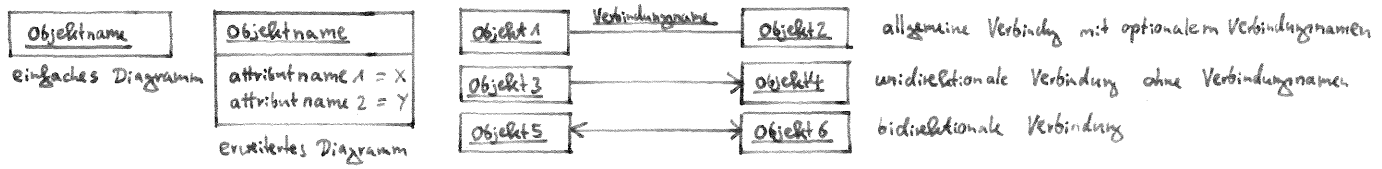
Objekte

- Objekt := materielle oder immaterielle Entität der Realwelt mit Zustand (Eigenschaften) und Operationen
- In Softwareentwicklung: Abstraktion von Objekten, die nur für den Kontext relevante Eigenschaften enthält
- Zustand eines Objekts = Konkrete Wertebelegung der Attribute
- Operationsarten
  - ↳ Modifikatoren: ändern den Zustand eines Objekts
  - ↳ Selektoren: liefern nur Attributwerte und verändern den Zustand somit nicht
- Identität: jedes Objekt besitzt eine eindeutige, zustandsunabhängige Identität
- Name: Teilmenge der Attributmenge (analog Datenbankschlüssel)
  Namensarten
  - ↳ indirekt: Objektreferenzen
  - ↳ anonym: ausschließlich indirekte Namen
  - ↳ Alias: mehrere indirekte Namen auf das identische Objekt
- Objektgleichheit
  - ↳ Zustands- bzw. Wertegleichheit: Objekte mit selbem Verhalten und Attributen besitzen übereinstimmende Attributwerte
  - ↳ referentielle Gleichheit: Referenzen sind Aliase auf ein Objekt
- Objekte klonen: erzeuge ein zustandsgleiches Objekt mit eigener Identität

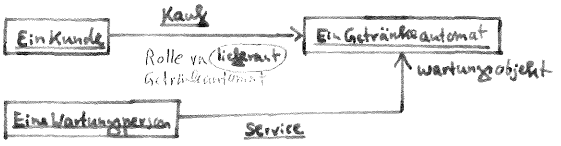
Verbindung

- Motivation: Dienstnutzer (client) nutzt Operationen eines Dienstleisters (supplier)
  - ↳ Voraussetzung: Dienstnutzer kennt Dienstleister
- Verbindungsarten:
  - unidirektional: A kennt B, aber B kennt nicht A
  - bidirektional: A kennt B, B kennt A
- Rolle bzgl. Verbindung := Teilmenge der Angebotenen Objektoperationen, definiert das Verhalten gegenüber verbundenen Objekten

UML



Beispiel:



Klassen

- Klasse := Abstraktion von Objekten mit identischen Eigenschaften
  - ↳ beschreibt Objektmenge mit denselben Attributen und identischem Verhalten
- Klassen werden durch ihren Namen identifiziert (CamelCase)
- Instanz := von einer Klasse beschriebenes Objekt
- Eine Klasse definiert einen abstrakten Datentyp (abstrahiert von Identität, Zustand und Existenz)

Attribute

- Attributsspezifikation: Name (klein geschrieben) und Typart (in UML optional)
  - ↳ elementar (int, real, boolean ...)
  - ↳ Aufzählungstyp (enumeration type)
  - ↳ öffentliche Schnittstelle einer Klasse bzw. Interface
- mögliche zusätzliche Angaben:
  - ↳ Sichtbarkeit
  - ↳ Defaultwert
  - ↳ unbeschränkt änderbar / Konstant

  Beispiel: public menge Integer = 0 {changeable}
- abgeleitete Attribute := Attribute, die aus anderen Attributen berechnet werden (z.B. dauer = ende - start)
  - ↳ UML: Kennzeichnung mit /
  - ↳ Keine Aussage über physische Speicherung oder Berechnung
- Klassenattribut := Attribut existiert genau einmal für alle Instanzen einer Klasse
  - ↳ UML: Kennzeichnung mit unterstrichenem Attributnamen

Operationen

- Operationsarten
  - ↳ Modifikatoren: Aufruf verändern den Zustand der Instanz
  - ↳ Selektoren: Aufruf fragt Zustand ohne Änderung ab („isQuery")
- Operationssignatur: [Sichtbarkeit] Name ([Übergabeart] [Parametername] : [Parametertyp]) : [Rückgabetyp] [{Operationsart}]

  Beispiel: public monatsumsatz (in abrMonat : Monat) : Geld {isQuery}
- Übergabearten für Parameter
  - ↳ in: Wertparameter → Kann von Operation nicht modifiziert werden
  - ↳ inout: Referenzparameter → Kann von Operation modifiziert werden
  - ↳ out: nur Rückgabe als Referenz
- Klassenoperation := Operation existiert auf Klassenebene. Ausführung ist ohne Instanz möglich → Kein Zugriff auf Instanzattribute!
  - ↳ MML: Kennzeichnung durch Unterstreichen
- Standardoperationen: Existenz wird ohne explizite Angabe in MML-Diagrammen angegeben
  - ↳ Instanzoperationen

    gibAtt() : T setzeAtt(in wert : T)

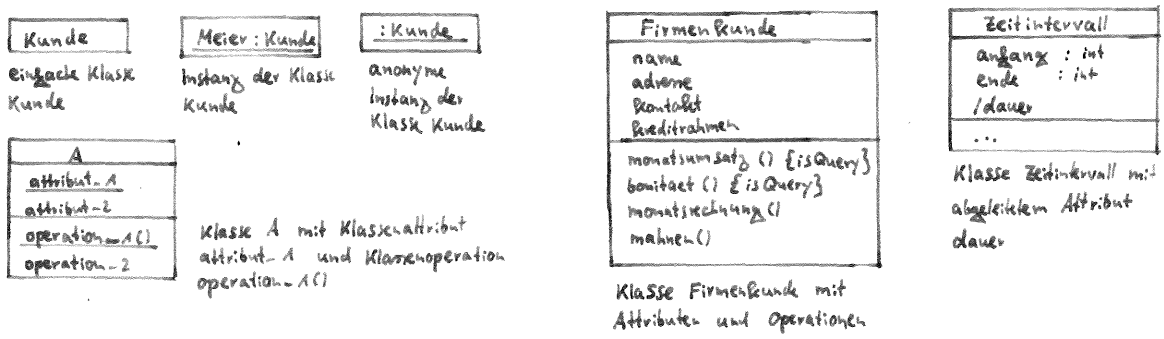
    gibAss()

    VerbindeAss(in einB : B) loeseAss(in einB : B)

    zerstoere()
  - ↳ Klassenoperation

    erzeuge()

UML: Attribute & Operationen



Assoziation

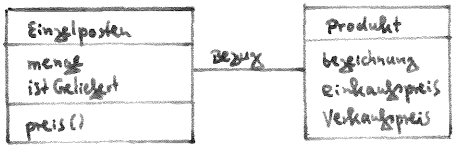
- Assoziation := Verbindung von Instanzen einer Klasse zu Instanzen derselben oder anderer Klasse
  - ↳ „reflexive Assoziation"
- MML: analog Objektverbindung inkl. Multiplizität
- bei zwei miteinander verbundenen Instanzen muss mindestens eine Instanz die andere Instanz kennen
- Multiplizität: Gibt eine obere / untere Grenze für die Anzahl der jeweiligen verbundenen Instanzen an
  - ↳ 1..1 genau eine verbundene Instanz (Kurz: 1)
  - ↳ 0..* beliebig viele verbundene Instanzen (Kurz: *)
  - ↳ 0..1 Keine oder eine verbundene Instanz
  - ↳ 1..* beliebig viele verbundene Instanzen, mindestens eine

  Abkürzungen:
  - ↳ 1:1-Assoziation: beide Assoziationsenden haben die Multiplizität 1 oder 0..1
  - ↳ 1:n-Assoziation: ein Assoziationsende hat Multiplizität 1 oder 0..1, ein Assoziationsende hat Multiplizität 0..n, n oder *
  - ↳ n:m-Assoziation: Sonst

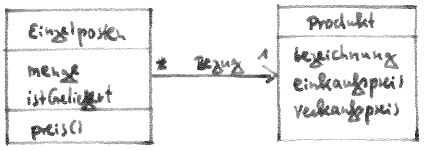
  Situative Notation:
  - ↳ exakte Zahlen
  - ↳ abgeschlossene Intervalle
- Konvention: Assoziation ist Tupelmenge → Zwei Objekte können auch bzgl. einer mehrwertigen Assoziation nur jeweils eine Verbindung miteinander eingehen
- abgeleitete Assoziation: indirekte Assoziation, die sich aus beteiligten Assoziationen ableitet (Beispiel: Geschwister ableitbar aus Kind-von)
- Assoziationsklasse: modelliert Assoziationen mit zusätzlichen Attributen, Operationen und anderen Elementen

  Wichtig: Jede Assoziationsklasse repräsentiert genau eine Assoziation!
- Anmerkung: Steht Klasse A in mehr als einer Assoziationsbeziehung, so können die Instanzen von A unterschiedliche Rollen für die Instanzen der assoziierten Klassen spielen

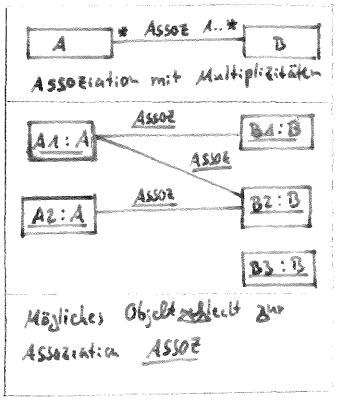
UML

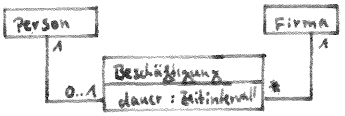


ungerichtete Assoziation „Bezug" zwischen Klassen

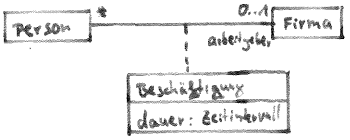


Einseitig navigierbare 1:n Assoziation

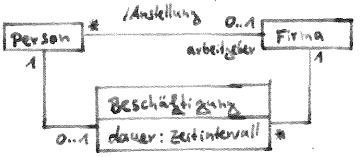




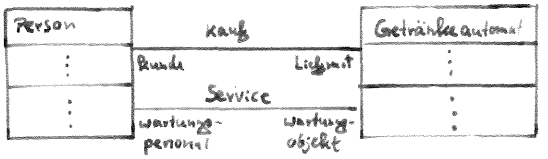
Attributierung einer Assoziation mit einer regulären Klasse



Assoziation Beschäftigung mit Assoziationsklasse



abgeleitete Assoziation „Anstellung"

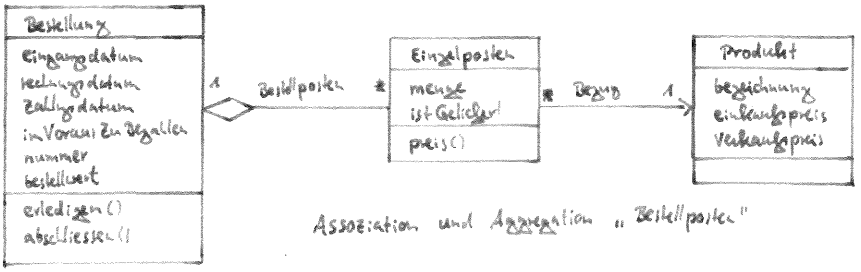


unterschiedliche Rollen bzgl. verschiedener Assoziationen

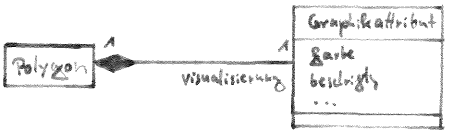
Aggregation & Komposition

- Aggregation := spezielle „Teil-Ganzes"-Assoziation, die eine Hierarchie aus der verbundenen Instanzen definiert
  - ↳ Ganzes-Instanz hat Verantwortung für Teil-Instanzen
- Komposition := Aggregation mit folgenden Einschränkungen:
  - ↳ Teilobjekte dürfen nur durch Operationen der Ganzes-Klasse entfernt oder ausgetauscht werden
  - ↳ Teilobjekte dürfen nicht Teil mehrerer Kompositionen sein
  - ↳ Teilobjekte werden beim Zerstören des Ganzes-Objekts kaskadierend mitzerstört

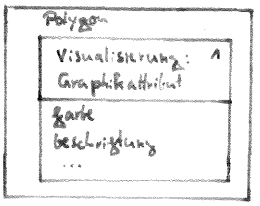
UML



Assoziation und Aggregation „Bestellposten"



Darstellung einer Komposition im Klassendiagramm

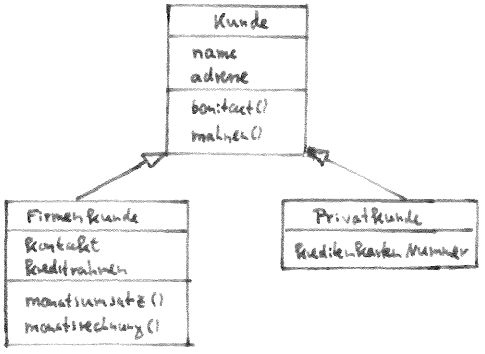


Alternative Darstellung der Komposition

Generalisierungsbeziehungen

- Generalisierung := Beziehung zwischen allgemeiner Oberklasse und spezieller Unterklasse
  - ↳ alle Merkmale (Attribute, Operationen und Assoziationen) der Oberklasse steht den Unterklassen zur Verfügung
  - ↳ Substitutionsprinzip: Jede Instanz der Unterklasse ist auch Instanz der Oberklasse
- Nutzen: Attribute und Operationen spezieller Klassen in allgemeinen Klassen zusammenfassen

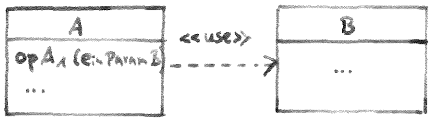
UML



Abhängigkeit

- Abhängigkeit := gerichtete Beziehung zwischen abhängigen und unabhängigen Elementen
  ↳ ohne weitere Angabe: Modellelemente hängen „irgendwie" von anderen Elementen ab bzw. abhängige Elemente „kennen" unabhängige Elemente
- Arten von Abhängigkeiten
  - ↳ Benutzungsabhängigkeit (<<use>> - dependency)
  - ↳ Realisierungsabhängigkeit (<<realizes>> - dependency)

UML

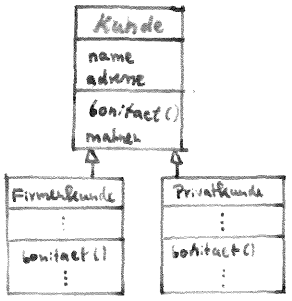


Sichtbarkeit

- Modifikatoren für Attribute & Operationen
  - ↳ Public (UML: +)
  - ↳ Private (UML: -)
  - ↳ Protected: nur in definierender Klasse und ihren Unterklassen sicht- und manipulierbar (UML: #)
- Schnittstellenarten
  - ↳ öffentliche Schnittstelle: alle öffentlichen Attribute & Operationen
  - ↳ private Schnittstelle: interne Realisierung
  - ↳ geschützte Schnittstelle: Modifizieren von Realisierungsdetails in Unterklasse

Abstrakte Klassen

- besitzt mindestens eine abstrakte Operation
  - ↳ Klasse gibt lediglich Existenz und Signatur der Operation vor
  - ↳ Operation wird in Unterklassen (reguläre Klassen) implementiert
- keine Instanziierung möglich
- UML: Namen abstrakter Klassen und Operationen werden kursiv geschrieben oder mit Präfix abstract gekennzeichnet

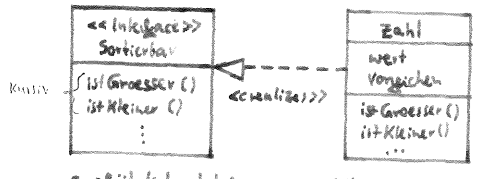


Abstrakte Klasse Kunde mit abstrakter Operation bonitaet() und regulären Unterklassen

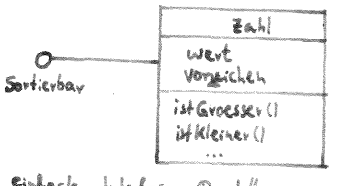
Interfaces

- besitzen keine Attribute, keine Assoziationen und geben lediglich Operationssignaturen an
- keine Instanziierung
- UML: Klasse mit Stereotyp <<interface>> oder kleiner Kreis mit Interfacenamen daneben

Beispiel: <<realizes>> - Abhängigkeit

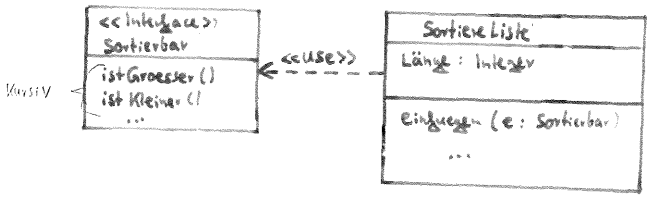


ausführliche Interface-Darstellung



einfache Interface-Darstellung

Beispiel: <<use>> - Abhängigkeit



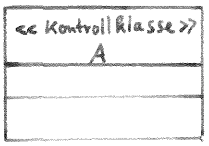
Klasse SortierteListe nutzt Interface als Parametertyp (auch möglich: Attributtyp)

Nutzen abstrakter Klassen und Interfaces

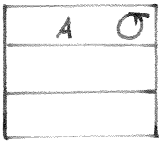
- Strukturierung von Modellen & Software
  - ↳ Gemeinsamkeiten extrahieren
  - ↳ Vorgaben für Schnittstellen definieren
- Einschränkung der öffentlichen Schnittstellen von Klassen mittels Interfaces

Stereotype

- Stereotyp := Satztextuell spezifizierter Charakteristika / Zusicherungen zu Klassen oder Assoziationen
- Stereotyp wird durch eindeutige Bezeichner und optionales graphischen Symbol identifiziert werden
- UML:



Mit Stereotypbezeichner



Mit graphischen Symbol

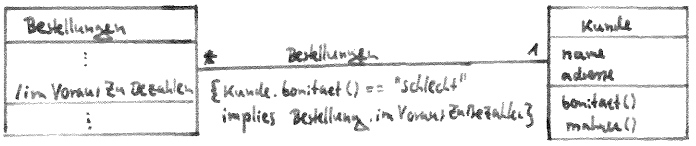
Darstellungsvarianten einer mit << Kontrollklasse >> stereotypisierten Klasse A

Zusicherungen

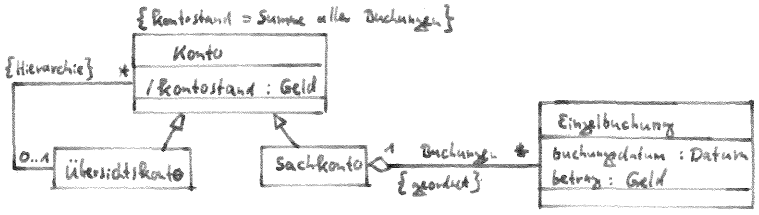
- Zusicherungen (Constraints) := Vorschriften für (Mengen von) Modellierungselemente
  - ↳ UML: { Constraint }
    - ↳ Umgangssprache
    - ↳ Pseudocode
    - ↳ Object Constraint Language (OCL)

- Beispiele:

↳ OCL: constraint (Bestellungen):
{ Kunde.bonitaet() == "schlecht" implies Bestellung.imVoraus Zu Bezahlen }



↳ Umgangssprache:
- { geordnet }: mehrwertige Assoziation als lineare geordnete Menge
- { Kollektion }: mit möglichen Duplikaten
- { Hierarchie }: hierarchische Ordnung
- { DAG }: gerichteter, azyklischer Graph
- { Oder }: Klasse hat Assoziationen zu mehreren anderen Klassen, aber stets nur eine Verbindung pro Instanz

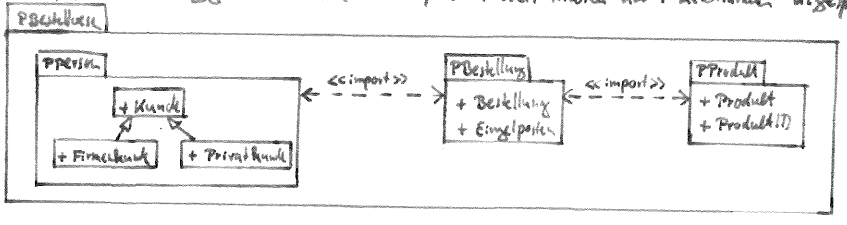


Textuelle Spezifikation

- listet Verantwortlichkeiten und Features einer Klasse auf
- Operationssemantik wird präzisiert durch:
  - ↳ Vorbedingungen: Welche Voraussetzungen müssen vor einem Operationsaufruf gelten?
  - ↳ Nachbedingungen: definiert das Ergebnis unter Annahme erfüllten Vorbedingungen (bezieht sich i.d.R auf Objektzustand bzw. Parameter)
  - ↳ Klasseninvariante: erlaubt einmalige Notation gemeinsamer Teile aller Vor- und Nachbedingungen. Gilt vor und nach jede öffentliche Operation der Klasse
- Beispiel: S. 111 →

Pakete

- Paket := Menge von Modellierungselementen, insbesondere auch andere Pakete
- Jedes Modellierungselement ist höchstens einem Paket zugeordnet → Pakete sind disjunkt
- Paket bildet einen eigenen Namensraum
- Sichtbarkeit
  - ↳ öffentlich (UML: +): von Paket exportierte Modellierungselemente
  - ↳ paketsichtbar (UML: ~): Modellierungselement ist innerhalb des Paketes und aller hierarchisch enthaltenen Pakete sichtbar
  - ↳ privat (UML: −)
- Abhängigkeit
  - ↳ Import-Abhängigkeit: Namen aller öffentlichen Elemente des importierten Pakets werden in den Namensraum des importierenden Pakets übernommen
  - ↳ Access-Abhängigkeit: Elemente des anderen Pakets müssen über Pakетnamen angesprochen werden

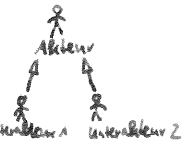


! Achtung: Keine automatische Sichtbarkeit von Unterpaketen in höheren Paketen
→ Modellierung von Import-Abhängigkeiten explizit notwendig !

Anwendungsfalldiagramm

Akteur

- Rolle := Menge von Funktionalitäten, die Benutzern menschlichen oder maschinellen Nutzern eines Anwendungssystems zur Verfügung stehen
- Akteur := Abstraktion einer Rolle von Nutzern aus der Realwelt
- Zwischen Akteuren können Generalisierungsbeziehungen bestehen
- UML:



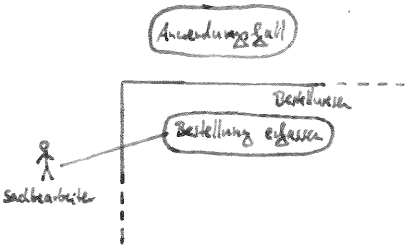
Anwendungsfall (use case)

- Anwendungsfall := abgeschlossene Teilfunktionalität des Anwendungssystems, die für mindestens einen Akteur ein Ergebnis liefert
- Anwendungsfälle beschreiben extern beobachtbare Funktionen des Anwendungssystems
  - ↳ Funktionalitäten
  - ↳ Abläufe
  - ↳ Ausnahmebehandlungen
  - ↳ Geschäftsregeln
  - ↳ externe Schnittstellen
- Anwendungsfälle werden mit den Anwendern ermittelt → Ergebnis: textuelle Beschreibung aus Sicht der Akteure → Anwendungsfalldiagramm
  - ↳ später oft. um Vor- und Nachbedingungen ergänzt
- spielen zentrale Rolle in Kommunikation Anwender und Analyst
- textuelle Beschreibung
  - Beschreibung normaler Ablauf (main flow)
  - Beschreibung abweichender Abläufe in eigenen Abschnitten
    - ↳ alternative Abläufe (alternative flow): Anwendungsfall wird weiterhin erfolgreich beendet, also Nachbedingung ist erfüllt
    - ↳ Ausnahmebehandlung (exceptional flow): erfolgreicher Abschluss ist gefährdet, Angabe einer geänderten Nachbedingung

  ! Keine UML-Vorgabe für textuelle Beschreibung !
- Szenario := Instanziierung eines Anwendungsfalls (= Funktionalität für vorgegebenen Zustand des Anwendungssystem präzisieren)
- Beispiel textuelle Beschrg.:

use case Kunde deaktivieren
actors
Sachbearbeiter
precondition
–
main flow
Der Sachbearbeiter bestimmt den zu deaktivierenden Kunden anhand der Kundennummer. Die Deaktivierung muss vom Sachbearbeiter bestätigt werden.
alternative flows Kundennummer unbekannt
Der Sachbearbeiter sucht zunächst den zu deaktivierenden Kunde aus einer Liste aller Kunden aus und deaktiviert ihn dann.
postcondition
Der Kunde ist deaktiviert.
exceptional flows Offene Rechnung
Hat der angegebene Kunde noch Rechnungen offen, so daß er nicht deaktiviert werden.
postcondition
Keine Deaktivierung durchgeführt.
exceptional flows Falsche Kundennummer
Zur angegebenen Kundennummern existierte kein Kunde, so daß auch keine Deaktivierung vorgenommen werden kann.
postcondition
Keine Deaktivierung durchgeführt
end Kunde deaktivieren

- UML:



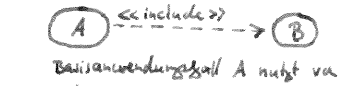
Assoziation zwischen Akteur & Anwendungsfall im Beispiel

Beziehungen zwischen Anwendungsfällen

- Komplexe Anwendungsfälle oder große Anzahl macht Strukturierungen notwendig:
  - ↳ Variante 1: logisch zusammenhängende Anwendungsfälle in Pakete zusammenfassen
  - ↳ Variante 2: include- und extend-Beziehungen (füge Zusatzfunktionen zu einer Basisfunktion hinzu)
    - + Redundanzvermeidung
    - + Strukturierung komplexer Anwendungsfälle
    - + Darstellung von Abhängigkeiten zwischen Anwendungsfällen

include-Beziehung

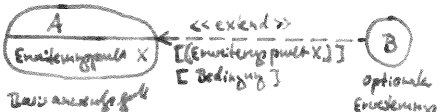
- Situation: verschiedene Anwendungsfälle beinhalten identische Teilfunktionalität
  - ↳ wird in separatem Anwendungsfall beschrieben
  - ↳ Basisanwendungsfälle nutzen separate Beschreibung



Basisanwendungsfall A nutzt von B beschriebene Teilfunktionalität

extend-Beziehung

- Basisanwendungsfall enthält Erweiterungspunkt (extension point) und kann unter bestimmten Bedingungen durch die von Anwendungsfall B beschriebene Funktionalität erweitert werden

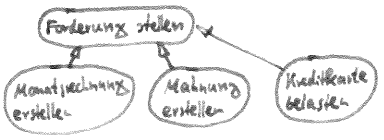


Basisanwendungsfall

optionale Erweiterungs-Anwendungsfall

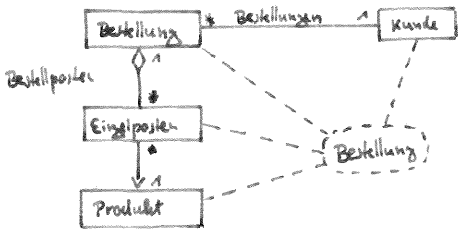
Generalisierungsbeziehung

- spezieller Anwendungsfall enthält die Funktionalität des allgemeineren Anwendungsfalls, aber beschreibt einige Teilaufgaben in spezieller Art und Weise
- UML:



Mechanismen

- Mechanismus := partielles Klassendiagramm, in dem der Anwendungsfall gestrichelt angedeutet und durch gestrichelte Linien mit allen beteiligten Klassen verbunden ist



- Beispiel für komplexes Anwendungsfalldiagramm: Abb. 13.11, S. 131

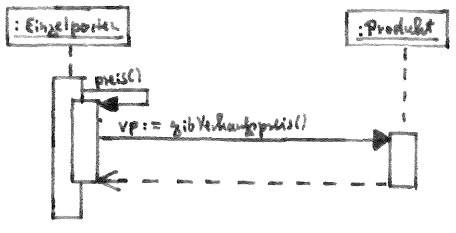
Interaktionsdiagramme

- Interaktionsdiagramme präzisieren den Ablauf von Anwendungsfällen und Operationen
- jedes Interaktionsdiagramm erfasst einen konkreten Ablauf (Zustand des Systems zu Beginn der Operationsausführung, aktuelle Parameter)
- Stärke: Einfachheit in Relation zur Ausdrucksmächtigkeit
  - ↳ Wichtig: Interaktionsdiagramme möglichst einfach halten → möglichst ein Diagramm für den Hauptablauf und wesentlichen Varianten
    - → bei vielen Schleifen und Fallunterscheidungen: verwende gesonderte Diagramm für konkrete Abläufe (Szenarien)
- Arten von Interaktionsdiagrammen
  - ↳ Sequenzdiagramme (ablauforientiertes Verhalten)
  - ↳ Kollaborationsdiagramm (Zusammenhang strukturellen und ablauforientierten Modell)

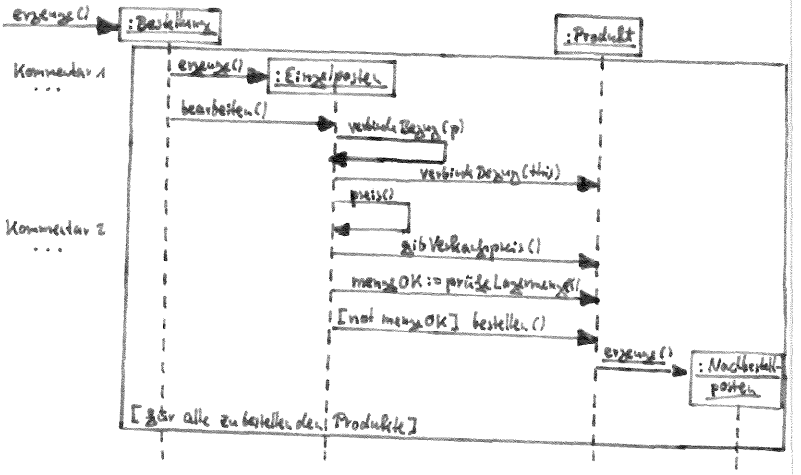
Sequenzdiagramme

- modelliert konkreten Ablauf von Komplexen Operationen im Klassenmodell unter Einbeziehung der beteiligten Objekte
- Objekte als Rechteck mit gestrichelter vertikaler Lebenslinie (Erzeugen durch Pfeil, zerstören durch Kreuz auf Lebenslinie)
- Operationen als Pfeil → von Dienstnutzer zu Dienstleister, Rückmeldung optional mit ←- -
- Rückgabewerte von Operationen durch Variablenzuweisung weiterverwendbar
- Kommentare zu Dokumentationszwecken in Textfeld links vom Sequenzdiagramm einfügen (optional)
- Kontrollinformation
  - Bedingung: [boolscher Ausdruck] Operationsname()
  - Iteration: * [Iterationsausdruck] Operationsname() ; Beispiel: * [i := 1..10] messwert erfassen(i)
    - mehrere Operationsaufrufe pro Iteration zu einer Folge zusammenfassen (Rechteck mit Iterationsausdruck in linker unteren Ecke)
- Nachrichtenarten
  - ↳ synchrone Nachricht: Aufrufer der Operation wartet untätig auf Ergebnis (Symbol: →)
  - ↳ asynchrone Nachricht: mehrere Objekte können gleichzeitig Operationen ausführen; Aufrufer kann parallel zum Empfänger weiterarbeiten (Symbol: →)
- Aktivitätselemente
  - ↳ Aktivierungsbalken := Zeitintervalle vom Nachrichtenversand bis zum Erhalten einer Antwort (Symbol: ▯)
  - ↳ aktives Objekt := Objekt mit paralleler Ablaufkontrolle zu anderen Objekten bei asynchroner Nachricht (Symbol: fetter Rechteck: ▮)
    - ↳ Aktivierung eines Objekts: Pfeil zeigt auf Anfang des Aktivierungsbalkens
    - ↳ Nachricht an bereits aktiviertes Objekt: Pfeil endet innerhalb des Aktivierungsbalkens
- UML:

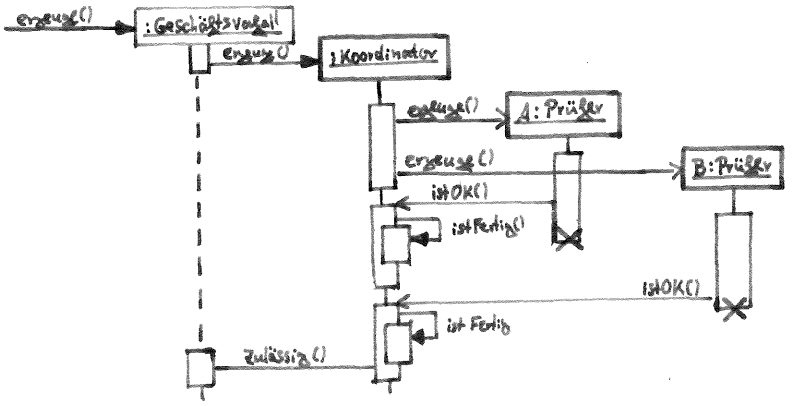
Iteration eines Aufrufs mit Zerstörung der beteiligten Instanzen



Sequenzdiagramm mit Aktivierungsbalken mit expliziter Rückkehr nach Operationsabschluss



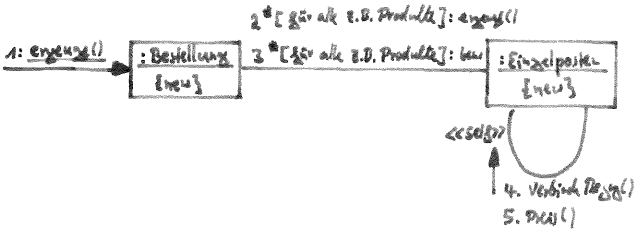
Iterativer Aufruf einer Operationsfolge mit Objekterzeugung, Selbstdelegation (preis()), Variablenzuweisung und Bedingung



Prüfung eines Geschäftsvorfalls mithilfe asynchroner Nachrichten

Kollaborationsdiagramme

- Kollaborationsdiagramme := Objektdiagramme mit zusätzlicher Beschreibung des ablauforientierten Verhaltens
- Nachrichtenaustausch wird mittels Verbindungen modelliert
- Keine Zeitachse → zeitliche Abfolge durch zeitliche Nummerierung verdeutlichen
  - ↳ Ordinalzahlen
  - ↳ hierarchische Dezimalnotation
- Kennungen: {new} Objekt wird während des Ablaufs erzeugt
  {destroyed} Objekt wird während des Ablaufs zerstört
  {transient} Objekt wird während des Ablaufs erzeugt und zerstört



Zustandsdiagramme

- Zustandsdiagramme := Modellierung des zustandsorientierten Verhaltens von Objekten

Zustand

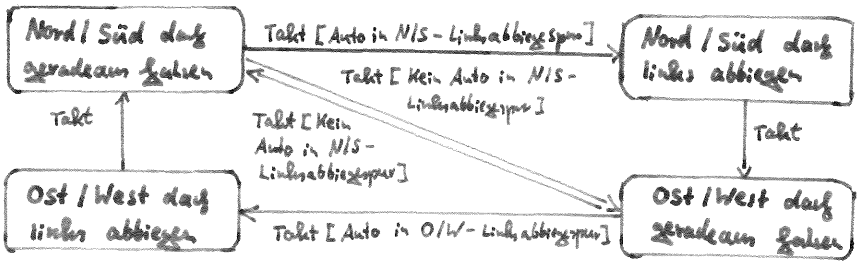
- Zustand eines Objekts := Konkrete Attributwerte und Verbindungen zu anderen Objekten zum aktuellen Zeitpunkt
- Wegen der großen Anzahl an Attributwertkombinationen abstrahieren Zustände meist von konkreten Attributwerten und Verbindungen
- Für das Verhalten Irrelevantes wird nicht modelliert
- Definitionsgemäß gilt:
  - ↳ mehrere Zustandssymbole ohne Bezeichner repräsentieren unterschiedliche Zustände
  - ↳ mehrere Zustandssymbole mit gleichen Bezeichnern repräsentieren denselben Zustand
- MML: abgerundetes Rechteck (Zustand A) mit festem Zustandsnamen

Ereignisse

- Ereignis := Eintritt eines benannten Sachverhalts mit im Modellierungskontext vernachlässigbarer Zeitdauer
- Ereignisarten
  - ↳ Aufrufereignis: Nachrichtenempfang veranlasst Operationsausführung
  - ↳ Änderungsereignis: Für das zustandsorientierte Verhalten relevante Änderung (z.B. Instanziieren / Löschen eines Objekts, Einrichten / Lösen von Verbindungen)
  - ↳ Zeitereignisse: Eintritt eines Zeitpunkts, Ablauf einer Zeitperiode
  - ↳ Signalereignisse: im Kontext des Kurses nicht relevant
- Bei lediglich Detailunterschieden zweier Ereignisse wird abstrahiert und Details über Ereignisattribute abgebildet
- MML: Analog Klassen, aber ohne Methoden, Assoziationen und keine Hierarchie zwischen Ereignis und Instanzen

Zustandsübergänge und Bedingungen

- Zustandsübergang (transition):= Objekt mit Ausgangszustand gelt bei Eintritt eines Ereignisses in einen Folgezustand über
  - ↳ reflexiver Zustandsübergang: Ausgangs- und Folgezustand identisch
- (Wächter-)bedingung := prädikatenlogischer Ausdruck, der u.A. Ereignisparameter und Attribute bzw. Verbindungen von betroffenen Objekten verwendet
  - ↳ Zustandsübergang wird nur bei erfüllter Wächterbedingung ausgeführt
- deterministisches zustandsorientierte Verhalten := Bedingungen aller Zustandsübergänge mit selbem Ausgangszustand und selbem Auslöserereignis müssen sich gegenseitig ausschließen
- MML: Ereignisname [Wächterbedingung] →



Zustandsdiagramm mit Bedingungen

- Zustandsübergang meist Reaktion auf Eintritt eines Ereignisses
- interne Transition := Ereignis veranlasst nur die Ausführung von Aktionen, aber keinen Zustandswechsel
- Zustandsübergang ohne Ereignis und ohne Wächterbedingung wird unmittelbar beim Betreten des Ausgangszustands ausgelöst

Aktionen

- Eingangsaktion := Aktion wird bei allen in einen bestimmten Zustand mündenden Zustandsübergängen ausgeführt
- Ausgangsaktion := Aktion wird bei allen einen Zustand verlassenden Zustandsübergängen ausgeführt

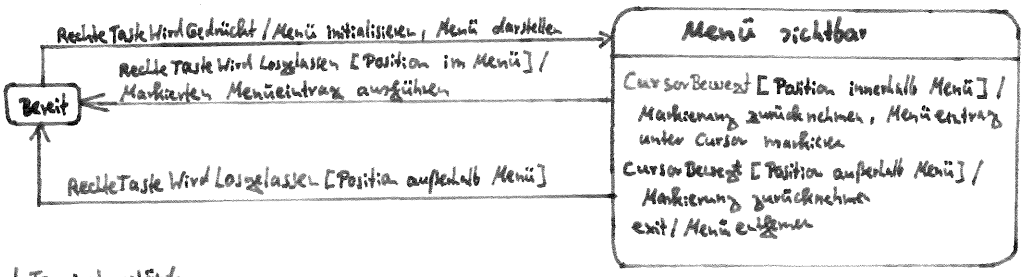
↳ reflexiver Zustandsübertragung: Eingang auslösendes Ereignis → Ausgangsaktion ausführen
→ Aktionen des Zustandübergangs
→ Eingangsaktion ausführen

↳ interne Transition: nur zugeordnete Aktionen werden ausgeführt, aber keine Ein- und Ausgangsaktionen

- UML: Syntax für Zustandsübergang: Ereignis [Wächterbedingung] / Aktionsfolge ^ Sendeaktion

Aktionsfolge: z.B. durch Kommata getrennte Aktionsnamen
Sendeaktion: Zielausdruck.Nachrichtenname (Argumente)

Beispiel: RechteTasteWirdLosgelassen (Position) [Position im Fenster] / Objekt := objektAuswählen (Position) ^ Objekt.invertiert()



interne Transition sowie Ein- und Ausgangsaktion werden im unteren, per waagrechtem Strich abgetrennten Teil angegeben.

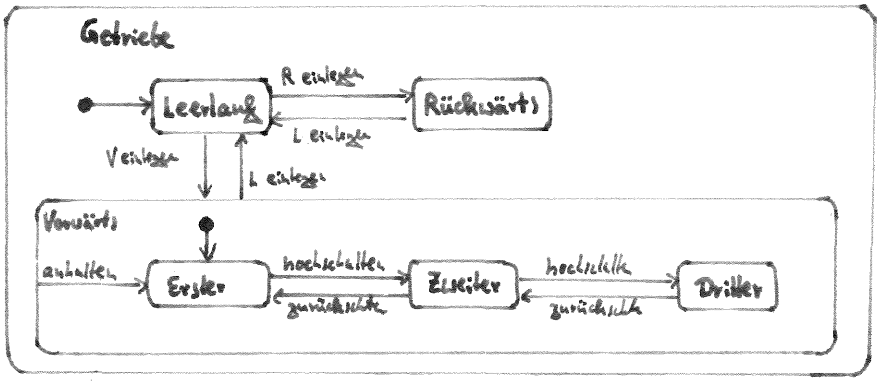
Ein-/Ausgangsaktion nach Schrägstrich mit Schlüsselworten entry bzw. exit

Initial- und Terminalzustände

- Initialzustand: Pseudozustand, dessen Übergang Objekt erzeugt.
  ↳ muss mit genau einem Zustandsübergang ohne Ereignis und Wächterbedingung zum Ausgangszustand des Objekts führen
  ↳ UML: ●
- Terminalzustand: Übergang entspricht Zerstörung des Objekts
  ↳ UML: ◉

Zusammengesetzte Zustände

- hierarchische Strukturierung zusammenhängender Teile des Zustandsdiagramm zu zusammengesetzten Zuständen
- Vorteile: bessere Strukturierung und Übersichtlichkeit komplexer Zustandsdiagramme
- UML: zusammengesetzter Zustand durch abgerundetes Rechteck mit Initialzustand



Zustandsdiagramm eines Getriebes mit Generalisierung

- Regeln für Zustandsübergänge bei zusammengesetzten Zuständen

a.) An der Kontur endende Zustandsübergänge münden logisch in den Initialzustand des enthaltenen Diagramms (siehe „V einlegen")

b.) Von der Kontur ausgehende Zustandsübergänge verbinden logisch alle Unterzustände mit Ausnahme des Initialzustands mit dem Folgezustand (siehe „L einlegen")

c.) Von der Kontur ausgehende Zustandsübergang innerhalb eines zusammengesetzten Diagramms gilt für alle Unterzustände. Sollbeispiel: Bei Eintritt des Ereignisses gelangt man von allen Unterzuständen in den angegebenen Zustand (siehe „anhalten")

Anforderungsermittlung

Vor der Anforderungsermittlung: Auftraggeber sammelt Basisanforderungen des Anwendungssystems im Lastenheft

- Anforderungsarten laut IEEE-Standard
  - funktionale Anforderungen
    - Funktionen
    - Eigenschaften
  - nichtfunktionale Anforderungen
    - Technische Anforderungen (Effizienz, Lastverhalten etc.)
    - Qualitätsanforderungen (Belastbarkeit, Zuverlässigkeit etc.)

Ziel der Anforderungsermittlung: Anforderungen in einer Anforderungsspezifikation zusammentragen

- Anforderungsspezifikation enthält Dokumente unterschiedlicher Abstraktion, Formulierung & Aufbereitung für die Projektbeteiligung
- Kernhalte:
  - Funktion (Was?)
  - Ziele (Warum?)
  - Klassen / Objekte (Womit?)
- Keine Vorgaben bzgl. der Realisierung
- Problemadäquatheit: möglichst natürliche Modellierung und Wiedergabe der Anwendersichtweise
- Inhalte der Anforderungsspezifikation
  - Anwendungsfallmodell
  - Domänen-Klassenmodell
  - Verhaltensmodelle (Interaktions- und Zustandsdiagramme)
  - grobe Beschreibung der graphischen Benutzeroberfläche

Vorgehensweise der Anforderungsermittlung

- Allgemein
  - Auf Basis des Lastenheft Überblick und Grundverständnis über die Problemwelt (Domäne) gewinnen
  - Konkrete Szenarien mit Anwendern besprechen
  - Aus Szenarien Anwendungsfälle destillieren & mit Anwender besprechen
- Teilaufgaben
  - Extraktion der Anforderungen
  - Verhandlung über Art und Umfang der Anforderungen mit Stakeholdern
  - Spezifikation der Anforderungen zur verbindlichen Basis für die nachfolgende Entwicklung und den Abnahmetest
  - Validierung der Spezifikation hinsichtlich der fachlichen Vollständigkeit und Konsistenz aus Anwender- und Auftraggebersicht

Merkregel

R 16.1: Jede Modellierungsaktivität in der Anforderungsermittlung orientiert sich ausschließlich an den Gegebenheiten und Erfordernissen der Problemwelt (Problemadäquatheit) und darf nicht Selbstzweck werden und sollte möglichst keine Gesichtspunkte der Realisierung vorwegnehmen. Insbesondere sind Modellelemente zu hinterfragen, die kein Pendant in der Problemwelt besitzen.

Anwendungsfallmodell

Schritt 1: Szenarien ermitteln und beschreiben

Beispiel: Szenario Bestellung - Alle Produkte vorhanden

Der Sachbearbeiter nimmt die Bestellung per Telefon entgegen. Er prüft die Adress- und Bankverbindungsangaben sowie die Bonität des Kunden und nimmt dann alle Bestellposten mit den jeweiligen Produkten und Bestellmengen auf. Am Ende Bestellung - Alle Produkte vorhanden

- Inhalt
  - betrachtete Funktion
  - Name
  - beteiligte Person
  - konkreter Ablauf

Schritt 2: Nutzer mit Akteuren modellieren

- Benutzer nach Rollen gruppieren und mit Akteuren modellieren
- Schnittstellen zu externen Systemen mit entsprechenden Akteuren modellieren

Merkregel

R 18.1: Für jeden menschlichen Akteur muss mindestens eine Person existieren, welche die Rolle des Akteurs spielen kann.

Schritt 3: Szenarien in Anwendungsfälle zusammenfassen

- Anwendungsfall soll wesentliche Teilfunktion des Lastenhefts abdecken
  - evtl. in kleinere Anwendungsfälle zerlegen
- bei textueller Spezifikation evtl. Vor- und Nachbedingungen notieren

Merkregeln

R 18.2: Jeder Anwendungsfall behandelt ein klar abgegrenzte Aufgabe und liefert ein relevantes Ergebnis.

R 18.3: Anwendungsfälle ohne Akteure sind unsinnig.

R 18.4: Anwendungsfälle beschreiben die Systembenutzung, nicht das System.

R 18.5: Anwendungsfälle werden von Analytikern, nicht von Anwendern geschrieben.

R 18.6: Einfachheit und Problemadäquatheit der Anwendungsfälle gehen vor die Eleganz des Anwendungsfallmodells.

R 18.7: Die textuelle Spezifikation eines Anwendungsfalls sollte ein Seite nicht überschreiten.

Schritt 4: Anwendungsfälle auf Abhängigkeiten untersuchen → Verknüpfungen & Pakete

- Verknüpfungsarten
  - include- / extends-Beziehung
  - Generalisierungsbeziehung
- bei Paketen: minimale Abhängigkeiten von anderen Paketen

Merkregeln

R 18.8: Beziehungen zwischen Anwendungsfällen sind möglichst spät und sparsam einzusetzen.

R 18.9: Beziehungen zwischen Anwendungsfällen modellieren keine Abläufe, sondern (statische) funktionale Zerlegungen.

R 18.10: Pakete von Anwendungsfällen werden nach ihrer fachlichen Zusammengehörigkeit gebildet.

Domänen - Klassenmodellierung

- Domänen - Klassenmodell bildet Strukturen der Problemwelt ab

Vorgehen

- identifiziere Objekte und Klassen
- bestimme Attribute und Assoziationen (grundsätzlich nur öffentliche Attribute angeben)
- Operationen werden nicht betrachtet
- Gemeinsamkeiten von Klassen können mit Generalisierungsbeziehungen modelliert werden

Merkregeln

R 19.1: Jedes Element des Domänen - Klassenmodells sollte ein Pendant besitzen, das ein relevanter Gegenstand oder Sachverhalt der Problemwelt ist. Ausnahmen müssen begründet und hinterfragt werden.

R 19.2: Auf jede Klasse des Domänen - Klassenmodells muss in mindestens einem Anwendungsfall oder Szenario Bezug genommen werden.

Generalisierungen

- Klassen auf strukturelle und verhaltensbezogene Gemeinsamkeiten untersuchen (meist Attribute und Assoziationen)
- Attribute & Assoziationen in Generalisierungshierarchie möglichst hoch ansiedeln

Merkregel

R 19.3: Die Problemadäquatheit von Domänen - Klassen hat unbedingten Vorrang vor der Zusammenfassung gemeinsamer Eigenschaften in Oberklassen.

Modellbereinigung

- vor aufwändiger textueller Spezifikation: durchsuche Domänen-Klassenmodell nach irrelevanten und redundanten Elementen

Merkregeln

R 19.4: Jede Domänenklasse sollte mehr als ein Attribut besitzen.

R 19.5: Ableitbare Attribute und Assoziationen einer Klasse sollten als abgeleitet markiert werden.

R 19.6: Zu jeder Domänenklasse sollte im Normalfall mehr als ein Objekt in der Problemwelt existieren.

Textuelle Spezifikation

- Anfertigung möglichst spät und nach Modellbereinigung
  - ↳ Attribute und Assoziationen genauer beschreiben
  - ↳ knappe textuelle Spezifikation von zu ungenauen Eigenschaften
- Sprache orientiert sich an Problemwelt

Pakete

- Kernklassen von Paketen: oberste Klassen von Aggregationen, Kompositionen und Generalisierungshierarchien + alle Klassen außerhalb solcher Beziehungen
- Nicht - Kernklassen ordnet man den Paketen ihrer Kernklasse zu.
- Hinweise zur hierarchischen Paketanordnung
  - ↳ Aufteilung von Klassen nach Beziehungen nicht notwendigerweise disjunkt ⇒ Durchschnitte durch geänderte Paketanordnungen möglichst minimieren
  - ↳ Paketübergreifende Beziehungen minimieren

Merkregel

R 19.7: Pakete von Domänenklassen werden nach der fachlichen Zusammengehörigkeit gebildet („hohe Kohäsion")

Benutzeroberfläche

- Benutzeroberfläche := nach außen sichtbarer Teil der Benutzerschnittstelle (Kommunikationsschnittstelle Mensch / Anwendungssystem)
- Aspekte der Oberfläche grundsätzlich beschreiben:
  - ↳ Bildschirmdarstellung (Layout): wesentliche Fenster mit Darstellungs- und Manipulationsmöglichkeiten sowie Verbindung untereinander
  - ↳ Dialogverhalten: Navigation zwischen Fenstern
- Empfehlung: rudimentäre Wegwerf - Prototypen („Mock-ups")

Verhaltensmodellierung

- in Anforderungsermittlung mäßig relevant, da präzise und feingranulare ⇒ hoher Entwicklungs- und Änderungsaufwand
- Verhaltensmodelle
  - ↳ ablauforientiertes Verhalten von Anwendungsfällen (Sequenzdiagramm)
  - ↳ zustandsorientiertes Verhalten von Anwendungsfällen (Zustandsdiagramm)

Validierung & Verifikation

- Validierung der Anforderungsspezifikation := Prüfe Spezifikation hinsichtlich der Anforderungen von Auftraggebern und Benutzern
  ↳ „Are we building the right product?"
- Verifikation der Anforderungsspezifikation := Abgleich der Modelle auf Vollständigkeit und Konsistenz
  ↳ „Are we building the project right?"
- Validierung konzentriert sich auf das Anwendungsfallmodell
  - ↳ Schritt 1 Reviews: Anwender prüfen die textuelle Beschreibung der Anwendungsfälle
  - ↳ Schritt 2 Walk-Throughs: Anwendungsfall wird mit ausgewählten Szenarien und betroffenen Domänenobjekten mit Anwendern durchgespielt und diskutiert
- Verifikation nach erfolgter Validierung: nur grober Abgleich des Domänen - Klassenmodells mit validiertem Anwendungsfallmodells

Analyse

- Analyse := Aufbereitung und Präzisierung der Anforderungsspezifikation
  - ↳ Ergebnis: Analysespezifikation als Basis für die Realisierung
- Inhalt der Analysespezifikation
  - ↳ Analyse-Klassenmodell (!)
  - ↳ Interaktionsdiagramme
  - ↳ Zustandsdiagramme
  - ↳ Dokumente zur Durchführung und Ergebnis der Verifikation
- Hauptziele
  a) textuelle Spezifikation der Anwendungsfälle + Domänen-Klassenmodell = Analyse-Klassenmodell
  b) Überarbeitung & Präzisierung des Analyse-Klassenmodells
  c) Verifikation (Konsistenz, Vollständigkeit)
- Vorgehensweise
  1.) Anwendungsfälle mit Domänen-Klassenmodell zu Analyse-Klassenmodell zusammenfassen und ggf. weitere Klassen ergänzen (a)
  2.) Analyse-Klassenmodell mit Operationen anreichern
  3.) ablauforientiertes Verhalten komplexer Anwendungsfälle mit Mechanismen und Interaktionsdiagrammen spezifizieren (b)
  4.) Analyse-Klassenmodell gemäß Heuristiken aufbereiten, bereinigen und in Pakete aufteilen (b)
  5.) Verifikation (c)

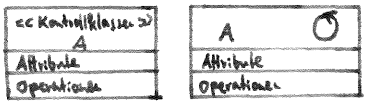
Klassenmodellierung

- Analyseklassen sind realisierungsnäher als Domänenklassen, aber immer noch Abstraktionen
  - ↳ Fokus auf funktionale Anforderungen
  - ↳ Attribute am Domäne, nicht am Programmierspeicher
  - ↳ Präzise Schnittstellenbeschreibung mit Signatur nur für komplexe Nicht-Standardoperationen

Klassenarten

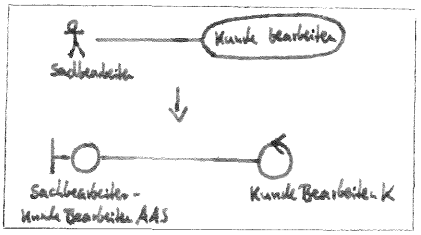
Entitätsklassen

- repräsentiert langlebige Informationen (Domänenklassen)
- Darstellungsvarianten:



Kontrollklassen

- Kapselt fachliche Logik eines Anwendungsfalls
- Koordiniert am Anwendungsfall beteiligte Instanzen von Entitätsklassen
- UML: Suffix „K"



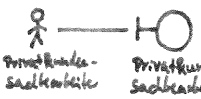
Schnittstellen- und Kontrollklasse für den Anwendungsfall „Kunde bearbeiten"

Schnittstellenklasse

- bündelt Interaktion zwischen Akteuren & Anwendung
  - ↳ Auswahl von Anwendungsfällen
  - ↳ Repräsentation und Ablaufgesteuerte Datenmodifikation
  - ↳ Umsetzung von Kommunikationsprotokollen für technische Akteure
- Schnittstellenklassenarten

Akteur-Schnittstellenklasse (AS-Klasse)

- erlaubt Akteuren Auswahl und Aktivierung der ihnen zur Verfügung stehenden Anwendungsfällen
- UML: Suffix „AS"



Anwendungsfall-Akteur-Schnittstelle (AAS-Klasse)

- wird für jeden Anwendungsfall und beteiligten Akteur modelliert
- zuständig für Interaktion des Akteurs mit dem Anwendungsfall
- Für Include- und Extendbeziehungen im Anwendungsfallmodell werden stereotypisierte Assoziationen zwischen AAS-Klassen eingeführt
- UML: Suffix „AAS"

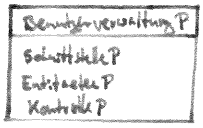
! Kontrollklasse vermitteln zwischen Schnittstellen- und Entitätsklassen ! 

Operationen

- Operationen ergänzen Aktionen im Analyse-Klassenmodell um
- Nicht-Standardoperationen aufnehmen und textuell Verantwortlichkeit beschreiben
- präzise Modellierung des Ablaufverhaltens erst im Entwurf!

Pakete

- Aufteilung von Analyseklassen zunächst nach Akteuren / Anwendungsfällen
- Dienstpakete bündeln für mehrere Akteure / Anwendungsfälle relevante Analyseklassen
- UML: Suffix „P"



Heuristiken

H25.1 Existieren im Laufe der Modellierung mehrere Alternativen als gleichwertig, so entscheidet man sich vorläufig für irgendeine der Alternativen.

1-1 Assoziation oder eine Klasse?

H25.2 Zwei verschiedene Klassen sind die bessere Wahl, wenn die 1-1 Assoziation zwischen diesen beiden Klassen in einer oder beide Richtungen optional, also mit Multiplizität 0..1 spezifiziert ist.

H25.3 Zwei verschiedene Klassen sind die bessere Wahl, wenn eine Instanz eine Verbindung nach außen eingehen kann.

H25.4 Wird eine Klasse in die ursprüngliche und eine neue Klasse mit einer neuen 1-1 Assoziation zwischen den beiden aufgeteilt, bleibt eine ursprüngliche Assoziation weiterhin auf die ursprüngliche Klasse bezogen, wenn Objektverbindungen bzgl. dieser Assoziation nicht geändert werden müssen, sobald sich eine Objektverbindung bzgl. der neuen 1-1 Assoziation ändert.

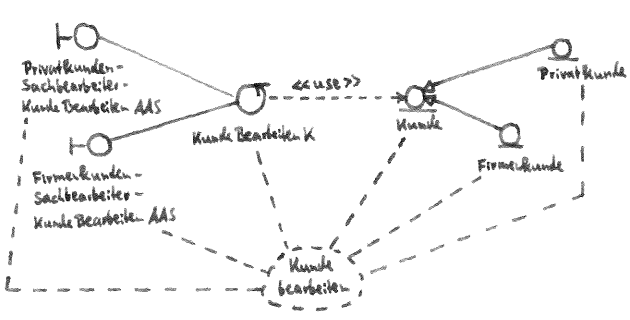
Attribute oder Generalisierung?

H25.5 Lassen sich Objekte mit denselben Attributen und denselben Verhalten in disjunkte Klassen aufteilen, werden sie in einer Klasse zusammengefasst und mit wenigen zusätzlichen Attributen unterschieden. Insbesondere wenn sich die unterscheidenden Eigenschaften dynamisch ändern können, wird die Modellierung mit Attributen vorgezogen.

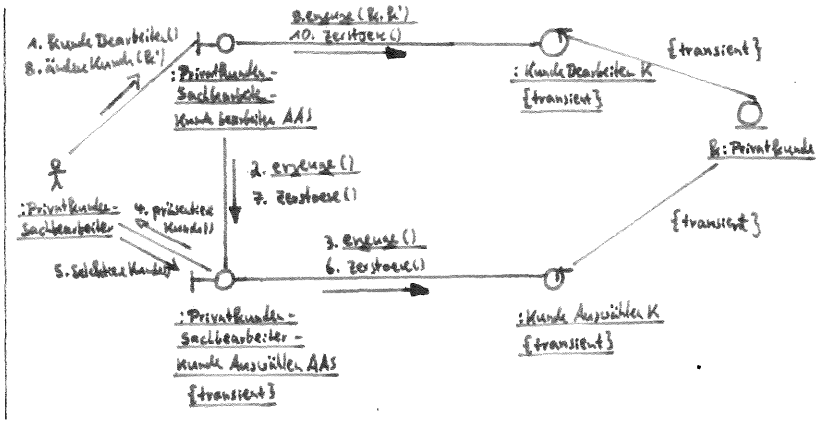
H25.6 Wenn sich die Objekte bzgl. Attribute und Verhalten unterscheiden, wird die Generalisierung benutzt. Sie wird auch für Spezialfälle verwendet, wenn sich die Objekte nur bzgl. ihre Attribute unterscheiden (in mehr als nur wenigen Attributen) und nicht bzgl. ihren Verhalten und umgekehrt.

Verhaltensmodellierung

- Interaktionsdiagramme aus Anforderungsermittlung werden in Analyse für Analyse-Klassenmodell verfeinert
- Vorgehen:
  1.) Die Anwendungsfall realisierenden Klassen mit Mechanismus kennzeichnen
  2.) Kollaborationsdiagramme mit Instanzen von Klassen aus Mechanismus für zugehörigen Szenarien anfertigen
    a.) nehme Objektkonstellation, die die Vorbedingung eines Anwendungsfalls erfüllt (meist AAS- und Entitätsobjekte)
    b.) ergänze die im Verlauf des Szenarios erzeugten Objekte und Verbindungen (Kontroll- und Entitätsobjekte erzeugen, modifizieren, zerstören)



Mechanismus für Anwendungsfall "Kunde bearbeiten"



Kollaborationsdiagramm mit einem Szenario zum Anwendungsfall „Kunde bearbeiten", der den Anwendungsfall „Kunde ansehen" benutzt

Verifikation

Intra-Modell-Verifikation

- prüft die Modelle auf
  - Vollständigkeit (Befolgt die Beschreibung der Modellelemente die Spezifikationsrichtlinien?)
  - Eindeutigkeit (Eindeutig interpretierbar?)
  - Konsistenz (alle Querbezüge verweisen auf existierende & richtige Modellelemente und keine redundanten oder widersprüchlichen Spezifikationen?)
- prüfen:
  - Anwendungsfallmodell: existieren Anwendungsfälle bzw. Erweiterungspunkte der Include- und Extends-Beziehungen?
  - Klassenmodell: - besitzen alle Assoziationen Multiplizitäten?
    - Existieren keine ableitbaren Attribute?
    - Existieren keine redundanten Features in Unterklassen?
  - Interaktionsdiagramme: - besitzt jede Kopf Operationsnamen?
    - wird jedes nachfolgende Objekt explizit erzeugt?

Inter-Modell-Verifikation

- Abgleich verschiedener Modelle
- Abgleich Analyse-Klassenmodell und Anwendungsfallmodell
  a.) Anwendungsfallszenarien stützen sich vollständig auf Operationen im Klassenmodell ab
  b.) jede Operation im Klassenmodell setzt eine Aktion in einem Szenario um
  c.) Konsistenz von Anwendungsfallspezifikation und Operationen im Klassenmodell
- Abgleich Analyse-Klassenmodell und Zustandsdiagrammen
  - Konsistenz der Operationen im Klassenmodell zu Aktionen d. Zustandsdiagramm,
    a.) Für jede Aktion eines Zustandsübergangs existiert eine Operation im Klassenmodell
    b.) Ausgangsbedingung eines Zustandsübergangs erfüllt Vorbedingung der zum Übergang gehörenden Operation
    c.) Nachbedingung der Operation passt zum Folgezustand

- Abgleich Interaktions- und Zustandsdiagramme
  - ↳ ist die im Interaktionsdiagramm angegebene Reihenfolge empfangener und gesendeter Nachrichten mit Zustandsdiagramm verträglich?
  - ↳ Ereignisfolge im Zustandsdiagramm = Folge empfangener Nachrichten im Interaktionsdiagramm
  - ↳ Folge der Sendeaktionen im Zustandsdiagramm = Folge gesendeter Nachrichten im Interaktionsdiagramm

Architekturentwurf

- Architekturentwurf := Zerlegung des Anwendungssystems in Teilsysteme und Komponenten
  - ↳ Vorteile: ⊕ Komplexitätsbeherrschung ⊕ Arbeitsteilung ⊕ Spezialisierung
- Vorgehen: Grundstruktur schrittweise ausfüllen, verfeinern und ergänzen
- Ergebnis: Architekturspezifikation
- Architekturentwurf ist der Anfang der Realisierung
- Zeichen guter Architektur
  - ↳ verwendet Standards / Konzepte aus den wichtigsten OOP-Sprachen
  - ↳ große Unabhängigkeit vom speziellen Anwendungsbereich
  - ↳ Wiederverwendung & Wiederverwendbarkeit

Schichtenarchitektur

- möglichst unabhängige Schichten mit getrennten Verantwortlichkeiten
- Für kommerzielle Informationssysteme: 3-Schichten-Architektur

Schicht 3: externe Schnittstellen
- ermöglicht Interaktion zwischen Anwendungssystem & Akteur
  - ↳ Benutzerschnittstelle ↳ Zugriff von / auf externe Systeme

Schicht 2: Anwendungskern
- Geschäfts- und Sachverhalte der Domäne
- fachliche Funktionalität
- Geschäftsregeln

Schicht 1: Datenhaltung
- realisiert persistente Speicherung von verarbeiteten Daten

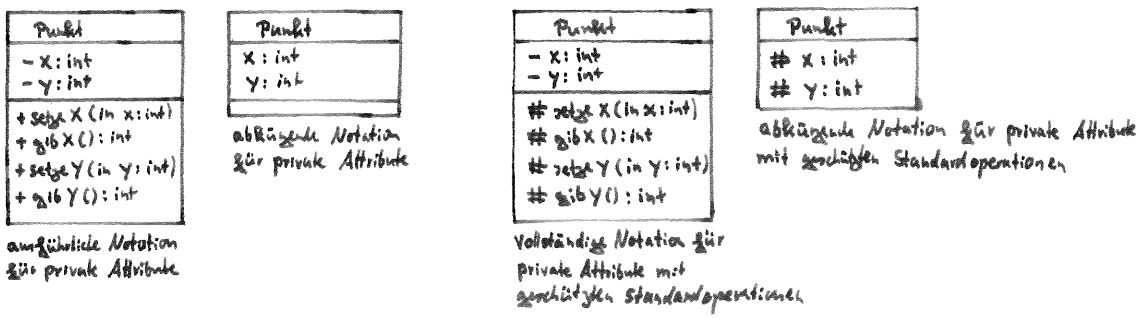
Grundlagen des Entwurfs

- Entwurf verfeinert die von der Architektur vorgegebenen Teilsysteme unter Berücksichtigung aller (funktionalen und nichtfunktionalen) Anforderungen
- Ausgangspunkt: Analysespezifikation, Architekturspezifikation und Teile der Anforderungsspezifikation
- Ergebnis: Entwurfsspezifikation
  - ↳ spezifiziert Aufgaben der einzelnen Module, ihre Beziehungen und ihr Zusammenwirken
  - ↳ präzises Entwurfs-Klassenmodell mit an Detaillierungsgrad angepasste Interaktions- und Zustandsdiagramme
- Vorgehen:
  - Schritt 1: Grobentwurf
    - ↳ Abbildung Analyseklassen → Architektur
    - ↳ abstrahiert noch von Implementationssprache
  - Schritt 2: Feinentwurf
    - ↳ verfeinert Grobentwurf
    - ↳ berücksichtigt Implementationssprache

Grundkonzepte

Geheimnisprinzip

- Geheimnisprinzip := Module verkapseln ihre Realisierung vor ihren Dienstnutzern. Diese können und dürfen nur die Schnittstelle kennen.
- Vorteile: + ermöglicht korrekte Nutzung des Dienstleisters
  + reduziert Fehlerwahrscheinlichkeit
  + unterstützt arbeitsteilige Entwicklung
  + Änderungen der Realisierungen bleiben modullokal
- Regel R35.1: Attribute werden im Entwurf immer als private deklariert.
  - ↳ In Generalisierungshierarchien:
    - Attribute als private deklarieren
    - Zugriff von Unterklassen über als protected deklarierte Standardoperationen
    - Im Entwurf Standardoperationen nicht aufführen und Attribut mit Präfix „#" kennzeichnen
- UML:



Kopplung & Kohäsion

- Kopplung := Beziehungskomplexität in einem Entwurf
  - Zwei nicht-elementare Typen sind gekoppelt, wenn:
    - eine Assoziation zwischen ihnen existiert
    - Operation eines Typs nutzt anderen Typ als Parameter
    - Instanz eines Typs werden im anderen Typ verwendet
    - zwischen ihnen eine Generalisierungsbeziehung besteht
    - ein Typ ist ein Interface, das der andere Typ implementiert
- Kohäsion := logischer Zusammenhang der von einer Klasse realisierten Aufgaben
- Ziel eines qualitativen Entwurfs: schwache Kopplung und starke Kohäsion
  - ↳ in Praxis Konflikt: Stärkung Kohäsion ⇒ Stärkung der Kopplung
    1.) Zu wenig Klassen ⇒ negative Wartbarkeit und Wiederverwendbarkeit
    2) Zu viele Klassen ⇒ Verletzung Geheimnisprinzip

Abschwächung der Kopplung

- Instanzen einer Klasse sollen so wenig wie möglich mit Instanzen anderer Klassen zusammenarbeiten
- Heuristiken:

H35.2: Sind nur sehr wenige Klassen (Anhaltspunkt: bis zu drei) eng gekoppelt, versuche die Klassen zu einer einzelnen Klasse zusammenzufassen.

H35.3: Sind mehrere Klassen (mehr als drei) stark gekoppelt, versuche die Kopplung durch eine Vermittlerklasse abzuschwächen.

H35.4: Ist eine Funktionalität für die starke Kopplung verantwortlich, versuche sie in eine andere Klasse zu verschieben.

H35.5: Sind innerhalb einer Generalisierungshierarchie mehrere Standardoperationen der Oberklasse für die starke Kopplung verantwortlich, versuche sie durch Aufruf einer (komplexeren) Operation der Oberklasse zu vermeiden.

H35.6: Jede „Ganze-Klasse" muss wissen, welche Teile-Klassen sie enthält, aber keine Teile-Klasse sollte ihre Ganze-Klasse kennen.

H35.7: Teile-Klassen derselben Ganze-Klasse sollten sich nicht gegenseitig benutzen.
(„Mitarbeiter plaudern nicht in Gegenwart ihrer Vorgesetzten")

H35.8: (Law of Demeter) In einer Operation o() der Klasse K sollten nur Operationen folgender Klassen benutzt werden:

- K selbst
- Klassen, die als Parametertypen von o() vorkommen
- mit K assoziierte Klassen
- Klassen, von denen bei der Ausführung von o() Instanzen erzeugt werden

Verstärkung der Kohäsion

- Grundsätzlich: Modul in kleinere Module zerlegen
  - Aber: Verstärkung der Kopplung nach Möglichkeit verhindern
- Heuristiken

H35.9: Lassen sich die Attribute und die Operationen einer Klasse in (weitgehend) disjunkte Teilmengen zerlegen, so dass jede Operationsmenge nur auf jeweils einer Attributmenge arbeitet, dann ist die Aufteilung der Klasse zu empfehlen.

H35.10: Erbringt eine Operation mehr als eine einzige, eng umrissene Funktionalität, ist die Funktionalität auf mehrere Operationen aufzuteilen.

Konformität

- Definition: Sei A eine Klasse mit Unterklasse B. Sei op_A die in A definierte und op_B die von B überschriebene Operation. Seien INV_A und INV_B die Invarianten.

  Wir nennen die Operation op_B konform zu op_A, wenn gilt:

  1) Signatur (op_A) = Signatur (op_B) ,
  2) Vorbedingung $(op_A) \Rightarrow$ Vorbedingung (op_B) und
  3) Nachbedingung $(op_B) \Rightarrow$ Nachbedingung (op_A).

  Wir nennen B konform zu A, wenn alle von A geerbten Operationen in B konform zur jeweiligen Operation in A sind.

- Generalisierung / Vererbung: Ist die Unterklasse konform zur Oberklasse, so bezeichnen wir die Beziehung als Generalisierung.
  Ist die Unterklasse nicht konform zur Oberklasse, so bezeichnen wir die Beziehung als Vererbung.

- Wichtig: Keinerlei Einschränkung für von der Unterklasse zusätzlich definierten Operationen
- Vorteil Generalisierung: geringerer Testaufwand
- Verletzung der Konformität oft durch Änderung an Oberklasse
- Heuristik zur Erweiterung der Generalisierungshierarchie:

H35.11: Ist eine Unterklasse nicht konform zu ihrer Oberklasse, weil die Oberklasse für die Unterklasse nicht benötigte Eigenschaften besitzt, sollte eine neue gemeinsame Oberklasse gebildet werden, die in Generalisierungsbeziehung zu beiden ursprünglichen Klassen steht.

- Minimierung des Überschreibens

H35.12: Klassen werden so definiert, dass sie für Unterklassenbildung offen, aber für unkontrollierte Zugriffe geschlossen sind (Open-Closed-Principle).
Um möglichst viele Operations-Implementierungen in Unterklassen wiederverwenden zu können, werden Operationen einer Oberklasse, die auch von (zukünftigen) Unterklassen benutzt werden, als protected gekennzeichnet

Entwurf mit Verträgen

- regelt Zusammenarbeit von Klassen bzw. Objekten (Analog Verträge: Dienstnutzer und Dienstleister)
- einfachste Techniken:
  1.) Schnittstellenspezifikation mit Vor- und Nachbedingungen sowie Klasseninvarianten
  2.) Klare Verantwortlichkeiten:
  - Dienstnutzer erfüllt Vorbedingung $\Rightarrow$ Dienstleister garantiert Nachbedingung
  - Dienstnutzer und Dienstleister erfüllen ihre Invarianten vor und nach jeder öffentlichen Operation
  3.) Systematische Ausnahmebehandlung
  - Dienstleister kann Aufgabe trotz erfüllter Vorbedingung des Dienstnutzers nicht erfüllen
    a.) Wiederanlauf (ggfs. alternative Lösung)
    b.) Operationsabbruch (organisierte Panik)

	Dienstnutzer	Dienstleister
Vorbedingung	Pflicht	Recht
Nachbedingung	Recht	Pflicht
Invariante	Pflicht	Pflicht

- Regel

R35.13: Vor- und Nachbedingungen enthalten keine vorhersehbaren Ausnahmefälle, die gesondert behandelt werden müssen.

- Vertragskonzept erleichtert Fehlerlokalisierung
  - verletzte Vorbedingung: Fehlverhalten des Dienstnutzers
  - verletzte Nachbedingung: Fehlverhalten des Dienstleisters

Interfaces

- definieren Menge sichtbarer Operationen, aber keine Attribute
- Interface kann nicht instanziiert werden
- definiert Typ von dem Variablen deklariert werden können
- definiert Vertrag, da von realisierenden Klassen und ihren Dienstnutzern einzuhalten sind
- ermöglicht einheitliche Verwendung von Instanzen verschiedener Klassen

Entwurfsmuster

- Rahmenwerk (framework): erlaubt Wiederverwendung ganzer Entwürfe für wesentliche Aspekte einer Gruppe von Anwendungssystemen
- Entwurfsmuster (design pattern): Beschreibung einer Lösungsfamilie für ein Entwurfsproblem
  - ↳ konstruktive Vorlage
  - ↳ deskriptives Beispiel für Dokumentation von Entwurfsentscheidungen
  - ↳ strukturelle Orientierung in einem komplexen Entwurf
- Definition GoF: „A design pattern systematically names, motivates and explains a general design that addresses a recurring design problem in object-oriented systems. It describes the problem, the solution, when to apply the solution and its consequences. It also gives implementation hints and examples. The solution is a general arrangement of objects and classes that solve the problem. The solution is customized and implemented to solve the problem in a particular context."
- Dokumentationsschema:
  - Zweck des Musters
  - Synonyme
  - Motivation (Einordnung, Zweck und Einsatzmotive)
  - Anwendbarkeit (Indikatoren für Einsatz)
  - Struktur
  - Beteiligte (Klassen / Objekte) & Zusammenspiel
  - Konsequenz (Vor- und Nachteile)
  - Implementierung
  - Praxiseinsatz
  - Querverweise

Anwendungsbereich	Wirkung: Erzeugung	Wirkung: Struktur	Wirkung: Verhalten
Klasse	Fabrikmethode	Adapter (Klasse)	Interpreter Schablonenmethode
Objekte	Abstrakte Fabrik Einzelstück Ersteller Prototyp	Adapter (Objekt) Brücke Dekorateur Fassade Fliegengewicht Kompositum Surrogat	Befehl Beobachter Besucher Iterator Schnappschuss Strategie Vermittler Zustand Zuständigkeitskette

- Arten von Entwurfsmustern
  - ↳ Erzeugende Muster: zielen auf Instanziierung ab
  - ↳ Strukturelle Muster: Zusammensetzung von Strukturen mehrer Klassen
  - ↳ Verhaltensmuster: Varianten des Zusammenspiels mehrere Objekte

Erzeugende Muster

- Abstrakte Fabrik (abstract factory): bietet Schnittstelle zum Erzeugen von Familien verwandter oder voneinander unabhängiger Objekte ohne konkrete Klassen zu benennen.
- Einzelstück (Singleton): stellt sicher, dass von einer Klasse nur eine einzige Instanz existiert und bietet globalen Zugriffsmechanismus auf die Instanz an.
- Ersteller (builder): trennt Aufbau eines komplexen Objekts von seiner Darstellung, so dass mittels desselben Konstruktionsvorgangs verschiedene Repräsentationen erzeugt werden können.
- Fabrikmethode (factory method): definiert Schnittstelle zur Objekterzeugung, aber überlässt den Unterklassen die Entscheidung, welche Klasse zu instanziieren ist.
- Prototyp (prototype): spezifiziert Art der zu erzeugenden Objekte über exemplarische Instanz und erzeugt neue Objekte durch Kopieren dieser Instanz.

Strukturmuster

- Adapter (adapter): wandelt Schnittstelle gegebener Dienstleisterklasse zur Verwendung durch Dienstnutzerklasse um.
- Brücke (bridge): entkoppelt Abstraktion von ihrer Implementierung
- Dekorateur (decorator): heftet dynamisch zusätzliche Verantwortlichkeiten (Operationen) an ein Objekt. Basiert auf Dienstnutzungen. Flexible Alternative zu Generalisierung.
- Fassade (facade): einheitliche Schnittstelle für eine Menge von Schnittstellen. Führt Strukturierungsebene oberhalb von Klassen ein.
- Fliegengewicht (flyweight): nutzt wenige (komplexe) Objekte mehrfach, um effizient eine große Anzahl kleiner Objekte zu handhaben
- Kompositum (composite): ermöglicht einheitlichen Zugriff auf Elemente hierarchischer Objektstrukturen. Bietet Schnittstelle zur Gleichbehandlung von individuellen Objekten (Blätter) und Kompositionen (innere Knoten)
- Surrogat (proxy): stellt über Stellvertreterobjekt eingeschränkten Zugriff auf eigentliches Objekt zur Verfügung.

Verhaltensmuster

- Befehl (command): verkapselt Operationsaufruf, damit Dienstnutzungen parametrisiert, aneinandergereiht, aufgezeichnet und ggf. später rückgängig gemacht werden können.
- Beobachter (observer): definiert 1:n-Abhängigkeit zwischen Objekten, um Zustandsänderungen eines Objekts an andere gemeldet werden können.
- Besucher (visitor): repräsentiert auf die einzelnen Elemente einer Objektstruktur anwendbare Operation. Ermöglicht Definition einer neuen Operation ohne Änderung an Klasse
- Interpreter (interpreter): interpretiert Sätze in einer gegebenen Sprache aufbauend auf einer Grammatikrepräsentation mittels der Generalisierung.
- Iterator (iterator): elementweiser Zugriff auf Struktur ohne ihre Implementierung offenzulegen.
- Schablonenmethode (template method): definiert Grundgerüst eines Algorithmus und überlässt Unterklassen die Implementierung von Details.
- Schnappschuss (memento): speichert Objektzustand, ermöglicht späteres Wiederherstellung
- Strategie (strategy): Definiert und verkapselt Algorithmenfamilie. Ermöglicht Austausch und Implementierungsvariation unabhängig von den aufrufenden Objekten.
- Vermittler (mediator): Objekt, das Interaktionen mehrerer anderer Objekte verkapselt. Schwächt Kopplung dieser Objekte und ermöglicht unabhängige Änderung einiger Interaktion.
- Zustand (state): erlaubt zustandsabhängiges Ändern eines Objektverhaltens
- Zuständigkeitskette (chain of responsibility): Vermeidet Kopplung eines Dienstnutzers an mehrere Dienstleister durch Weitergabe der Nachricht in der Kette

Auswahl von Entwurfsmustern

1.) Musterkategorie für die zu lösende Problemart auswählen
2.) Muster anhand Verwendungszweck und beschriebener Struktur auswählen

Einsatz von Entwurfsmustern

1.) Musterbeschreibung überblicksartig lesen
2.) beteiligte Klassen / Objekte und gegenseitige Abhängigkeiten verstehen
3.) Quellcode konkreter Beispiele ansehen
4.) Namensraum für vom Muster betroffenen Klassen: nicht zu sehr an Musterbeschreibung orientieren
5.) Klassenspezifikation: definiere Schnittstellen, Attribute, Operationen, Generalisierungsbeziehungen, Assoziationen
6.) Operationsbezeichner anwendungsspezifisch unter Rücksicht auf Musterbezeichner
7.) Erg. auf Beispielimplementierung zurückgreifen

Gründe für Entwurfsüberarbeitung

- starre Klassenangaben bei Instanziierung
- starre Abhängigkeiten von bestimmten Operationen
- Abhängigkeit von Software- und Hardwareplattform
- Abhängigkeit von Schnittstellen / Implementierungen, die Geheimnisprinzip verletzen und Lokalität von Änderungen aufweichen
- algorithmische Abhängigkeiten, die isolierte Änderung oder Austausch einzelner Operationen erschweren
- zu starke Kopplung
- Probleme in Generalisierungshierarchien

Grobentwurf

- Ziel: Grobentwurfsspezifikation hoher Stabilität
- Merkmale qualitativer Grobentwurfsspezifikation:
  - Gewährleistung aller Qualitätsanforderungen und technischer Vorgaben
  - Größtmögliche Erweiterungsstabilität
  - Berücksichtigung der Merkmale eines guten Entwurfs
  - Größtmögliche Unabhängigkeit von der Implementationssprache
- grundsätzliche Entwurfsentscheidungen für dieses Kurs:
  1.) Es wird nur auf Kopien der Entitätsobjekte gearbeitet
  2.) Prinzip der externen Kontrolle: Kontroll- und Entitätsklassen treten in Bezug auf Schnittstellenklassen ausschließlich als Dienstleister auf.

3-Schichtenarchitektur

- Grobentwurf: übertrage Analyseklassen in gewählte Architektur

Schnittstellenklassen → externe Schnittstelle
Kontrollklassen, Entitätsklassen → Anwendungskern

Externe Schnittstelle

- bündelt Interaktionen zwischen Benutzer & Anwendung:
  - Interaktion von Akteuren in anwendungsinterne Ereignisse übersetzen
  - Anwendungsdaten in eine für Akteure verwertbare Form überführen

Benutzerschnittstelle

- Schnittstellenklassen abstrahieren im Grobentwurf noch von tatsächlichen Fenstern und der technischen Realisierung
- neue Klassenart (im Grobentwurf):
  - Haupt-Schnittstellenklasse (HS-Klasse): modelliert Haupt- bzw. Startsicht der Anwendung
    Name: Name Anwendungssystem + „S"
- Klassenhierarchie für Schnittstellenklasse
  - HS-Klasse
  - AS-Klasse
  - AAS-Klasse

externe Systeme

- externes System ist meist prozedurenweise zugreifbar:
  - Implementierung, Konfig. & Änderungen allenfalls sehr eingeschränkt möglich
  - meist wohldefiniert, oft prozedurale Schnittstelle (aufgaben- und herstellerspezifisch)
  - autonome Datenformate
  - stützen sich selbst nur selten auf Fremdressourcen ab

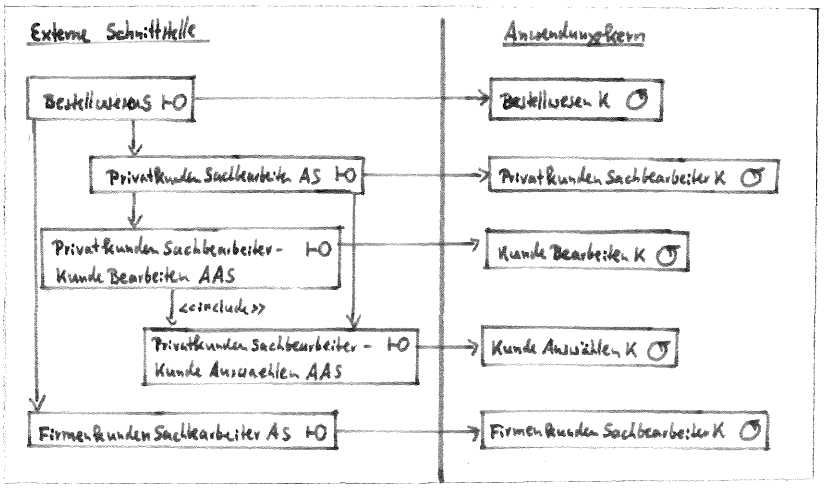
! DBMS ist kein externes System !

- Ziel bei Anbindung: gezielte Verkapselung externer Systeme um Schnittstellenänderungen möglichst lokal zu halten
  → für jedes externe System eine Akteur-Schnittstellenklasse entwerfen
  → Funktionalität im Grobentwurf nur aus Dienstnutzersicht spezifizieren

Anwendungskern

Kontrollklassen

- neue Klassen:
  - Haupt-Kontrollklasse: Koordiniert Start des Systems und Zugang zu allgemein verfügbaren Anwendungsfällen
    Name: Name Anwendungssystem + „K"
  - Akteur-Kontrollklasse: ermöglicht pro Akteur Zugang zu den zugeordneten Anwendungsfällen
    Name: Name Akteur + „K"



Haupt-, Akteur- und Akteur-Anwendungsfall-Schnittstellen und -Kontrollklassen im Grobentwurf im Beispielwesen

Standard-(Klassen-)Operationen für Entitätsklassen

- gibAlleNamen(): String[] liefert eine Liste mit den Namen aller Instanzen
- gibKopie(in name: String): liefert eine Kopie der Instanz mit dem Namen name
- gib(in name: String): liefert die Instanz mit dem Namen name
- KopiereAttribute(in instanz: E): kopiert alle Attribute der E-Instanz in die Attribute der ausführenden Instanz

Standardoperationen für Kontrollklassen

- gibENamen(): String[] liefert Liste mit den Namen aller E-Instanzen
- gibE(in name: String): E liefert eine Kopie der E-Instanz mit dem Namen name
- schreibeE(in kopie: E): veranlasst nach erfolgreicher fachlicher Prüfung die Übertragung der Attributwerte der Kopie einer E-Instanz in das Original
- erzeugeE(): E erzeugt eine neue E-Instanz und gibt eine Kopie dieser Instanz zurück
- löscheE(in name: String): löscht nach erfolgreicher fachlicher Prüfung die E-Instanz mit dem Namen name

Standardoperationen für Schnittstellenklassen

- öffnen(): stellt das entsprechende Fenster am Bildschirm dar
- schließen(): entfernt das entsprechende Fenster am Bildschirm und beendet den Anwendungsfall
- abbrechen()
- ausgeführt(): wird vom Schnittstellenobjekt eines benutzten Anwendungsfalls bei dessen Beendigung aufgerufen

Pakete

- Paketaufteilung aus Analyse übernehmen und neue Pakete zuordnen

Verhaltensmodellierung

- Vorgehen:
  1. Interaktionsdiagramme für ablauforientierten Verhalten komplexer Anwendungsfälle aus Analyse an die Detaillierung des Grobentwurfs anpassen
  2. erstmalige Modellierung der Abläufe komplexer Operationen (meist in separaten Interaktionsdiagrammen)
     - ↳ Ausgangsbasis: textuelle Spezifikation der Operation
       - Operation erzeugt / zerstört Instanzen?
       - Operation erzeugt / löst Verbindg?