Objektorientierte Modellierung anhand von UML

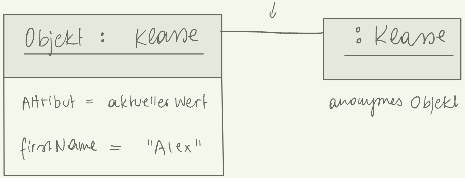
meistverwendetes: Klassendiagramm

gemeinsam mit Objektdiagramm → Strukturmodellierung

Klassen von Objekten: Objekte mit (min.) gleichen Eigenschaften/Verhalten

Instanz: ein Objekt der Klasse

Objektnotation



mehrere Objekte mit Links verbunden … Objektdiagr.

= Momentaufnahme des Systems

dienen vorwiegend der Illustration

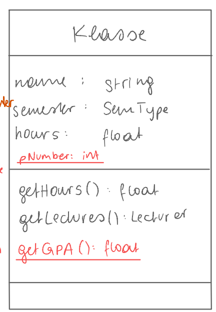
Klassennotation

meist counter ↑

Klassenvariable nur einmal /Klasse erhoben

suchen ↰

Klassenoperation kann auch ohne Instanz verw. werden



Klassenname

Attribute

↱ Abgeleitete Attribute z.B berechnet

[Sichtbarkeit][/] Name [: Typ] [[1..*]] [= Default] [{Eigensch, Eigensch, …}]

(Name ↓ eindeutig; Typ ↳ Datentyp; [1..*]: [min..max])

- + public
- − private : nur jene aus Klasse
- # protected : aus Klasse / Subklasse
- ~ package : alle Objekte eines Pakets

String, bool, int, …

<<primitive>> → (<<primitive>> Float | round() : void)

<<datatype>> zusammenges.

<<enumeration>>

{readonly}

{unique}/{nonunique}

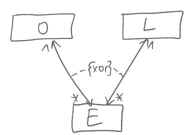
{ordered}/{unordered}

Operationen

[Sichtbarkeit] Name ([Parameter, …]) [:Typ] [{Eigenschaft, …}]

↳ [in / out / inout] Name [: Typ] [[0..*]] [= Default] [{Eigenschaft, …}]

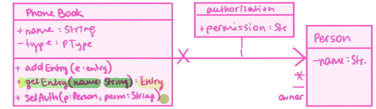
kann auch mehr Abschnitte geben z.B. noch constraints



Schlüsselattribute in UML nicht vorgesehen

↳ Erweiterung mittels Stereotypen

<<key>> - ssO : int




```
class PhoneBook {
public String name;
private PType type;
private Person owner;
public Hashtable authorization;
  // Key: Person (Type: Person)
  // Value: permission (Type: String)
public void addEntry (entry e) { ... }
public Entry getEntry (String name) { ... }
public void setAuth (Person p, String perm) { ... }
}
```

(zu `private Person owner;`) wenn das gegeben, muss bei Person angegeben werden bei PhoneBook dann immer ✗←…

(zu Key) wohin navigiert wird

(zu getEntry / setAuth) → das wird anyways ignoriert beim erstellen

Aggregation

Transitiv

schwach

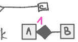
Teil kann in mehreren Ganzen enthalten sein

eingeschr. Propg. Semantik

Ganzes gelöscht → Teile nicht

unger. azykl. Graph

stark



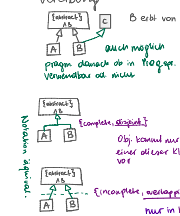
immer

Teil in einem bestimmten Ganzen enthalten sein

wenn gelöscht wird Ganzes gelöscht

Komposition Baum

Vererbung



B erbt von C

auch möglich

pragm. danach ob in Prog.spr. verwendbar od. nicht

Obj. kommt nur in einer dieser Klassen vor

nur in Konzeption mögl.

Assoziationen

wenn nicht navigierbar KEINE Multiplizität angeben

Q

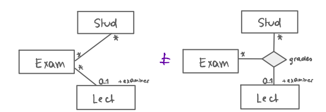
(Nicht) Navigierbarkeit

Rolle

Multiplizität mit wievielen Objekten der anderen Seite in Beziehung

weder Pfeil noch Kreuz heißt keine Angabe

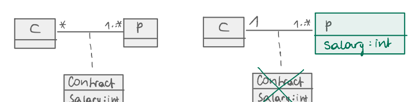
n-äre Bez.



bei ein Prüf mehrere Lect ein Stud

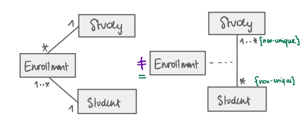
Jeder Stud genau 1 Lect bei jeder Prüfung

Assoziationsklasse



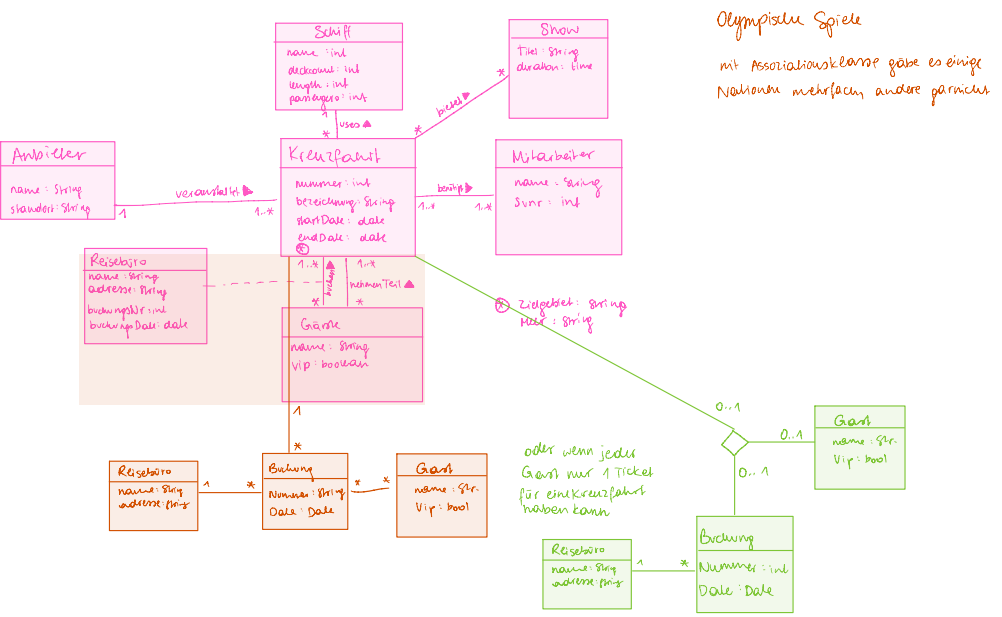
v.a. bei n-n Bez.

Personen können versch. Gehalt bei versch. C haben



Pro Stud/Study mehrere Enrollments

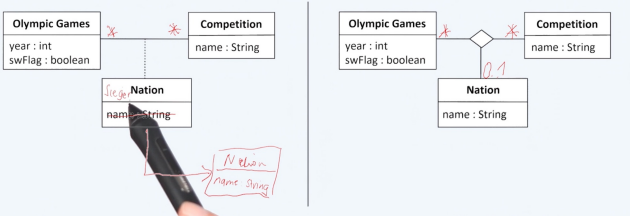
Pro Stud / Study nur 1 Enrollment außer {non-unique}



Beispiel: Olympische Spiele

Modellieren Sie den folgenden Sachverhalt:

Bei den olympischen Spielen gibt es mehrere Bewerbe. Von den Spielen wird das Jahr und, ob es sich um Sommer- oder Winterspiele handelt, gespeichert. Vom Bewerb wird die Bezeichnung gespeichert. In einem Austragungsjahr kann einen Bewerb genau eine Nation gewinnen. Von der Nation wird der Name gespeichert.



Ordnung und Eindeutigkeit von Assoziationen

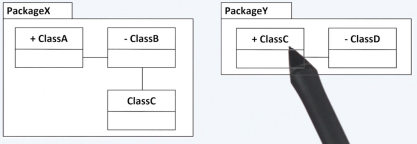
- Ordnung `{ordered}` ist unabhängig von Attributen

- Eindeutigkeit
  - Wie bei Attributen durch `{unique}` und `{nonunique}`
  - Kombination mit Ordnung `{set}`, `{bag}` und `{sequence}` bzw. `{seq}`

Eindeutigkeit	Ordnung	Kombination	Beschreibung
unique	unordered	set	Menge (Standardwert)
unique	ordered	orderedSet	Geordnete Menge
nonunique	unordered	bag	Multimenge (= Menge mit Duplikaten)
nonunique	ordered	sequence	Geordnete Menge mit Duplikaten (Liste)

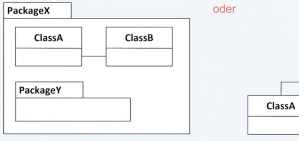
Verwendung von Elementen anderer Pakete

- Elemente eines Pakets benötigen Elemente eines anderen
- Qualifizierung dieser "externen" Elemente
  - Zugriff über qualifizierten Namen
  - Nur auf öffentliche Elemente eines Pakets



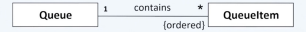
Hierarchien von Paketen

- Pakete können geschachtelt werden
  - Beliebige Tiefe
  - Paket-Hierarchie bildet einen Baum
- Zwei Darstellungsformen



Ordnung und Eindeutigkeit von Assoziationen

- Ordnung `{ordered}` ist unabhängig von Attributen



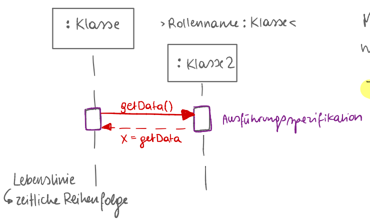
- Eindeutigkeit
  - Wie bei Attributen durch `{unique}` und `{nonunique}`
  - Kombination mit Ordnung `{set}`, `{bag}` und `{sequence}` bzw. `{seq}`

Eindeutigkeit	Ordnung	Kombination	Beschreibung
unique	unordered	set	Menge (Standardwert)
unique	ordered	orderedSet	Geordnete Menge
nonunique	unordered	bag	Multimenge (= Menge mit Duplikaten)
nonunique	ordered	sequence	Geordnete Menge mit Duplikaten (Liste)

Durch die Kombination der Werte von ordered beziehungsweise

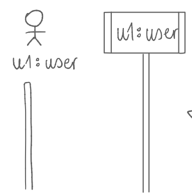
Interaktionsdiagramm

↳ Am häufigsten **Sequenzdiagramm**



Modellierung des Nachrichtenaustauschs nicht auf Typ sondern Instanzebene

Trace ... Abfolge von Nachrichten zwischen konkreten Objekten
wenn gleiche Lebenslinie → Trace festgelegt

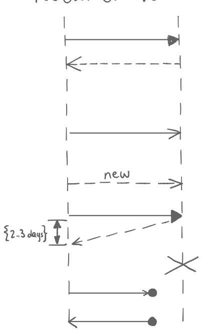


Der Kontrollfluss von Objekten wird durch Signal von anderen oder durch Operationsaufruf gestartet

← **Aktive Objekte** sind permanent aktiv, können unabh. von anderen Objekten operieren. (Doppelte Lebenslinie)

Nachrichten

Pfeil / Pfeilspitze signalisieren Art der Kommunikation



Synchrone Komm.: Sender wartet bis Beendigung d. ausgel. Interaktion
Antwortnachricht (optional) muss nicht att = msg (par 1, par 2) : val
(att: Name des Attr. der Rückgabewert bekommt; msg: Name der Nachricht; par 1, par 2: Parameter; val: Rückgabewert)

Asynchrone Komm.: Nachricht ist Signal, Sender wartet nicht auf Ende

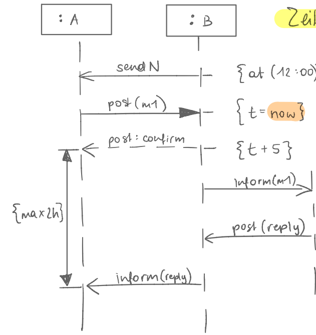
constructor, Objekterzeugung

Zeitkonsumierende Übertragung: Zeitspannenangabe optional

Objektzerstörung: überlebt nicht die gesamte Folge

verlorene Nachricht: Senden an unbekannte/irrelevante Kommunikationspartner

gefundene Nachricht: empfangen von ⊣⊢



Zeiteinschränkung

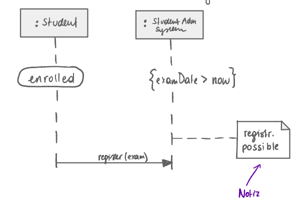
{at (12:00)} {after (5 sec)} {12.00 .. 13.00}

{t = now} aktueller Zeitpunkt

{t + 5} **duration** berechnen der Zeitdauer einer Transaction

Zustandsinvariante, wenn Bedingung nicht erfüllt; Fehler! → nicht weitergearbeitet

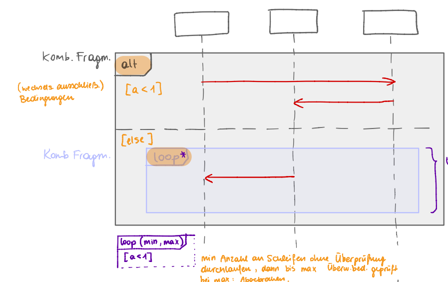
3 Varianten



Kombinierte Fragmente

Modellierung von Kontrollstrukturen

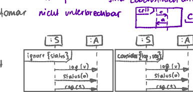
Nachrichtenfluss innerhalb von Operand



min Anzahl an Schleifen ohne Überprüfung durchlaufen, dann bis max Überw. bed. geprüft bei max: Abgebrochen.

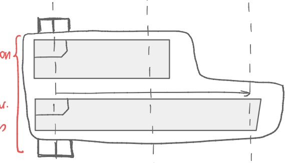
Art des Fragments
↳ default: **seq** schwache Ordn. nur ein Operand notw. Ausführungsreihenfolge nur durch Lebenslinie eingeschränkt

- **alt**ernative
- **opt**ional Üw.b. unbed. angeben
- **break** Ausnahmen Üwb ⊣⊢
  ↳ dann übergeordn Frag. beendet
- **loop** Schleifen
- **strict** strenge Ordn. hier Reihenfolge der Operanden einzuhalten, innerhalb Operand nur Lebenslinie
- **par** nebenläufig Reihenfolge der Operanden egal, nur innerh. von Operanden sind Lebenslinien einzuh.
- **critical** atomar nicht unterbrechbar c-a
- **ignore** irrelevant
- **consider** relevant
- **assert** zugesichert
- **neg** ungültig

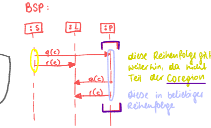


Coregion

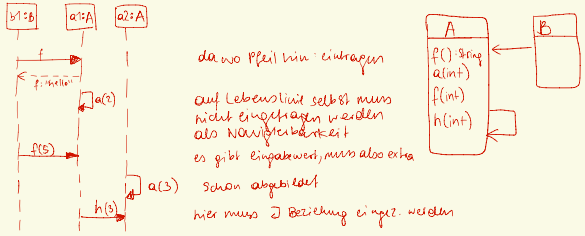
- nebenläufige Abläufe EINER Lebenslinie
- Reihenfolge der Ereignisse innerhalb NICHT beschr.
- Komb. Fragm. innerhalb Coregion als Ganzes in bel. Reihenfolge ausführen



BSP:



KD aus SD



da wo Pfeil hin: eintragen

auf Lebenslinie selbst muss nicht eingetragen werden als Navigierbarkeit

es gibt Eingabewert, muss also extra

a(3) schon abgebildet

hier muss 2 Beziehung eingez. werden

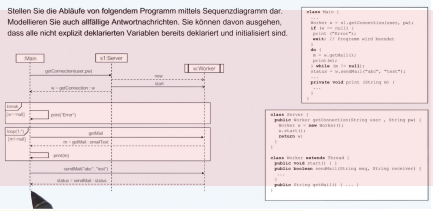
SD aus Code

do... while → loop (1..*)

Annahme (dem Email text)

Beispiel: Programmabläufe

Stellen Sie die Abläufe von folgendem Programm mittels Sequenzdiagramm dar. Modellieren Sie auch allfällige Antwortnachrichten. Sie können davon ausgehen, dass alle nicht explizit deklarierten Variablen bereits deklariert und initialisiert sind.



Beispiel: Bankkonto

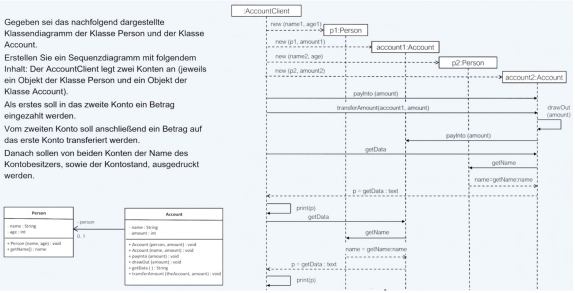
Gegeben sei das nachfolgend dargestellte Klassendiagramm der Klasse Person und der Klasse Account.

Erstellen Sie ein Sequenzdiagramm mit folgendem Inhalt: Der AccountClient legt zwei Konten an (jeweils ein Objekt der Klasse Person und ein Objekt der Klasse Account).

Als erstes soll in das zweite Konto ein Betrag eingezahlt werden.

Vom zweiten Konto soll anschließend ein Betrag auf das erste Konto transferiert werden.

Danach sollen von beiden Konten der Name des Kontobesitzers, sowie der Kontostand, ausgedruckt werden.



Beispiel: Patterns

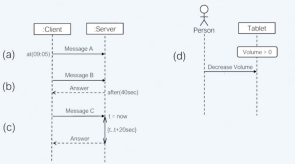
Wie können Sie die folgenden Sachverhalte in einem Sequenzdiagramm abbilden? Modellieren Sie die geschilderten Sachverhalte.

(a) Der Client schickt Nachricht A um 9:05 zum Server und erwartet keine Antwort.

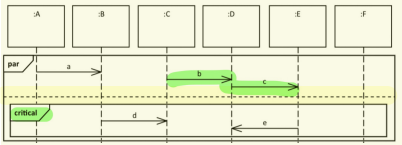
(b) Der Client schickt eine Nachricht B zum Server. Dieser antwortet nach 40 Sekunden.

(c) Der Server muss innerhalb von 20 Sekunden eine Antwortnachricht an den Client schicken, wenn dieser eine Nachricht C übermittelt hat.

(d) Eine Person drückt auf einem Tablet die Taste „Lautstärke verringern". In welchem Zustand bzgl. der Lautstärke muss sich das Tablet befinden, damit dies überhaupt möglich ist? Bilden Sie diesen Sachverhalt mit Hilfe des Konzepts „Zustandsinvariante" ab.

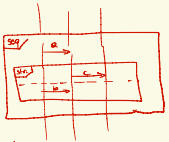


Nested Operanden



hier a - b - c | - (d e) unabhängig

makes the difference



hier muss a vor c sein weil gleicher Operand

auf äußerster Ebene wenn in Operanden Lebensl. geteilt werden müssen Blöcke auch in vorg. Reihenf. ausgef. werden

Transfer-Objekt – Assembler Pattern

Objekte direkt ansprechen Client stark an Obj geknüpft Assembler als Zwischenstufe

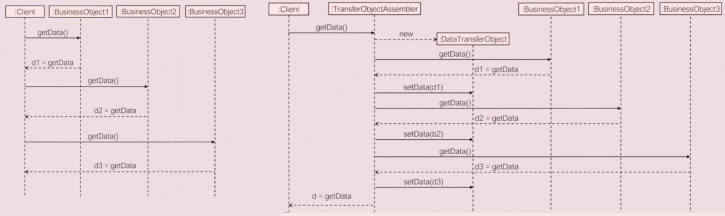
Beispiel: Transfer-Object-Assembler-Pattern

Bei diesem Lösungsansatz benötigt der Client Wissen über die Geschäftsobjekte, um an die notwendigen Daten zu gelangen. So ist der Client stark an die Geschäftsobjekte gekoppelt, was im Allgemeinen nicht wünschenswert ist.

Um diese Abhängigkeiten aufzulösen, wird das Transfer-Object-Assembler-Pattern eingesetzt. Hier wird ein eigener Assembler eingeführt. Der Client kommuniziert nur mit diesem Assembler, welcher die Daten aus mehreren Geschäftsobjekten in einem Transferobjekt zusammensetzt. Der Assembler retourniert dann dieses Transferobjekt an der Client. Somit erhält der Client die von ihm benötigten Daten in einer gekapselten Form.

Konkret wird das Pattern auf folgende Weise realisiert:
Der Client fordert mittels getData() die benötigten Informationen vom TransferObjectAssembler an. Dieser erzeugt zunächst ein DataTransferObject und befüllt dieses mit den Daten von den benötigten BusinessObjects. Die Daten eines BusinessObjects können ebenfalls mittels getData() abgefragt werden. Letztendlich gibt der TransferObjectAssembler das DataTransferObject an den Client zurück.

Modellieren Sie das Transfer-Object-Assembler-Pattern.



festlegen was an relevantem passiert und objektorientierte behandelt werden muss

Keine Abläufe modellieren!

andere Systeme als Schnittstelle

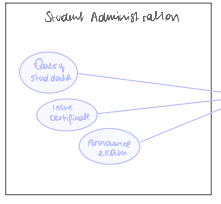
Anwendungsfalldiagramm (Use Case Diagram)

Anforderungserhebung: Anforderung der Kundinnen

Anwendungsfallbeschreibung → nicht in UML enthalten

- Name
- Kurzbeschreibung
- Vorbedingung
- Nachbedingung
- Fehlersituation
- Systemzust. im Fehler
- Akteurin die mit Anwf. kann.
- Trigger

Akteur·in: interagiert mit System — Interaktion immer im Kontext der Anw.fälle — zieht Nutzen!
nicht wie System funktioniert / System beschreiben nicht relevant!



System (was wird beschr.?)

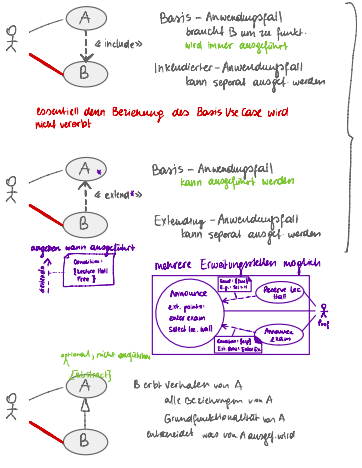
Akteur·in (Wer benutzt System?) — Strichmenschen

- muss mit min. 1 Anfall kommunizieren
- Assoziation ist binär
- Multiplizitäten können angegeben werden

primär Hauptnutznießer
sekundär nur für Funktion nötig

aktiv stoßen Fall an
passiv tun das nicht

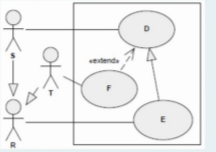
Anwendungsfälle des Profs (was machen die Akteurinnen?)



nicht vererbt

include / extend nur wenn absolut notwendig, es sollen keine Abläufe modelliert werden

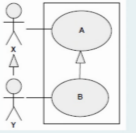
Folgendes Diagramm wurde streng nach UML2-Standard modelliert.
Welche Kombinationen von AkteurInnen können den Anwendungsfall E ausführen?



Wählen Sie eine oder mehrere Antworten:

- a. R
- b. R ∧ R
- c. S ∧ S ✓
- d. R ∧ S ✓

Folgendes Diagramm wurde streng nach UML2-Standard modelliert.
Welche Kombinationen von AkteurInnen können den Anwendungsfall B ausführen?



Wählen Sie eine oder mehrere Antworten:

- a. X
- b. Y
- c. X ∧ Y ✓
- d. Y ∧ Y ✓

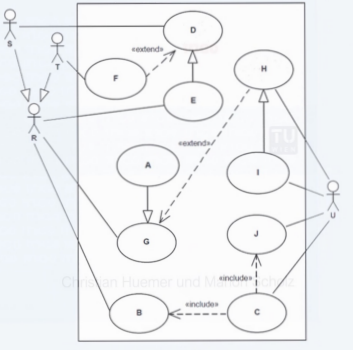
Beispiel: Anwendungsfalldiagramm lesen

Beantworten Sie folgende Fragen zu diesem Anwendungsfalldiagramm:

Welche Akteure (bzw. Kombinationen von Akteuren) können die einzelnen Anwendungsfälle ausführen?

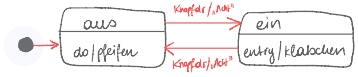
A	R ∨ S ∨ T
B	R ∨ S ∨ T
C	U
D	S
E	S ∧ (R ∨ S ∨ T)
F	T
G	R ∨ S ∨ T
H	U
I	U ∧ U
J	U

Muss B ausgeführt werden, wenn auch C ausgeführt wird? ✓
Muss H ausgeführt werden, wenn G ausgeführt wird? ✗
Ist J oder C der Basis Use Case? C
Ist F oder D der Basis Use Case? D
Kann I auch A erweitern? ✓



Zustandsdiagramm

in welchen Zuständen können sich Obj. befinden
welche Ereignisse lösen wechsel aus



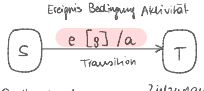
Zustand — Syrenn kann sich dauernaft befinden

Zustandsübergang — inc() / hours := (hours+1) mod 24

●→	Startzustand	Pseudozustand: es wird sofort ausgehende Transition ausgelöst Angabe von Aktivitäten erlaubt aber keine Bedingungen und Ereignisse
→◉	Endzustand	kein Pseudozustand keine ausgehenden Transitionen
→✕	Terminierungsknoten	Objekt hört auf zu existieren Informationen aus Zuständen gehen verloren

events — Ereignis

entry /
exit /
do /
event / z.B. Knopfdruck /



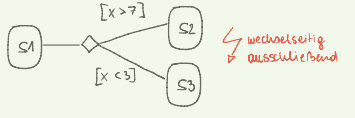
Quellzustand — Zielzustand

- Wenn e und [g] nicht erfüllt: in S bleiben
- Wenn kein e → completion event
- kein [g]: [true]

Ereignistypen

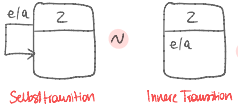
- CallEvent: Empfang einer Nachricht occupy()
- SignalEvent: Empfang eines Signals rightmousedown
- TimeEvent: relativ after(5 seconds)
  absolut when(time == 16:00)
- ChangeEvent: when(x > y) ... bool, dauernd überprüft
- CompletionEvent: wenn alle Aktionen (do) abgeschlossen
- all: irgendein Ereignis, das keine andere Transition auslöst

Modellierung mit Entscheidungspunkten



wechselseitig ausschließend

Zustandsübergänge



Selbsttransition — Innere Transition

hier äquivalent wenn nicht entry/ oder exit/ angegeben!

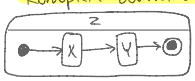
→ immer bei Eintritt bzw Verlassen ausgeführt

Ausführungsreihenfolge

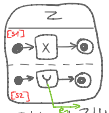
S1 aktiv: entry
e trifft ein: Bed. überprüfen → passt → exit → \a → entry von S2
→ passt nicht → bleibt in S1 (kein exit/entry)

geschachtelte Zustände

Komplexe Zustände



Z/X oder Z/Y



Z/X und Z/Y

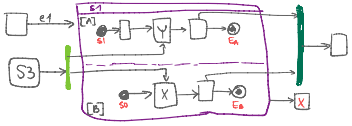
Orthogonaler Zustand
Subzustände nebenläufig in unterschl. Regionen [S1]/[S2]

X u. $\odot_{S2}$
X u. Y
$\odot_{S1}$ u. $\odot_{S2}$ → compl. event
$\odot_{S1}$ u. Y

e_1 Z wird verlassen (auch Region S1)

Ein-/Austreten

exit von innen nach außen
Übergangspkt.
entry von außen nach innen
Bedingungen vor exit prüfen



bei e1 werden S1 und S0 aktiviert
wenn an anderer Position starten:
Parallelisierung-/Synchronisierungskanten

Zielzust. in unterschl. Regionen
Quellzust. außerhalb orth. Zust.
wenn E_A und E_B → X

Subzustandsfolge

Verfeinerungssymbol

umfasst in anderen Zust. diagr. wieder verwendet

Notation: Zustand: Unterautomat

Unterzust. mit Startzust. aktiviert

Historischer Zustand

merken sich letzten internen Zustand im Komplexen Zustand
beim verlassen des komplexen Zustands
beim Rückkehren werden entry/ wieder ausgeführt

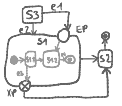
Flacher H-Zustand merkt sich Ebene (Subzustände und Zustände auf der Ebene) (H)

tiefer H-Zustand werden alle Zustände auf Schachtelungsebene gespeichert (H*)

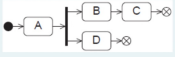
H*x alle Ebenen H* 2 Ebenen

Einsprungspunkt / Ausprungspunkt

Kapselungsmechanismus Transition in Subzustand ohne das äußere T. Aufbau des komplexen Zustands kennen muss



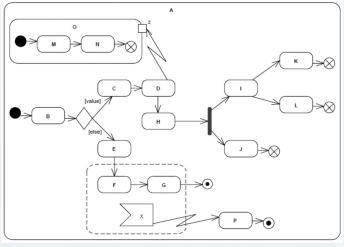
Gegeben ist folgendes Aktivitätsdiagramm. Welche Folgen von abgeschlossenen Aktionen sind im Zuge eines Durchlaufs möglich?



Wählen Sie eine oder mehrere Antworten:

- a. A → B → C
- b. A → B → D → C ✓
- c. A → D
- d. A → D → B → C ✓

Welche Aussagen treffen zu?



Wählen Sie eine oder mehrere Antworten:

- a. Falls während der Durchführung von D der Fehler z auftritt, so wird stattdessen 0 ausgeführt. Anschließend geht der Ablauf mit der Durchführung von H regulär weiter. ✓
- b. Sobald die Aktion P zu Ende durchgeführt wurde, wird die gesamte Aktion A beendet.
- c. Falls während der Durchführung von F oder G das Ereignis X eintritt, so wird P ausgeführt und anschließend ist A beendet. ✓
- d. Es ist möglich, dass im Zuge eines einzigen Durchlaufs die Aktivität 0 und die Aktion H durchgeführt werden. ✓

Aktivitätsdiagramm

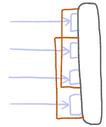
prozedurale Verarbeitungsaspekte

in welchem Schritt werden Infos/Daten übergeben

Kontrollfluss   Datenfluss zwischen Arbeitsschritten

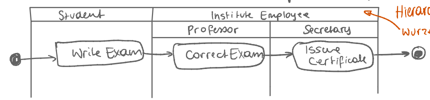
im Kurs grafisch, in echt auch andere Notation z.B. Pseudocode

Parametersatz



alternative Gruppen von Ein- bzw. Ausgabewerten

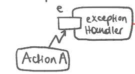
Partition (→ „swimlanes") wer/was ist für Ausführung zuständig ist



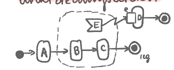
Hierarchische Partition
Wurzelpartition

Ausnahmen

exception Handler: substituierend vordefinierte Ausnahmen. Fehler e tritt ein, Token in A gelöscht → E.H. statt A ausgeführt, danach regulär weiter



Unterbrechungsbereich: mehrere Aktionen eingeschlossen, wenn async. Ereign. E eintritt werden alle Token aus Bereich entfernt und Ausnahmeben. tritt ein. nach D wird nicht in Ablauf zurückgegangen: ◎reg wird nicht erreicht



Kontrollfluss überlagert Aktivitätsfluss

Objektfluss verbindet Aktivitäten nicht direkt sondern über Objektknoten

diese bestimmen Typ der zu transportierenden Objekte

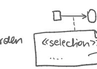
Transport- / Kontrollfunktion

Steuerungsmöglichkeiten

↳ Reihenfolge
- FIFO first in first out   {ordering = FIFO}
- LIFO last in first out   LIFO
- geordnet Selektionsv. angeben! ordered
- ungeordnet eintreffen hat k. Auswirkung unordered

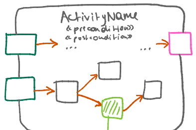
↳ Kapazität (im Objektknoten) {upperBound = 3} / Gewicht (bei Ausgabe) {weight = 7}

↳ Selektionsverhalten selekt. welche weitergegeben werden an Objektknoten selbst oder an Objektflusskante



→ Pseudocode oder natürlich-sprachlich

↳ Transformationsverhalten



Eingabeparameter   Ausgabeparameter

Aktionen ← Knoten

atomar

Sprachunabh.

kann auch weitere Aktivität sein (Schachtelung)

↳ Kommunikationsbezogen: Signale u. Ereignisse
↳ Objektbezogen: Erzeugen / Löschen v. Objekten
↳ Strukturmerkmal: Variablenwerte setzen / löschen
↳ Linkbezogen: Erz. / Löschen von Links zwischen Obj.

Kante legt Abläufe fest

Kontrollflusskanten [a] →
↳ Vorgänger bzw. Nachfolger

Objektflusskanten □→□ oder →□→
↳ transportieren Daten

pins

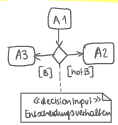
Überwachungsbedingung möglich (guard)

Objektknoten
Inhalt ist Datentoken
verbindet zwei Aktionen, Ergebnis der einen Eingabe f. andere

zur Speicherung und Weitergabe
Transient: Löscht Datentoken sobald weitergegeben

zur Speicherung und Weitergabe
Permanent: bewahrt Datentoken auf und gibt Duplikate weiter

Alternative Abläufe: Entscheidungsknoten



Token geht einen Weg

entweder so / oder so mit [true] [false]

Vereinigungsknoten: Sobald 1 Token eingeht wird er an ausgehende Kante weitergeschickt

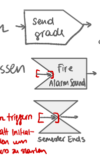
auch Kombinierte Entsch.-/Verei.-Knoten

Aktivität erst wenn Token an beiden Kanten eingehen

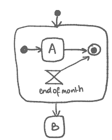
ereignisbasierte Aktionen

senden von Signalen → Send grade → bei Versenden an jemanden od. etwas anderes

Empfangen von Ereignissen asynchrones Ereignis



Asynchrones Zeitereignis: kann triggern; ↳ statt Initialknoten um Flows zu starten



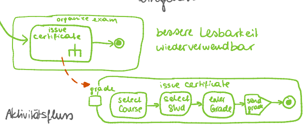
Start / Ende von Abläufen

● Initialknoten: jede ausgehende Kante wird mit Token versorgt. bei mehreren Initialknoten: alle werden mit Token versorgt

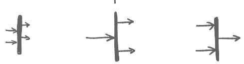
◉ Aktivitätsendknoten: beendet alle Abläufe einer Aktivität und Lebenszyklus eines Objekts

der erste Token der ankommt beendet Aktivität. mehrere Endknoten erlaubt. Kontrolltoken werden gelöscht, Datentoken an Ausgabepins nicht

⊗ Ablaufendknoten: beendet Ablauf einer Aktivität



bessere Lesbarkeit wiederverwendbar



kombinierter Knoten

Parallelisierungsknoten: vervielfacht eingeh. Token

Synchronisierungsknoten: Verbindet eingehende Token zu einem ↳ erst wenn alle da sind

Token beschreiben durch ihren Fluss Aktivitätsfluss

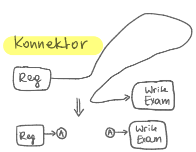
wenn Überwachungsbedingung nicht erfüllt → Token verbleibt im Zustand

Kontrolltoken „Ausführungserlaubnis" für Nachfolgeknoten

Datentoken Transport von Datenwert od. Referenz auf Objekt

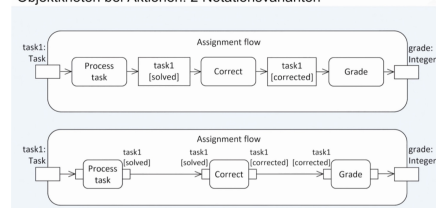
Gewicht einer Kante {weight = 30} → braucht 30 Token; default: 1

Konnektor

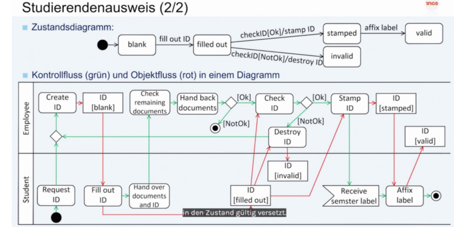


Kontrollknoten
- Ablaufendknoten
- Vereinigungsknoten
- Parallelisierungsknoten
- Entscheidungsknoten

Objektknoten bei Aktionen: 2 Notationsvarianten



Studierendenausweis (2/2)



Steininger • 11811359 • Software Engineering VU

(one/lone/some) (abstract) sig ... declares set of atoms
- 1 set, ≤1 set, ≥1 set, class; immutable object

set any set

extends: subset relations (e.g. sig File extends FSObject {})

sig abs {
fields : e
}
associations
fields is a binary relation with domain A and range given by expression e

- → cross product
- relational join
- ~ transposition
- * reflexive transitive closure
- + union
- & intersection
- - difference
- in subset

- <: domain restriction
- :> range restriction
- ++ override
- iden identity relation
- = equality
- # cardinality
- none empty set
- univ universal set

predicate boolean
function returns value
fact constraints that always hold; value | structural invariants of model

Scopes: run xyz for 5 but exactly ...
alloy analyser searches exhaustively for structures up to a given size
assert a {F}
check a [for ...]

Constraints
- ! not
- && and
- || or
- => implies
- else
- <=> iff

some e ≥1, no e =0, lone e ≤1, one e =1

under / overconstrained models
- missing facts, weak facts
- unnecessary facts, inconsistencies
- simulate model
- use assertions, more than facts

Alloy: formal modelling language based on set theory; specifies collection of constraints that describe a set of structures

Analyze models

Consistency checking
by RUN command
positive answers are definite
→ structures
- translate constr. | formula → formula over booleans
- check if formula has satisfying assignments → SAT solvers
- YES: F is consistent → translate back to model
- NO: F is inconsistent within scope → UNSAT, no structure within scope → larger structures may exist; not necessarily inconsistent

$\exists s \cdot C(s) \land F(s)$

(in)validity checking
by CHECK command
negative answers are definite
→ counterexamples
- translate constr. | ¬formula → formula over booleans
- check if formula has satisfying assignments
- YES: F is invalid
- NO: F is valid within scope
- instead validity → invalidity → looks for counterexamples

$\forall s \cdot C(s) \Rightarrow F(s)$ ≡ $\exists s \cdot C(s) \land \neg F(s)$

modelling
abstraction of reality
simplification
dealing with complexity

(+) describe views of system; specify information informally; graphical notation good f. communication
(-) precise meaning often unclear; incomplete / informal models & tool support; many details hard to depict visually

types of models:
- UML models (informal)
- Formal models
- Design by contract

Formal models: based on math → precise; describe aspect of system; enable automated analysis

formal, precise, verifiable interface specifications
what does code expect / guarantee / maintain?

invariants

preconditions
requirements on input state, must be true at entry
- return error value
- throw exception (IndexOutOfBoundsException)
- use assertions (can be enabled in Java)

weak or strong precond?
- burden on client
- easy method implementation
- easy for clients
- burden on method: what to do with invalid

	obligation	benefit
caller	establish precond. for all calls	assume precondition upon method entry
method	establish postcond. of called method	assume postcondition upon return from call

postconditions
guarantees about the result and output state; must be true at the normal exit of the method
- use assertions (e.g. assert (x <= y.length) — clients not enough capacity)

inheritance and contracts → to fulfill Liskov substitution principle: subclass as substitution for base class without breaking expected behavior
- precondition: stronger / less wide — not okay
- postcondition: weaker / more wide — not okay
- dann nicht schwächer

SDLC
software development life cycle

- Requirements
- Design
- Implementation
- Validation = Testing?!

External Interfaces: detailed description of all inputs / outputs

Functionality
- outputs / inputs relationship
- abnormal situations
- exact sequence of operations
- validity checks
- effect of parameters

Functional / Non-Functional

static numerical requirements
- # terminals supported
- # simultaneous users supported
- amount of information handled

dynamic numerical requirements
- number of requests processed within time period (average | peak workload)

Performance

Design constraints
- standard compliance (report format etc.)
- implementation requirements (tools, prog. language)
- operation requirements (administration / management of system)
- legal requirements (licensing, regulation, certification)

Quality criteria
- correctness (clients view)
- consistency (no contradictory requirements)
- completeness (all (including exceptional) scenarios described)
- clarity (only one way to interpret)
- realism (requir. can be implemented and delivered)
- traceability (each feature can be traced to a set of functional requ.)
- verifiability (repeatable tests)
- portability, maintainability, security
- requirements validation: quality assurance step

Activities
- identify actors: groups supported by system; groups executing (main) functions, (second) (admin, maint.); what external hardware / software
- scenarios: describe common cases; focus on understandability; description of what people do and exp. as they use the system; what tasks actors want to perform; what info does the actor access; what information is passed by actor / system
- use cases: generalize scenarios to describe all possible cases; focus on completeness; list of steps describing interaction between actor and system
- non-funct. requirements: set of them contains conflicts

Principles of maintainable test code
- fast tests (stubs / mocks)
- cohesive, independent, isolated
- have reason to exist
- repeatable, not flaky
- strong assertions
- break if behaviour changes
- single + clear reason to fail
- easy to write, change, evolve

high test coverage → code well tested
low ≠ not well tested
how well it's tested depends on coverage criterion

how to implement with testability in mind

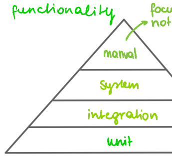
effective and systematic testing
- focus on right tests
- max detected bugs for min amount of tests
- every developer should come up with same tests

unit testing / larger tests / intelligent testing
automated tools to look for tests

unit... piece of code that implements one functionality
(+) is favorable: fits well with the way developers work; fast, easy to control, easy to write
(-) lacks reality, some bugs are not caught

- integration tests: integration between code and external parties
- system tests: system in its entirety; given input X → provides output Y; only for most important behaviour

focus on validation, not automated



complex, real, more / flaky, writeable, controllable, speed: more hard slower
less: less easy faster
× size of slice: number of tests to carry out

€ testing is expensive but worth it
- effective: testing everything is impossible → test right things
- know when to stop: use adequacy criteria
- use / combine different strategies (pesticide paradox)
- bugs happen in some place more than others
- bugs will still happen no matter how much testing
- testing is context dependent

verification: having the system right: implementation correct
validation: having the right system: customer needs

TDD
test-driven development
TDD rapid feedback about implementation of small parts

1. write test for next small feature → fails first
2. implement the feature → passes
3. refactor what we wrote

RED GREEN Refactor cycle: write test → fails → fix it → passes

(+) looking at requirements first; control over pace of writing production code; quick feedback about problems; testable code; feedback about design

test guide to development: test driven development; des. f. testability; des. by contracts
builds feature — Developer — units — autom. test suite
requirements analysis

several units (classes | methods) with different contracts that implement required functionality with testability in mind

developing for testability: how easy it is to write automated tests for the system / class / method under test

right time to think about it: during implementation (+ always)
good testable code: €+ Quality+

DOMAIN core of the system
INFRASTRUCTURE handles external dependencies

Architecture: clear separation of responsibilities
Domain depends on ports: define what infrastructure can do
Infrastructure with adapters: implementations of the ports to connect to infr.

improves testability: easy control what to test
if domain classes depend on ports → exercise behavior by implementing fake ports

dependency injection: implementation strategy; allows to easily separate domain from infrastr. code; receiving dependencies via constructor

dependency inversion principle: high level modules (domain code) should not depend on low-level modules (adapter) but on abstractions (interfaces); details should depend on abstractions

Test doubles
testing class with dependencies might be too slow | hard | much work
mimics behavior of software component for testing purpose
(+) more control, speed, how to think how classes should interact

- dummy objects: passed to class under test but never used
- fake objects: working implementations; do the same task in a much simpler way
- stubs: hard coded answers to performed calls; no working implementation (few stubs real return, different things)
- mocks: act like stubs and save interactions with mocked obj.; allow asserting the method is called once / never with specific parameter / called certain # times with certain par.
- spies: not simulate object but record interactions with it

used most: less realistic, more coupled with code they test

property-based testing
defining a property that the program should satisfy and letting test framework choose examples (goal: find counterexample)
(+) explores input domain much better
(-) more complex; requires more creativity + practice to automate
issues: generating data may be expensive / impossible; failing to express boundaries of a property; ensuring passed input data is fairly distributed

example-based testing
picking out concrete example from all possible ones and writing test case
eg. specification based testing

Testing
- Unit Testing
- Specification Testing
- Boundary Testing
- Structural Testing
- Property-based Testing
- Mocks, stubs, fakes
- Larger Tests
- Integration Testing
- System Testing
- Intelligent Testing
- Mutation Testing

way to go: highly beneficial; we insert bugs in code and check whether the test suite breaks, to proof it is does

specification-based & boundary testing
uses program requirements as testing input
recommended to use first
7-step approach to systematically derive tests
1. understand requirement: what should the program do; what are the inputs; what are the outputs
2. explore program: play with the program to increase your understanding; stop when you have a clear mental model
3. identify partitions: equivalent inputs are combined into partitions; each input variable individually; interactions with others: dependencies / constraints
4. analyze boundaries: making program behave correctly when inputs are near a boundary (think of < instead ≤); test what happens to the program when inputs go from one boundary to the other; for each boundary test on point and off point(s)
5. devise tests: combine all partitions for each of the inputs, decide which ones should be combined: exceptions only once
6. automate tests: write automated tests, name them well
7. augment test suite: develop interesting variations

partition on point does not belong to

BUGS WILL STILL HAPPEN
use different techniques; iterative process (not linear)

structural testing
using structure of source code to guide testing (coverage report)
structural testing ↔ coverage criteria

modified condition / decision coverage
test complex conditions more efficiently than testing all possible combinations
exercises each condition so it can independently of other conditions affect outcome of entire decision
we always need n+1 tests, n = #conditions
VIDEO!

Condition + Branch coverage = exec (branches + cond. values) / total (branches + cond. values) · 100%
↳ each individual condition evaluates to true and false and corresp. branch; statement also evaluates to true and false at least one

MC/DC → Decision coverage → branch coverage → condition & branch coverage
C+B coverage; Path coverage → Branch Coverage → Statement Coverage
more thorough
faster to achieve

control flow graph (CFG) as coverage criterion

Loop coverage = exec. loops with 0, 1, >1 iterations / total loops × 3 · 100%
↳ typically combined with other criteria

Path coverage = exec. paths / total paths · 100%
↳ amount of ways to walk through code; not feasible for input-dependent loops

Branch coverage = exec. br. / total br. · 100%
↳ all conditions count all options

Statement coverage = exec. st. / total st. · 100%

BASIC BLOCK: no jumps within basic block except for last instruction
when first instruction is run → whole block is run (entry / exit are not basic blocks)

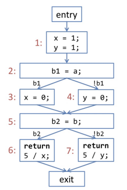
DU pairs: reaching-definitions algorithm
x y — x_predecessor, y_successor

variable definition: basic block v that assigns to v
variable use: basic block that reads value from v
v_n: variable generated in step n: gen(n)
v_m: kill(m) deletes the variables that were defined in m
if I generate v in a block I immediately kill all other (previous) definitions

Definition-clear path: paths where n_i var. def. n_k var use and no n_j: $n_i < n_j < n_k$ with var. def

Reach(n) is all the predecessors; so in n=1 it's empty (there are no preds)
ReachOut(n) is after being in the block; subtract the ones that where killed, but it's empty and none were killed

DU pairs: consider only paths where computation in one part of path affects computation of another
data flow coverage
control flow coverage



n	Reach(n)	ReachOut(n)
1	∅	x_1, y_1
2	x_1, y_1	x_1, y_1
3	x_1, y_1	x_3, y_1
4	x_1, y_1	x_1, y_4
5	x_3, y_1, x_1, y_4	x_1, x_3, y_1, y_4
6	x_1, x_3, y_1, y_4	x_1, x_3, y_1, y_4
7	x_1, x_3, y_1, y_4	x_1, x_3, y_1, y_4

Compute DU: DU pairs of x: (1,6) (3,6)
Reach is 6 because that's where x is used; Definition from index 1, 3

bei Schleifen Tabelle im Nachhinein updaten
→ kann auch (x, x) ergeben also gleiche Zahl