Softwaretechnik

Sommer 2019: Teil 8: Komponenten und Schnittstellen

Henning Schnoor

Institut für Informatik, Christian-Albrechts-Universität zu Kiel

Komponenten und Schnittstellen

Übersicht

Übersicht

Modularer Entwurf: Zur Historie

- David Lorge Parnas
  *10.2.1941 in New York
- Erfinder des Geheimnisprinzips (1971)
- Wegbereiter des Modularen Entwurfs (1972)
  - Spezifikation von Modulen
  - Trennung von Spezifikation und Implementierung
  - Modulare Softwarearchitektur
- Gesammelte Werke: [HW01]

Übersicht

Übersicht

Komponenten in der UML [09, Seite 148]

- A Component represents a **modular** part of a system that encapsulates its contents and whose manifestation is replaceable within its environment.
- A component is a **self-contained** unit that encapsulates the state and behavior of a number of classifiers.
- A component specifies a formal contract of the services that it **provides** to its clients and those that it **requires** from other components or services in the system in terms of its provided and required interfaces.

Komponenten in der UML [09, Seite 148]

- A component is a **substitutable** unit that can be replaced at design time or run-time by a component that offers equivalent functionality based on compatibility of its interfaces. As long as the environment obeys the constraints expressed by the provided and required interfaces of a component, it will be able to interact with this environment.
- Similarly, a system can be **extended** by adding new component types that add new functionality.
- The required and provided interfaces may optionally be organized through **ports**, these enable the definition of named sets of provided and required interfaces that are typically (but not always) addressed at run-time.

Komponenten in der UML [09, Seite 148]

- Optionally, a **behavior** such as a protocol state machine may be attached to an interface, port, and to the component itself, to define the external view more precisely by making dynamic constraints in the sequence of operation calls explicit.
- Other behaviors may also be associated with interfaces or connectors to define the **contract** between participants in a collaboration (e.g., in terms of use case, activity, or interaction specifications).

Ziel des Systementwurfs

- Entwicklung eines Softwaresystems (insb. Code, Testrahmen, Testpläne) aus einer schrittweisen Definition und Zusammensetzung von modularen Bausteinen
  - Modulare Architekturen
- Zentraler Aspekt: Änderungsfreundlichkeit (Design for change)
- Modularisierung als Grundlage für Teamarbeit:
  - Gleichzeitige, unabhängige Arbeit mehrerer Entwickler an verschiedenen Komponenten eines komplexen Systems ermöglichen
  - Schnittstellen zwischen den einzelnen Modulen zwischen den Entwicklern absprechen und einhalten

Benutzt-Beziehungen sollten hierarchisch sein!

- So können dann architekturelle Ebenen geschaffen werden.
  Typisches Beispiel für betriebliche Informationssysteme:
  - Benutzungsschnittstelle
  - Anwendungslogik
  - Datenhaltung
- Das schafft Abstraktionsebenen.
  - Schichtenarchitektur
- Parnas:
  „If the structure is not hierarchical, we may end up with a system in which nothing works until everything works.“
- Ein Modul sollte möglichst wenig andere Module benutzen und von möglichst vielen Modulen benutzt werden.
  (siehe auch das Prinzip Modularisierung aus der Einleitung)

Wiederverwendung von Komponenten

Kandidaten zur Wiederverwendung:

- Eigene Komponenten aus früheren Entwicklungen
- Altsysteme (legacy systems)
- Auf dem Markt erhältliche Komponenten
  - COTS (Commercial-Off-The-Shelf) systems
- Open-Source Software

Spezifische Prozessaktivitäten:

- Analyse von Komponenten
- Anpassung der Anforderungsspezifikation
- Systementwurf mit Wiederverwendung
  - design with reuse
- Komponentenentwicklung und -integration

Historie: Wiederverwendung von Komponenten

- Wiederverwendung findet in den traditionellen Ingenieurwissenschaften statt.
  - Elektrotechnik
  - Maschinenbau
  - Bauingenieurwesen
- Die Idee gibt es für Software schon seit den 60er Jahren
  - McIllroy auf der ersten Internationalen Software Engineering Konferenz, 1968
- Gute Einführung: [Szy02]
- Entwicklungsprozess muss Wiederverwendung berücksichtigen
  - Prozess muss erlauben, bestehende Komponenten einzubinden
  - Neue Komponenten müssen wiederverwendbar gestaltet werden

Entwurfsstrategien

top-down-Entwurf:

- Entwurf wird auf der höchsten abstrakten Ebene oder Schicht begonnen.
- Von oben nach unten wird jede Ebene/Schicht weiter verfeinert.
- Verfeinert wird solange, bis Basismodule erreicht werden, deren Leistungen durch zugrunde liegende Basissoftware realisiert werden können.

bottom-up-Entwurf:

- Begonnen wird mit den Modulen der niedrigsten abstrakten Ebene.
- Aufbauend auf diesen Modulen werden dann die Module der nächsthöheren Schicht entwickelt.
- Sind Module aus vorherigen Entwicklungen vorhanden, dann werden aus diesen Modulen nach und nach höhere abstrakte Ebenen zusammengebaut.

top-down-Entwurf vs. bottom-up-Entwurf

- Top-down Entwurf betont Aufteilbarkeit und
  bottom-up Entwurf betont Kombinierbarkeit.
- Ein ‚guter' Entwurf erfordert eine Kombination aus bottom-up und top-down Entwicklung
  - evtl. jo-jo – Entwurf
- Trotzdem: die Präsentation sollte immer top-down erfolgen!
- Die getroffenen Entwurfsentscheidungen beeinflussen die Änderungsfreundlichkeit grundlegend.
  - Design for change
- Entwurf und Implementierung werden bei beiden Methoden oft verzahnt abgewickelt.

Entwurf mit Komponenten

- Zwei grundsätzliche Ansätze können unterschieden werden:
  1. Komponenten dienen als Entwurfsphilosophie,
     - Unabhängig davon, ob existierende Komponenten wiederverwendet werden sollen.
  2. Komponenten werden als verfügbare Bausteine angesehen, um ein System zu entwerfen und zu implementieren.
- Aspekte:
  - Bibliotheksmanagement
    - Wo finde ich die Komponenten
  - Integration externer Dienste
  - Buy, don't build
  - Buy big things, build small things
  - ‚Not-invented-here' Syndrom
- Viele der zu lösenden Probleme sind nichttechnischer Natur.

Verwendung von externen Komponenten

- Evaluation vor Kauf
- Verwendung von möglichst einfachen Komponenten: Vermeidung von „vendor lock-in“
- Für Änderbarkeit: externe Komponente isolieren, bsp. mit Facade Entwurfsmuster
- Unit-Tests für externe Komponenten!

Technische Ziele der Modularität

Kopplung zwischen Komponenten

- Geringe Kopplung ist angestrebt.
- Entkopplung erhöht die Flexibilität.

Kohäsion innerhalb von Komponenten

- Hohe Kohäsion ist angestrebt.
- Kohäsion erhöht die Unabhängigkeit.

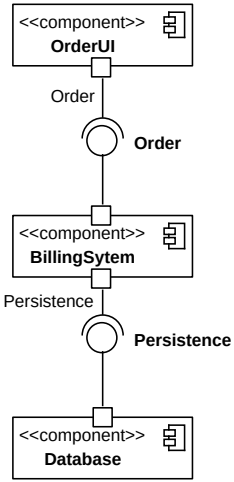
Managementziele der Modularität

- Wiederverwendung bewährter Bausteine
- Hohe Qualität
- Hohe Flexibilität für die Wartung
- Schnellere Konstruktion von Softwaresystemen
- Kostenreduktion

Von Objekten zu Komponenten

- Mehrere Klassen/Objekte können ein Modul/Komponente formen.
- Granularität ist hier ein wichtiger Aspekt!
- Man kann Komponenten auch ohne objektorientierte Techniken realisieren, OO Techniken machen das Leben aber leichter.
- OO Programmiersprachen unterstützen zumeist keine expliziten Benutzt-Beziehungen.
  - Für das Zusammensetzen von Komponenten ist das aber sehr wichtig. Siehe dazu auch Moduldefinitionssprachen: [GJM03]
- Dagegen sind Vererbungs-Beziehungen zwischen Komponenten sehr problematisch.

Komponentendiagramme in der UML



Java Modularisierung

https://www.heise.de/developer/artikel/Modulare-Java-Zukunft-Das-Java-Platform-Module-System-erklaert-3700766.html

Modularisierung: Definierende Eigenschaften

- eindeutiger Name
- eindeutige Abhängigkeiten
- API / Information Hiding

Modularisierung in Java

- Klassen (*private*/*public*)
- Packages (*package private*/*protected*)
- JARs
  - Identifikation über Dateinamen
  - Keine Abhängigkeiten: Sichtbarkeit „innerhalb eines JARs“ nicht vorgesehen
  - aber: Abhängigkeiten werden über JARs definiert (bsp. Maven)

Java Modularisierung

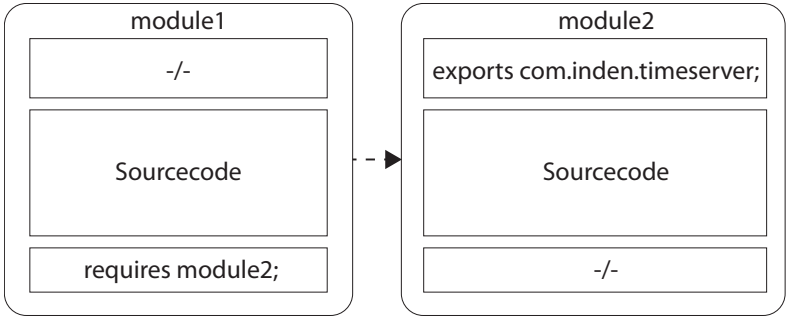
Konsequenz

- kein Information Hiding innerhalb von JARs
- Methoden, die innerhalb eines JARs sichtbar sind, müssen global sichtbar sein
- führte zu Sicherheitsproblemen im JDK, Wartbarkeitsproblemen in JUnit 4

Ausweg

- JARs können jetzt **Module** deklarieren
- Explizite Angabe von
  - exportierten Elementen
  - verwendeten Modulen
- Trennung von `Classpath` / `Module Path`

Modularisierung in Java 9 [Ind18]



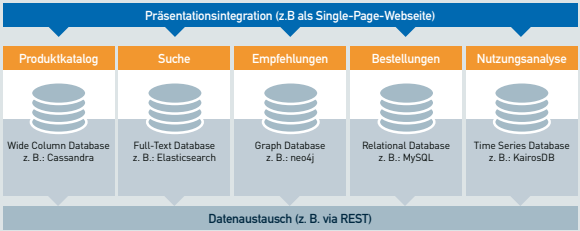
Microservice-Architekturen als Enabler von DevOps

Domänenschnitt und Bounded Context für Self-Contained Systems

Unabhängiges Deployment einzelner Microservices ist hier essentiell

Microservices nutzen Polyglot Persistence:

- Jeder Microservice verwaltet seine eigenen Daten um unabhängig deploybar und skalierbar zu sein.
- Eine Konsequenz ist „eventual consistency" der Daten

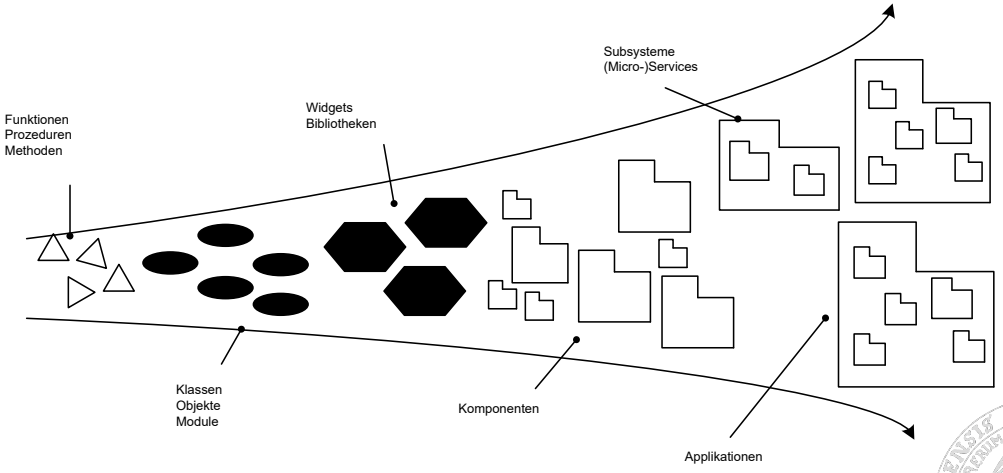


Beispiel für eine vertikale Aufteilung in Mircoservices

Container- und Cluster-Technologien für Microservices

- Container ermöglichen das leichtgewichtige Deployment von Microservices.
  → z. B. Docker oder RedHat Ansible Tower
- Cluster ermöglichen die Verteilung, Load Balancing und Skalierung u.a. in der Cloud.
  → z. B. Kubernetes

Granularität von Komponenten

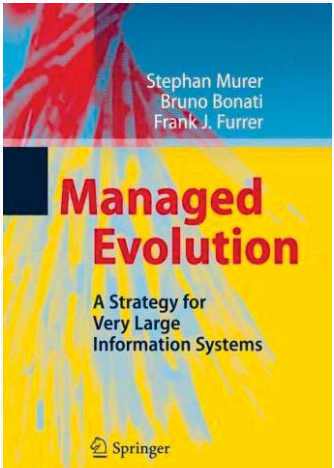


Detaillierungsgrad der Schnittstellen

- Die Schnittstelle stellt eine Abstraktion des Moduls für seine Kunden dar.
- Eine wichtige Entwurfsentscheidung ist der Detaillierungsgrad der Schnittstellen
  - Eine zu kleine Schnittstelle macht (große) Module schwer handhabbar (Eisberg-Analogie).
  - Eine zu große Schnittstelle macht Module kompliziert in der Benutzung und bringt evtl. zu viele Abhängigkeiten zwischen den Modulen.
  - Hier sind wichtige Entwurfsentscheidungen zu treffen, für die es kein ‚Kochrezept' gibt!

Managed Evolution of Very Large Systems Requires Stable Interfaces

- **Hard to replace a very large information system as a whole**
  - High cost (> CHF 1 bn, estimate for Swiss platform)
  - Development time too high (> 5 years)
  - High risk, as both technical and business prerequisites shift over time
- **Managed evolution is the only feasible approach**
  - Stepwise transformation of landscape, renewing component after component
  - Multi-year effort which gradually implements the target architecture
- **Well encapsulated components exposing managed interfaces are a prerequisite for managed evolution**
  - Technically renew components, without affecting clients
  - Consolidate redundant data and functionality behind common interfaces and simplify
  - Seamlessly operate a technically heterogeneous system resulting from the evolutionary approach



Übersicht

Übersicht

Keller als algebraische Spezifikation

ADT Stack[X];

Operationen:

empty:	Stack[X]	$\rightarrow$ BOOLEAN
new:		$\rightarrow$ Stack[X]
push:	Stack[X],X	$\rightarrow$ Stack[X]
pop:	Stack[X]	$\nrightarrow$ Stack[X]
top:	Stack[X]	$\nrightarrow$ X

Vorbedingungen:

pre pop (s:Stack[X]) = **not** empty (s)

pre top (s:Stack[X]) = **not** empty (s)

Regeln: For all e:X, s: Stack[X]:

empty (new ()) = true

not (empty (push (s,e))) = true

top (push (s,e)) = e

pop (push (s,e)) = s

Verschlüsselung als algebraische Spezifikation

ADT Ciphertext
ADT Plaintext
ADT Key

Operationen:

encrypt:	Plaintext $\times$ Key	$\rightarrow$	Ciphertext
decrypt:	Ciphertext $\times$ Key	$\rightarrow$	Plaintext

Regeln: For all p:Plaintext, k:Key:

decrypt(encrypt(p,k),k)=p

Was fehlt?

Algebraische Spezifikationen vs. Klassen

- Vorbedingungen für Methoden wurden spezifiziert.
  - Das geht z.B. in Eiffel, s.u.
- Die Vorbedingungen müssen in einer Implementierung für die entsprechenden Funktionen erfüllt sein, sonst muss eine Ausnahmebehandlung eingeleitet werden.
  - Diese Implementierungsaspekte werden mit algebraischen Spezifikationen nicht betrachtet.
- Die Operationen sind konstruierend (new), lesend (empty, top) oder transformierend (push, pop, encrypt, decrypt)
- Im Beispiel werden die Operationen pop und top zu partiellen Funktionen, was durch das Zeichen $\nrightarrow$ (statt $\rightarrow$) angegeben wurde.

Algebraische Spezifikation: Anmerkungen

- Spezifikation ist implementierungsunabhängig.
- Vollständig formale Spezifikation, daher für Verifikation und automatische Werkzeuge einsetzbar.
- Beispiel: Verifikation von kryptografischen Protokollen mit algebraischen Spezifikationen für Primitive
- Die Wirkung von Zugriffsoperationen ergibt sich durch wechselseitige Abhängigkeiten der Axiome, nicht durch isolierte Betrachtung.
- Erfordert mathematische Kenntnisse zum Erstellen und Verstehen.

Übersicht

Übersicht

Übersicht

Übersicht

Zur Historie:

- Bertrand Meyer
  *1950 in Paris
  Zurzeit Politecnico di Milano (Italy) und Innopolis University (Kazan, Russia), Vorher ETH Zürich, Monash University, Melbourne und Präsident der Firma ISE Inc. (Interactive Software Engineering) in Santa Barbara, USA
- Erfinder der Programmiersprache Eiffel (1985/86)
- Buch: [Mey01]

Was ist Eiffel?

- Eine objekt-orientierte Programmiersprache
- Unterstützt den Entwurf und die Implementierung
- Bietet die Möglichkeit, ADTs (fast) formal zu spezifizieren
- Unterstützt Design by Contract

Programmieren als Vertrag

Motivation:

- Wiederverwendung hängt ab von
  - Verständlichkeit:
    Sinn und Zweck einer Klasse müssen verstanden werden.
  - Sicherer Verwendbarkeit:
    Klassen müssen als korrekt und robust eingeschätzt werden können.
- Die Objekt-Orientierung liefert hier zunächst nur Klassennamen und Signaturen von Methoden.
  - Es fehlt die Spezifikation von Semantik.

Idee:

- Vor-/Nachbedingungen für den Aufruf einer Methode angeben, d.h. Semantik (teilweise) festlegen
- Klasseninvarianten definieren

Zusicherungen in Eiffel

Zusicherungen werden formuliert als prädikative Ausdrücke, z.B.:

- `n>0 and not (x = void)`
- `and` kann hier auch durch `;` ersetzt werden, z.B.:
  `n>0; not (x = void)`
- Durch `;` getrennte Ausdrücke können zur Dokumentation mit Namen versehen werden, z.B.:
  `Positiv: n>0; NichtLeer: not (x = void)`
- Zusicherungen können optional vom Laufzeitsystem überprüft werden.
  - Eine Verletzung führt zu einer Ausnahme.

Vor-/Nachbedingungen

- Eine Vorbedingung drückt die Eigenschaften aus, die beim Aufruf einer Methode immer gelten müssen, damit die Methode korrekt arbeiten kann.
  - Vorbedingungen werden durch das Schlüsselwort `require` eingeleitet, das dann von einer Zusicherung gefolgt wird.
  - Die Einhaltung der Vorbedingung ist eine Verpflichtung des Kunden.
- In Nachbedingungen wird festgelegt, welche Eigenschaften nach der Ausführung einer Methode gewährleistet werden.
  - Nachbedingungen werden durch das Schlüsselwort `ensure` eingeleitet, das dann von einer Zusicherung gefolgt wird.
  - Wenn ein Kunde die Vorbedingungen erfüllt, dann garantiert der Lieferant die Nachbedingungen.
- Vor-/Nachbedingungen spezifizieren damit teilweise die Semantik einer Methode als partielle Funktion.
- Achtung: Semantik dadurch nicht vollständig spezifiziert.

Klasseninvarianten

- Klasseninvarianten definieren globale Eigenschaften, die von allen exportierten (somit öffentlichen) Methoden einer Klasse eingehalten werden müssen.
- Sie gelten nach jedem Aufruf einer exportierten Methode der entsprechenden Klasse, einschließlich der Erzeugungsmethode.
- Sie müssen auch vor jedem Aufruf einer exportierten Methode gelten.
- Klasseninvarianten werden durch das Schlüsselwort `invariant` am Ende einer Klassendeklaration eingeleitet, das dann von einer Zusicherung gefolgt wird.
- Sie werden konjunktiv an die Vor- und Nachbedingungen gefügt.
- Sie sind Rahmenbedingungen für Verträge zwischen Kunden und Lieferanten.

Zusicherungen: Beispiel 1

```
class Buch
    export ...
    feature
    ...
        Seitenzahl: INTEGER;

        Seitenzahl_Setzen (t : INTEGER) is
            require t > 10
            do
            ...
            ensure Seitenzahl = t
        end; – Seitenzahl_Setzen
    ...

    invariant
        Seitenzahl > 10
        Publ_Dat > 1800; Publ_Dat <= 2018
    end; – Buch
```

Zusicherungen: Beispiel 2

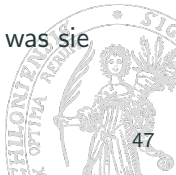
```
class Buch_Liste
    export Buch_Löschen
    feature
        B_Liste : ARRAY [Buch];

        Buch_Löschen (b : Buch) is
            require size (B_Liste) > 0
            do
                ...
            ensure
                    size (B_Liste) = old size (B_Liste) -1
        end; – Buch_Löschen
    end; – Buch_Liste
```

Ist das eine sinnvolle Kombination aus Vor- und Nachbedingung?

Korrektheit von Klassen

Notation: {P} A {Q} bedeutet, dass nach Ausführung von A Q gilt, falls vorher P galt.

- Eine Klasse wird als korrekt in Bezug auf seine Zusicherungen bezeichnet, falls gilt:
  - Für Create:
    { Default and preCreate } doCreate { INV }
  - Für alle exportierten Eigenschaften E (außer Create):
    { preE and INV } doE { postE and INV }
- Dabei ist Default die Zusicherung, dass die Attribute die Default-Werte erhalten:
  - Die Default-Werte müssen die Klasseninvariante einhalten, falls Create nicht vorhanden ist.
- INV ist die Klasseninvariante.
- Ist die Vorbedingung nicht erfüllt, so kann die entsprechende Methode tun was sie will.



Zusicherungen und Vererbung

Vertragsbedingungen werden an Erben weitergegeben, die Erben müssen mindestens den originalen Vertrag erfüllen:

- Ein Erbe (sozusagen der Zulieferer) muss mindestens die Leistung erbringen, die der Vorfahr als Lieferant versprochen hat.
- Ein Erbe darf höchstens die Vorraussetzungen fordern, die der Vorfahr als Lieferant gefordert hat.
- Eine Vorbedingung darf in einer Unterklasse somit nur disjunktiv erweitert werden:
  - Die Bedingung wird abgeschwächt.
- Eine Nachbedingung darf in einer Unterklasse somit nur konjunktiv erweitert werden:
  - Die Bedingung wird verschärft.

Vor-/Nachbedingungen und Vererbung

```
class Anbieter
feature
    Dienstleistung (x: INTEGER) is
        require x > 5
        do ...
        ensure x > 10
    end;
end;

class Zulieferer
inherit Anbieter
redefine Dienstleistung
feature
    Dienstleistung (x: INTEGER) is
        require x > 3
        do ...
        ensure x > 12
    end;
end;
```

A: Anbieter;
Z: Zulieferer;
...
A := Z;

A gibt durch polymorphe Zuweisung von Z die Verträge mit seinen Kunden an Z weiter.

Vor-/Nachbedingungen und Vererbung

```
        require PreO
O:      feature f
        ensure PostO

        require PreU
U:      redefine f
        ensure PostU
```

- Bei der Redefinition muss die Unterklasse den Vertrag der Oberklasse übernehmen.
  - Es muss gelten:
    PreU ist schwächer als PreO: PreO $\Rightarrow$ PreU
    PostU ist stärker als PostO: PostU $\Rightarrow$ PostO
- Das gilt insbesondere auch für die konkrete Definition von abstrakten Methoden.
- Die Prüfung der korrekten Vertragsübernahme ist i.a. nicht entscheidbar.

Klasseninvarianten und Vererbung

- Invarianten werden entlang der Vererbungsrelation verschärft:
  - Die Gesamtinvariante der Klasse InstitutInformatikBericht ist: #Autoren > 0 and #Seiten > 50 and #Referenzen > 50
- Die Randbedingungen für Verträge werden durch Vererbung komplexer.

Programmieren als Vertrag

	Pflicht	**Recht**
Kunde	Aufruf einer Methode nur mit Zustand und Parametern, die die Vorbedingungen erfüllen.	Die Methode erzeugt dann keine Ausnahme und erfüllt die Nachbedingungen.
Lieferant	Sicherstellen, dass eine Methode bei Aufruf mit erfüllter Vorbedingung korrekt arbeitet und am Ende die Nachbedingungen erfüllt.	Keine Fehlerabfrage notwendig, wenn Vorbedingung verletzt wird.

Programmieren als Vertrag

Zusicherungen

- legen (einige) semantische Eigenschaften von Methoden und Klassen fest;
- beschreiben einen Vertrag zwischen Kunde und Lieferant, also für Benutzt-Beziehungen;
- ergeben durch Vererbungsbeziehungen die Übernahme von Verträgen,
  - dadurch soll undisziplinierten Überschreiben vermieden werden;
- werden genutzt zur/zum
  - Dokumentation,
  - Inspektion, Review,
  - Erleichterung von Wiederverwendung,
  - Debugging,
  - Disziplinierte Ausnahmebehandlung.

Übersicht

Übersicht

Wie lässt sich das in der UML spezifizieren?

Die OCL ist eine

- Sprache, mit der Zusicherungen (Constraints) für OO-Modelle formuliert werden können.
- Formale Sprache typisierter Ausdrücke: OCL-Ausdrücke basieren auf Mengenlehre und Prädikatenlogik.
- Insbesondere auch zur Spezifikation von Schnittstellen geeignet.

Constraints sind gültige OCL-Ausdrücke vom Typ Boolean; dazu zählen:

- Invarianten von Klassen und Typen in Klassendiagrammen,
- Vor- und Nachbedingungen von Operationen / Methoden,
- Bedingungen, z.B. in Zustandsdiagrammen (Wächter, Guards)

Literatur zur OCL: [WK03].

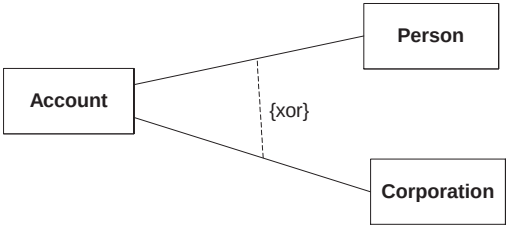
Was ist die OCL? (Forts.)

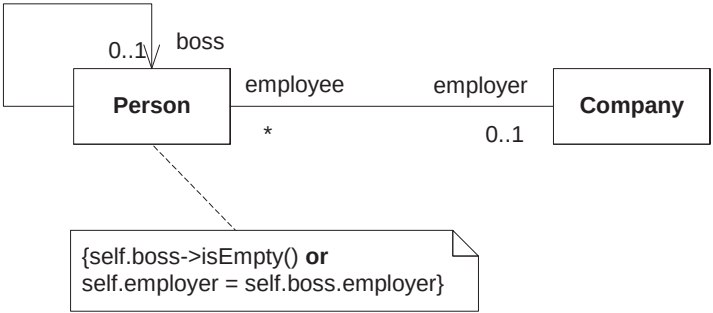
OCL ist eine deklarative Sprache, d.h. Ausdrücke haben keine Seiteneffekte:

- Auswertung von OCL-Ausdrücken liefert Werte zurück.
- Berechnung des Werts führt nicht zu Veränderungen des Modells, auf dem der zugehörige OCL-Ausdruck ausgewertet wird.
  - Zustand eines Systems (das Modell), das mittels OCL-Ausdrücken beschrieben wird, wird also nicht durch deren Auswertung verändert.
- Aber: OCL-Ausdrücke können genutzt werden, um Bedingungen für Zustandsveränderungen zu spezifizieren.
- Vergleich: Integritätsbedingungen für Datenbanken

Eher FO-Logik als Java! OCL ist eine Modellierungs- und Spezifikationssprache, d.h. Implementierungsaspekte können (und sollen) in OCL nicht formuliert werden.

OCL im Klassendiagramm



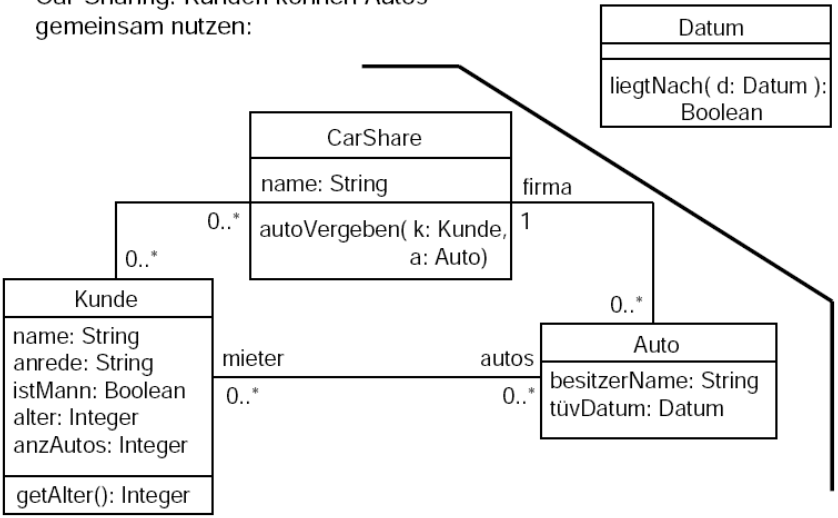


Kontext von OCL-Ausdrücken

- Jeder OCL-Ausdruck gilt im Kontext einer Instanz eines spezifischen Typs.
- In OCL-Ausdrücken kann mit dem Schlüsselwort self auf diese Instanz Bezug genommen werden.
  - Beispiel: Wenn ein OCL-Ausdruck im Kontext eines Typs „Person“ gilt, so bezeichnet self eine (aktuell betrachtete) Instanz des Typs „Person“.
- Kontext eines OCL-Ausdrucks kann ...
  - entweder durch gestrichelte Linie zwischen OCL-Ausdruck und Modellelement
  - oder durch explizite Spezifikation in OCL-Ausdruck angegeben werden.

Ein Beispiel

Car-Sharing: Kunden können Autos gemeinsam nutzen:



Invarianten für Attribute

Aussage: „Kunden müssen mindestens 18 sein.“:

```
Context: Kunde
inv: alter ≥ 18
```

- **Context** gibt die bezogene Klasse aus dem Modell an.
- Dann stehen Attribute und Methoden der Klasse zur Verfügung um Boolesche Ausdrücke (Constraints) zu bilden.

Zugriff auf Methoden

Auch Attribute, deren Typen Klassen sind, sind zugreifbar:

```
Context: Auto
inv: tüvDatum.liegtNach(<heute>)
```

Datum

Methode von Datum
Über **Punktnotation** ist Zugriff auf
Methoden und Attribute von Datum möglich

Achtung bei Methoden: In Constraints dürfen nur reine Abfragen verwendet werden, d.h. Methoden, die Klassen nicht modifizieren!

Zugriff auf assoziierte Klassen

- Einfacher Fall: Assoziation eindeutig
  $\Rightarrow$ Bezug ist klar
- Was, wenn Multiplizität größer 1 ist?
  $\Rightarrow$ Keine Aussagen über einzelne Objekte möglich, aber über Menge
- Auf Mengen sind spezielle Operationen definiert, auf die mittels Pfeilnotation zugegriffen wird.
- Aussage: „Ein Kunde darf nicht mehr als 5 Autos mieten.“

```
Context: Kunde
inv: autos->size ≤ 5
```

- Zugriff erfolgt über den entsprechenden Rollennahme an der Assoziation.
- Damit sind auch Aussagen über Eigenschaften der Elemente möglich (s.u.).

Kontext für Invarianten

Verbindung zum Modell wird über den Kontext hergestellt:

Syntax: Context: <Klassenname>
inv: <Constraints>

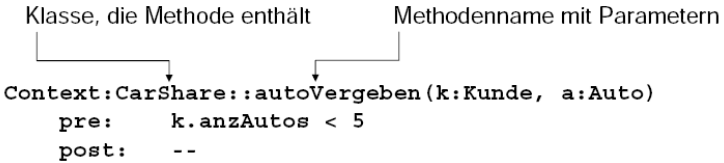
Danach sind die sog. Eigenschaften der Klasse (Attribute; Methoden, welche keine Seiteneffekte haben; Assoziierte Klassen) in den Constraints verwendbar.

- Schlüsselwort `self` zum expliziten Bezug auf Klasse (ist optional)

→ Invariante muss für jede Instanz der Klasse gelten.

Methoden und ihre Vor- / Nachbedingungen

- Aussagen über das Verhalten einer Methode
- Beispiel: „Ein Kunde soll nicht mehr als 5 Autos bekommen.“



- Parameter der Methode sind direkt zugreifbar.
- Attribute und Methoden der umgebenden Klasse sind mittels
  `self.`<Attribut- / Methodenname>
  zugreifbar.

Nachbedingungen

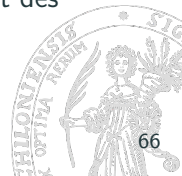
- Besonderheit bei Nachbedingungen: Zugriff auf den Zustand eines Teilausdrucks vor dem Methodenaufruf mit Suffix @pre

```
Context:CarShare::autoVergeben(k:Kunde,a:Auto)
   pre:  k.anzAutos < 5
   post: k.anzAutos = k.anzAutos@pre + 1
```

Wert von k.anzAutos vor Aufruf von autoVergeben(...)

Wert von k.anzAutos nach Aufruf von autoVergeben(...)

- @pre bezieht sich nur auf den direkt vorangehenden Teilausdruck. Der Rest des Ausdrucks bezieht sich auf die Objekte nach dem Methodenaufruf.



Kontext für Vor- / Nachbedingungen

Kontext für Vor- / Nachbedingungen ist die bezogene Methode:

Syntax: Context:<Klassenname>::<Methodenname>
(<Parm1>:<Klasse1>...):<Ergebnistyp>
pre: <Constraints für Vorbedingung>
post: <Constraints für Nachbedingung>

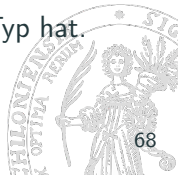
Danach sind

- übergebene Parameter
- Ergebnis der Methodenauswertung mittels Schlüsselwort `result`
- Eigenschaften der umgebende Klasse mittels Schlüsselwort `self`

zugreifbar.

Anmerkungen zur OCL i

- Die OCL ist eine formale Sprache um Ausdrücke auf UML Diagrammen zu beschreiben. Mit ihrer Hilfe kann man typischerweise Invarianten einbringen, welche semantisch in den UML Diagrammen nicht auszudrücken sind.
- OCL ist eine Spezifikationssprache, d.h. dass ein OCL Ausdruck insbesondere keine Seiteneffekte hat. OCL Ausdrücke liefern nur Ergebnisse und ändern nicht(!) das Modell.
- OCL ist keine Programmiersprache. Aus diesem Grunde eignet sie sich auch nicht dazu Programmlogik in die Modelle einzubringen.
- OCL ist eine typisierte Sprache, sodass jeder OCL Ausdruck einen festen Typ hat.



Anmerkungen zur OCL ii

- Die OCL nutzt einerseits gewisse Schlüsselwörter und Ausdrücke (and, or, <>, @pre, etc.) andererseits die im Modell vorkommenden Typen mitsamt ihrer Attribute und Operationen.
- Umgangssprachliche Ausdrücke sollten in einem OCL Ausdruck nicht vorkommen.
- Wird auf Java-Typen gearbeitet (java.util.Date beispielsweise), so müssen auch Java-Methoden verwendet werden.
- Es gibt zwei Arten von Zugriffsoperatoren. Mit . wird auf die Attribute und Operationen von Objekten zugegriffen. Mit -> wird auf Collections (beispielsweise Mengen oder Listen) gearbeitet.
- Der Ausdruck self bezieht sich auf das Objekt in dem gegebenen Kontext.

Modellierung mit der UML (Gesamtüberblick für dieses Modul)

Statische Struktur	Klassendiagramme
	Komponentendiagramme
	Verteilungsdiagramme
Struktur der Spezifikation	Paketdiagramme
Funktionen	Anwendungsfallbeschreibungen
Dynamik	Zustandsautomaten
	Protokollautomaten
	Aktivitätsdiagramme
	Sequenzdiagramme
Bedingungen	OCL

Literatur:

UML [RQ+12; Oes11; Rum11; Sei+12]

Klassiker [BRJ98; RBJ98; JBR99]

Übersicht

Übersicht

Lernziele

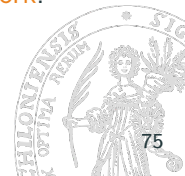
- Modularität würdigen
- Unterschied zwischen syntaktischen und semantischen Spezifikationen kennen
- Das Vertragsmodell verstehen
  - Eiffel
  - Java
  - OCL

References i

- Booch, Grady, James Rumbaugh und Ivar Jacobson (1998). The Unified Modeling Language User Guide. Addison-Wesley.
- Ghezzi, C., M. Jazayeri und D. Mandrioli (2003). Fundamentals of Software Engineering. 2nd. Prentice Hall.
- Hoffman, D.M. und D.M. Weiss, Hrsg. (2001). Software Fundamentals, Collected Papers by David L. Parnas. Addison-Wesley.
- Inden, Michael (2018). Java 9 - Die Neuerungen. dpunkt.verlag.
- Jacobson, Ivar, Grady Booch und James Rumbaugh (1999). The Unified Software Development Process. Addison-Wesley.
- Meyer, B. (2001). Object-oriented Software Construction. 2nd edition. Prentice Hall.

References ii

- Oestereich, Bernd (2011). Analyse und Design mit UML 2.5. 11., umfassend überarbeitete und aktualisierte Neuauflage. Oldenbourg Verlag. ISBN: 978-3-486-71667-2.
- Rumbaugh, James, Grady Booch und Ivar Jacobson (1998). The Unified Modeling Language Reference Manual. Addison-Wesley.
- Rumpe, Bernhard (2011). Modellierung mit der UML. 2. Auflage. Springer-Verlag.
- Rupp, C., S. Queins u. a. (2012). UML 2 glasklar: Praxiswissen für die UML-Modellierung. 4. Auflage. Hanser. ISBN: 978-3-446-43057-0.
- Seidl, Martina, Marion Brandsteidl, Christian Huemer und Gerti Kappel (2012). UML@Classroom: Eine Einführung in die objekt-orientierte ModellierungWork. dpunkt Verlag.



References iii

- Szyperski, C. (Second 2002). Component Software: Beyond Object-Oriented Programming. ISBN 0-201-74572-0. Adiison-Wesley.
- OMG Unified Modeling Language Superstructure Version 2.2 (Feb. 2009). http://www.omg.org/spec/UML/2.2/Superstructure/PDF/.
- Warmer, Jos und Anneke Kleppe (2003). The Object Constraint Language: Getting Your Models Ready for MDA. Second. Addison-Wesley.