Vorlesungsinhalt OOSE SoSe 2022

Inhalt

[1. Objektorientierte Systementwicklung (OOSE) 4](#_Toc109487580)

[1.1 Warum überhaupt Objektorientierung? 4](#_Toc109487581)

[1.2 Einstieg durch Begriffsbildung 4](#_Toc109487582)

[1.3 Paradigma der OOSE 5](#_Toc109487583)

[2. Grundlagen der Programmierung 7](#_Toc109487584)

[2.1 Überblick 7](#_Toc109487585)

[2.2 JavaScript 7](#_Toc109487586)

[2.3 Charakteristika von JavaScript 8](#_Toc109487587)

[3. 10 Grundkonzepte der OO-Programmierung nach B. Meyer 9](#_Toc109487588)

[3.1 Architektur 9](#_Toc109487589)

[3.2 Klassen 9](#_Toc109487590)

[3.3 Instanzen 10](#_Toc109487591)

[3.4 Eingeschränkte Kommunikation 10](#_Toc109487592)

[3.5 Abstraktion 11](#_Toc109487593)

[3.6 Design by Contract 12](#_Toc109487594)

[3.7 Vererbung 13](#_Toc109487595)

[3.8 Polymorphismus und dynamisches Binden 13](#_Toc109487596)

[3.9 Statische Typbildung 14](#_Toc109487597)

[3.10 Automatische Speicherverwaltung 14](#_Toc109487598)

[4. Requirements Engineering (RE) 15](#_Toc109487599)

[4.1 What the customer really needed… 15](#_Toc109487600)

[4.2 The whole Picture 15](#_Toc109487601)

[4.3 Softwareprojekte in Schieflage 15](#_Toc109487602)

[4.4 Software-Qualitätsmerkmale nach ISO/IEC 9126 16](#_Toc109487603)

[4.5 Anforderungen an Softwaresysteme 18](#_Toc109487604)

[4.6 Strukturierte Anforderungsdokumente 20](#_Toc109487605)

[5. Anforderungen im Detail 22](#_Toc109487606)

[5.1 Spezifikationen 22](#_Toc109487607)

[5.2 Kontext von Software 22](#_Toc109487608)

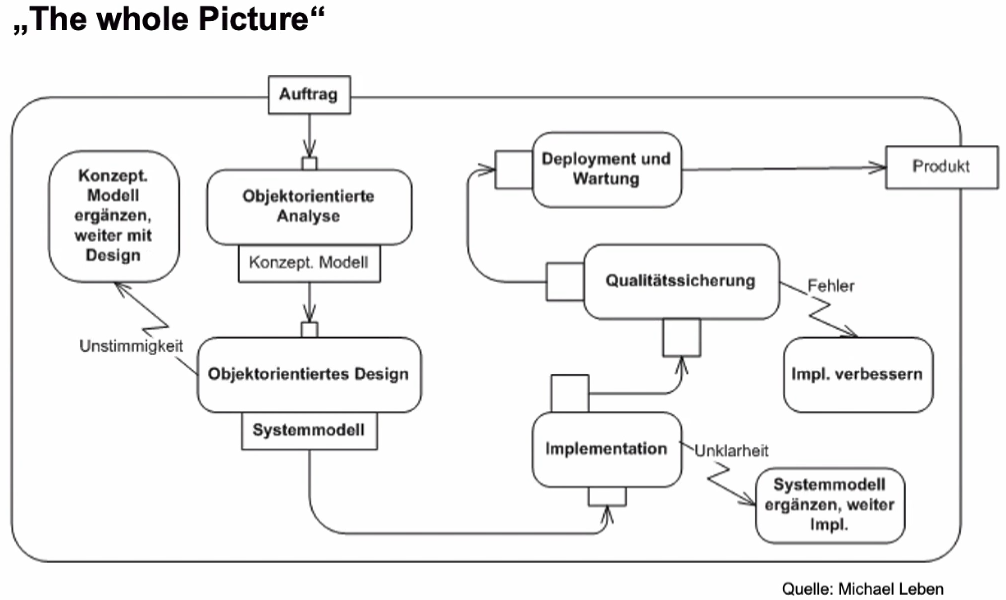
[5.3 Vorgehen 23](#_Toc109487609)

[5.4 Klassifikation 24](#_Toc109487610)

[5.5 Priorisierung 24](#_Toc109487611)

1. Objektorientierte Systementwicklung (OOSE)
   1. Warum überhaupt Objektorientierung?

* Objektorientierung steht für eine **bestimmte Sichtweise** auf die Entwicklung von Software.
* OO gibt **Vorgehensweisen** in der Softwareentwicklung vor, die den Fokus auf **modulare Systeme** legt.
* Aus der OO lassen sich Konzepte und Methoden **für andere Fachgebiete ableiten**
* Das Verhältnis von OO hilft jedem, der beruflich im Bereich von Softwareentwicklung tätig ist:
  + **User Experience**
  + **Technische Dokumentation**

****

* 1. Einstieg durch Begriffsbildung

**Objekt**

* Allgemein etwas unspezifiziert, eine Sache, ein Gegenstand, ein Ding.
* Im Sinne der Dialektik das, worauf ein Subjekt seine beobachtende, sinnliche, empirische und praktisch verändernde Aktivität richtet (Objekt in der Philosophie).
* Ein Satzglied in der Grammatik.
* Ein Himmelskörper (Astronomisches Objekt).
* Eine Einheit in einem Geoinformationssystem (Geoobjekt).
* **In der Informatik: ein zentrales Strukturelement der Objektorientierten Programmierung.**

**System**

* Wörtlich: das Gebildete, Zusammengestellte, Verbundene
* Bezeichnet ein **Gebilde**, dessen wesentliche **Elemente (Teile)** so aufeinander bezogen sind, dass sie eine **Einheit (ein Ganzes)** abgeben.

Systeme organisieren und erhalten sich durch Strukturen:

* **Muster** (Formen) der Systemelemente, deren **Beziehungsgeflechte**, durch die ein System funktioniert (d.h. entsteht und sich erhält).
* Jedes System besteht aus **Elementen**, die zueinander in Beziehung stehen und sich zumeist **wechselseitig beeinflussen**.
* Durch Definition zweckmäßiger **Systemgrenzen** lässt sich das System von seiner Umwelt weitgehend abgrenzen. So ist eine isolierte Betrachtung des Systems möglich.
* Eine **strukturlose Zusammenstellung** mehrerer Elemente wird als **Aggregat** bezeichnet.

**Paradigma**

* *Para* = neben und *deiknynai* = zeigen, begreiflich machen, bedeutet Beispiel, Vorbild, Muster, auch Abgrenzung.
* Plural: Paradigmen bzw. Paradigmata
* In Wissenschaft, Technik und Wirtschaft mit sehr unterschiedlichen Bedeutungen verwendet; insbesondere im Computerbereich und der Managementliteratur.
* **Besondere, fokussierte Sichtweise auf einen (möglichst grundlegenden) Aspekt des jeweiligen Fachgebietes.**
* Beispiele:
  + Paradigma der Wiederverwendbarkeit von Software
  + Paradigma der Teamarbeit
  + Paradigma der schlanken Produktion (lean production)
  + Programmierparadigma; das einer Programmiersprache oder Programmiertechnik zugrundeliegende Prinzip.
  1. Paradigma der OOSE

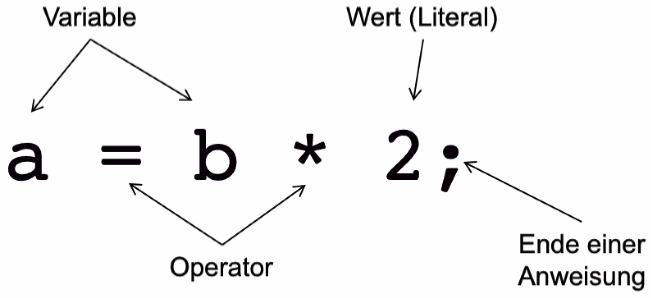
Die OOSE stellt leistungsfähige Konzepte bereit für:

1. **Zuverlässigkeit**  
   Fähigkeit, Systeme zu produzieren, die fehlerfrei sind und schon beim ersten Einsatz funktionieren.
2. **Erweiterbarkeit**  
   Fähigkeit, Systeme zu produzieren, die bei veränderten äußeren Anforderungen oder neuen technischen Voraussetzungen mit vertretbarem Aufwand angepasst werden können.
3. **Wiederverwendbarkeit**Fähigkeit, ein System aus vorgefertigten Teilen aufzubauen und sicherzustellen, dass seine eigenen Teile auch künftigen Entwicklungen verwertbar sind.
4. **Portabilität**Fähigkeit, Software zu produzieren, die zu annehmbaren Kosten auf unterschiedliche Hardware-Plattformen portiert werden kann.
5. **Effizienz**Fähigkeit, leistungsstarke Software zu produzieren.
6. Grundlagen der Programmierung
   1. Überblick

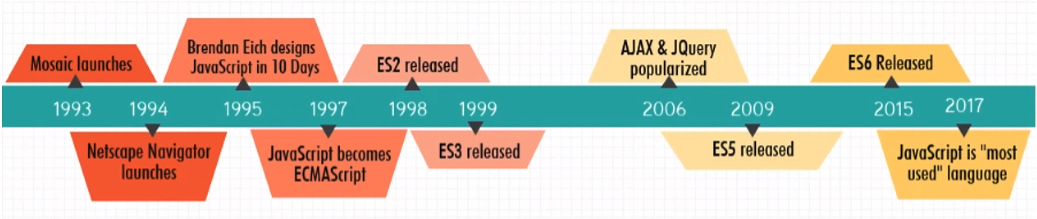
* Erstellung eines Programms -> auch Quellcode, Code genannt
  + Zusammenstellung von *speziellen Anweisungen*,
  + Die dem Computer sagen, welche *Aufgaben* er *auszuführen* hat.

Die Regeln zur Formulierung und Kombination von Anweisungen für den Computer werden *Programmiersprache* genannt -> bzw. Syntax einer Programmiersprache.

* Eine Gruppe von Wörtern, Zahlen und Operatoren die eine bestimmte Aufgabe erfüllen, werden als *Anweisung* bezeichnet:



* „Nimm den Wert, der in der Variablen b gespeichert ist, multipliziere ihn mit 2 und speichere das Ergebnis in der Variablen a.“
  1. JavaScript
* JavaScript (kurz „JS“) ist eine vollwertige und dynamische Programmiersprache
* Wenn man sie zu einem HTML-Dokument hinzufügt, sorgt sie für Dynamik und Interaktion mit dem Nutzer
* JS wurde 1995 von Brendan Eich erfunden
* Der als ECMAScript (ECMA 262) standardisierte Sprachkern von JavaScript beschreibt in den Anfängen eine dynamisch typisierte, objektorientierte, aber klassenlose\* Skriptsprache (\*seit ES6 auch mit Klassen)



* JavaScript wird nicht wie andere Programmiersprachen **kompiliert**, sondern zur Laufzeit **interpretiert**
* **Kompilierte Programmiersprachen (C, Rust, Fortran, uvm.):**
  + Ein Programm wird im Ganen dem Compiler übergeben.
  + Der Compiler wandelt den geschriebenen Code in „maschinenlesbaren“ Code um.
  + Sehr schnelle Ausführung des Codes möglich.
  + Fehler werden während des Kompilierens entdeckt und der Vorgang wird gestoppt.
  + Dieser Vorgang findet nur einmal vor der „Auslieferung“ des Codes statt.
* **Interpretierte Programmiersprachen (JavaScript, Python, uvm.):**
  + Ein Programm wird Zeile für Zeile eingelesen (Top-Down) und direkt ausgeführt.
  + Dabei wird der maschinenlesbare Code nicht vorher erzeugt, sondern zur Laufzeit.
  + Im Vergleich zu kompiliertem Code, verringerte Geschwindigkeit bei der Ausführung.
  + Fehler werden dann entdeckt, wenn der Interpreter bei der Ausführung auf sie stößt.
  + Dieser Vorgang wird bei jeder Ausführung wiederholt.
  1. Charakteristika von JavaScript

Beispiele für den Einsatz von JavaScript in Webbrowsern

* Dynamische Manipulation von Webseiten über das Document Object Model (DOM)
* Plausibilitätsprüfung (Datenvalidierung) von Formulareinträgen noch vor der Übertragung zum Server
* Anzeige von Dialogfenstern
* Senden und Empfangen von Daten, ohne das der Browser die Seite neu laden muss (Ajax)
* Vorschlagen von Suchbegriffen währen der Eingabe
* Werbebanner oder Laufschriften
* Verschleierung von E-Mail-Adressen zur Bekämpfung von Spam
* Schreib- und Lesezugriff auf Cookies und den Web Storage innerhalb des Browsers

1. 10 Grundkonzepte der OO-Programmierung nach B. Meyer
   1. Architektur

Die OO-Architektur beschreibt die Organisation des Software-Systems:

* Seine zusammenhängenden Teile (oder Module), sowie
* Die Beschreibung der Interaktion dieser Module.

Die Objekttechnologie wirkt sich hauptsächlich auf die Architektur von Software-Systemen aus.

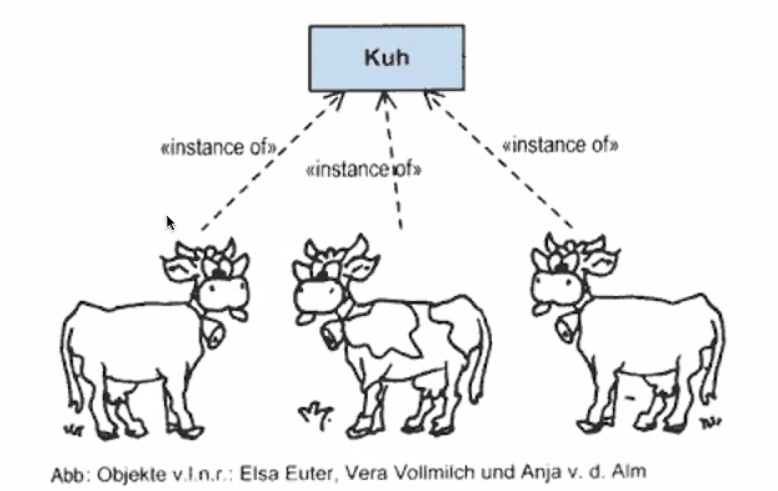
* Erweiterbare und wiederverwendbare Software erfordert vor allem eine flexible Struktur und autonome Module.
* Zuverlässigkeit von Software geht einher mit einer möglichst einfachen Architektur.

„Komplexität ist der schlimmste Feind der Zuverlässigkeit“   
*(Bertrand Meyer)*

* 1. Klassen

Grundeinheiten eines objektorientierten Software-Systems

* Beschreiben Datentypen, die durch ihre äußeren Eigenschaften abstrakt definiert werden.
* Werden abgeleitet von für die Anwendung relevanten Datentypen.

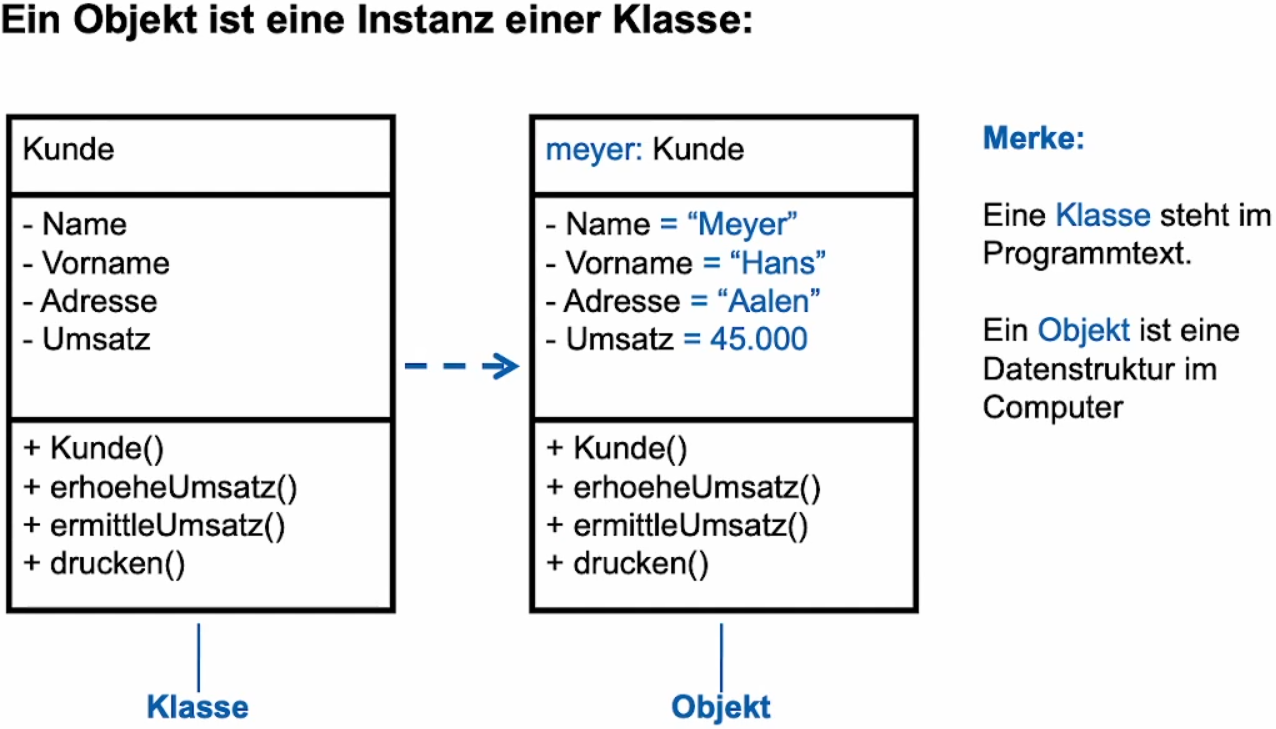


Aufbau einer Klasse:

Ein Bild, das Tisch enthält.

Automatisch generierte Beschreibung

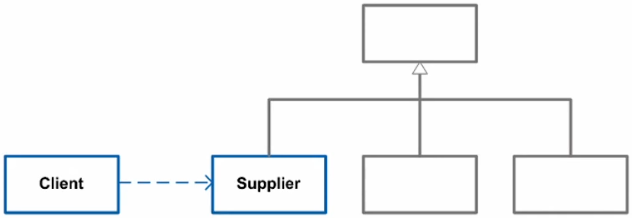
* 1. Instanzen
* Ein objektorientiertes System muss in der Lage sein, während der Ausführung eine beliebige Anzahl von Datenstrukturen zu erzeugen, die einer Beschreibung durch eine gegebene Klasse entsprechen.
* Solche Datenstrukturen heißen **Instanzen der Klasse** oder **Objekte**.  
  (siehe Bild unter „2. Klassen“)



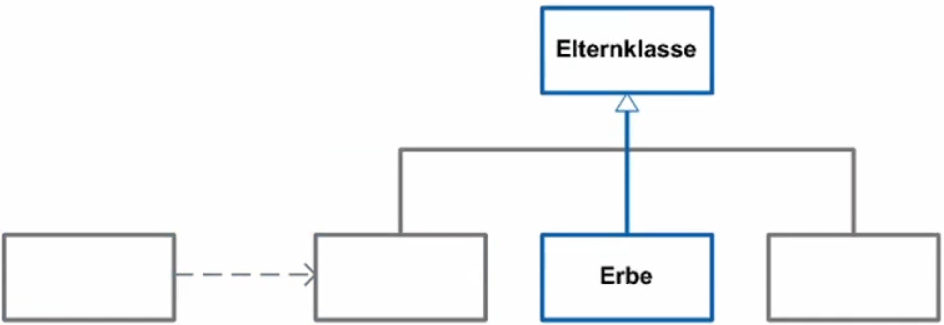
* 1. Eingeschränkte Kommunikation

Ein rein objektorientierter Ansatz erlaubt nur zwei Beziehungen zwischen den Klassen:  
**Client** und **Erbe**

**Client:**  
Eine Klasse ist ein Client einer anderen Klasse – ihres Supplier –, wenn sie ihre eigenen Aufgaben mit Hilfe von Leistungen erbringt, die der Supplier bereitstellt.

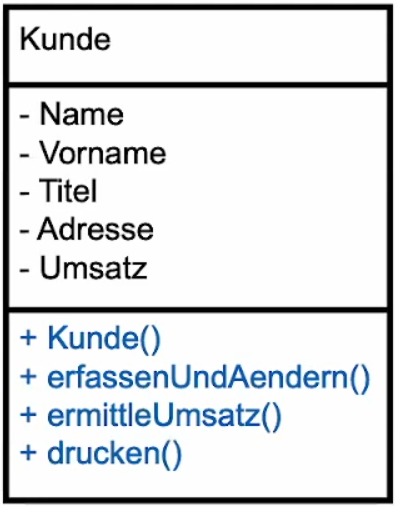


**Erbe:**  
Eine klasse ist **Erbe einer anderen Klasse** – ihrer Elternklasse – wenn sie ihr eigenes Leistungsspektrum um das der Elternklasse erweitert (Vererbungskonzept).



* 1. Abstraktion

Damit eine Klasse von anderen Klassen genutzt werden kann, beschreibt die objektorientierte Methode eine Klasse ausschließlich durch eine Liste der Operationen, die auf ihre Instanzen angewendet werden können.



**Datenabstraktion**  
ist die entscheidende Voraussetzung für die Qualitätskriterien

* Erweiterbarkeit
* Wiederverwertbarkeit und
* Zuverlässigkeit

Eine Klasse, die eine andere Klasse als Supplier nutzt, ignoriert die internen Eigenschaften des Supplier.

Sie nimmt lediglich jene Operationen und Merkmale zur Kenntnis, die eine Supplier-Klasse ihren Clients offiziell zur Verfügung stellt.

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

* **Informationskapselung:**  
  Für jede Klasse wird spezifiziert, welche Eigenschaften der Klasse den Clients zugänglich sein sollen und welche – als Geheimnisse der Klasse – nur für den internen Gebrauch bestimmt sind.
* **Geheimnisprinzip:**  
  Was der Client nicht sieht, kann er nicht verändern (Geheimnisprinzip):  
  Die internen Details des Suppliers sollen den Entwicklern der client-Klasse jedoch nicht vorenthalten werden, sondern im Gegenteil: Es soll ihnen erspart bleiben, sich mit diesen Details befassen zu müssen.

„Sag mir nicht, was du bist;  
sage mir, was du für mich tun kannst“  
*(Bertrand Meyer)*

* 1. Design by Contract
* Die Nutzung von Supplier-Merkmalen durch eine Client-Klasse sollte durch eine möglichst präzise Beschreibung der gemeinsamen Rechte und Pflichten geregelt sein.
* Die Pflichten der einen Partei entsprechen den Rechten der anderen Partei.

Ein Bild, das Tisch enthält.

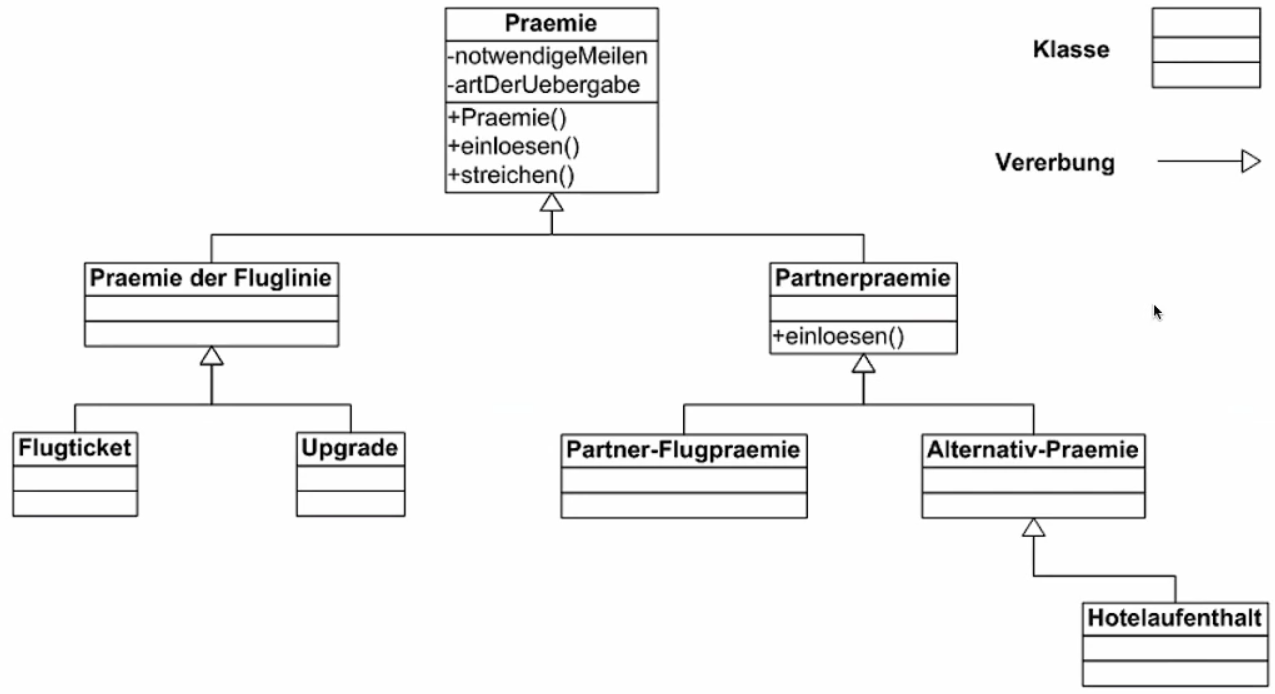
Automatisch generierte Beschreibung

* **Client-Pflichten (Supplier-Rechte):**  
  Bedingungen, die die Client-Klasse vor dem Aufrufen eines Merkmals erfüllen muss, um die richtige Ausführung der Merkmals sicherzustellen – sogenannte Preconditions (Vorbereitungen).
* **Client-Rechte (Supplier-Pflichten):**  
  Ergebnisse, die das aufgerufene Merkmal sicherstellen muss – so genannte Postconditions (Nachbedingungen).

Ein Bild, das Tisch enthält.

Automatisch generierte Beschreibung

* 1. Vererbung
* Verwandte Klassen können so organisiert werden, dass die Gemeinsamkeiten zwischen ihnen genutzt werden und für eine verständliche und handhabbare Klassenstruktur gesorgt ist.
* Ziel sind nutzbringende, wohlstrukturierte Hierarchien.



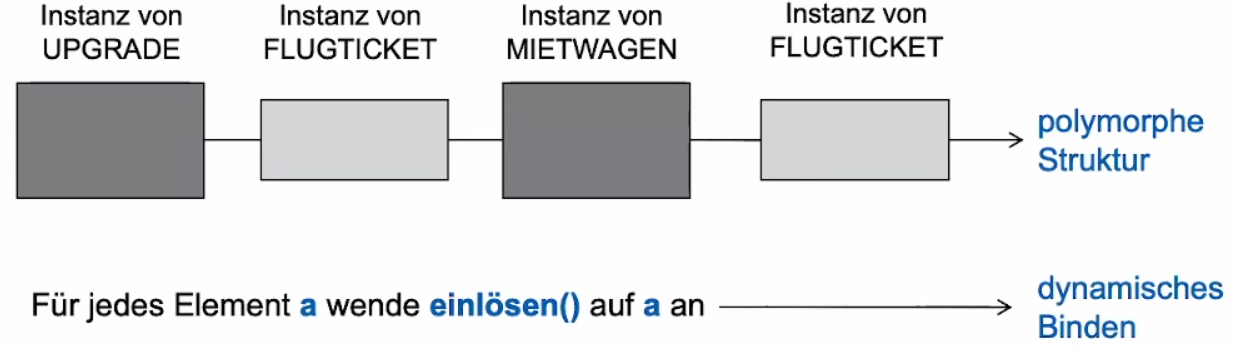
* 1. Polymorphismus und dynamisches Binden

Polymorphie ist eine der wichtigsten Fähigkeiten, die bei der Umsetzung von Objektorientierten Systemen genutzt werden kann. Wörtlich übersetzt bedeutet Polymorphie **Vielgestaltigkeit**.

Im Bereich der Objektorientierung bezieht sich Polymorphie darauf, dass **verschiedene Objekte** bei Aufruf **derselben Operation unterschiedliches Verhalten** an den Tag legen können. Polymorphie bildet die Grundlage dafür, dass objektorientierte Systeme so entwickelt werden können, dass sie innerhalb von Grenzen **flexibel auf Änderungen von Anforderungen** reagieren können.

In den meisten Fällen ist es allerdings **zur Übersetzungszeit** eines Programms **noch gar nicht klar**, mit welchen Objekten eine Variable konkret belegt sein wird. Deshalb kann in der Regel erst zur Laufzeit eines Programms entschieden werden, welche Methode beim Aufruf einer Operation ausgeführt werden soll. Der **Mechanismus der *späten Bindung*** erlaubt es, die Zuordnung auch erst zu diesem Zeitpunkt zu treffen.

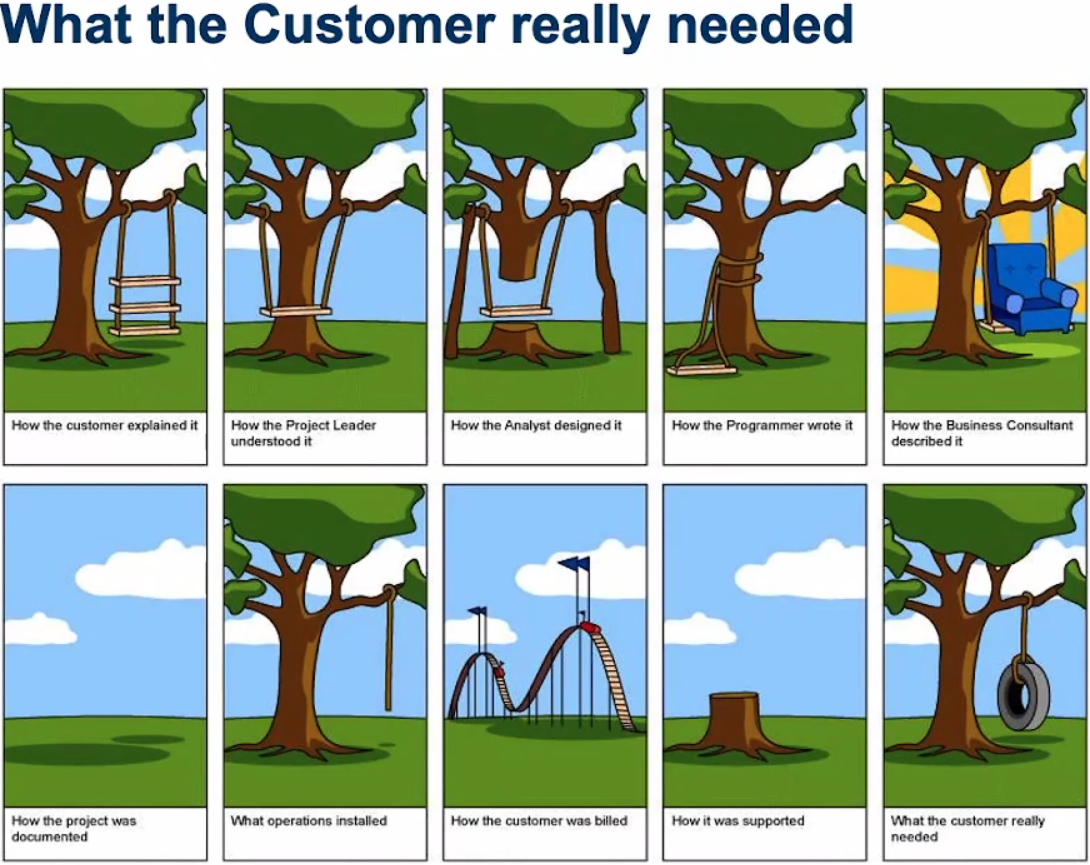
Die objektorientierte Softwareentwicklung ermöglicht es, Strukturen aus Objekten unterschiedlicher, aber miteinander verwandter Typen zu bilden und sicherzustellen, dass sich jede Operation automatisch an den Typ ihres Zielobjekts anpasst.



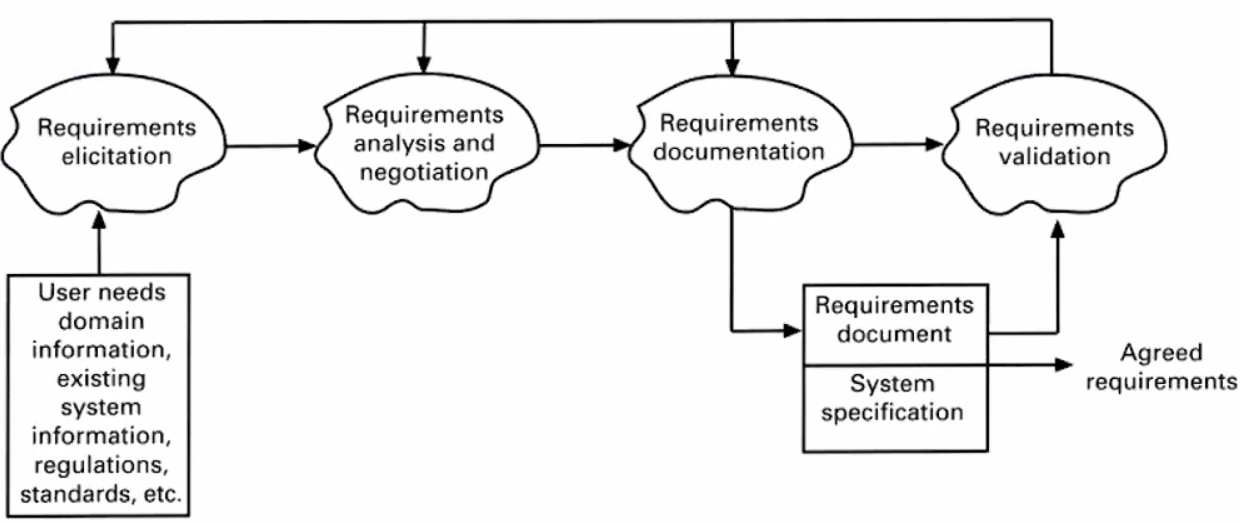
**Exemplarische Anwendungsfälle:**

* „Überprüfen Sie die Sicherheitsmaßnahmen sämtlicher Werke.“
* „Legen Sie die Jahressonderzahlungen für alle Mitarbeiter fest.“
* …
  1. Statische Typbildung
* Mit jeder Entität des software-Textes sollte ein Typ assoziiert werden können.
* Auf diese Weise kann ein Compiler oder ein anderes Werkzeug vor der Ausführung prüfen, ob die Objekte die Operationen, die zur Laufzeit auf sie angewendet werden, auch wirklich ausführen können.
  1. Automatische Speicherverwaltung
* Eine objektorientierte Umgebung sollte automatisch den Speicher freigeben, der von Objekten beansprucht wird, auf die die Anwendung zur Laufzeit nichtmehr zugreifen kann.
* Garbage Collector spürt in regelmäßigen Abständen nicht mehr erreichbare Objekte auf und gibt deren Speicherplatz wieder frei.

1. Requirements Engineering (RE)
   1. What the customer really needed…



* 1. The whole Picture



* 1. Softwareprojekte in Schieflage

**Ausganssituation (Standish Group, 1995):**

* Nur 16& der untersuchten Software-Projekte waren erfolgreich, 31% ohne Ergebnis, 53% nicht im Plan.
* Zeitüberschreitungen bis zu 22%.
* Kostenüberschreitung von durchschnittlich 189%.
* 40% der festgestellten Fehler stammen aus der Analysephase.

**Was geht schief?**

* Unvollständige Anforderungen: 13,1%
* Kunden nicht ausreichend einbezogen: 12,4%
* Mittel mit ausreichend: 10,6%
* Unrealistische Erwartungen: 9,9%
* Mangelnde Unterstützung durch Management: 9,3%
* Änderungen in den Anforderungen: 8,7%
* Mangelnde Planung: 8,1%

**Bilanz:**

* 75% aller Projekte sind Misserfolge.
* Die Hälfte aller Projekte überschreitet das Budget um mindestens 200%.
* 31% aller Projekte werden gestoppt.

**RE: People: Barry W. Boehm**

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

* 1. Software-Qualitätsmerkmale nach ISO/IEC 9126

**Wie steht es mit der Produktqualität?**  
**ISO/IEC 9126 liefert folgende Qualitätsmerkmale:**

* Funktionalität
* Zuverlässigkeit
* Benutzbarkeit
* Effizienz
* Wartbarkeit
* Übertragbarkeit

**Qualitätsmerkmal „Funktionalität“**

* **Angemessenheit:**  
  Eignung von Funktionen für spezifizierte Aufgaben, z.B. aufgabenorientierte Zusammensetzung von Funktionen aus Teilfunktionen.
* **Richtigkeit:**  
  Liefern der richtigen oder vereinbarten Ergebnisse oder Wirkungen, z.B. die benötigte Genauigkeit von berechneten Werten.
* **Interoperabilität:**  
  Fähigkeit, mit vorgegebenen Systemen zusammenzuwirken.
* **Sicherheit:**  
  Fähigkeit, unberechtigten Zugriff, sowohl versehentlich als auch vorsätzlich, auf Programme und Daten zu verhindern.
* **Konformität:**Grad, in dem die Software Normen oder Vereinbarungen zur Funktionalität erfüllt.

**Qualitätsmerkmal „Zuverlässigkeit“**

* **Reife:**  
  Geringe Versagenshäufigkeit durch Fehlerzustände.
* **Fehlertoleranz:**  
  Fähigkeit, ein spezifiziertes Leistungsniveau bei Software-Fehlern oder Nicht-Einhaltung ihrer spezifizierten Schnittstelle zu bewahren.
* **Robustheit:**  
  Fähigkeit, ein stabiles System bei Eingaben zu gewährleisten, die gar nicht vorgesehen sind.
* **Wiederherstellbarkeit:**  
  Fähigkeit, bei einem Versagen das Leistungsniveau wiederherzustellen und die direkt betroffenen Daten wiederzugewinnen. Zu berücksichtigen sind die dafür benötigte Zeit und der benötigte Aufwand.
* **Konformität:**  
  Grad, in dem die Software Normen oder Vereinbarungen zur Zuverlässigkeit erfüllt.

**Qualitätsmerkmal „Benutzbarkeit“**

* **Verständlichkeit:**  
  Aufwand für den Benutzer, das Konzept und die Anwendung zu verstehen.
* **Erlernbarkeit:**  
  Aufwand für den Benutzer, die Anwendung zu erlernen (z.B. Bedienung, Ein-, Ausgabe).
* **Bedienbarkeit:**  
  Aufwand für den Benutzer, die Anwendung zu bedienen.
* **Attraktivität:**  
  Anziehungskraft der Anwendung gegenüber dem Benutzer.
* **Konformität:**  
  Grad, in dem die Software Normen oder Vereinbarungen zur Benutzbarkeit erfüllt.

**Qualitätsmerkmal „Effizienz“**

* **Zeitverhalten:**  
  Antwort- und Verarbeitungszeiten sowie Durchsatz bei der Funktionsausführung.
* **Verbrauchsverhalten:**  
  Anzahl und Dauer der benötigten Betriebsmittel bei der Erfüllung der Funktionen. Ressourcenverbrauch, wie z.B. CPU-Zeit, Festplattenzugriffe usw.
* **Konformität:**  
  Grad, in dem die Software Normen oder Vereinbarungen zur Effizienz erfüllt.

**Qualitätsmerkmal „Wartbarkeit“**

* **Analysierbarkeit:**  
  Aufwand, um Mängel oder Ursachen von Versagen zu diagnostizieren oder um änderungsbedürftige Teile zu bestimmen.
* **Änderbarkeit:**  
  Aufwand zur Ausführung von Verbesserungen, zur Fehlerbeseitigung oder Anpassung an Umgebungsänderungen.
* **Stabilität:**  
  Wahrscheinlichkeit des Auftretens unerwarteter Auswirkungen von Änderungen.
* **Prüfbarkeit:**  
  Aufwand, der zur Prüfung der geänderten Software notwendig ist.
* **Konformität:**  
  Grad, in dem die Software Normen oder Vereinbarungen zur Wartbarkeit erfüllt.

**Qualitätsmerkmal „Übertragbarkeit“**

* **Anpassbarkeit:**  
  Möglichkeit, die Software an veränderte Bedingungen anzupassen.
* **Installierbarkeit:**  
  Aufwand, der zum Installieren der Software in einer festgelegten Umgebung notwendig ist.
* **Koexistenz:**  
  keine negativen Interferenzen mit anderen Softwaresystemen.
* **Austauschbarkeit:**  
  Möglichkeit, diese Software anstelle einer spezifizierten Anderen in die Umgebung jener Software zu verwenden, sowie der dafür notwendige Aufwand.
* **Konformität:**  
  Grad, in dem die Software Normen oder Vereinbarungen zur Übertragbarkeit erfüllt.
  1. Anforderungen an Softwaresysteme

**Viele Anforderungen an Software-Projekte…**

* sind nicht vollständig.
* entsprechen nicht den wahren Bedürfnissen des Auftraggebers oder der Benutzer.
* sind zu vage (= lassen den Systementwickler zu viel Interpretationsraum).
* führen damit zu Fehlentwicklungen/Fehlverhalten des Systems.
* sind für eine Systemabnahme ungeeignet.

**Ziel der Anforderungsanalyse:**

**Erstellung guter Anforderungen**

Nur: **Was sind gute Anforderungen?**

**„Gute Anforderungen“ sind:**

* adäquat – sie beschreiben das, was der Kunde will bzw. braucht.
* vollständig – es wird alles beschrieben, was der Kunde will bzw. braucht.
* widerspruchsfrei – sonst ist die Spezifikation nicht realisierbar.
* eindeutig – damit Fehler durch Fehlinterpretationen vermieden werden.
* prüfbar – es kann festgestellt werden, ob das realisierte System die Anforderungen erfüllt.

**Output des RE: Anforderungsdokument**

* verständlich
* kompakt
* praxisorientiert
* vollständig
* eindeutig
* aktuell

**Anforderungsdokumente**

* keine fixen Vorgaben
* teils unternehmensinterne Standards
* IEEE 830-1993 als öffentlicher Standard
* ISO/IEC/IEEE 29148:2018 als öffentlicher Standard
* Texte in natürlicher Sprache
* Strukturmodelle, Interaktionsmodelle, formale Modelle
* **Das Anforderungsdokument muss von Auftraggeber verstanden werden können!**
  1. Strukturierte Anforderungsdokumente

**RE: People: Martin Glinz**



**Struktur eines Anforderungsdokument (M. Glinz)**

**Kapitel 1: Einleitung**

1.1 Anlass

* Wozu wird das Dokument benötigt
* Projekt, Kunde, Umfeld

1.2 Ziele

* Zielsetzung für das zu spezifizierende System im Überblick;
* Bei größeren Systemen mehrstufig: Ziele, Teilziele

1.3 Einsatzbereich

* Wo wird das spezifizierte System verwendet
* Umgebung, Zielplattform

**Kapitel 2: Überblick**

2.1 Kontext

* Betrachtungsebene, Kontext des spezifizierten Systems

2.2 Struktur der Problemstellung

* Sachliche Struktur der dem spezifischen System zugrunde liegenden Problemstellung
* Gegebenenfalls Gliederung in Teilprobleme

2.3 Globale Attribute

* Systemweite gültige Attribute
* Leistungsanforderungen, besondere Qualitäten, Randbedingungen, …

2.4 Annahmen

* Annahmen, auf denen das spezifizierte System basiert

2.5 Perspektiven

* Entwicklungsperspektiven für das spezifizierte System

**Kapitel 3: Einzelanforderungen**

* Auflistung alles Anforderungen, sinnvoll gegliedert

**Möglichkeiten:**

* Hierarchisches Objektmodell:  
  hierarchische Gliederung des Problems in geschlossene Teilprobleme
* Unterkapitelgliederung:  
  die Unterkapitel fassen logisch zusammengehörige Teile zusammen; Unterkapitel in funktionale Anforderung und Attribute gegliedert

**Anhänge**

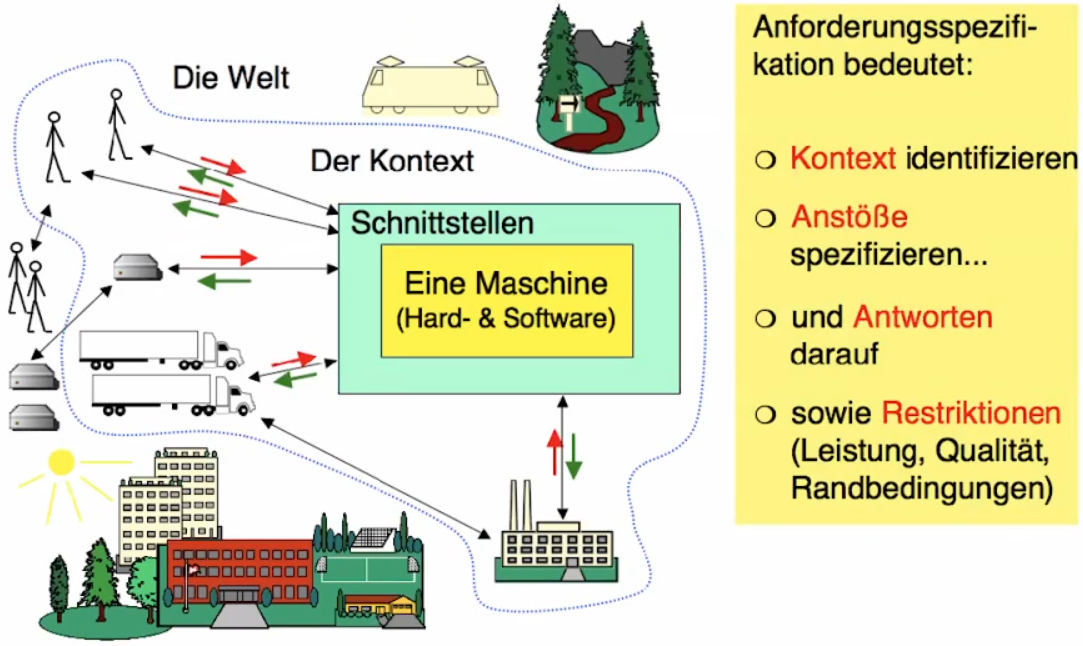
* Glossar
* Verzeichnis alle verwendeter Fachbegriffe mit Definitionen, Abkürzungen, Synonymen etc.
* Verzeichnis referenzierter Dokumente
* Nachweis alles Dokumente, auf die in der Anforderungsspezifikation Bezug genommen wird.

1. Anforderungen im Detail
   1. Spezifikationen

Requirements Engineering **bildet** in Form von Spezifikationen **Bedürfnisse als Lösungen ab.**

Es werden zwei grundlegende Perspektiven in den Spezifikationen unterschieden:

* **Anforderungsspezifikation**  
  Beschreibt was und wofür etwas gemacht werden soll. Wird häufig auch als Lastenheft oder Grobkonzept bezeichnet. Deckt Marktanforderungen ab. Gehört dem Auftraggeber und ist vertragsrelevant.
* **Lösungsspezifikation**  
  Beschreibt wie etwas gemacht werden soll. Wird häufig auch als Pflichtenheft, Systembeschreibung oder Fachkonzept bezeichnet. Deckt Produktanforderungen und Teile der Kompetenzanforderungen ab. Gehört dem Auftraggeber und ist Grundlage für die weitere Entwicklung.
  1. Kontext von Software
* Der weitaus größte Teil von Software-Entwicklung entfällt auf Systeme, welche Anwendungsprobleme **betrieblicher oder technischer Natur** lösen bzw. **Menschen oder Maschinen** bei der Lösung solcher Probleme zu unterstützen.
* Bei der Spezifikation der Anforderungen an solchen Systemen ist es von erheblicher Bedeutung, sich darüber im Klaren zu werden, wie ein solches System **in sein Umfeld** eingebettet sein soll und **wo die Systemgrenzen** liegen.
* Ein Software-System kann man sich als eine spezialisierte Problemlösungsmaschine vorstellen, welche **über Schnittstellen mit ihrer Umgebung interagiert**. Ein solches System ist typisch in einem **Anwendungsbereich** situiert.

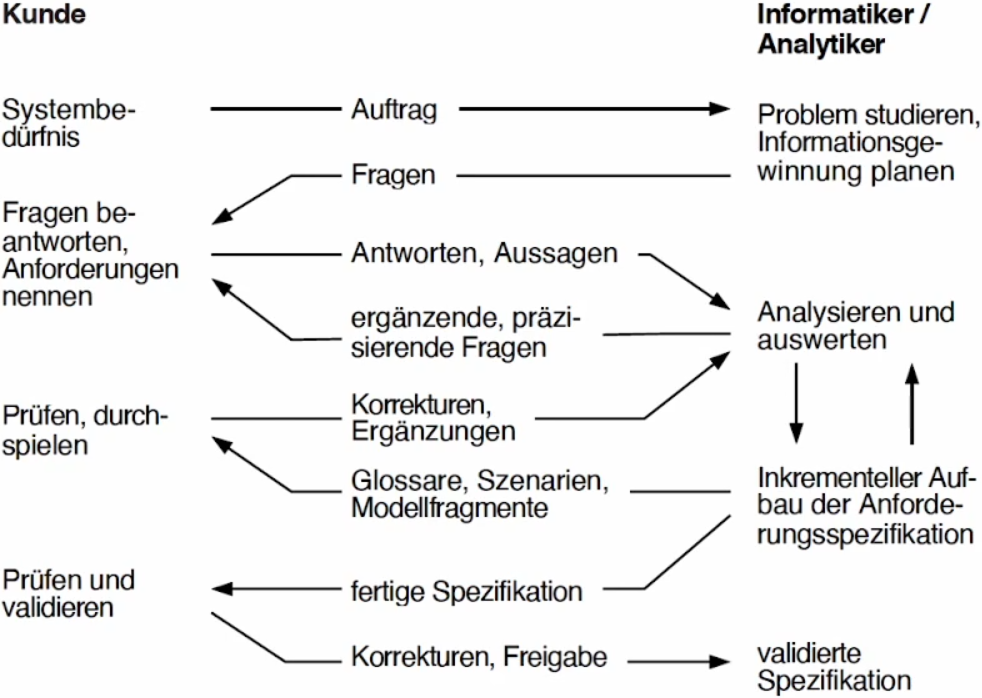


* Bei der Identifikation des Kontextes kommt es entscheidend auf die Betrachtungsebene an, und in vielen Projekten gibt es mehr als eine solche Ebene. Typisch gibt es mindestens:
  + Geschäftsebene
  + Systemebene
  + Softwareebene
  1. Vorgehen

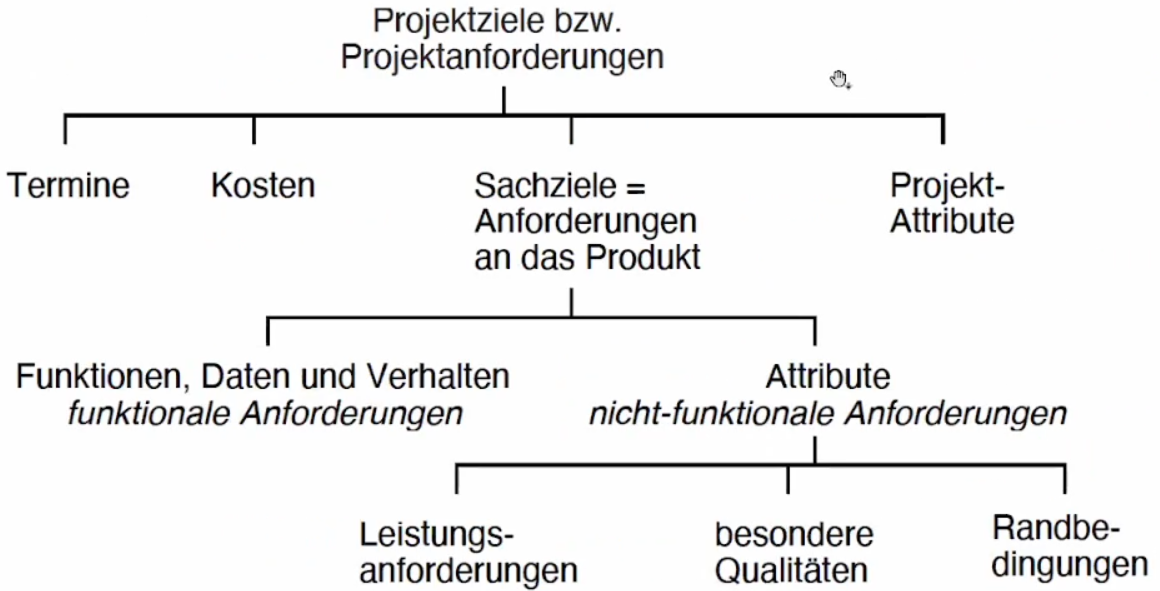
Kriterien, nach denen sich das Vorgehen richten kann:

* Muss die Spezifikation „wasserdicht“ sein, das heißt so gestaltet, dass sie als Vertrag für eine Realisierung der Software durch an der Spezifikation nicht beteiligte Dritte verwendet werden kann?
* Sind die Kunden und zukünftigen Benutzer bekannt und können sie in die Erstellung der Spezifikation einbezogen werden?
* Wird das zu spezifizierende System im Kundenauftrag oder für den Markt entwickelt?
* Wird das Projekt als Ganzes nach einem linearen oder einem inkrementellen Prozess abgewickelt?
* Soll für die Lösung Standardsoftware zum Einsatz kommen? (In diesem Fall müssen sich die Anforderungen zu erheblichen Teilen nach den Fähigkeiten der Standartsoftware richten.)

Möglicher Ablauf des Spezifikationsprozesses und Interaktionen der Beteiligten:



* 1. Klassifikation



* Funktionaler Aspekt
  + Daten: Struktur, Verwendung, Erzeugung, Speicherung, Übertragung, Veränderung
  + Funktion: Ausgabe, Verarbeitung, Eingabe von Daten
  + Verhalten: Sichtbares dynamisches systemverhalten, Zusammenspiel der Funktionen
  + Fehler: Normalfall und Fehlerfälle
* Leistungsaspekt
  + Datenmengen (durchschnittlich/im Extremfall)
  + Verarbeitungs-/Reaktionsgeschwindigkeit (durchschnittlich/im Extremfall)
  + Verarbeitungszeiten und -intervalle
  + wo immer möglich: messbare angeben!
  1. Priorisierung
* Anforderungen müssen oft nach ihrer Wichtigkeit priorisiert werden
* Hilft bei Einschränkungen durch Kosten/Zeit
* Planung für inkrementelle Entwicklungsprozesse
* Wenn Anforderungen zur Beschaffung eines (bestehenden) Systems dienen -> Auswahlkriterien

**Geläufige Priorisierungs-Klassen**

* **Muss**-Anforderungen: sind unverzichtbar und müssen in jedem Fall erfüllt werden
* **Soll**-Anforderungen: sollten erfüllt werden, sind aber bei zu hohen Kosten verzichtbar
* **Wunsch**-Anforderungen: werden nur erfüllt, wenn dies mit vertretbaren Kosten möglich ist (oft auch: **Kann**-Anforderung)