Zusammenfassung: Software Engineering

zielorientierte Bereitstellung; systematische Verwendung v. Prinzipien, Methoden, Werkzeugen für die arbeitsteilige, ingenieurmäßige Entwicklung; Anwendung v. Umfangreichen Softwaresystemen

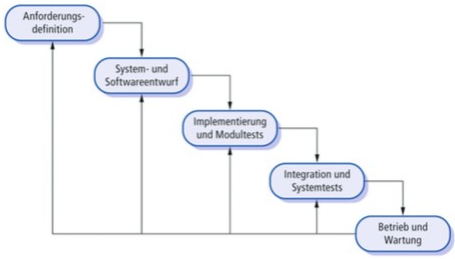
Begriffsdefinitionen

Prozess: Folge logisch zusammenhängender Methoden, Techniken zur Erstellung/ Veränderung eines Objekts mit definiertem Anfang/Ende (SCRUM, V-Modell XT; Methodologie, Verfahren)

Methode: Planmäßig angewandte, begründete Vorgehensweisen zur Erreichung von festgelegten Zielen unter Einhaltung vorgeg. Richtlinien, Normen, etc. (Brainstorming, objektorientierte Analyse)

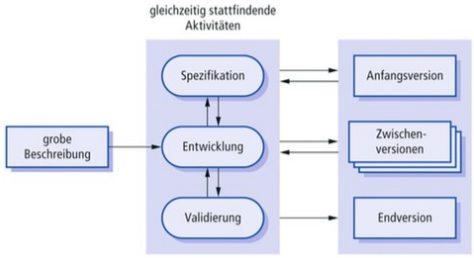
Technik: Festlegung zur Durchführung v. Teilaufgaben durch Regeln, Vorschriften (Entscheidungstabellen, Anwendungsfälle; Methode, Technologie)

Werkzeug: Softwareprodukte, die eine automatisierte Bearbeitung v. Aufgaben gestatten (Programmgenerator, GUI-Builder, IDE; Tool)

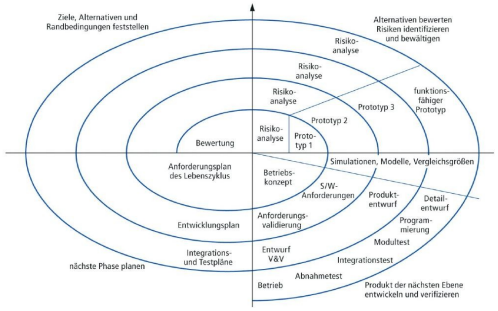


Wasserfallmodell (grundlegende Prozessabläufe als eigene Phasen)

Modelle



Inkrementelle Entwicklung (verknüpft Aktivitäten, Spezifikation, Entwicklung, Validierung, harmoniert oft nicht mit bürokratischen Abläufen))



Spiralmodell (kombiniert Vermeidung von Veränderungen mit Änderungstoleranz, jede Windung steht für eine Phase)

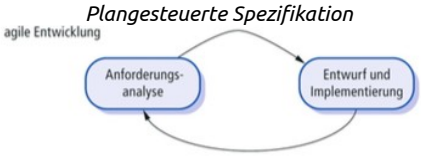
Agile Softwareentwicklung

inkrementelle Entwicklungsverfahren, die sich auf schnelle Entwicklung, häufige Softwarereleases, das Reduzieren von zusätzlichem Prozessaufwand und die Produktion von hochqualitativem Code konzentrieren; Kunde wird unmittelbar in Entwicklungsvorgang einbezogen

Prinzipien

- Einbeziehung des Kunden: Angeben neuer Systemanforderungen, Festlegen von Prioritäten, Bewerten d. Iterationen d. Systems
- Inkrementelle Auslieferung: Kunde legt Anforderungen fest, die in einem Inkrement enthalten sein sollen
- Menschen statt Prozesse: Entwickler ohne enge Verfahrensvorschriften auf eigene Weise vorgehen lassen
- offen für Änderungen: Erwartung, dass Systemanforderungen sich ändern → System soll angepasst werden können
- Einfachheit: komplizierte Elemente aktiv aus System entfernen

Plangesteuerte Spezifikation



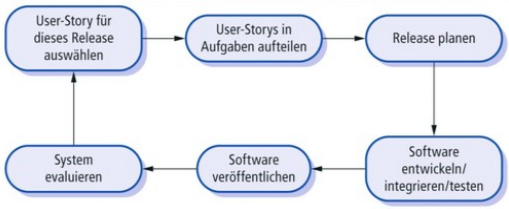
Agile Spezifikation

Extreme Programming(XP)

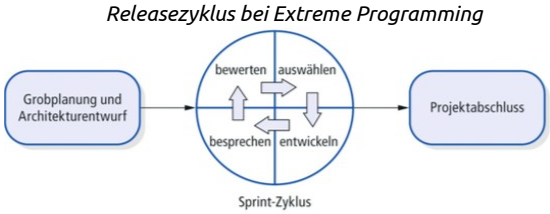
Entwicklung automatisierter Tests vor Erstellen einer Programmfunktion
→ alle Tests erfolgreich, damit ein Inkrement in System integriert werden kann

Prinzipien und Verfahren

- inkrementelle Planung: Anforderungen auf Story-Cards → Priorität entscheidet, ob in Release mit aufgenommen
- kleine Releases
- einfacher Entwurf
- Test-First-Entwicklung: Vor Impl. Test schreiben
- Refactoring: Verbesserungsmanagement, beim Code sofort anwenden → einfach und wartbar
- Paar-Programmierung
- kollektives Eigentum: jeder arbeitet an allen Bereichen, keine „Experteninseln“
- kontinuierliche Integration: sobald Aufgabe abgeschlossen → Integration in Gesamtsystem → Durchlaufen aller Einzeltests
- Erträgliche Arbeitsgeschwindigkeit: große Mengen an Überstunden nicht akzeptabel
- Kunde vor Ort: Mitglied des Entwicklerteams, teilt Systemanforderungen mit
- SCRUM-Prozess: Folge von Sprints (festgelegte Zeitperiode, in



Releasezyklus bei Extreme Programming



SCRUM-Prozess

der ein Inkrement entwickelt wird)

Analysephase

Aufgaben

- Klärung und Konkretisierung der Aufgabenstellung; Ermittlung der Kernanforderungen
- Analyse der Ist-Situation
- Abgrenzung der zu entwickelnden Systeme; Feststellung von zu beachtenden Restriktionen
- Aufzeigen von alternativen Lösungsmöglichkeiten zur Zielerreichung
- Überprüfung der Durchführbarkeit von Lösungsalternativen in technischer, ökonomischer, fachlicher, personeller Hinsicht
- Empfehlung und Entscheidung über Lösungsalternativen

Ablauf, Lastenheft, Pflichtenheft

1) Zusammenstellung der Benutzeranforderungen: Beschreibung der benötigten Systemfunktionen und Randbedingungen
2) **Lastenheft**: vom Auftraggeber erstellt, Systemanforderungen, Problembeschreibung *aus Kundensicht*
3) Erstellung einer Vorstudie: Lösungsalternativen, Untersuchung d. Realisierbarkeit + Wirtschaftlichkeit
4) Vorstudie: beschreibt grob die Lösungsalternative; Empfehlung für eine Alternative aus Entwicklersicht
5) Entscheidung für eine Lösungsalternative
6) **Pflichtenheft**: baut auf Lastenheft auf, spezifiziert Anforderungen einer Lösungsalternative aus Entwicklersicht, Realisierungsvorgaben für Softwareentwicklung, sollte vertraglicher Bestandteil sein

Einbringen d. Beteiligten (sog. Stakeholder)

- um Akzeptanz zu erreichen
- jede Gruppe hat spezifische Sicht auf System → spezifische Anforderungen
- nachträgliche Anforderungen sind kostenintensiv → Anforderungen aller Beteiligten vor Realisierung

Anforderungen

Eine **funktionale Anforderung** legt fest, was das System tun soll, z. B.:
„Das System sollte dem Bibliothekar die Möglichkeit bieten, einen manuell ausgelösten Datentransfer mit den Nachbarsystemen durchzuführen."
Unterkategorien: *Verhaltensanforderungen* (Zustandsdiagramm), *Strukturanforderungen* (Klassendiagramm), *Funktionsanforderungen* (Aktivitätsdiagramm)

Eine **nicht-funktionale Anforderung** legt fest, welche Eigenschaften das System haben soll, z. B.:
„Das Bibliothekssystem soll den Datentransfer zu den Nachbarsystemen in der Zeitspanne von 22:00 Uhr bis 06:00 Uhr durchführen."
Unterkategorien: *Qualitätsanforderungen* (beziehen sich auf Qualitätsmerkmale, die nicht durch funktionale Anforderungen Abgedeckt sind z. B. Qualität, Effizienz, Zuverlässigkeit), *Randbedingungen* (z. B. Schnittstellen, Hardware, rechtliche Aspekte, die Umsetzungsmöglichkeiten einschränken)
sollten zeitlich zusammen mit den funktionalen definiert werden; Bezug zu den funktionalen Anforderungen sollte klar gekennzeichnet sein; können durch zusätzliche funktionale Anforderungen konkretisiert werden

Anforderungsspezifikation

systematisch dargestellte Sammlung von Anforderungen., die vorgegebenen Kriterien genügen

- Gründe: zentrale Bedeutung von Anforderungen, rechtliche Relevanz, Komplexität, Zugreifbarkeit
- Verwendung: Änderungsmanagement, Architekturentwurf, Planung, Text, Nutzung und Wartung, Implementierung, Vertragsmanagement
- **Qualitätskriterien nach IEEE 830-98**: bewertet (nach rechtlichen Verbindlichkeiten/Prioritäten), eindeutig, korrekt, konsistent, prüfbar, verfolgbar (Ursprung, Traceability, Beziehungen), vollständig (auch formale Gesichtspunkte), modifizierbar, erweiterbar (Inhalt und Struktur sollten Änderbarkeit unterstützen)
- zusätzliche Merkmale: klare Struktur, angemessener Umfang, hohe Qualität aller einzelnen Anforderungen, abgestimmt, gültig/aktuell, realisierbar, verständlich, nur eine Anforderung pro Satz, kurze Sätze/Absätze (max. 7)
- Glossar: Steigerung der Verständlichkeit, Vermeidung von Missverständnissen, kontextspezifische Fachbegriffe, Abkürzungen und Akronyme, alltägliche Begriffe mit spezieller Bedeutung, Synonyme, Homonyme (Begriff mit verschiedenen Bedeutungen)

Perspektiven

- Strukturperspektive: statisch-strukturell (Ein- und Ausgabestruktur), Nutzungs- und Abhängigkeitsbeziehungen im Systemkontext
- Funktionsperspektive: Welche Daten fließen vom System in den Kontext, welche werden durch System manipuliert?
- Verhaltensperspektive: zustandsorientiert, System und Einbettung in Kontext (Reaktion auf Ereignisse)

Natürlichsprachliche Dokumentation

Vorteile: jedem verständlich, kein Erlernen einer Notation nötig, für alle Anforderungen Einsetzbar, von Stakeholdern eher akzeptiert
Nachteile: mehrdeutig/missverständlich, Isolation der Anforderungen für nur eine Perspektive schwierig

Modellbasierte Dokumentation

Vorteile: Anforderungen können isoliert in 3 Perspektiven dargestellt werden, kompakte und für geübten Leser eindeutig verständliche Darstellung, Vermeidung v. Missverständnissen
Nachteile: kein universaler Einsatz möglich, Kenntnis der Notation nötig

Standardgliederungen

erleichtern Einarbeiten neuer Mitarbeiter, ermöglichen selektives Lesen/Überprüfen, schnelleres Erfassen ausgewählter Inhalte, einfache Wiederverwendung von Inhalten

- Volere: Aufnahme von Randbedingungen, zugeschnitten auf objektorientierte Entwicklung, eher bei SW-Entwicklung, guter Blick auf ganzes System
- IEEE 830-98:
  - Einleitung: einführende Informationen (Zweck, Begriffsdefinitonen)
  - allgemeine Übersicht: allgemeine Beschreibung der Software (Perspektive, Einschränkungen)
  - Anforderungen: spezifische Anforderungen (funktionale Anforderungen, Performance)

- V-Modell d. Bundes:
  - Lastenheft: wird von Auftraggeber erstellt, Was?, Wofür?, Anwendersicht, Gesamtheit der Forderungen
  - Pflichtenheft: setzt Lastenheft um, Konkretisierung der Anforderungen, Realisierungsvorgaben vom Auftragnehmer, auf Vorgaben des Lastenhefts

Standardinhalte

- Einleitung (Zweck, Umfang, Stakeholder, Definitionen, Referenzen, Übersicht)
- Allgemeine Übersicht (Systemumfeld, Architekturbeschreibung, Funktionalität, Randbedingungen, Nutzer&Zielgruppen, Annahmen)
- Anhang: weiterführende Informationen
- Index: Inhaltsverzeichnis, Indexverzeichnis

Anforderungen natürlichsprachlich dokumentieren

- Auflösen von Normalisierungen (Speicherung, Archivierung, …)
- keine schwammigen Substantive (der Anwender, die Meldung, die Funktion, …) verwenden
- Vorsicht mit Mengenwörtern wie alle, jeder, immer, …
- keine unvollständige Bedingungsstrukturen (wenn, …, dann …, im Falle von, falls …)
- W-Fragen beantworten

Satzschablone

einheitlich, automatische Reduktion von sprachlichen Effekten, für alle Arten von Systemen einsetzbar

Das System *X* (z. B. Das Bibliothekssystem) — verpflichtend: muss / wünschenswert: soll / verpflichtend, in Zukunft: wird — fähig sein *wem?* die Möglichkeit bieten *Objekt* & *Ergängzung* (ein Leihobjekt als beschädigt zu markieren) *Prozess*

Entwurfsphase (aka Definitionsphase, Systemspezifikation)

Ergebnisse: Fachentwurf und Fachkonzept

Fachentwurf

- verfeinerte Analyse der zugrunde liegenden Informationsstrukturen und deren Beziehungen aus fachlicher Sicht
- weitere Zergliederung der Fachaufgaben nach dem Top-Down-Prinzip
- Entwurf der Benutzeroberfläche (Menüs, Masken, Fenster, Listen, falls noch nicht geschehen)
- Vorbereitung der Tests durch die Spezifikation fachbezogener Testfälle
- Erarbeitung eines Organisationskonzeptes zur Integration der Anwendung
- Aufstellung eines Schulungsplans

Fachkonzept

- vollständige Beschreibung der fachlichen Anforderungen, Grundlage für IT-technische Realisierung, höherer Detaillierungsgrad als Vorstudie, für alle Beteiligten verständlich
- Ergebnistypen: Informationsmodell (ER), Funktionsmodell (HIPO, SAOT), Objektmodell (OOA), Ereignismodell, Oberflächenmodell, Organisationsmodell

IT-Entwurf → IT-Konzept

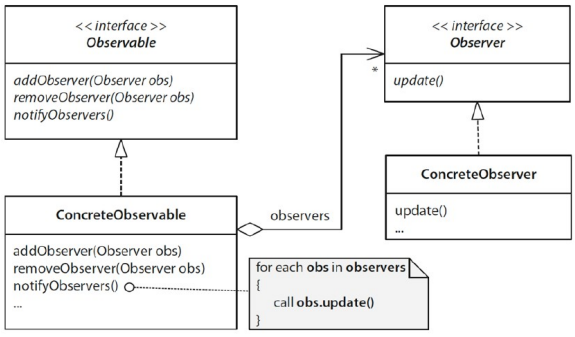
Aufgaben:

- Überarbeitung/Präzisierung der fachlichen Modelle
- Ergänzung der fachlichen Modelle um informationstechnisch erforderliche Klassen
- Festlegung der Datei-/Datenbankkonzepts auf der Grundlage des Informationsmodells
- Strukturierung/Zusammenfassung der Anwendung zu Softwarekomponenten mit klar spezifizierten Schnittstellen
- Festlegung der vertikalen Systemarchitektur mit klar definierten, abgegrenzten Schichten
- Festlegung der horizontalen Systemarchitektur (Verteilung der Komponenten, Datenbanken, … auf Rechnerknoten)
- Festlegung der Kommunikation zwischen den einzelnen Systemkomponenten
- Fortschreibung der Testkonzepte und Schulungskonzepte

Ergebnistypen: Datei- und Datenbankentwurf; Softwarearchitektur mit: festgelegten Komponenten mit spezifizierten Schnittstellen/ festgelegten Softwareschichten, Verteilungskonzept, Sicherungskonzept, Sicherheitskonzept, Kommunikationskonzept, Fehlermanagement, Test- und Schulungskonzept

Patterns

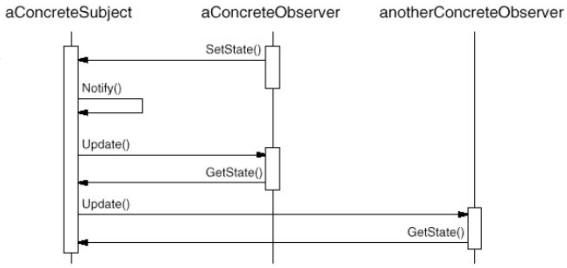
- Regeln, um häufig auftretende Probleme bei der Erstellung von Software zu lösen
- Vorgehensmuster, die Programmierer unterstützt, ein konkretes Problem zu abstrahieren, zu strukturieren, eine Lösung zu erarbeiten und die Lösung auf das konkrete Problem abzubilden
- Abstraktionen und Vokabular zur Kommunikation unter Programmieren
- unterstützen Dokumentation von Software Architekturen
- Katalysatoren in der Softwareentwicklung, um Wiederverwendung, Formbarkeit (malleability) und Modularität zu erreichen (**hohe Kohäsion**, **lose Kopplung**)



Observer Pattern (Design Pattern)

- Kontext: Objektsystem mit Zustandsabhängigkeiten zwischen Objekten
- Beispiel: grafische Bedienoberfläche → mehrere Sichten (Observer) auf das gleiche Datum (Observable/Subject)
- Problem: Der Zustand mehrerer Objekte, die miteinander in Beziehung stehen, soll konsistent gehalten werden.

Das bedeutet im Detail:
    - Veränderung eines Objekts soll „automatische“ Anpassung des Zustands eines anderen Objekts zur Folge haben
    - Objekte, welche konsistent gehalten werden sollen, sind Instanzen von Klassen, die einander nicht „kennen“ → lose Koppelung: Es soll erreicht werden, dass beteiligte Klassen unabhängig voneinander wiederverwendet werden können
- Teilnehmer/Rollen:
    - „Observable/Concrete Subject“: abstrakte Klassen zur Verwaltung einer beliebigen Zahl von „Observer“-Instanzen, hat Liste von „Observer“ Instanzen (kennt aber nicht deren konkrete Klasse), hat Schnittstelle zum An- und Abmelden von diesen, hat Schnittstelle zur Benachrichtigung von „Observern“ über Änderungen
    - „Observer“: Schnittstelle für Objekte, die an Zustandsänderungen eines „Concrete Subject“-Objekts interessiert sind
    - „Concrete Subject“: definiert Zustand, an dem „Observer“-Objekte Interesse haben; benachrichtigt alle „Observer“ bei Zustandsänderungen; hat Schnittstelle zur Abfrage des aktuellen Zustands
    - „Concrete Observer“: reagiert auf Zustandsänderungen der assoziierten Instanz von „Concrete Subject“



- Verhalten:
    - „Observer“ haben sich vorher beim Objekt in der „Concrete Subject“-Rolle registriert.
    - „SetState()“ ruft „Observable::Notify()“ auf, welches wiederum „Update()“ auf registrierten „Observer“-Objekten aufruft.
    - Diese fragen neuen Zustand mittels der „GetState()“ Methode des „Concrete Subject“ ab um ihren Zustand zur Änderung des „Concrete Subject“-Objekts konsistent zu halten.

Implementierungsphase: Java-Code-Conventions

- Wichtig, da ca. 80% d. Lebenszeit eines Systems ist Wartung, Software wird praktisch niemals nur von einem Autor gewartet, Lesbarkeit wird erhöht, Software soll sauber sein (wie jedes andere zu verkaufende Produkt)
- Reihenfolge: Kommentare zu Beginn, Paket & Import, Klasse/Interface-Doku, Klasse/Interface, Implementierungskommentar, statische Felder, Felder (erst public, protected, dann private), Konstruktoren, Methoden (von wichtig nach unwichtig, private bei Erstverwendung)
- Einrückung:
    - 4 Leerzeichen Einrückung
    - Zeilen mit mehr als 80 Zeichen vermeiden
    - Zeilenumbruch: nach Komma, vor einem Operator, Ausrichten d. Ausdrucks an vorheriger Zeile, falls Ergebnis d. Vorhergehenden Regeln verwirrender Code → Einrückung um 8 Leerzeichen
    - Beachten von Bindungsrichtlinien
- Kommentare
    - don't comment bad code – rewrite it
    - unzweckmäßig für Informationen, die besser in anderem System aufgehoben sind
    - Änderungshistorie füllt Source Code nur mit uninteressanten, historischem Text
    - Metadaten (Autor, Änderungsdatum,…) im allgemeinen nicht geeignet für Kommentar
    - sollen sich ausschließlich Code und Architektur aus technischer Sicht widmen
    - Vermeidung von obsoleten und redundanten Kommentaren
    - keine Offensichtlichkeiten, kurz fassen
    - auskommentierter Code: löschen, da über Versionsmanagementsystem wiederherstellbar
- Leerzeilen: erhöhen Lesbarkeit; 2 Leerzeilen zwischen Abschnitten einer Source-Datei bzw. Klassen/Interface-Deklarationen; 1 Leerzeile zwischen Methoden, lokalen Variablen und erster Anweisung in Methode, vor einem Block/Ein-Zeilen-Kommentar, zwischen logischen Abschnitten in einer Methode
- Leerzeichen: Java-Schlüsselwort gefolgt von Klammern (while (true)), nach Kommas in einer Argumentliste, alle binären Operatoren (außer „.“), in for-Anweisung, nach cast Anweisung
- Namenskonventionen: Interface-/Klassennamen als Nomen in PublicCase, Methoden als Verben in camelCase Variablen als Kurzbezeichner in camelCase, Konstanten in Großbuchstaben
- Programmierpraxis: Zugriff auf Klassenmethoden über Klassenname, nummerische Konstanten nicht direkt codieren, Vermeidung von mehreren Wertzuweisungen in einer Zeile, Klammerung trotz Vorrangregeln, Struktur soll Einrückung widerspiegeln

Testphase

Fehlerbegriff

- Fehler: Nichterfüllung einer festgelegten Anforderung; Abweichung zw. Ist- und Sollverhalten
- Mangel: Nicht angemessene Erfüllung einer gestellten Anf. Oder berechtigten Erwartung
- Jeder Fehler/Mangel seit Zeitpunkt d. Entwicklung vorhanden, er kommt jedoch erst bei Ausführung d. SW zum Trage → failure (Fehlerwirkung, Fehlfunktion, äußerer Fehler, Ausfall)
- Zwischen Auftreten einer Fehlerwirkung und deren Ursache muss unterschieden werden → fault (Fehlerzustand, Defekt, innerer Fehler) → über Debugging herausfinden
- wird Fehlerzustand durch einen/mehrere andere Fehlerzustände kompensiert → Fehlermaskierung

Testbegriff

- Definition: jede Ausführung eines Testobjekts, die der Überprüfung d. Testobjekts dient; Vergleich Ist- & Sollverhalten;

festgelegte Randbedingungen
- Ziele: Fehlerwirkungen nachweisen, Qualität bestimmen, Vertrauen in Programm erhöhen, Fehlerwirkungen vorbeugen
- Testen kann Fehlerfreiheit nicht nachweisen!
- Validieren(Entwickle ich das richtige System?) ↔ Testen/Verifikation(Lauffähiger Prototyp konform zur Anforderung)

Komponententest (Unit-Test)

erstmals systematischer Test, der erstellten Softwarebausteine (Testobjekte) → Ausschließung komponentenexterner Einflüsse

Grundlegende Testarten

- Funktionaler Test: subsumiert alle Testmethoden, mittels derer das von außen sichtbare Ein-/Ausgabeverhalten eines Testobjekts geprüft wird (Blackbox), basiert auf funktionalen Anforderungen
- Nicht-funktionaler Test: Lasttest, Performancetest, Stresstest, Test der (Daten-)Sicherheit, Test auf Benutzungsfreundlichkeit
- Strukturbezogener Test: basiert auf internen Architektur der Software (Whitebox) → Kontrollfluss, Aufrufhierarchie, Menüstruktur
- Regressionstest: erneuter Test eines bereits getesteten Systems nach dessen Modifikation mit dem Ziel nachzuweisen, dass durch die Änderungen keine neuen Defekte eingebaut oder Fehlerzustände freigelegt werden

Blackbox-Verfahren

innerer Aufbau eines Testobjekts nicht bekannt → Testfälle aus Spezifikation abgeleitet
Äquivalenzklassenbildung: Unterteilung aller möglichen konkreten Eingabewerte in Äquivalenzklassen (Testobjekt verhält sich für Werte in einer Äquivalenzklasse gleich), Test eines Repräsentanten ist ausreichend, ungültige Werte müssen auch getestet werden, die Klassen müssen ja nach Anwendungsfall weiter aufgeteilt werden

Allgemeine Prinzipien des Softwaretestens

1. Testen zeigt Anwesenheit von Fehlern → verringert Wahrscheinlichkeit, dass noch unentdeckte Fehlerzustände im Objekt vorhanden sind (aber keine Aussage über Qualität des Codes!)
2. Vollständiges Test ist nicht möglich! → Tests = Stichproben; Testaufwand nach Risiko und Prioritäten steuern
3. Mit Testen frühzeitig beginnen → Fehler werden frühzeitig erkannt (TestDrivenDevelopment)
4. Häufung von Fehlern → dort wo Fehlerwirkungen nachgewiesen wurden, finden sich meist noch weitere, beim Testen muss flexibel darauf reagiert werden können
5. zunehmende Testresistenz → regelmäßiges Prüfen und Ergänzen der Tests
6. Testen ist abhängig vom Kontext → keine 2 Systeme sind auf exakt gleiche Art zu testen
7. Trugschluss: keine Fehler bedeutet ein brauchbares System
8. für Tester besonders wichtige Qualitätsanforderungen: Prüfbarkeit und Vollständigkeit (im Vergleich zu Robustheit und Konsistenz)

Logischer Testfall	1	2	***Konkreter Testfall***	1	2
Eingabewert x (Firmenzugehörigkeit)	x <= 3	3 < x <= 5	Eingabewert x (Firmenzugehörigkeit)	2	4
Erwartetes Ergebnis (Gratifikation in %)	0	50	Erwartetes Ergebnis (Gratifikation in %)	0	50

zuerst definieren (später Konkretisierung)
Auswahl erfolgt auf jeweiliger Testbasis → Spezifikation der Testobjekte (Blackbox) oder Grundlage des Programmtextes (Whitebox)

Konkretisierung der logischen Testfälle
→ konkrete Eingabewerte

UML (= Useless Modeling Language)

Ebenen

- Instanzebene (Metaebene 0), z. B. Sequenzdiagramm, Objektdiagramm
- Typebene (Metaebene 1), z. B. UML-Klassendiagramm
- Metaebene 2, z. B. Sprachdefinition

Nutzen

Analyse des Anwendungsbereichs → besseres Verstehen, Abgrenzung des Systems nach außen → Beschreibung der externen Schnittstellen, Herleitung der benötigten Klassen, Auftragsbestandteil

Stereotyp/Verwendungskontext

- gibt Verwendungszusammenhang an
- ermöglicht Einführung von Modellierungselementen zur Modellierungszeit
- z. B. Angabe, dass es sich um ein «Interface», «Enumeration», «Actor» oder «Use Case» handelt

Objektorientierung, Klassen vs. Objekte

Objektorientierung: beinhaltet gemeinsame Betrachtung von Datenstrukturen und Methoden in programmtechnischen Einheiten

Objekt
- Gegenstand der Erkenntnis, Wahrnehmung, des Denkens und Handels
- ein Objekt der objektorientierten Analyse ist eine Abstraktion des Problemfeldes

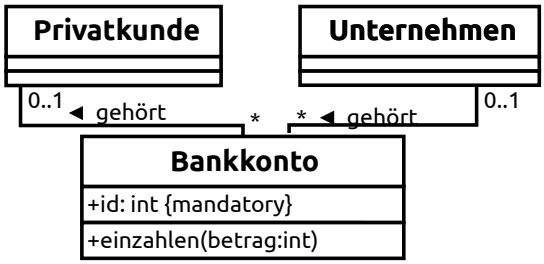
Klasse:
- Beschreibung einer Menge von Objekten mit gleichen Attributen, Operatoren, Beziehungen und Semantiken
- Abstraktionen, die Teil der Problem- bzw. Lösungsdomäne sind
- beinhaltet kleine, wohldefinierte Menge von Verantwortlichkeiten und führt diese gut aus
- mehrere Klassen können untereinander in Beziehung stehen (sollten dann im gleichen Klassendiagramm stehen)

CRC (Class-Responsibility-Karten)

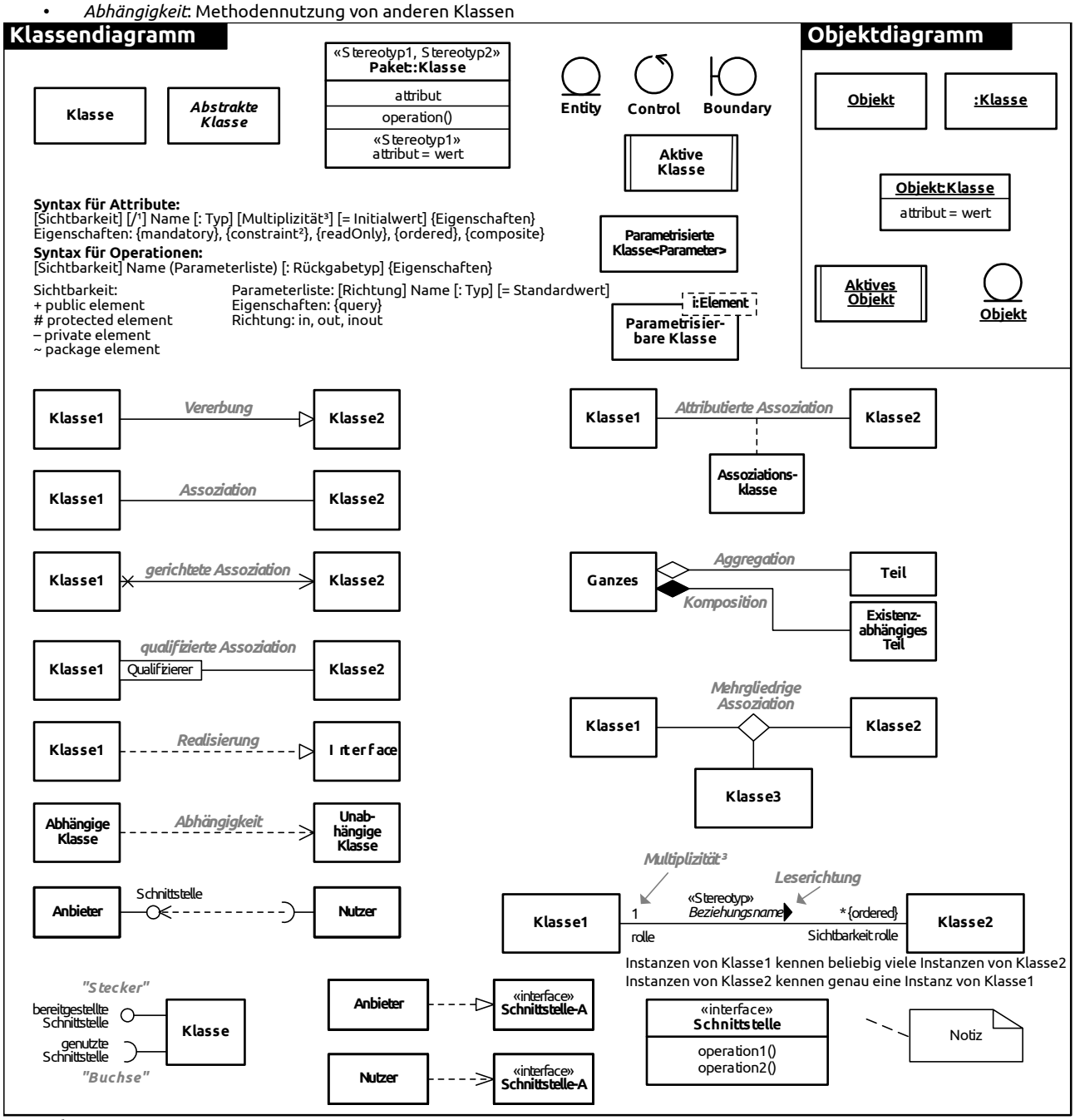
eine Karteikarte pro Klasse mit Klassenname, Verantwortlichkeiten und Klassen mit denen zusammengearbeitet wird

Klassen- und Objektdiagramm

Beziehungen: Abhängigkeit, Assoziationen, Realisierung



- *Assoziation*: Kenntnisbeziehung (und damit auch Abhängigkeiten) zwischen Klassen/Objekten
- *Aggregation*: Is-Part-of-Beziehung/Teil-Ganzes-Beziehung (keine Einschränkung bezüglich Multiplizitäten)
- *Komposition*: Spezialfall der Aggregation (jedes Objekt der Komponentenklasse darf nur Komponente von genau einem Objekt der Oberklasse sein; Teilobjekte werden gelöscht, wenn Aggregatobjekt gelöscht wird)
- *Kardinalität*: Anzahl der Instanzen, mit denen eine Instanz tatsächlich in Beziehung steht
- *Multiplizität*: zulässiger Wertebereich der Kardinalitäten
- *Abhängigkeit*: Methodennutzung von anderen Klassen



Anmerkungen

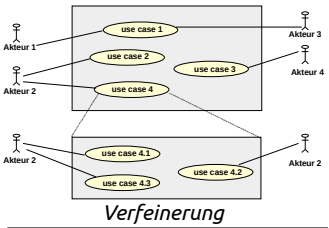
[1] *Abgeleitetes Attribut*: konkreter Wert wird aus Werten anderer Attribute hergeleitet (beispielsweise Kreisfläche aus Radius)

[2] *Constraint*: Bedingung z. B. {a > 0}; auch Wertbereiche möglich, z. B. {0, 1, …, 9}

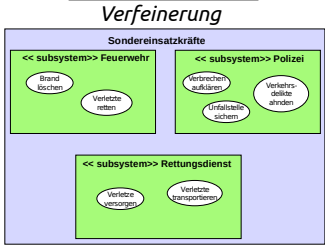
[3] *Multiplizität*: gibt an, wie oft ein Attribut verwendet wird bzw. bei Beziehungen die Anzahl der Instanzen, mit denen eine Instanz tatsächlich in Beziehung steht

Klassenattribute und -operatoren: meint statische Felder/Funktionen; durch Unterstreichung kennzeichnen

Use Cases



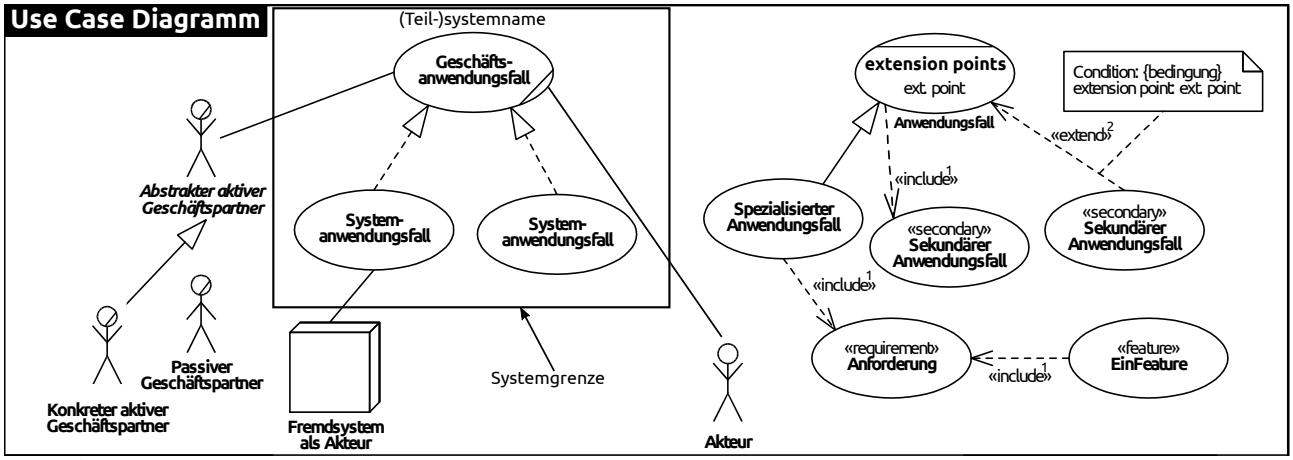
Verfeinerung



- können/sollten Bestandteil des Pflichtenheftes sein
- sind Abstraktionsebene oberhalb von Klassen (aus Sicht des Benutzers)
- bestehen aus mehreren zusammenhängenden Aufgaben
- haben festgelegte Reihenfolge
- Ausnahmen, Alternativen möglich
- evtl. Verweis auf Ablaufdiagramme
- können hierarchisiert werden (siehe Bsp. „Sondereinsatzkräfte“)
- Assoziationen zwischen Use Cases sind verboten
- Akteure sind im Use Case Diagramm über Assoziationen mit Use Cases und evtl. auch anderen Akteuren verbunden; Use Cases dürfen nicht nicht alleine stehen
- „Verfeinern“ von Use Cases möglich (siehe Bsp.)
- kennzeichnen das Verhalten eines Systems aus Benutzersicht
- spezifizieren *nicht*, wie eine Aufgabe später implementiert werden soll

Akteur

etwas oder jemand außerhalb des Systems; ein Use Case muss durch Akteur initiiert werden; helfen benötigte Systemfunktionalitäten aufzudecken; kann auch anderes IT-System sein
Variante: Menge von Rollen, die Anwender/andere Systeme im Bezug auf Use-Case wahrnehmen



Anmerkungen

[1] «*include*»: zum „Outsourcen“ von gemeinsamen Verhalten, das zwei oder mehr Use Cases besitzen; inkludierte Use Cases können nicht alleine existieren, sondern werden immer als Teil eines anderen Use Cases ausgeführt.

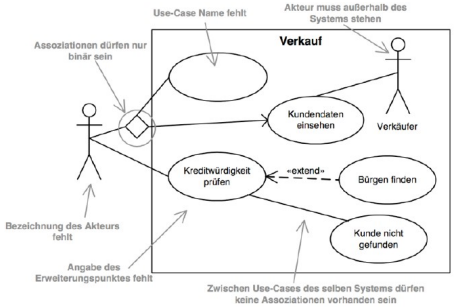
[2] «*extend*»: kennzeichnet Erweiterung (zusätzliches Verhalten unter bestimmten Bedingungen); die erweiternden Use Cases kennzeichnen Ausnahmefälle

Use Case Beschreibung

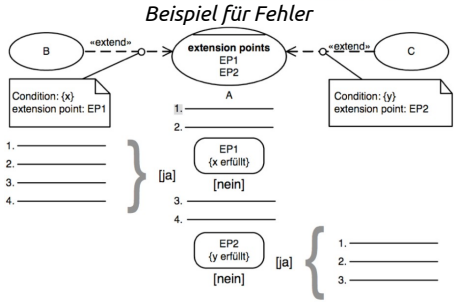
Sollte möglichst formal erfolgen, z. B.:

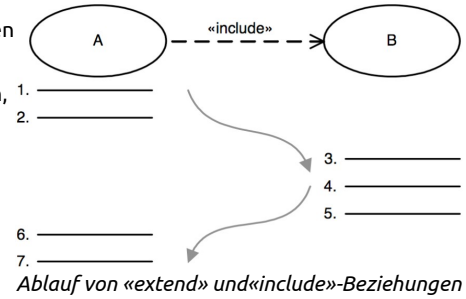
Name des Use Case:	z. B. Linie folgen
Nummer:	eindeutige Zahl
Geltungsbereich:	zugehöriges System/Teilsystem, z. B. Roboter
Level:	z. B. Hauptfunktionalität/Unterprogramm
Kategorie:	z. B. primär, sekundär oder optional
Beteiligte Klassen/Objekte:	z. B. Movement, LightSensor
Vorbedingungen:	z. B. Programm gestartet
Nachbedingungen (Erfolg):	z. B. ins Menü zurückkehren
Nachbedingungen (Fehlschlag):	z. B. Fehlermeldung auf Display ausgeben
Akteure:	etwas oder jemand aktives außerhalb des Systems, z. B. Anwender
Auslösendes Ereignis:	z. B. Programm im Menü gestartet
Auszuführende Aktionen:	Auflistung der einzelnen Aktionen in ihrer Reihenfolge
Nachbedingungen:	z. B. ins Menü zurückkehren
Erweiterungen:	entfallen eigentlich immer
Aktionen für Ausnahmen:	beziehen sich auf einzelne auszuführende Aktionen

reduzierte Use Cas Beschreibung: Name, Nummer, Art, Kurzbeschreibung, Auslöser, Ergebnis, Akteure, Eingehende Informationen, Vorbedingungen, Nachbedingungen, Essenzielle Schritte



Beispiel für Fehler





Ablauf von «extend» und«include»-Beziehungen

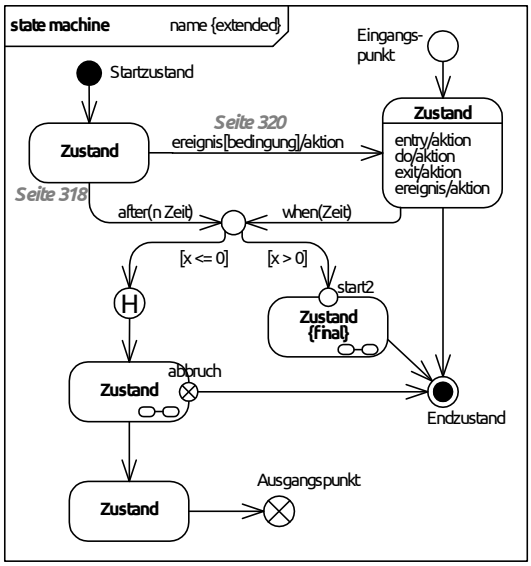
Zustandsdiagramme

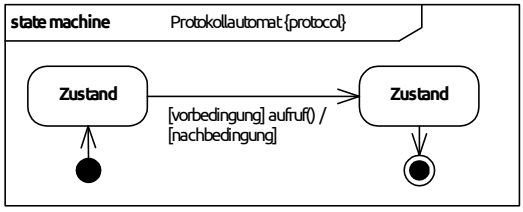
- stellen Reaktionen der Klassenobjekte auf Ereignisse dar
  → hilft Methoden zu finden, Lebenszyklus einer Klasse zu beschreiben, Test systematisch durchzuführen
- stellen Zustände der Klassenobjekte, Zustandsübergänge (Transitionen),

Ereignisse (entsprechen systemseitig dem Aufruf von Methoden), Bedingungen, und Aktionen (entsprechen Methoden) dar

- stellen Aktionen, die während eines Zustandes auszuführen sind dar
  - zu Beginn eines Zustandes
  - beim Verlassen eines Zustandes
  - während eines Zustandes
  - während eines Zustandes in Abhängigkeit von einem Ereignis, ohne dass sich der Zustand ändert
- ermöglichen somit, das dynamische Verhalten einer Klasse zu modellieren

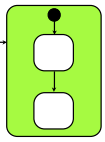
Zustandsdiagramm



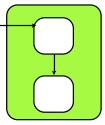


Betreten von zusammengesetzten Zuständen

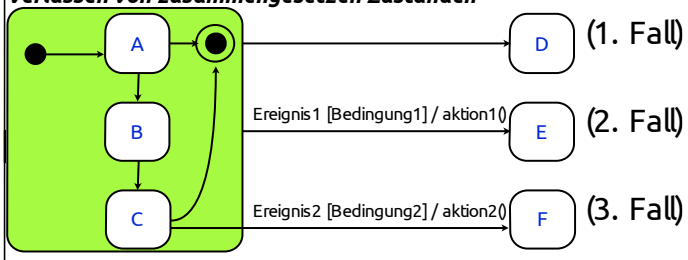
1. Fall: default_entry
Der zusammengesetzte Zustand wird mit dem Startknoten begonnen; ein Startknoten ist in diesem Fall notwendig.



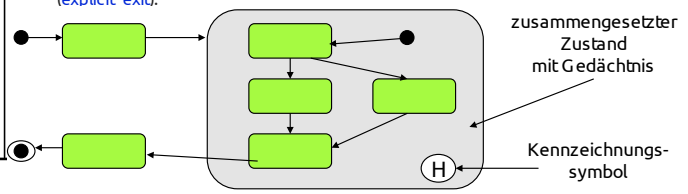
2. Fall: explicit_entry
Der zusammengesetzte Zustand wird mit dem Zustand begonnen, auf den die Transition zeigt; ein Startknoten ist nicht notwendig.



Verlassen von zusammengesetzen Zuständen



1. Fall: Der zusammengesetzte Zustand trifft auf das Endesymbol, der Zustand D wird angenommen (default_exit).
2. Fall: Der zusammengesetzte Zustand ist in einer der Zustände A, B oder C und das Ereignis1 tritt ein; danach wird der Zustand E angenommen, falls die Bedingung 1 erfüllt ist (explicit_exit).
3. Fall: Der zusammengesetzte Zustand ist im einfachen Zustand C; das Ereignis2 tritt ein und der Zustand F wird angenommen, falls die Bedingung 2 erfüllt ist (explicit_exit).



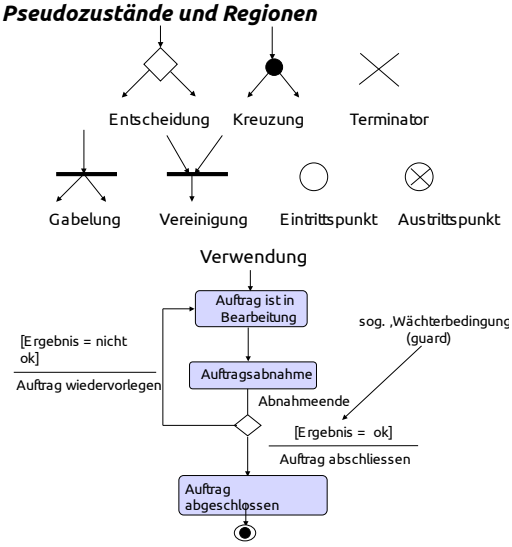
Anmerkungen

Jedes Zustandsdiagramm muss mindestens einen Endknoten besitzen. Innerhalb eines Zustandes können mit „do“ auch Ereignisse dargestellt werden, die keine Transition auslösen.

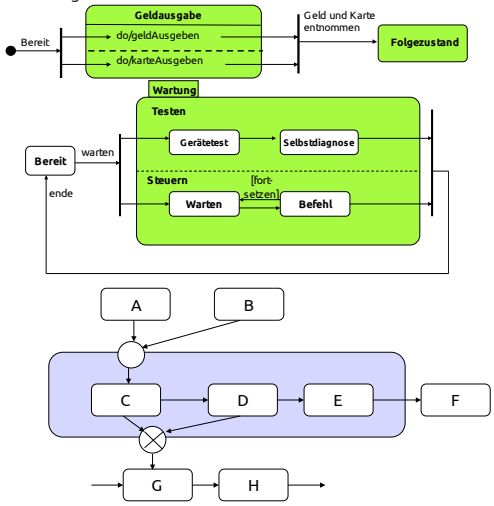
Zusammengesetzter Zustand mit Gedächtnis

Beim Verlassen des Zustandes wird sich der letzte Unterzustand gemerkt. Bei erneuten Eintritt in den Zustand erfolgt die Aktivierung nicht beim Startzustand, sondern bei dem zuletzt aktiven Unterzustand.

Pseudozustände und Regionen

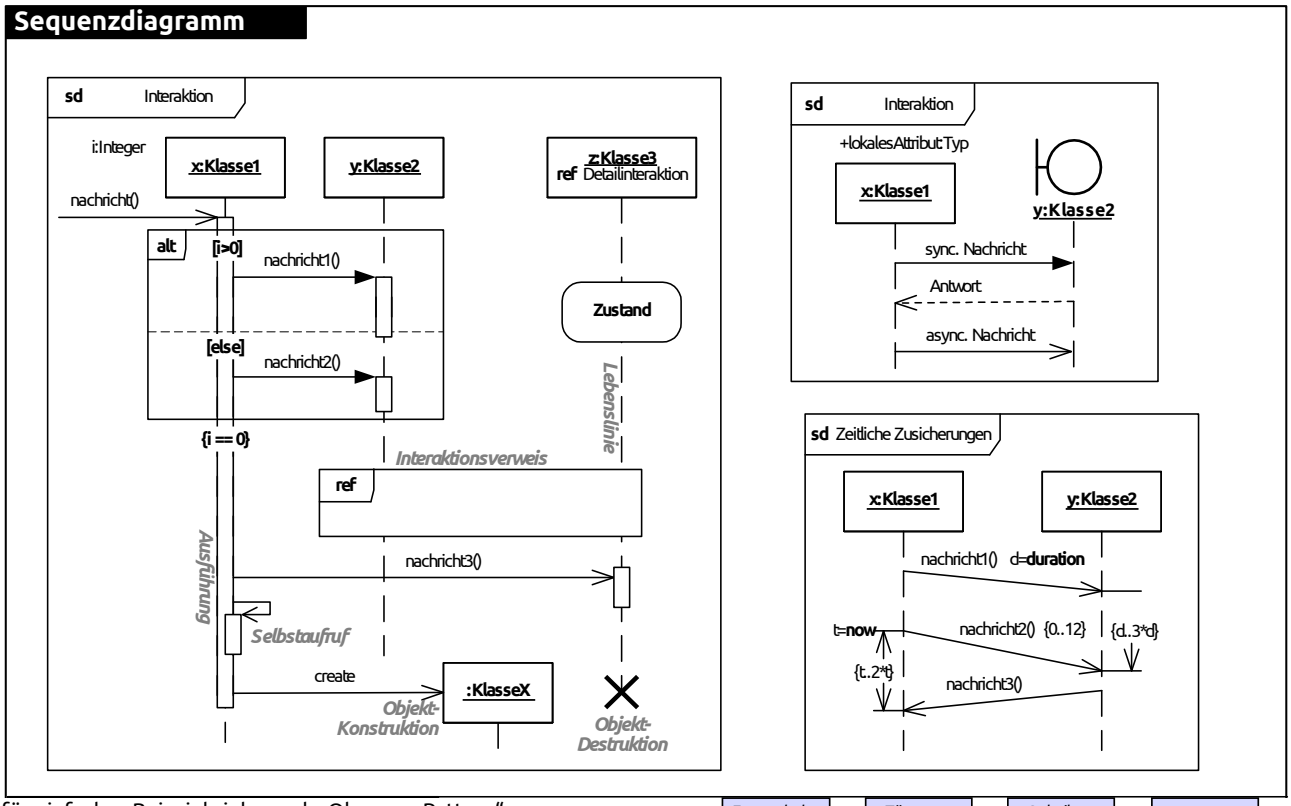


Sofern in einem Zustand parallele Aktionen auszuführen sind, kann der Zustandsautomat in Regionen aufgeteilt werden, die durch gestrichelte Linien getrennt dargestellt werden.



Sequenzdiagramme

- zeigen Interaktionen zwischen mehreren Kommunikationspartnern (vor allem Instanzen von Klassen)
- geben Aufschluss darüber, wer mit wem Informationen austauscht
- zeigen den zeitlichen Ablauf der Kommunikation („von oben nach unten“)
- relativ detaillierte Darstellung einer Interaktion
- Erstellung allerdings zeitintensiv
- nicht jeder Ablauf in einem komplexem System kann modelliert werden
- hoher Detaillierungsgrad erfordert hohen Wartungsaufwand



für einfaches Beispiel siehe auch „Observer Pattern“

Asynchrone Nachrichten

Nach dem Versenden einer Asynchronen Nachricht, muss ein Kommunikationspartner nicht auf eine Antwort warten, sondern kann direkt weitere Aktionen durchführen.

Zustandsinvariante

während des Szenarios geltende Bedingung

- Zustand, der von Instanz eingenommen wird, sobald alle *vorhergehenden* Ereignisse der Lebenslinie aufgetreten sind
- Element (durch Lebenslinie repräsentiert), muss ich in abgebildeten Zustand befinden, wenn Interaktion fortgesetzt wird