

Ampelsystem Entwurf

Karlheinz Hug

Informatik 3

Studiengang Medien- und Kommunikationsinformatik

Fakultät Informatik, Hochschule Reutlingen

4. November 2005 – aktualisiert 8. Dezember 2010

Substantive suchen

Ampelsystem, Verkehrsampel,
Fußgängerampel, Straße, Zebrastreifen,
Richtung, Zustand, Straßenkreuzung, Lampe,
Normalbetrieb, Blinkbetrieb, Verkehrsphase,
Fußgängerphase, Ampeltaste, Gelbphase,
Rotgelbphase, Grünphase, Rotphase,
Straßenverkehr, Warteschlange, Fahrzeug,
Fahrtrichtung, Ampelanlage, Verkehr,
Ereignis, Zyklus, Länge, Wartezeit,...

Anwendungsklassen finden

- Ampelsystem = A.anlage = **System**
- Verkehrsampel = **TrafficLight**
- Fußgängerampel = **PedestrianLight**
- Lampe = **Lamp**

Abstraktionen finden

- Verkehrsampeln und
- Fußgängerampeln
- sind spezielle Ampeln = **Lights**

Bestandteile finden

- Ampelsysteme bestehen aus Ampeln
- Ampeln bestehen aus Lampen

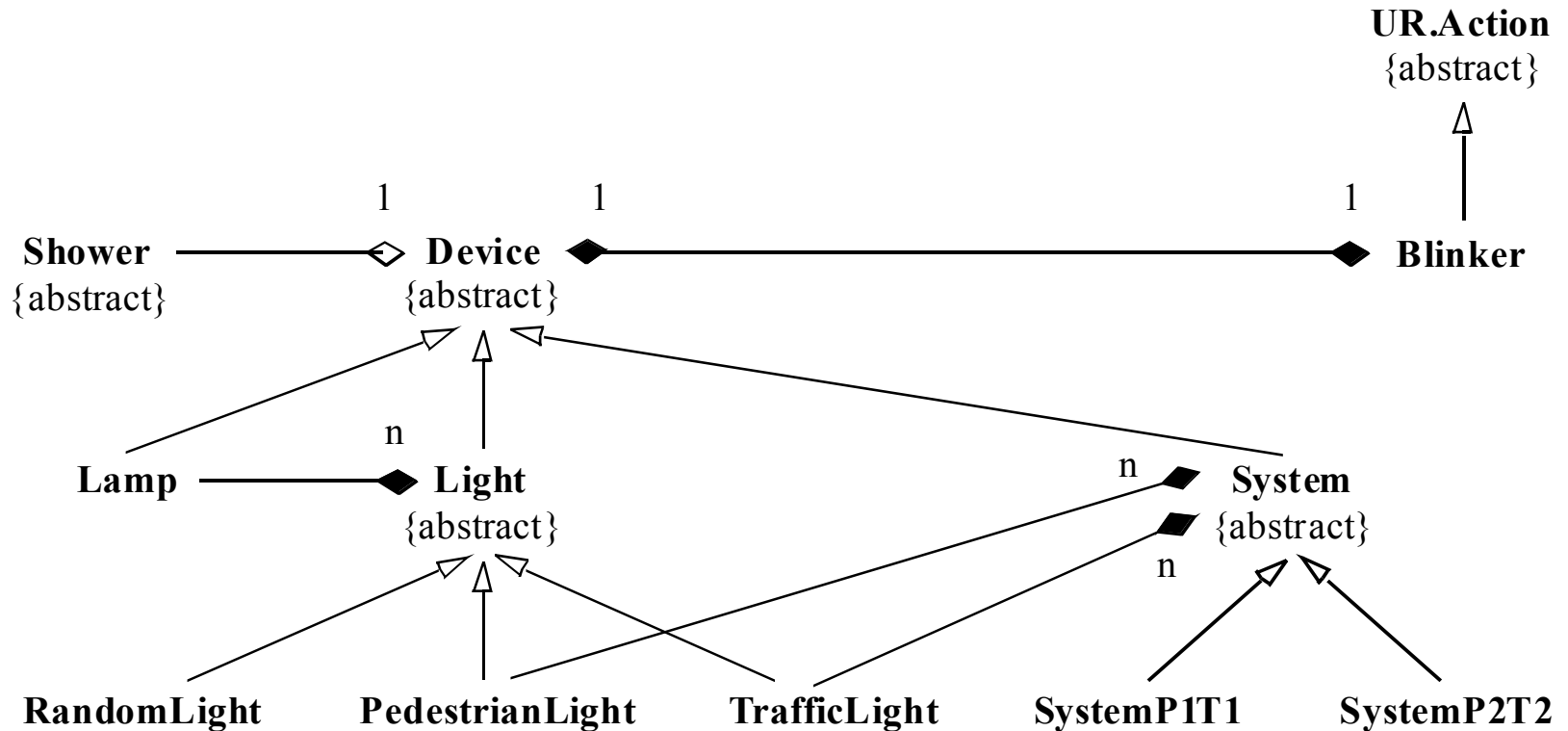
Gemeinsames Verhalten finden

- Ampelsysteme durchlaufen zyklische Phasen
- Ampeln durchlaufen zyklische Zustände

Abstraktionen finden

- Ampelsysteme und
- Ampeln
- sind spezielle
zyklische Zustandsautomaten =
Geräte = **Devices**

Klassendiagramm erstellen



Entwurfsziele beachten

- Lokalität
- Lose / starke Kopplung
- Änderbarkeit
- Austauschbarkeit
- Wiederverwendbarkeit
- Erweiterbarkeit
- Wartbarkeit

Entwurfsklassen finden

- **Shower**
- **Blinker**

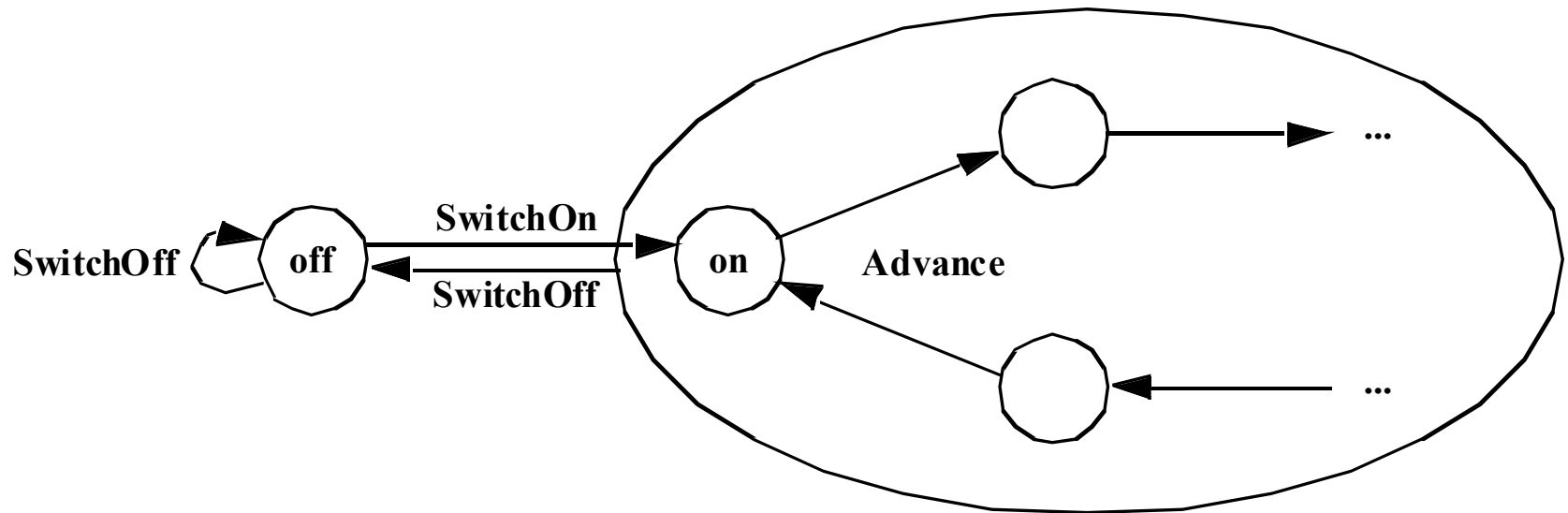
Entwurfsmuster anwenden

- **Kompositum**
 - Lamp – Light – Device
 - Light – System – Device
- **Schablonenmethode**
 - Shower
 - Blinker (implementiert UR.Action)
- **Fabrikmethode**
 - Lamp, Light, System
- **Beobachter**
 - Shower

Device-Klassen entwerfen

- Jede **Device**-Klasse modelliert einen endlichen Automaten mit zyklisch durchlaufenen Zuständen
- Zu jeder **Device**-Klasse gibt es ein Zustandsdiagramm
- Die abstrakte Klasse **Device** modelliert das abstrakte Verhalten zyklischer Automaten

Zustandsdiagramm abstrahieren

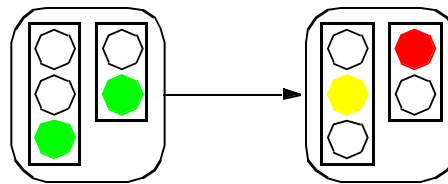


Nur *ein* Zustandsübergang
pro Operationsaufruf!

Warum nur *ein* Zustandsübergang pro Operationsaufruf?

- **Device-Objekt** verharrt im Zustand
- Kunde entscheidet, wann es nächsten Zustand einnimmt
- Kunde soll mehrere **Device-Objekte** flexibel schalten können

Beispiel:

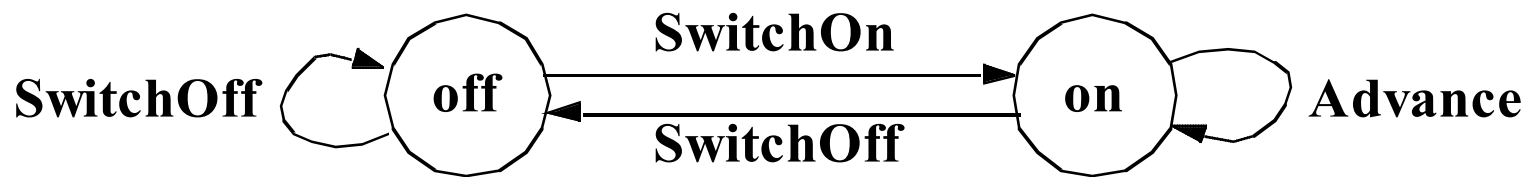


Zustände abstrahieren

- Aus den Komposita ergibt sich eine Hierarchie von Zustandsdiagrammen
- Komponierte Klassen wiederholen nicht alle Zustandskombinationen ihrer Teile, sondern definieren umfassende Zustände

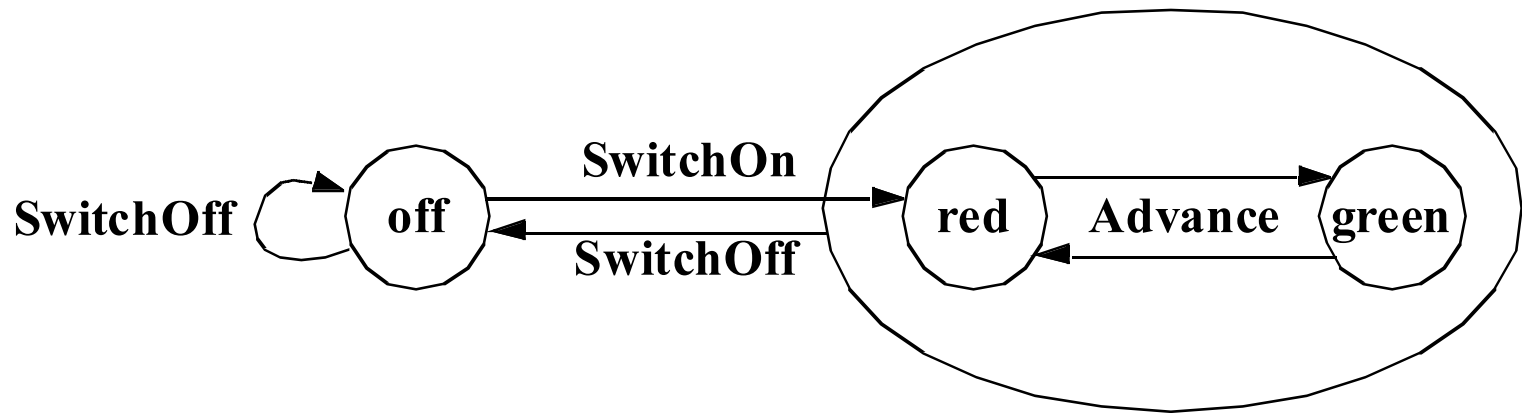
Zustandsdiagramm

Device, Lamp, RandomLight



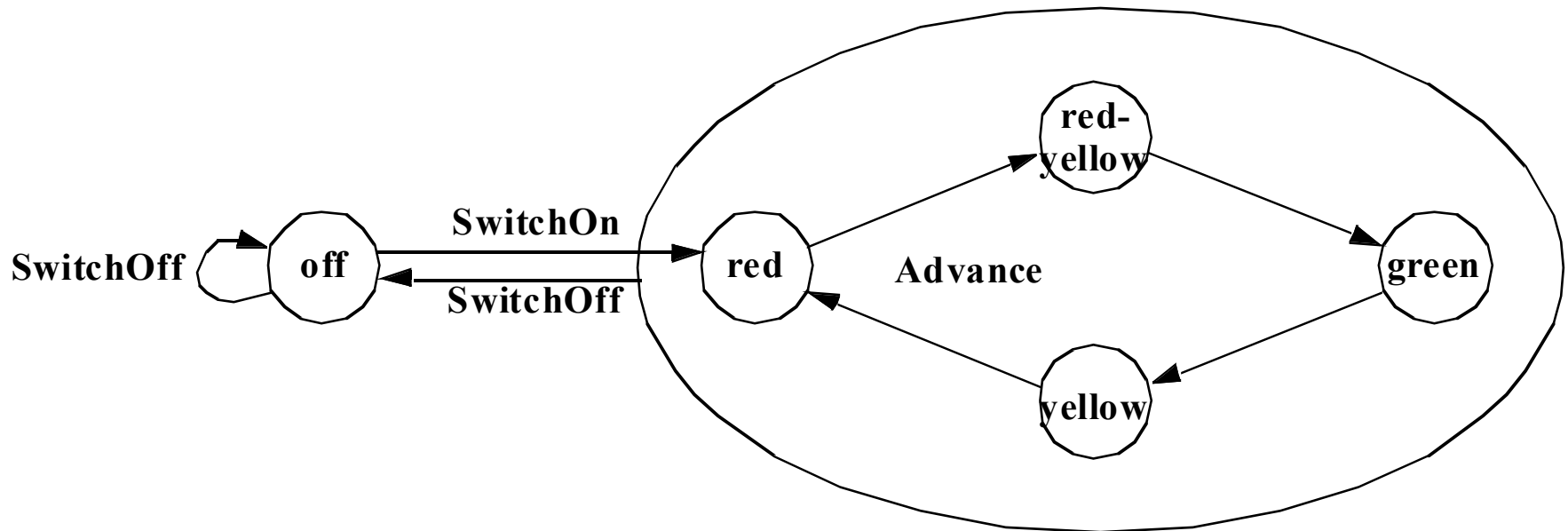
Nur *ein* Zustandsübergang pro Operationsaufruf!

Zustandsdiagramm PedestrianLight



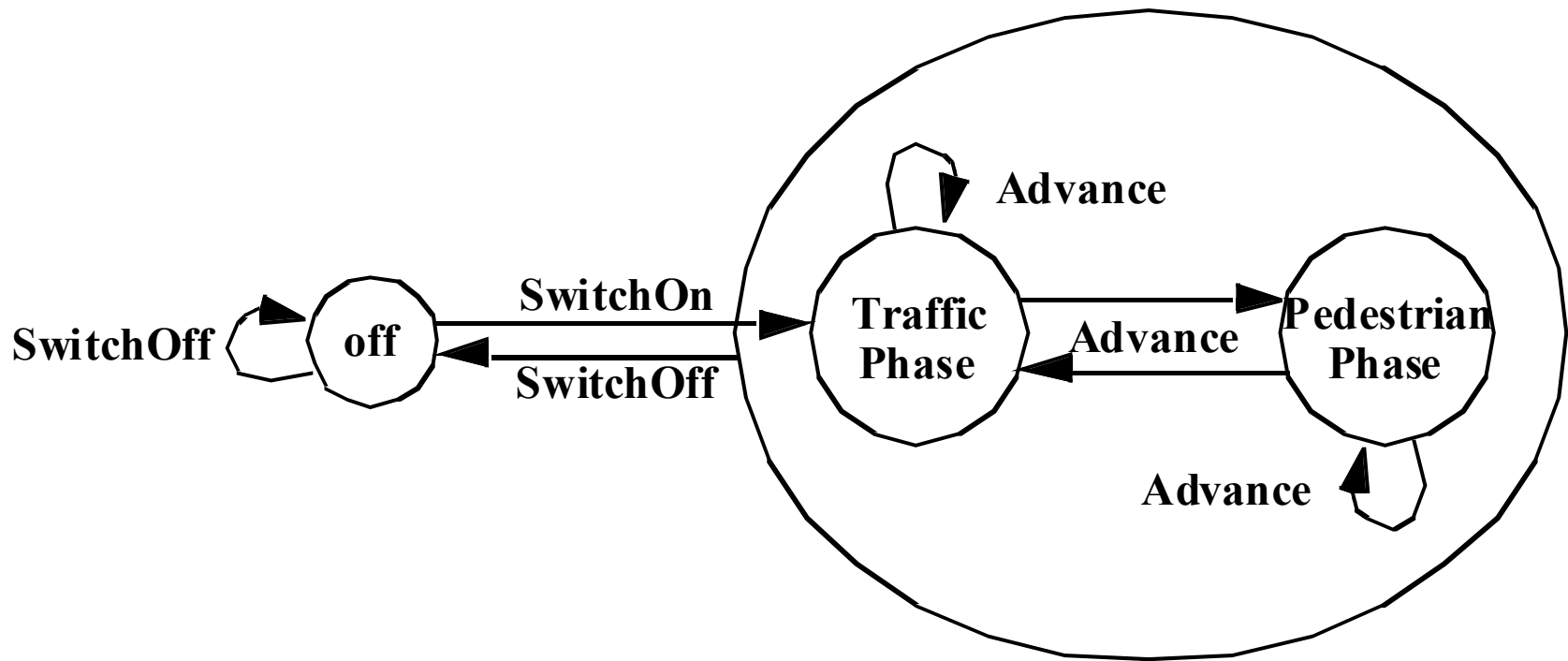
Nur *ein* Zustandsübergang pro Operationsaufruf!

Zustandsdiagramm TrafficLight



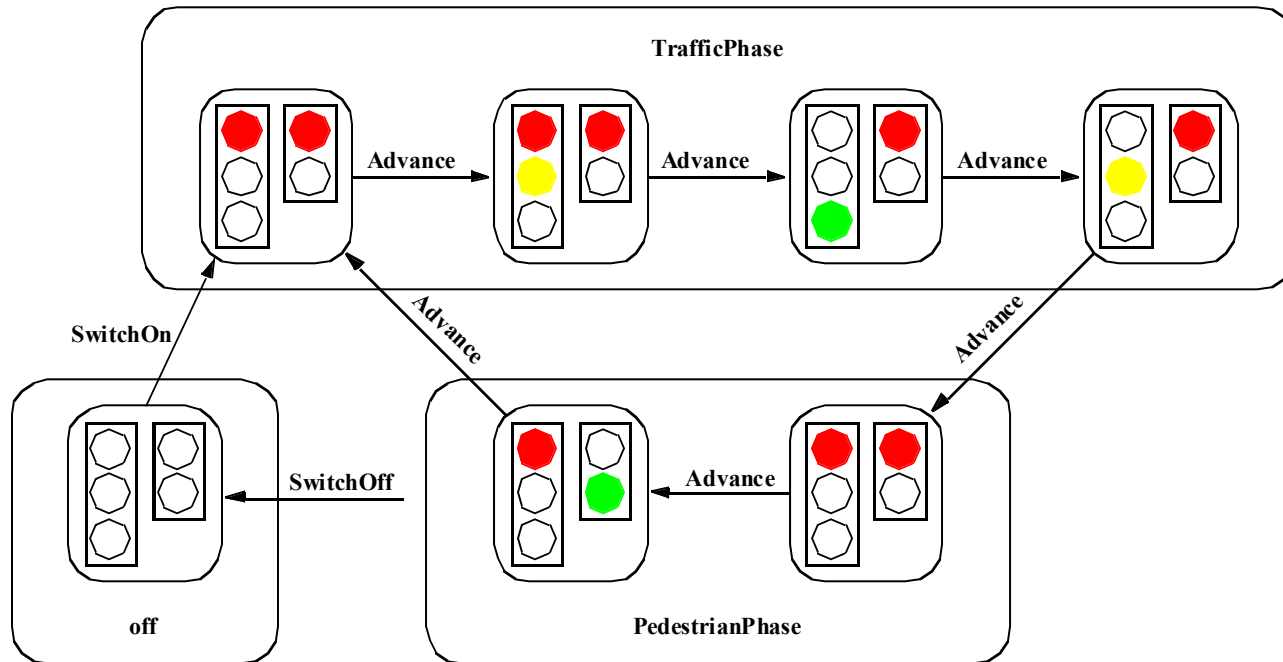
Nur *ein* Zustandsübergang pro Operationsaufruf!

Zustandsdiagramm SystemP1T1

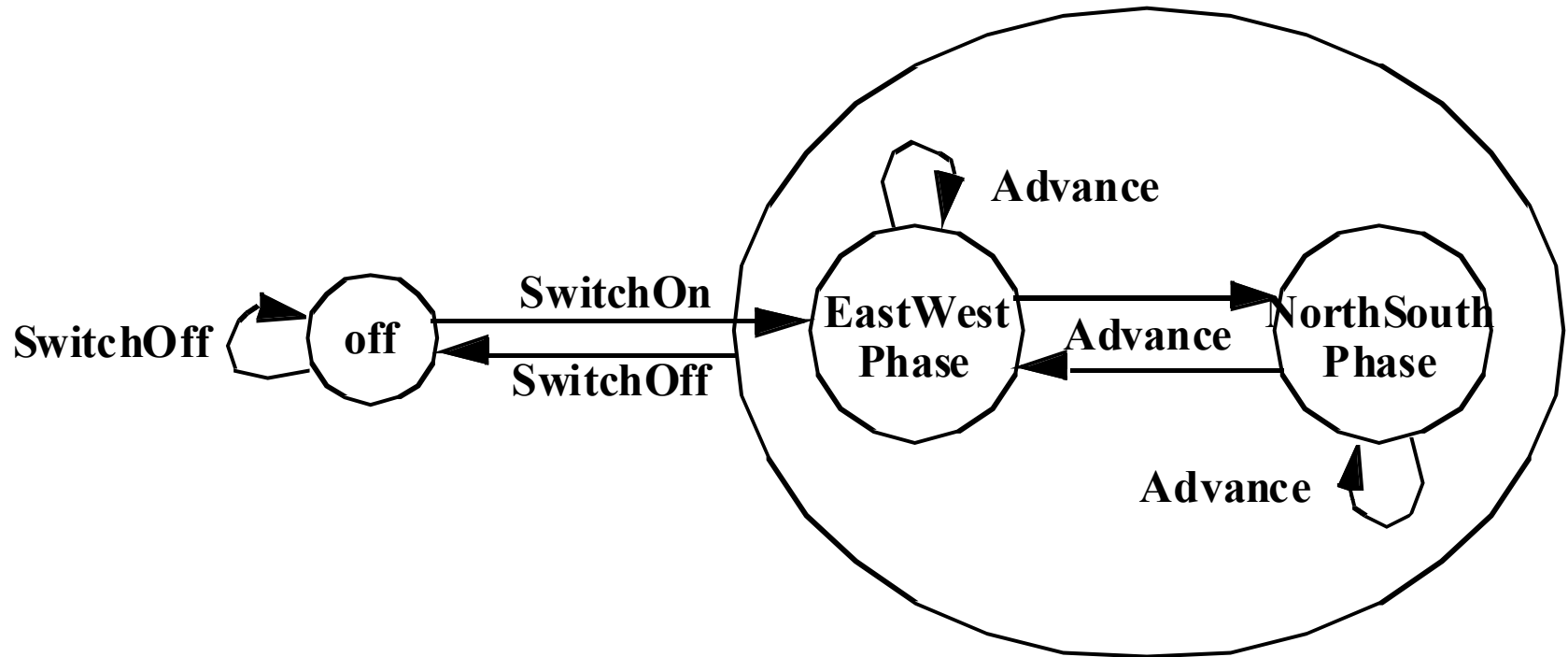


Höchstens *ein* Zustandsübergang bei jeder Komponente
pro Operationsaufruf!

Zustandsablauf SystemP1T1



Zustandsdiagramm SystemP2T2



Höchstens *ein* Zustandsübergang bei jeder Komponente pro Operationsaufruf!

Zustandsablauf SystemP2T2

*Die Darstellung
des Ablaufs
der Ampelzustände
an einer Kreuzung
ist dem Leser
als Teilaufgabe überlassen!*

Wie hängen die Zustände komponierter Klassen zusammen?

- Zu jedem Zustand einer komponierten Klasse kann es mehrere erlaubte Kombinationen der Zustände der Komponenten geben
- Diese Zusammenhänge sind in Invarianten festzulegen

Abstrakte und konkrete Übergangsoperationen

- Dem Zustandsmodell zufolge müssen die Übergangsoperationen

**SwitchOn,
SwitchOff,
Advance**

redefinierbar sein

Vorbedingungen der Übergangsoperationen

SwitchOn
PRE off

- Wiederholtes Anschalten im laufenden Betrieb verhindern,
da sonst bei Komposita inkonsistente Zustände entstehen

Vorbedingungen der Übergangsoperationen

SwitchOff

PRE -

- Keine Vorbedingung,
damit Abschalten stets möglich

Vorbedingungen der Übergangsoperationen

Advance

PRE # off

- Soll normalen Zyklus ungestört ausführen können
- Wird z.B. wiederholt von asynchron laufendem Prozess aufgerufen

Vorbedingungen der Übergangsoperationen

Advance

PRE # off

- Asynchrones Abschalten verboten, da es die Prozesssynchronisation verkomplizieren würde
- Regel:
 - Erst asynchronen Prozess stoppen,
 - dann abschalten

Blinken

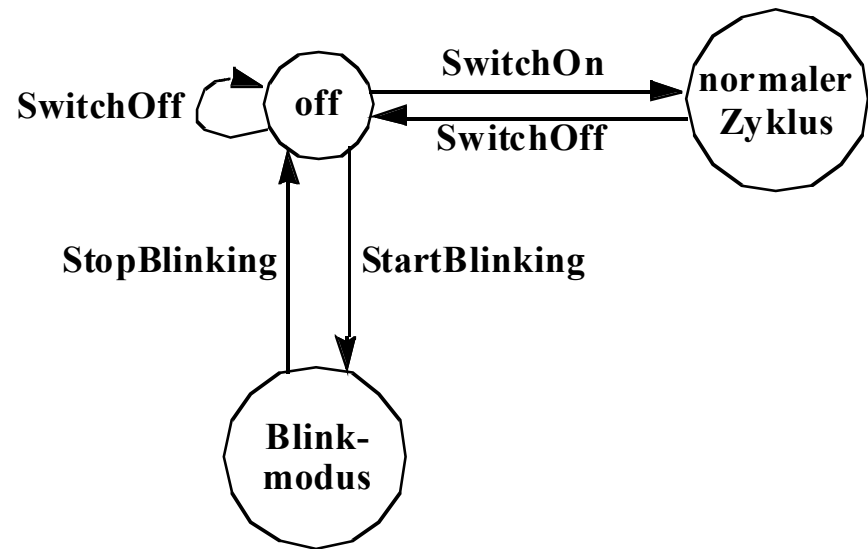
- Problem: Blinken passt nicht zu normalem zyklischen Verhalten
- Zwei sich abwechselnde Zustände:
aus <-> **an**
- Blink-An-Zustand einer komponierten Klasse kann von allen zyklischen Zuständen verschieden sein
- Beispiel: 2 Verkehrsampeln, Blink-An = **gelb/gelb**, kein erlaubter zyklischer Zustand

Blinken

- Lösung: orthogonal
zum zyklischen Zustandsmodell
ein Blink-Zustandsmodell:
nicht blinkend <-> blinkend
- realisiert als Flag **blinking** in **Device**
- **Advance** und Blinken dürfen nicht
überlappen, da sonst inkonsistente
Zustände entstehen

Blinken

- Ein System ist entweder im
 - Auszustand, oder
 - normalen Zyklus, oder
 - Blinkmodus



Bedingungen verstärken

Advance

PRE # **off AND NOT blinking**

POST # **off AND NOT blinking**

StartBlinking

PRE **off AND NOT blinking**

POST **blinking**

StopBlinking

PRE **blinking**

POST **off AND NOT blinking**

Erlaubte Zustände

on	angeschaltet, # off
blinking	Blinkmodus
running	normaler Zyklus

Aus Wahrheitstabelle bestimmen:

running IMPLIES (on AND NOT blinking)

≡

**(on AND NOT blinking) OR
NOT running**

Blinken entwerfen

- **Blinker** als Implementationsklasse zur Schablonenklasse **UtilitiesRepeater.Action**
- **Blinker.Do**
 PRE **blinking**
 DO **device.Blink**
 privat, final
- **Device.Blink**
 öffentlich, redefinierbar
- **Device.Start/StopBlinking**
 öffentlich, final

Blinken entwerfen

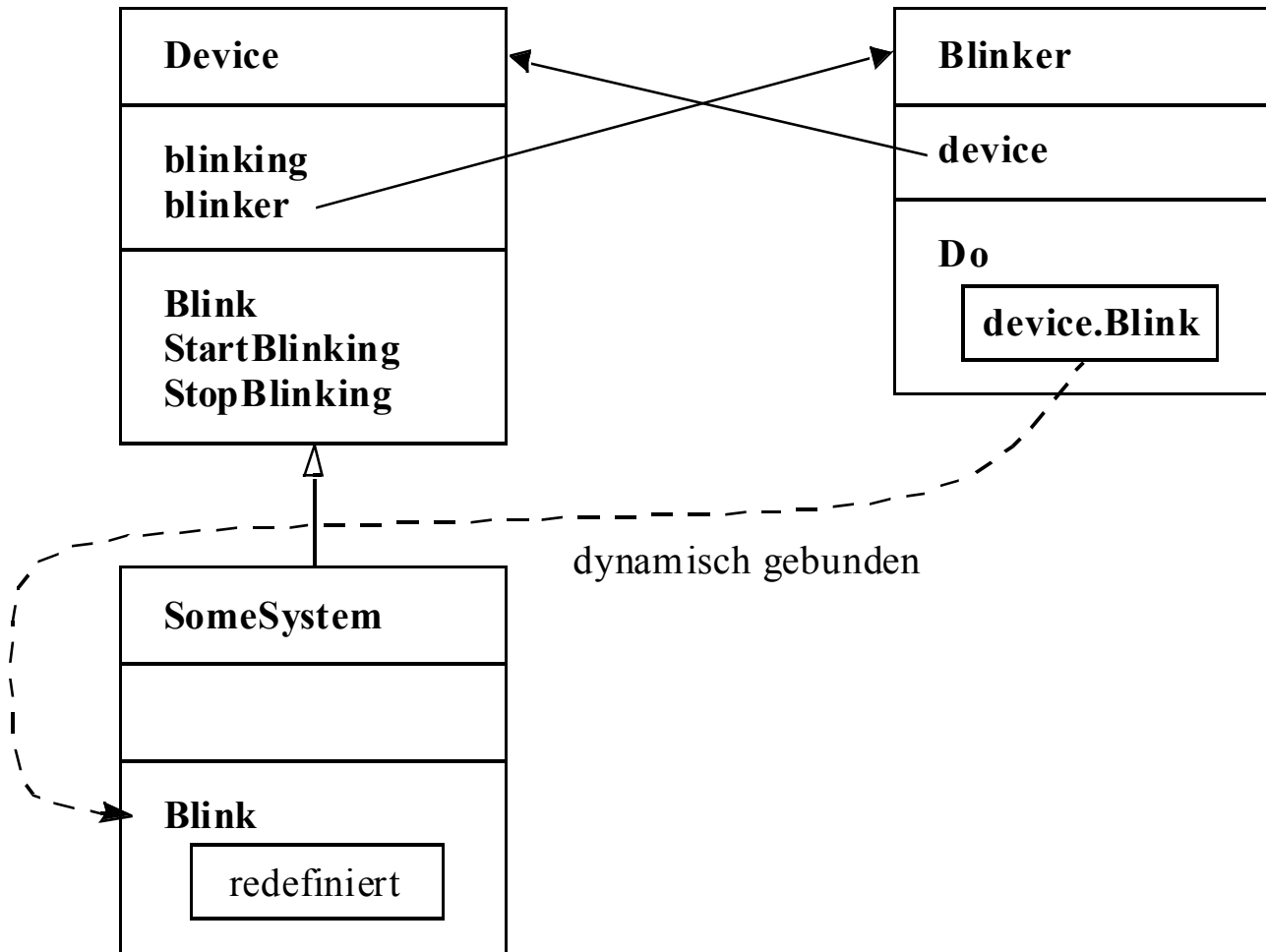
Vorteile:

- **Blinker-Klasse** fertig implementiert
- flexibel einsetzbar
- effizient
- **Start/StopBlinking** fertig implementiert

Nachteil:

- **Device** etwas komplexer

Blinken entwerfen



Blinken entwerfen

Lamp.Blink

DO IF **off** THEN **SwitchOn**
ELSE **SwitchOff** END

Light.Blink

DO some **lamp.Blink** (z.B. gelbe Lampe)
Precursor

System.Blink

DO some **light.Blink** (z.B. Verkehrsampeln)
Precursor

Blinken entwerfen

- Jedes **Device**-Objekt ist fest mit einem **Blinker**-Objekt verbunden
- flexible Möglichkeiten
- Bei komponierten Klassen wird tatsächlich nur das **Blinker**-Objekt des komponierten Objekts genutzt, nicht die **Blinker**-Objekte der Komponenten

Vorteile:

- leicht zu implementieren
- alle Komponenten blinken synchron

Invarianten bestimmen

PedestrianLight

aus: alle Lampen aus

normal: Lampen abwechselnd an

blinkend: rote Lampe blinkt

blinking IMPLIES greenLamp = off

NOT blinking IMPLIES

(redLamp = off OR greenLamp = off)

Invarianten bestimmen

TrafficLight

aus: alle Lampen aus

normal: rot, gelb, grün, rotgelb

blinkend: aus, gelb

blinking IMPLIES

(redLamp = off AND greenLamp = off)

...

Invarianten bestimmen

SystemP1T1

aus: alle Ampeln aus

normal: TrafficPhase =>

 trafficLight Normalzyklus, pedLight rot

 PedestrianPhase =>

 pedLight Normalzyklus, trafficLight rot

blinkend: trafficLight aus <-> gelb,

 pedLight aus

**blinking IMPLIES (pedLight = off AND
 trafficLight IN {off, yellow})**

Invarianten bestimmen

SystemP2T2

aus: alle Ampeln aus

normal: EastWestPhase =>

 EaWeLights Normzyklus, NoSoLights rot

 NorthSouthPhase =>

 NoSoLights Normzyklus, EaWeLights rot

blinkend: trafficLights aus <-> gelb,

 pedLights aus

NOT blinking IMPLIES

(pedLight = off $\Leftrightarrow$ trafficLight = off) OR ...

Module

	Klassen	DIN-A4-Seiten
Devices	2	4
Lamps	1	3
Lights	4	7
LightSystems	3	7
LightSimulation	0	2
LightSimulationDB	10	9
	20	33