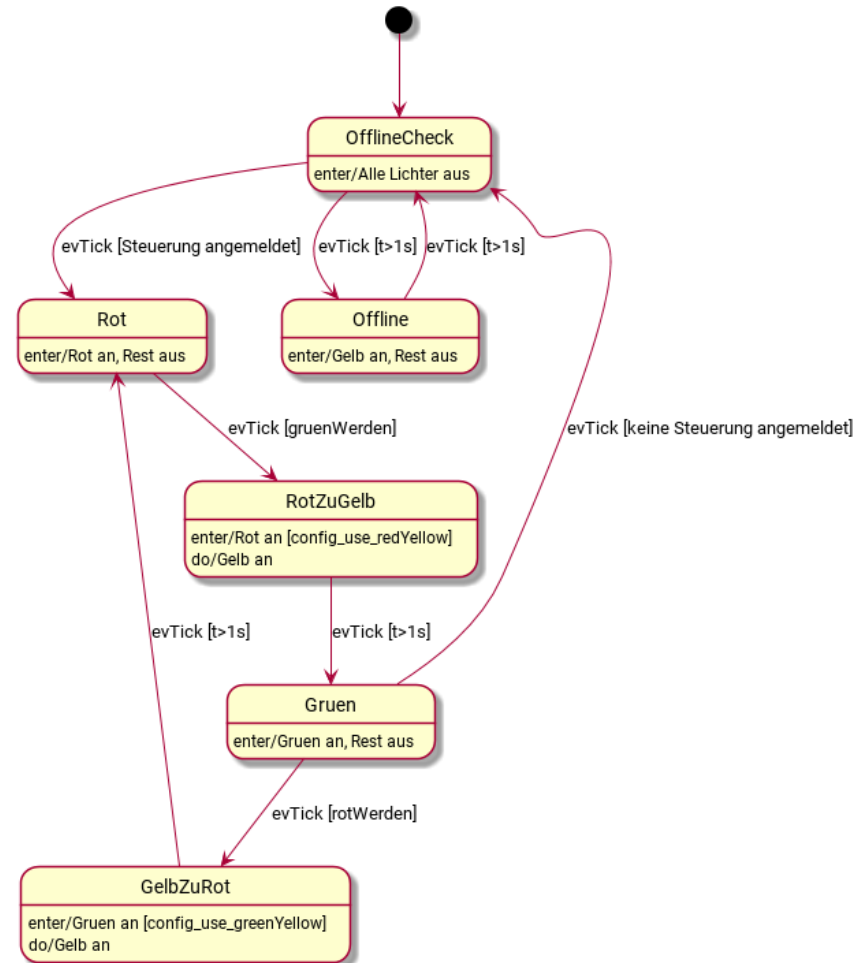
- Herleitung des Ampel-Steuerautomaten mit UML Zustandsdiagrammen
- Herleitung der Steuerautomat-Klassen UML Klassendiagrammen

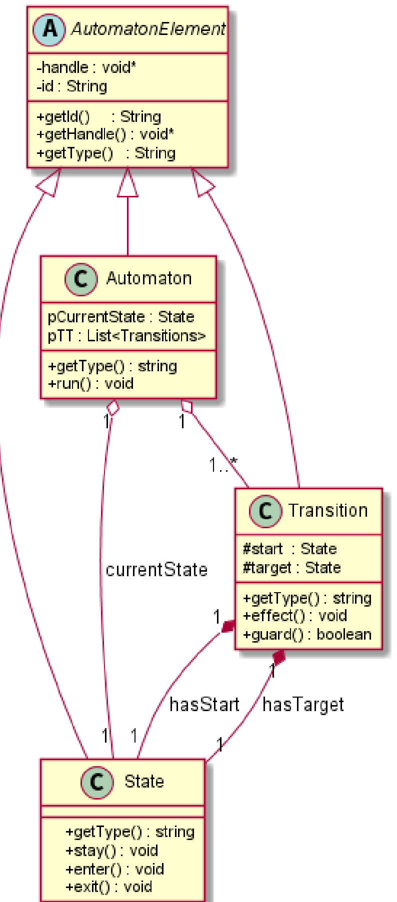
C++ Konzepte

- Grundlagen Klassen, Sichtbarkeit, Vererbung
- Vertiefung `const`
- Freundkonzept
- Spezialisierung von Klassen
- virtuelle Methoden und abstrakte Klassen

Entwurfsmuster

- Strategiemuster





Ausblick auf die nächste Vorlesung

Kurzeinführung Formale Sprachen und Grammatiken

- Wie kann ich prüfen, ob sich eine Ampel richtig verhält?
- Das beobachtbare Verhalten einer Ampel kann als Satz einer Sprache verstanden werden
- Eine Sprache besteht mindestens aus Worten und einer Grammatik
- Prüfen, ob die Ampel die Grammatik beherrscht

Wiederverwendung des endlichen Automaten als Bibliothek

- Wiederholung MRT1: Statische und dynamische Bibliotheken
- Projekteinstellungen zum Erstellen und Nutzen von Bibliotheken in Eclipse

Formale Sprachen und Grammatiken

- Formulierung einer Taschenrechner-Grammatik
- Nutzung der in letzter Vorlesung erstellten Automaten-Klasse als Akzeptor
- Konstruktion eines Automaten

Vielen Dank für Ihre Aufmerksamkeit

Beispielcode der Slides auf github.com

Lehrveranstaltung MRT1/2 auf OPAL

Literaturverzeichnis

Christoph Kecher, Alexander Salvanos, Ralf Hoffmann-Elbern„„UML 2.5: Das umfassende Handbuch“, Rheinwerk Verlag GmbH, 8. Auflage, 2018, ISBN 978-3-8362-6020-6

B. Oestereich; „Analyse und Design mit der UML 2.5“, Oldenbourg Verlag, 2012

Alan Shalloway, James R. Trott; “Design Patterns Explained: A New Perspective on Object Oriented Design, 2nd Edition (Software Patterns) 2nd Edition”, Addison-Wesley Educational Publishers Inc, 2. Auflage, 2004, ISBN-13: 978-0-3212-4714-8

Russ Miles, Kim Hamilton; “Learning UML 2.0”, O’Reilly Media, 1st Edition, 2006, ISBN 978-0-596-00982-3

Philippe Kruchten; “Architectural Blueprints—The “4+1” View Model of Software Architecture”, IEEE Software 12, 1995