Objektorientiertes Programmieren mit C++ für Fortgeschrittene

Kapitel 7

7. Entwicklung von OOP-Programmen

7.1. Entwicklungsprozess

7.2. Modellierung (UML)

7.3. Technische Hilfsmittel

Der Softwareentwicklungsprozess (Überblick)

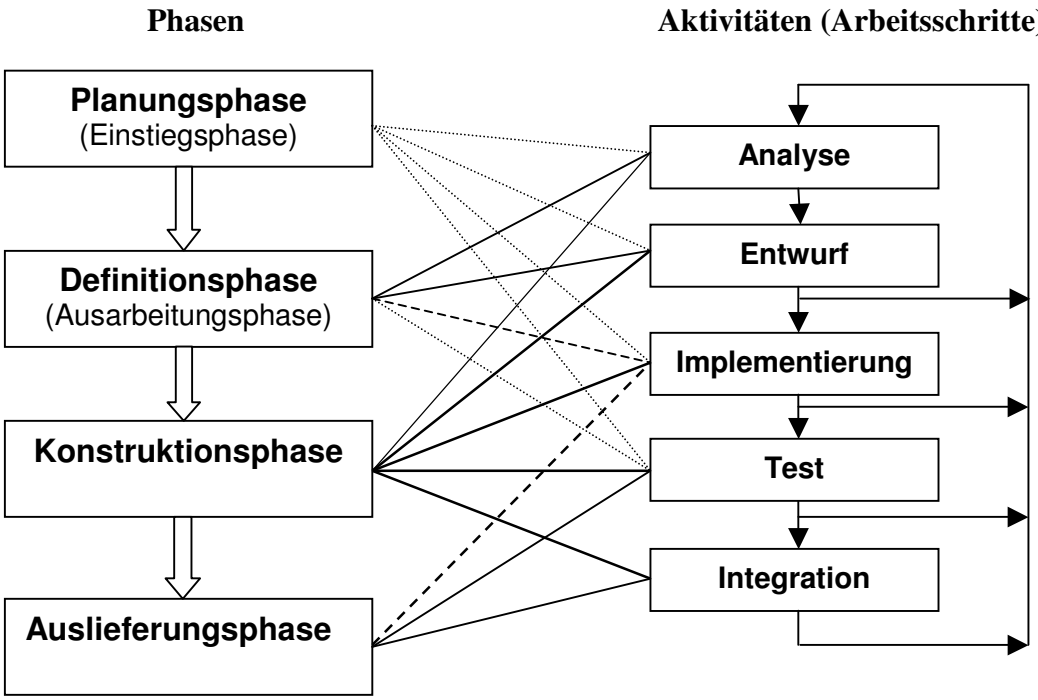
- **Entwicklungsziel** :

  ◇ Das **Ergebnis eines erfolgreichen Softwareentwicklungsprozesses** soll ein **Software-System** sein, dass
  - ▻ die vom Benutzer gestellten **funktionalen** und **nichtfunktionalen Anforderungen und Erwartungen** erfüllt
  - ▻ **zeitgerecht** und **wirtschaftlich** entwickelt wurde
  - ▻ **änderungs**- und **anpassungsfähig** ist

  ◇ Ein derartiger Prozess muss einerseits ein ausreichendes Maß an **Kreativität** und **Innovation** unterstützen, andererseits muss er die notwendige **Überprüfung** und **Steuerung des Entwicklungsfortschritts** sicherstellen.

  ◇ Diese Forderungen werden von einem **iterativen** und **inkrementellen** Entwicklungsprozess erfüllt.
  Ein derartiger **Prozess** besteht aus zeitlich nacheinander ablaufenden **Phasen**, innerhalb denen bestimmte **Aktivitäts-Zyklen** (Arbeitsschrittzyklen) **wiederholt** - mit teilweise unterschiedlicher Gewichtung - ausgeführt werden.

- **Phasen und Aktivitäten des Entwicklungsprozesses** :



  ◇ Insbesondere die **Konstruktionsphase** besteht aus zahlreichen **Iterationen**.
  In jeder Iteration wird ein vollständiger **Aktivitätszyklus** - z. Teil in sich auch wieder zyklisch – durchlaufen, wobei jeweils Software erstellt, getestet und integriert wird, die eine **Teilmenge der Gesamtanforderungen** des Projekts abdeckt.
  → Schrittweise Weiterentwicklung von **Systemprototypen** mit zunächst unvollständiger Funktionalität bis zum fertigen System.
  Ein vollständiger Aktivitätszyklus kann aber auch bereits in der **Planungsphase** und der **Definitionsphase** zur schnellen Erzeugung **explorativer Prototypen** durchlaufen werden.

  ◇ In jeder Entwicklungsphase können sich **Rückwirkungen** auf Ergebnisse und Festlegungen **früherer Phasen** ergeben, insbesondere kann derartig die Konstruktionsphase auf die Definitionsphase rückwirken.

  ◇ In **allen Phasen** und **Tätigkeiten** sollten die jeweiligen Ergebnisse und Festlegungen geeignet **dokumentiert** werden.

Die Phasen des Softwareentwicklungsprozesses

- **Planungsphase** (Einstiegsphase, Anfangsphase, Vorphase, inception phase)

  ◇ **Ziel** : Prüfung, ob ein Projekt realisiert werden soll.

  ◇ Festlegung von Zweck und Ziel eines Projektes,
  Durchführung von Trendstudien und Marktanalysen,
  Untersuchung der Durchführbarkeit
  (Aufwands-, Bedarfs- und Termin-Abschätzung, Risikoanalyse, Wirtschaftlichkeitsrechnung)
  ⇒ **Durchführbarkeitsstudie** (feasability study)
  bestehend aus : Lastenheft (grobes Pflichtenheft), Projektkalkulation und Projektplan

- **Definitionsphase** (Ausarbeitungsphase, Elaborationsphase, elaboration phase)

  ◇ **Ziel** : Festlegung der vollständigen, konsistenten, eindeutigen und durchführbaren Projektanforderungen

  ◇ Ermittlung und Analyse der Anforderungen (Systemanalyse)
  Problembereichanalyse (→erste Exploration von Klassen), Abgrenzung des Software-Systems zur Systemumgebung
  → Problembereichsmodell
  Identifikation der **Nutzungsfälle** (Use Cases) → Nutzungsfallmodell
  ⇒ **Anforderungsspezifikation (Pflichtenheft)**

  Entwurf der grundlegenden **Systemarchitektur** (→weitere Exploration von **Klassen**)
  Konzipierung der externen Schnittstellen (z.B. Benutzeroberfläche)
  Risikoanalyse und Priorisierung der Nutzungsfälle
  Festlegung der Entwurfsstrategie
  Planung der Systementwicklung in inkrementellen Entwicklungsschritten (→ Evolutionsplan, Entwurfsmodell)
  ⇒ **Entwurfsspezifikation**

- **Konstruktionsphase** (construction phase)

  ◇ **Ziel** : Erstellung eines die festgelegten Anforderungen erfüllenden Softwaresystems (Programm(e))

  ◇ Entwicklung einer detaillierten Systemstruktur
  Identifikation und Definition weiterer **Klassen/Objekte**
  Konzipierung und Analyse der statischen Klassenbeziehungen → **Klassendiagramme**
  Konzipierung und Analyse der dynamischen Objektinteraktion → **Interaktionsdiagramme**, **Kollaborationsdiagramme**
  → Erstellung **statischer und dynamischer Modelle**

  ◇ Erstellung des Systems in einer Folge von Iterationen (**zyklische Evolution**)
  (jeweils Analyse, Entwurf, Implementierung, Test und Integration)
  → inkrementelles Hinzufügen weiterer Funktionalitäten
  → iterative Änderung des Codes (z.B. Umstrukturierung)
  Validierung jeder Iteration durch einen geeigneten Test
  ⇒ **ausgetesteter Code mit Dokumentation**
  Benutzer- und Installationshandbücher
  Prüf- und Testprotokolle

- **Auslieferungsphase** (Abnahme- u. Einführungsphase, Überleitungsphase, transition phase)

  ◇ **Ziel** : Auslieferung eines fertiggestellten Produkts an den Kunden/Anwender

  ◇ Abnahme des Produkts durch den Kunden (→ Abnahmetest → **Abnahmeprotokoll**)
  Gegebenenfalls Korrektur von festgestellten Fehlern
  Installation und Inbetriebnahme des Produkts
  Schulung der Benutzer

Klassenkategorien

• Allgemeines

◇ Ein **objektorientiertes Softwaresystem** (OO-Programm) ist als eine Ansammlung **interagierender Objekte** organisiert.

◇ Die einzelnen Objekte decken dabei unterschiedliche Aufgabenbereiche ab. Entsprechend ihrem jeweiligen Aufgabenbereich können sie und damit die sie beschreibenden Klassen unterschiedlichen **Kategorien** zugeordnet werden.
→ Objekte und Klassen besitzen einen **Stereotypen**.

◇ Auch die gegebenenfalls zwischen Klassen bestehenden **Beziehungen** lassen sich in **verschiedene Arten** einteilen.

• Gebräuchliche Klassenkategorien

▻ **Entity-Klassen**
Das sind Klassen, deren Objekte Bestandteil des Problembereichs sind (**Domänen-Klassen**).
Sie reflektieren entweder **reale Objekte** des Problembereichs oder sie werden zur Wahrnehmung **interner Aufgaben** des Systems benötigt.
Es kann sich dabei um **konkrete technische Objekte** (z.B. Pkw) handeln, es können aber auch **Personen** und deren **Rollen** (z.B. Student, Kunde) **Orte**, (z.B. Hörsaal), **Organisationseinheiten** (z.B. FAKULTÄT FÜR),
Ereignisse (z.B. Unfall),
Informationen über (Inter-)Aktionen (z.B. Kaufvertrag), **Konzepte** (z.B. Entwicklungsplan), sonstige **allgemeine** oder **problembereichsbezogene Begriffe** (z.B. Lehrveranstaltung) **usw**. sein
Objekte von Entity-Klassen sind typischerweise **unabhängig von der Systemumgebung** und von der Art und Weise, wie das System mit der Umgebung kommuniziert.
Häufig sind sie auch **applikationsunabhängig**, d.h. sie lassen sich in mehreren Applikationen einsetzen.

▻ **Interface-Klassen** (*Boundary Classes*)
Objekte dieser Klassen sind für die **Kommunikation** zwischen der Systemumgebung und dem Inneren des Systems zuständig.
Sie realisieren die **Schnittstelle** des Systems **zum Systembenutzer** (Benutzeroberfläche) bzw **zu anderen Systemen**.
→ Sie bilden den **umgebungsabhängigen** Teil eines Systems.

▻ **Controller-Klassen** (*Control Classes*)
Objekte von Controller-Klassen **steuern** den **Ablauf**, d.h. die Zusammenarbeit der übrigen Objekte zur Realisierung eines oder mehrerer Use Cases.
Es handelt sich um **aktive** Objekte.
Typischerweise sind sie **applikationsspezifisch**.

▻ **Service-Klassen**
Objekte von Service-Klassen stellen anderen Objekten (insbesondere den Controller-Objekten) **Dienste** zur Verfügung.
Häufig handelt es um Klassen aus einer **Bibliothek**.
Sie können aber auch durch **Auslagerung von Funktionalitäten** anderer Klassen gebildet werden.

▻ **Utility-Klassen**
Sie **kapseln** Daten und Operationen (Funktionen), die **global** verfügbar sein sollen.
Im Allgemeinen werden sie **nicht instantiiert**.
Beispiel : mathematische Konstanten u. mathematische Funktionen.

Die konsequente Unterscheidung und Trennung von Entity-, Interface- und Controller-Klassen führt zur "***Model-View-Controller***"-Architektur (MVC-Architektur), ein allgemein eingeführtes OOP-Paradigma.
Das User-Interface (View) ist von der eigentlichen Problembearbeitung (Model) getrennt und weitgehend entkoppelt.
Die Verbindung zwischen beiden wird über einen Controller hergestellt.

Beziehungen zwischen Klassen

- **Vererbungsbeziehung**

  ◇ Beziehung zwischen Klassen, deren Komponenten sich teilweise überdecken

  ◇ Eine abgeleitete Klasse erbt die Eigenschaften und Fähigkeiten (Komponenten) der Basisklasse(n).
  "ist"-Beziehung → ein Objekt der abgeleiteten Klasse ist auch ein Objekt der Basisklasse(n)

  ◇ Ordnungsprinzip bei der Spezifikation von Klassen.
  → **Generalisierung / Spezialisierung**

- **Nutzungsbeziehungen**

  ◇ Unter einer (statischen) Nutzungsbeziehung versteht man eine in einem konkreten Anwendungsbereich geltende Beziehung zwischen Klassen, deren Instanzen voneinander Kenntnis haben und die dadurch miteinander **kommunizieren** können
  → Nutzungsbeziehungen sind notwendig für die **Interaktion von Objekten**

  ◇ **Assoziation**
  - Spezielle Beziehung zwischen Klassen bzw. Objekten, bei der die Objekte **unabhängig** voneinander existieren und **lose** miteinander **gekoppelt** sind
    Beispiel : einem Objekt wird ein anderes Objekt als Parameter übergeben.
  - **Name** : Kennzeichnung der Semantik der Beziehung zwischen den Klasseninstanzen
  - **Navigationsrichtung** : legt die Kommunikationsrichtung und die Richtung, in der ein Objekt der einen Klasse ein Objekt der anderen Klasse referieren kann, fest.
    bidirektional (Kommunikation in beiden Richtungen möglich) oder unidirektional
  - **Rolle** : Ein Name für die Aufgabe, die ein Objekt der assoziierten Klasse aus der Sicht eines Objekts der assoziierenden Klasse wahrnimmt.
  - **Kardinalität** (*multiplicity*) : bezeichnet die mögliche Anzahl der an der Assoziation beteiligten Instanzen einer Klasse.

  ◇ **Aggregation**
  - Spezielle Beziehung zwischen Klassen bzw Objekten, bei der die Objekte der einen Klasse **Bestandteile** (Komponenten) eines oder mehrerer Objekte der anderen Klasse sind. → zwischen den Objekten besteht eine **feste Kopplung**
  - **"hat"**-Beziehung bzw. **"ist Teil von"**-Beziehung
  - Das "umschließende" Objekt bildet einen Container für das bzw. die enthaltene(n) Objekt(e)
  - Aggregation kann als **Spezialfall der Assoziation** aufgefasst werden.
    → die für Assoziationen möglichen Kennungen (Name usw.) lassen sich auch für Aggregationen verwenden.
  - eine Aggregation ist i.a. aber eine **unidirektionale** Beziehung (Navigation vom umschließenden Objekt zu den Komponenten-Objekten)

  Je nach dem **Grad der Kopplung** unterscheidet man:

  ▻ **einfache Aggregation**
  Das umschließende Objekt (Aggregat) und die Komponenten sind **funktionell aneinander gebunden**.
  Eine Komponente kann zusätzlich noch **weiteren** Aggregaten der gleichen oder einer anderen Klasse zugeordnet sein.
  Bei Löschung des Aggregats **dürfen** die Komponenten unabhängig vom Aggregat auch erhalten bleiben.
  Beispiel : Klasse mit dynamisch erzeugten Komponenten

  ▻ **echte Aggregation** (Komposition, *composite aggregation)*
  Die Komponenten können **nur einem** Aggregat zugeordnet sein und nur **innerhalb** des Aggregats **existieren**.
  Bei Löschung des Aggregats werden auch die Komponenten gelöscht.
  Beispiel : Klasse mit statisch allozierten Komponenten

  ◇ **Anmerkung** :
  In der Praxis kann es im Einzelfall sehr schwierig sein, zwischen Assoziation und Aggregation und den verschiedenen Formen der Aggregation zu unterscheiden.

Identifikation von Klassen und ihren Beziehungen

• Allgemeines

◇ Das **Identifizieren** von **Klassen** und ihren **Beziehungen** ist eine **zentrale Aufgabe** der **Analyse-**Tätigkeit innerhalb des OO-Entwicklungsprozesses.
Es handelt sich um eine sehr schwierige, äußerst kreative Tätigkeit, die methodisch kaum unterstützt wird und für die kein allgemeingültiges "Kochrezept" angegeben werden kann.
Zum Erwerb der notwendigen eigenen Erfahrungen kann man sich lediglich von Empfehlungen, Tipps und Tricks (**Heuristiken**) , die einschlägige Experten mittlerweile in der Literatur veröffentlich haben, leiten lassen.

◇ Da der Entwicklungsprozess iterativ verläuft, wird sich die **Liste der gefundenen Klassen** im Laufe dieses Prozesses **ändern** → die im ersten Schritt gefundenen Klassen, werden i.a. nicht der endgültig realisierten Liste entsprechen, neue Klassen werden hinzukommen, bereits identifizierte Klassen werden modifiziert (z.B. geteilt oder zusammengelegt) bzw. wieder verworfen.
Analoges gilt für die **Attribute** (Datenkomponenten) und **Operationen** (Funktionskomponenten) der Klassen, sowie für die **Beziehungen** zwischen den Klassen.

• Identifikation von Klassen

▻ Ermittlung der **konkreten technischen Objekte** des Problembereichs (**Fachklassen**).

▻ Ermittlung von Objekten/Klassen aus im Problembereich eingesetzten **Formularen** (Formularanalyse).

▻ Untersuchung der Beschreibung der Szenarien der einzelnen *Use Cases* (oder sonstiger verbal formulierter Anforderungen an das System, wie z.B. das Pflichtenheft) nach **Hauptworten**. Diese sind i.a. Kandidaten für Klassen.

▻ Einsatz von **CRC-Karten** (CRC – *Class*, *Responsibilities*, *Collaborations*)
Anlegen kleiner Karten, auf denen der Klassenname, die Zuständigkeiten (*responsibilties*) und die mit ihnen kollaborierenden Klassen (zu denen also Assoziationen bestehen müssen) vermerkt werden.
Überprüfung der einzelnen Anwendungsfälle mittels der CRC-Karten auf Erfüllung der geforderten Funktionalität.
Übervolle Karten bedeuten entweder schlecht ausgearbeitete Zuständigkeiten oder nichtkohäsive Klassen.

▻ Untersuchung jedes **Aktor-/Scenario-Paares** zur Ermittlung der **Interface-Klassen**.

▻ Hinzufügen je einer **Controller-Klasse** für jedes **Aktor/Scenario-Paar**.

▻ Zuordnung der gefundenen Klassen zu den verschiedenen **Klassenkategorien**.

▻ Wählen **aussagekräftiger Namen** für die gefundenen Klassen.

▻ Zuordnen von **Operationen** und **Attributen**, im ersten Schritt allerdings nur soweit es zur **Unterscheidung** der Klassen und zur Abdeckung der Zuständigkeiten notwendig ist.

▻ **Filterung** der gefundenen Klassenmenge hinsichtlich **tatsächlicher Problembereichszugehörigkeit**, echter **Substanz**, eindeutiger **Abgrenzung**, **Überschneidungen**, **Redundanz** usw.

▻ Gegebenenfalls sind Klassen wieder zu verwerfen, zusammenzufassen, zu teilen oder Komponenten in gemeinsame Basisklassen auszulagern (**Refaktorierung**).

• Identifikation der Klassenbeziehungen

▻ Untersuchung der gefundenen Klassen auf Gemeinsamkeiten (→ Vererbungsbeziehungen)

▻ Ermittlung der Nutzungsbeziehungen kann parallel zur Identifikation der Klassen beginnen (CRC-Karten !)

▻ Untersuchung der einzelnen Scenarios auf **Verben**. Diese können auf zwischen Objekten auszutauschende Botschaften hinweisen. Der Austausch von Botschaften erfordert Nutzungsbeziehungen.

▻ Rangordnung der beteiligten Klassen überprüfen.
Über-/Unterordnung kann auf Aggregation hinweisen.

▻ Überprüfung, ob "hat"- bzw "ist Teil von"-Beziehung vorliegt.

▻ Im Zweifelsfall Assoziation wählen.

▻ Kennungen (Name, Navigationsrichtung, Rollenname, Kardinalität usw) der gefundenen Beziehungen festlegen.

Modellierung

• Allgemeines

◇ Das zu entwickelnde **Software-System** sowie der von ihm abzudeckende **Problembereich** sind häufig so **komplex**, dass sie sich nur mit Hilfe geeigneter **Hilfsmittel** ausreichend erfassen lassen.
Ein derartiges Hilfsmittel stellt die **Modellierung** dar.

◇ I.a. ist es wenig sinnvoll das Gesamtsystem in einem einzigen – alle Einzelheiten erfassenden – Modell darzustellen. Vielmehr setzt man **unterschiedliche Modelle** ein, die jeweils verschiedene Teilaspekte des Systems repräsentieren. Diese Modelle erleichtern nicht nur das Problembereichs- und Systemverständnis, sondern bilden auch ein wesentliches Kommunikationsmittel aller an der Systementwicklung beteiligten Personen und stellen damit auch einen wichtigen Bestandteil der Systemdokumentation dar.

◇ Voraussetzung für die Bildung adäquater, aussagekräftiger, eindeutiger und leicht verständlicher Modelle ist eine **geeignete Notation** zur Modelbeschreibung.
Für den Bereich der objektorientierten Systementwicklung ist dies die **Unified Modeling Language** (**UML**), mit der Modelle in überwiegend **graphischer** Notation (→ **Diagramme**) dargestellt werden können.

◇ Die **UML** stellt Sprachmittel zur Formulierung zahlreicher **unterschiedlicher Diagramme** zur Verfügung, die zur Beschreibung der verschiedenen Modelle geeignet sind.

• Modelle und zugeordnete UML-Diagramme

◇ Modell der Systemnutzung

▻ **Nutzungsfallmodell** (Anwendungsfallmodell) → **Use-Case-Diagramm**

◇ Logisches Modell

▻ **Statisches Modell**
grobe Systemarchitektur (Aufbaustruktur) → **Paketdiagramm**
detaillierte Systemarchitektur → **Klassendiagramm**

▻ **Dynamisches Modell**
Zusammenarbeit mehrerer Objekte (Objekt-Interaktion) → Interaktionsdiagramme:
Sequenzdiagramm
Kollaborationsdiagramm
Objektverhalten (in mehreren Use Cases) → **Zustandsdiagramm**
Systemverhalten → **Aktivitätsdiagramm**

◇ Physikalisches Modell

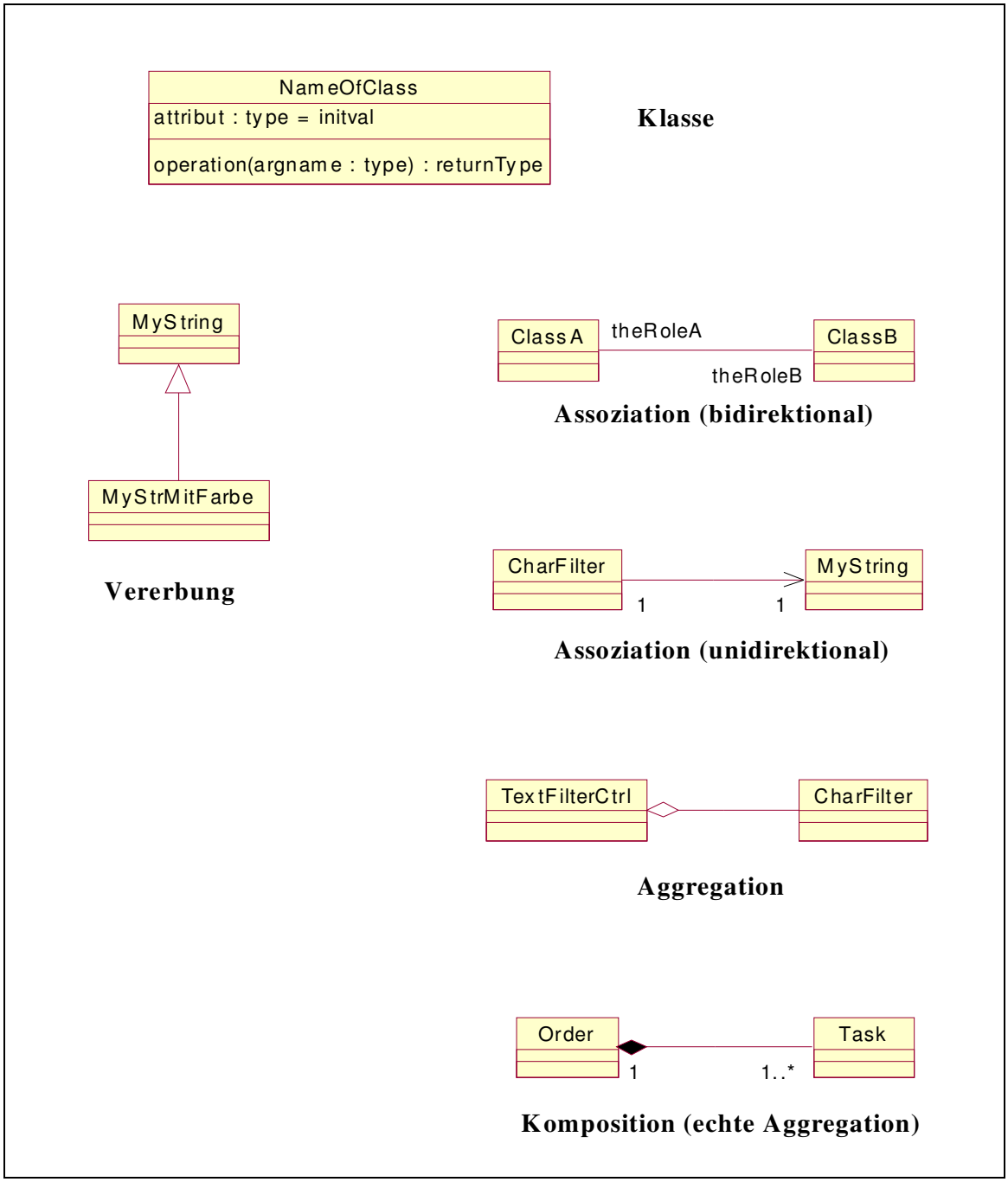
▻ **Implementierungsmodell** → **Komponentendiagramm** (Moduldiagramm)

▻ **Konfigurierungsmodell** → **Verteilungsdiagramm** (Deployment Diagram)
(topologische Strukturierung und Verteilung/Zuordnung der Soft- und Hardwarekomponenten des Gesamtsystems)

Unified Modelling Language (UML) (1)

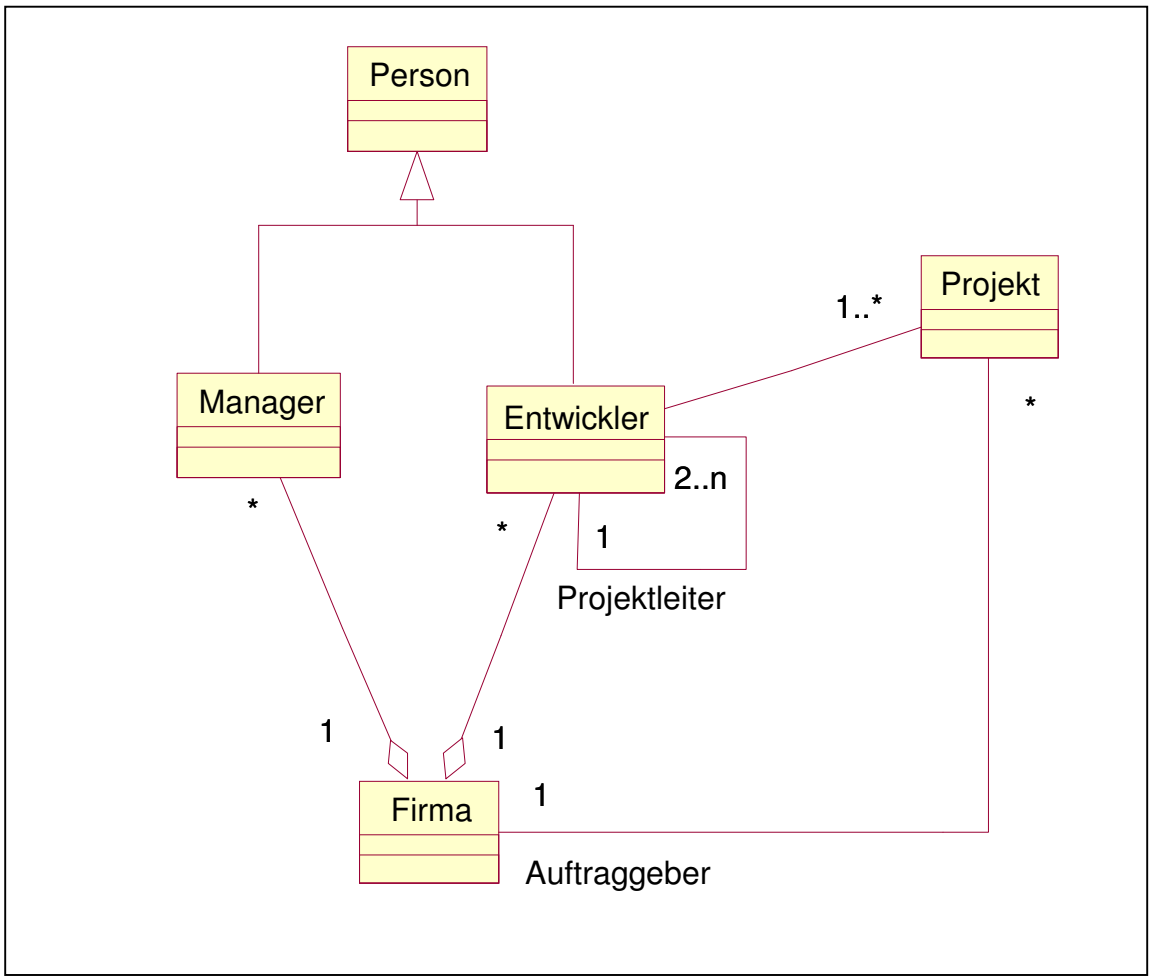
- **Klassendiagramm**
  - ◇ Ein Klassendiagramm beschreibt die im System eingesetzten **Klassen** und ihre statischen **Beziehungen** (Vererbungsbeziehungen und Nutzungsbeziehungen) → **detailliertes statisches Systemmodell**
  - ◇ Zur Erhöhung der Übersichtlichkeit kann die Gesamtheit der Klassen eines Systems auf mehrere **Teildiagramme** aufgeteilt sein.

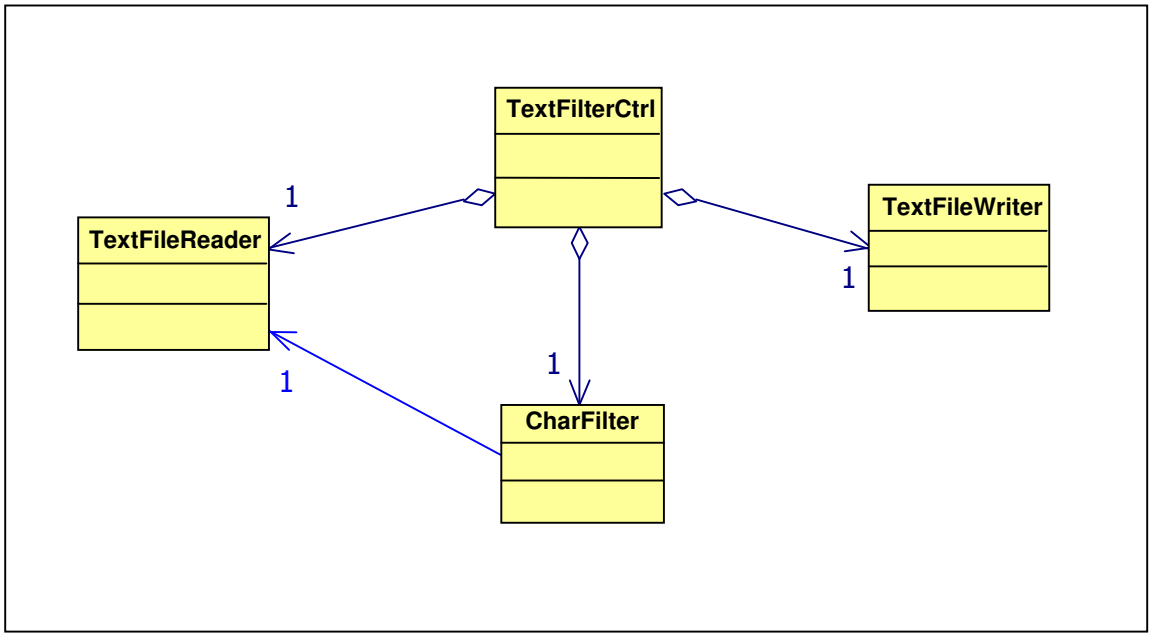
- **Elemente des Klassendiagramms**



Unified Modelling Language (UML) (2)

- **Beispiel für Klassendiagramme**

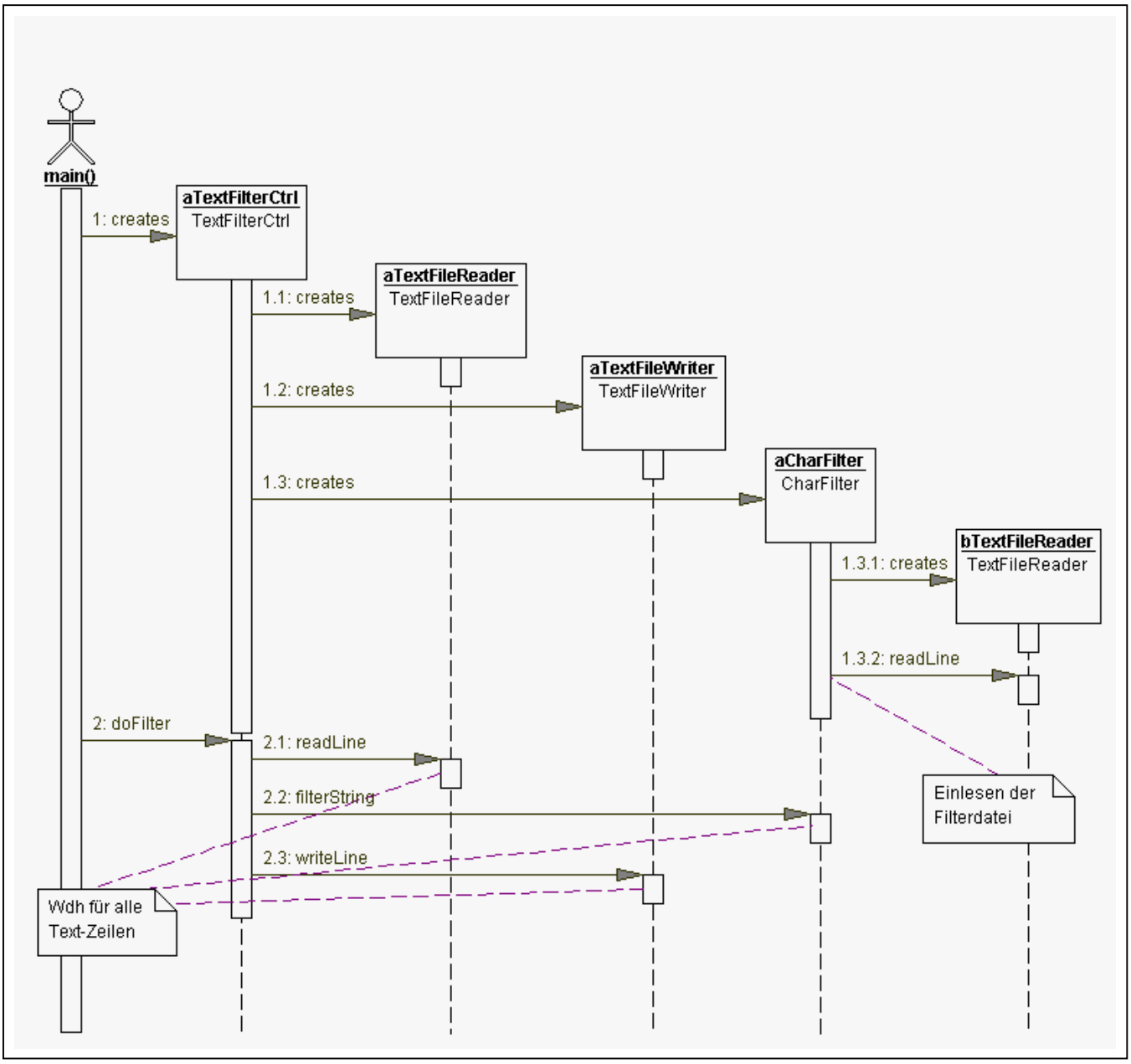




Unified Modelling Language (UML) (3)

- **Sequenzdiagramm**
  - ◇ Ein Sequenzdiagramm beschreibt die **Interaktion mehrerer Objekte**.
  - ◇ Für jedes Scenario jedes Use Cases sollte ein eigenes Diagramm erstellt werden.

- **Beispiel für ein Sequenzdiagramm**



Technische Hilfsmittel (1)

- **Allgemeines**
  - ◇ An jeder der Entwicklungsphasen sind **personelle Ressourcen** (*human resources*) nötig und beteiligt. Zentrales Thema des Softwareentwicklungsprozesses ist deshalb auch die **Kommunikation**, der am Entwicklungsprozess beteiligten Personen.
    Diese erstreckt sich über alle Entwicklungsphasen und Aktivitäten hinweg.
  - ◇ Es sollten deshalb **Konventionen** der Kommunikation vorab festgelegt werden, die eine **gemeinsame** Art der Kommunikation für bestimmte Aktivitäten festlegen. Dies geschieht auch durch die Auswahl und Festlegung **technischer Hilfsmittel**.
  - ◇ Technische Hilfsmittel zur Unterstützung der personellen Ressourcen sollen deshalb:
    - Die Beteiligten mit **aktuellen** und **relevanten** Informationen des Softwareentwicklungsprozesses versorgen.
    - **Einfach** zu handhaben sein und am Besten in gewissen Punkten auch **automatisierte Mechanismen** zur Verfügung stellen.
    - die nötigen Informationen möglichst **ortsunabhängig** (und über **Zugriffsrechte geregelt**) die nötigen Informationen zur Verfügung zu stellen.
  - ◇ Ein gewähltes technisches Hilfsmittel begleitet den Entwicklungsprozess **phasenübergreifend** vom Zeitpunkt des Einsatzes bis zum Ende des Entwicklungsprozesses.
  - ◇ Änderung eines technischen Hilfsmittels während eines Entwicklungsprozesses ist möglich und manchmal auch nötig (z. B. Wechsel der Entwicklungsumgebung), oft aber nicht gewünscht.
    ➔ Migration der Daten ➔ Mehraufwand
  - ◇ Ziel der Entwicklung solcher Hilfsmittel ist es, so früh wie möglich im Softwareentwicklungsprozess zum Einsatz zu kommen.
  - ◇ Solche Produkte werden wie folgt bezeichnet:
    - Projektmanagement-Tools
    - Softwareentwicklungstools
    - CASE-Tools (*Computer-Aided Software Engineering*)
  - ◇ Meist umfassen diese aber nicht alle Entwicklungsphasen, deshalb trifft man (besonders im Opensource-Entwicklungsbereich) auf eine Sammlung mehrerer technischer Hilfsmittel.

- **Einige Beispiele solcher unterstützenden technischen Hilfsmittel**
  - ◇ **Projektorganisation/-verwaltung**
    (z. B. MS Project, ...)
    - Dieses Werkzeug kommt bereits ab der **Planungsphase** zum Einsatz.
    - Diese Werkzeuge besitzen oft auch automatisierte Mechanismen zur Erstellung von Berichten, Analysen und Präsentationen ➔ **Kommunikation**.
  - ◇ **Office-Produkte**
    (z. B. MS Office, OpenOffice, KOffice, ...)
    - Kommen ebenfalls bereits ab der **Planungsphase** zum Einsatz.
    - Die wichtigsten Komponenten sind Textverarbeitung, Präsentationssoftware, Tabellenkalkulation.
    - Dienen zur Kommunikation (Präsentation, Dokumentation) in jeder Entwicklungsphase.
    - Werden inzwischen teilweise durch andere Werkzeuge (Projektorganisationsssoftware, Webpräsentation, u.ä.) abgelöst oder über diese aufgerufen.

Technische Hilfsmittel (2)

- **Einige Beispiele solcher unterstützenden technischen Hilfsmittel (Forts.)**

◇ **Modellierungssoftware**
(z. B. Together Control Center, microTOOL objectiF, ...)

- Dieses Werkzeug kommt meist ab der **Definitionsphase** (u. U. auch schon in der Planungsphase) zum Einsatz, verliert aber **leider** manchmal in den späteren Entwicklungsphasen (Konstruktion, Auslieferung) an Bedeutung.
- Die gemeinsame Kommunikation findet hier über **UML** statt.

◇ **Versionsverwaltung**
(z. B. Revision Control System (rcs), Concurrent Versions System (cvs), Subversion (svn), LinCVS, WinCVS, …)
Dient zur zentralen Ablage (Repository) von Quellcode und Dokumentation. Dieses Repository kann **auch** auf einem **entfernten Server** liegen, somit wird der Softwareentwicklungsprozess **ortsunabhängig**.

- Historisch bedingt kommt die Versionsverwaltung meist erst in der **Konstruktionsphase** zum Einsatz, jedoch wäre auch ein früherer Einsatz denkbar (Ablegen von Spezifikationen, Lastenheft, u. ä.) .
- Einzeländerungen können **kommentiert** in das Repository "eingecheckt" werden → **Kommunikation** (Dokumentation).
- Änderungen können wieder rückgängig gemacht werden.
- Frontends zum Zugriff auf die Versionsverwaltung von Dateien werden bereits in Integrierte Entwicklungsumgebungen und Internetdiensten eingebettet.

◇ **Integrierte Entwicklungsumgebung (*IDE*)**
(z. B. Visual Studio, KDevelop, Anjuta, Borland C++, ...)

- Dieses Werkzeug kommen meist ab der **Konstruktionsphase** zum Einsatz.
- Die Entwicklung der IDEs geht aber bereits in Richtung Integration der Modellierung (z. B. über UML), um sie somit bereits in der in der **Definitionsphase** einsetzen zu können.
- Die Kommunikation findet hier meist über **kommentierten Code** und in das Projekt **integrierte Dokumentation** statt (z. B. API-Dokumentation, CHANGELOG, ...).

◇ **Internetdienste**
(z. B. Newsgruppen, Mailinglisten, Wiki, elektronische Foren, Bugtracking-Systeme)

- Gewinnen besonders ab der **Konstruktionsphase** an Bedeutung.
- Werden meist zur **Kommunikation** nach außen eingesetzt → interaktive Informationsplattform für den Anwender (Kunden).
  So z. B. zur **Produktpräsentation** (Anleitungen, HOWTOs, ...) und als **Feedback-Möglichkeit** (Bugreports, Verbesserungsvorschläge, ...).
- Auch unmittelbare Beteiligte des Softwareentwicklungsprozesses nutzen diese Formen der Kommunikation untereinander (z. B. geschlossene Newsgruppen).
  In diesem Fall findet man diese Werkzeuge bereits in **früheren Phasen** der Softwareentwicklung.
- Erreichbarkeit der Beteiligten wird durch die Verwendung dieser Werkzeuge erhöht → **Ortsunabhängigkeit**.