Inhalt

Was ist Softwareengineering?

Beschäftigt sich mit der Softwareherstellung. Schnittstelle zwischen Angewandte Ingenieurwissenschaften und Informatik. Hier geht es um die Entwicklung in der Praxis und die Herstellung von Software.

Kontrolle während diesem Prozess sehr wichtig. Prozess je nach Branche/Unternehmen verschieden.

Anforderungsanalyse – Architektur – Implementieren – Testen – Verkaufen, Freigeben

1. **Requirements (Anforderungen) definieren**: Was soll die Software machen? Hilfsmittel UML (Unified Modeling Language) Mit dieser Sprache kann man Modelle herstellen (Wie soll sich diese Software verhalten). Von einem Konsortium entwickelt namens OMG, da sind alle Grossunternehmen drin wie z.B. IBM. Use-Case und Activity Diagramme sind weitere Hilfsmittel.

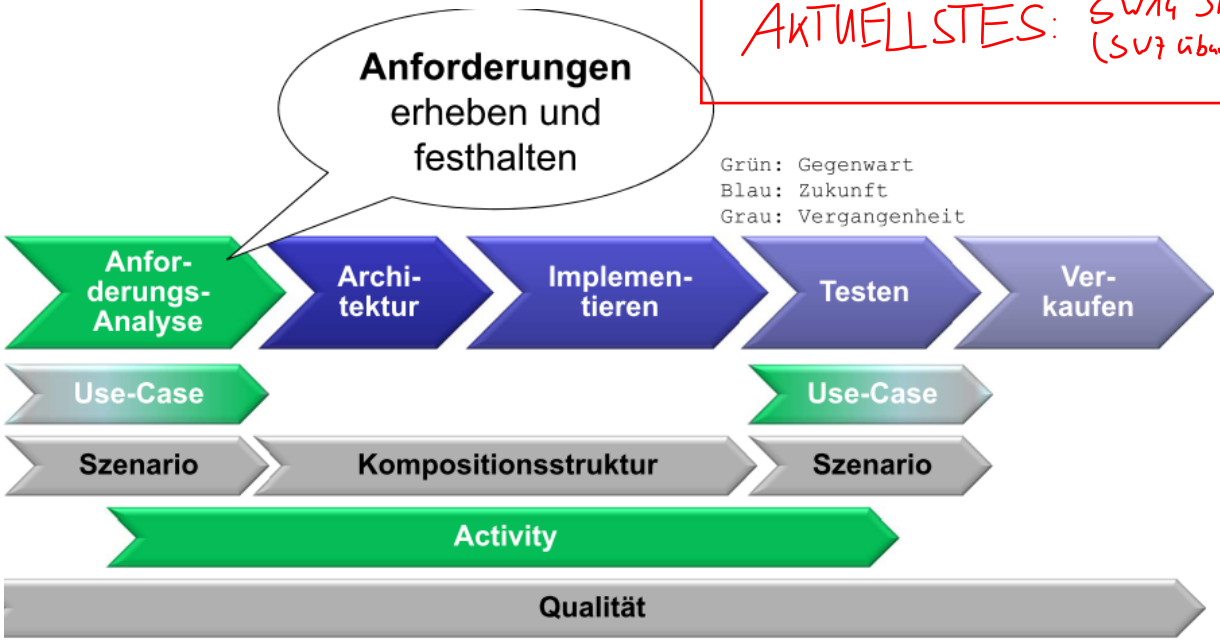
2. **System und Software Architektur festlegen:** Basisarchitekturen (Pattern), Frameworks, Komponenten, Datenstrukturen. Hilfsmittel: UML Klassendiagramme. Objektorientierte Programmierung.

3. **Implementation**: Programmcode schreiben, übersetzen und testen. Hilfsmittel: Frameworks, Entwicklungsumgebungen, Debugger (Ein **Debugger** (von engl. *bug* im Sinne von *Programmfehler*) ist ein Werkzeug zum Diagnostizieren und Auffinden von Fehlern in Computersystemen)

Code schreiben – Kompilieren – Linken – Ausführen – Debuggen – Code schreiben… (Zyklus)

1. Anforderungsanalyse

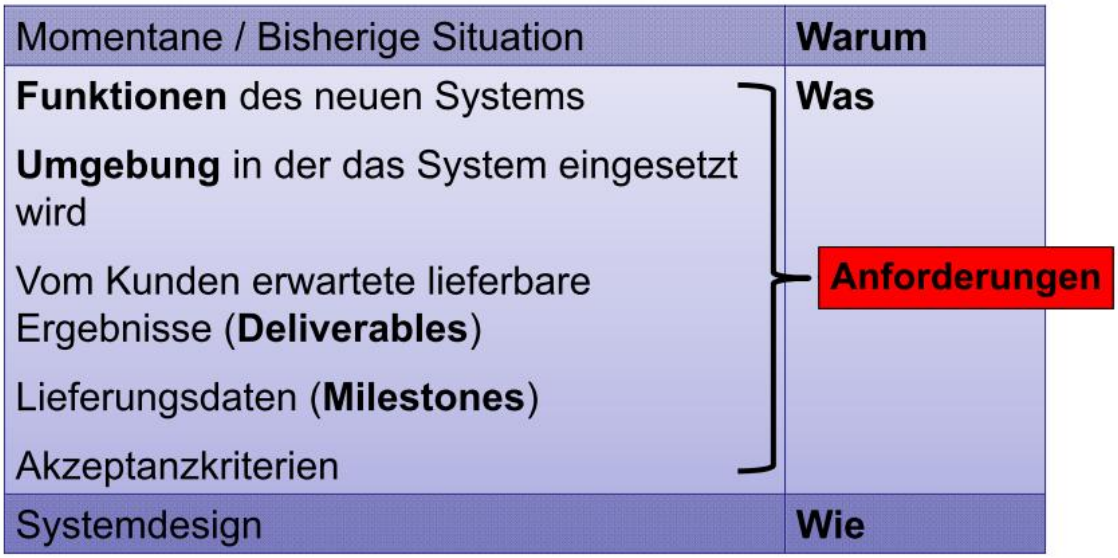
AKTUELLSTES: SW14 Slides (SW7 übungen)



Anforderungen

Was sind Anforderungen (Warum, Was, Wie)?

Benutzersicht auf ein System/Software. Die Anforderungen identifizieren das **WAS**!
Beispiele für Anforderungen: Funktionalität, Interaktion, Fehlerbehandlung, Umgebungs-Bedingung.



Was sind keine Anforderungen?
Systemstruktur, Systemdesign, Implementierungstechnologien, Entwicklungsmethodik

Wie kann man Anforderungen festhalten?
- Text: Lastenheft, Funktionsbeschreibung, Bilder: Skizze, UML, Film, Prototyp, Mock-up (kleines Modell das was kann und anzeigt wie das System schlussendlich werden sollte)

Funktionale Anforderungen = Aktion/Verb = WAS?	
Funktionalität	Was macht das System? - Input/Output-Beziehung - Reaktion auf abnormale Situationen - Exakte Sequenz von Operationen - Validierung von Input Funktionale Anforderungen werden oft als Aktion oder **Verb** formuliert: - Der Kunde kann am Bankomaten Bargeld **abheben**. - Der Kunde kann am Bankomaten den Kontostand **abfragen**.
Externe Schnittstellen	Interaktion mit Benutzern und anderen Systemen und beinhaltet die detaillierte Beschreibung aller In- und Outputs. - Beschrieb des Zwecks - Quelle des Inputs, Destination des Outputs - Datenformat, Wertebereich, Genauigkeit, Toleranz - Beziehungen und Abhängigkeiten zu anderen In- und Outputs
Nicht-funktionale Anforderungen = WIE? = Eigenschaften (kann nicht in Use-Case dargestellt werden!)	
Qualitätskriterien für Anforderungen 7 Stück!	- **Korrektheit**: Anforderungen entsprechen der Kundensicht - **Vollständigkeit**: Alle Anforderungen wurden erhoben, inkl. ausserordentliches Verhalten - **Konsistenz**: Anforderungen wiedersprechen sich nicht - **Eindeutigkeit**, Klarheit: Anforderungen können nur auf eine Art interpretiert werden - **Realistisch**: Anforderungen können umgesetzt werden - **Überprüfbarkeit**: Erfüllung der Anforderungen können geprüft werden - **Nachvollziehbarkeit**: Funktionen beziehen sich auf Anforderungen
Einschränkungen	Standards, existierende Hardware- und Softwareumgebung, rechtliche Anforderungen..

Qualitätskriterien (ALLE)

Funktionale Anforderungen

Korrektheit	Mathematisch beweisbar. Wird gefördert durch: - Einsatz angemessener Werkzeuge (*high-level-Sprachen*) - Benutzung Standardalgorithmen und *Libraries* - Einsatz von bewährten Entwicklungsprozessen	
Nicht-Funktionale Anforderungen		
Überprüfbarkeit	Überprüfen ob die schriftlichen Anforderungen erfüllt worden sind oder nicht.	Wird gefördert durch: - Überprüfbare Anforderungen - Modulares Design
Robustheit	Software muss zu 99.9 % der Anforderungen entsprechen. Die 0.1 % sollten sich "*vernünftig*" verhalten.	Kann gefördert werden durch: - Zusicherungen von Vor- und Nachbedingungen einzelner Prozesse - Softwareüberwachung, sichere Zustände
Security	Sichere Software gegen unautorisierten Zugang zu Information oder Manipulation.	Security kann gefördert werden durch: - Kryptographie - Sichere Protokolle (https, ssh-Verbindung)
Safety	Sichere Software minimiert das Risiko, dass durch die Software Menschen oder Umwelt gefährdet wird.	Kann gefördert werden durch: - Formale Ansätze bei der Programmierung (beweisbare Korrektheit/Mathematisch beweisen) - Einsatz von bewährten Entwicklungsprozessen - Reduktion der Anzahl sicherheitsrelevanter Komponenten(Software so einfach wie möglich gestalten, um sicher zu gehen, dass sie wirklich funktioniert).
Performanz	Software ist schnell, konsumiert wenig Speicherplatz.	Kann gefördert werden durch: - Berücksichtigung von Performanz während dem Entwurf der Softwarearchitektur - Optimierung des Programmcodes (schlanken Code gestalten).
Skalierbarkeit	Performanz skalierbarer Software kann erhöht werden, indem mehr Ressourcen (z.B. Hardware) eingesetzt werden.	Kann erreicht werden durch: - Dezentrale Architekturen (Anzahl Benutzer können verdoppelt werden ohne dass auf meiner Seite mehr Kosten entstehen) - Niedrige Komplexität der Algorithmen
Benutzbarkeit	Software, die leicht und intuitiv zu bedienen ist.	Kann erreicht werden durch: - Bereitstellung unterschiedlicher Interfaces - *Adaptive Interfaces* - Neue Formen von human *computer interaction (HCI)*
Interoperabilität	Software kann mit anderen Systemen koexistieren und kooperieren.	Kann gefördert werden durch: - *Gut dokumentierte Schnittstellen* (Dateiformate, Protokolle wie http) - Standardschnittstellen (XML)
Wartbarkeit	Software, die sich später erweitern oder verändern lässt.	Kann erreicht werden durch: - Unterteilung des Systems in lose gekoppelten Untersystemen, welche individuell gewartet werden können. - Entwurf und Einsatz von allgemeingültigen Schnittstellen, - Formate und Protokolle. - Erstellung guter Dokumentation. - Erstellung von Testsuiten
Wiederverwend-barkeit	Eher selten, da zeitlich begrenzt. Wiederverwendbare Software kann verwendet, angepasst und zusammengefügt werden, um neue Systeme zu entwickeln.	Wiederverwendbarkeit kann gefördert werden durch: - Modularem Design, allgemeingültigen Schnittstellen, - Formaten und Protokollen - Gute Dokumentation - Objekt-orientierten Technologien

Anforderungen erheben (Arten von Anforderungserhebung)

Interface Engineering	Service eines existierenden Systems in einer neuen Umgebung anbieten. Ausgelöst durch: - Neue Technologien oder Bedürfnisse
Re-engineering	Re-Design und/oder Re-implementierung eines bestehendes System (Bsp: Einführung neuer Funktionen). Ausgelöst durch: - Neue Technologien oder Bedürfnisse
Greenfield Engineering (Grüne Wiese)	Von Grund auf ein neues System entwickeln. Anforderungen werden von Kunden und Benutzern erhoben, Ausgelöst durch: - Benutzerbedürfnisse.

Informationsquellen für Anforderungserhebungen

- Kunde (Sitzungen, Gespräche, Auftrag)
- Existierende Dokumentation (Benutzerhandbücher, Unternehmensstandarte)
- Benutzer (Wenn möglich auch mit Benutzer sprechen, nicht nur mit Kunde)
- Beobachtung (Benutzer beim Verrichten ihrer Arbeit beobachten)

Anforderungen festhalten

	Szenario	Use-Case	Activity	Sequenz-diagramm
Zielpublikum (Typisch)	Kunde, Benutzer	Kunde, Benutzer, Entwickler	Entwickler	Entwickler
Form	Natürliche Sprache	Diagramm, Natürliche Sprache	Diagramm	Diagramm
Detaillierungsgrad Vollständigkeit				
Einfachheit				

Iteratives Vorgehen zur Erfassung von Anforderungen

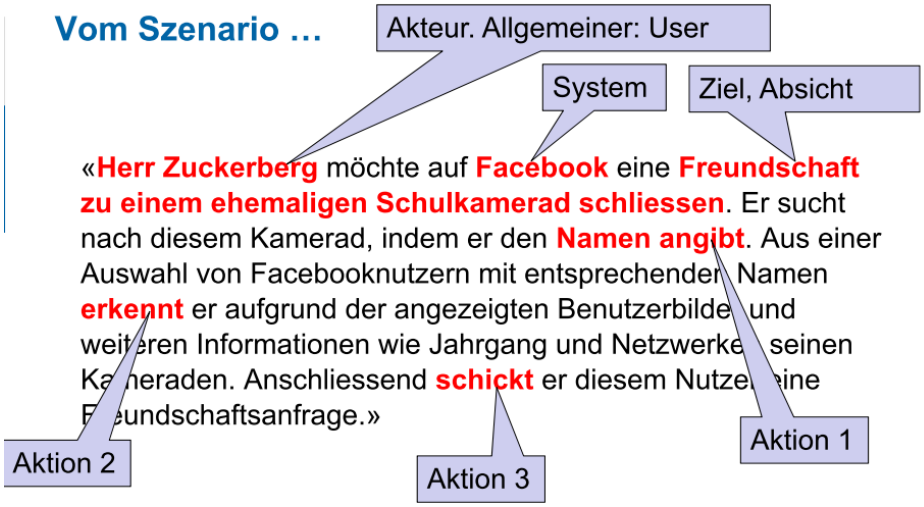
- Der *Requirements Engineer* hilft dem Kunden, die *Anforderungen zu formulieren*.
- Der *Kunde* hilft dem Requirements Engineer, die *Anforderungen zu verstehen*.
- Die Anforderungen entwickeln sich weiter, während die Szenarien entwickelt werden.
- **Achtung: Der Kunde versteht die Applikationsdomäne, nicht die Lösungsdomäne** (Benutze Applikationsdomäne-Ausdrücke. Zum Beispiel, für FB, "Person" statt "Profil").

Szenario

Erzählerische Beschreibung davon, was Benutzer machen und erfahren, wenn sie ein System benutzen. Das Szenario behandelt sowohl Normal- wie auch Ausnahmesituationen.

- Eine handelnde Person (Akteur) & System
- Der Akteur verfolgt ein Ziel oder verrichtet eine Aufgabe
- Der Handlungsstrang (Aktion = Verb), der aus einer Sequenz von Aktionen und Ereignissen = Ereignissequenz besteht, erzählt aus der Perspektive des Akteurs, wie diese Aufgabe verrichtet wird. Der Handlungsstrang kann mit Hintergrundinformationen (Interessen, Vorwissen, Absichten) ergänzt werden.

Szenario Beispiel:



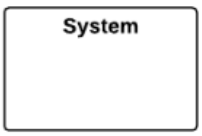
Identifikation von Akteuren

Akteure repräsentieren eine Art von Benutzern, nicht einen bestimmten Benutzer (z. B. Rollen)

- Ein Benutzer kann mehrere Rollen haben
- Mehrere Benutzer können dieselbe Rollen haben
- Was für Arten/Gruppen von Benutzern werden durch das System unterstützt?
- Welche Benutzergruppen führen die Hauptfunktionalitäten des Systems aus?
- Welche Benutzergruppen führen Nebenfunktionalitäten aus? (Wartung, Administration, …)
- Mit welcher externer Hardware und Software interagiert das System? Identifikation von Use-Cases
- Was für Aufgaben soll das System für Akteure erledigen?
- Auf was für Informationen greift der Akteur zu?
- Wer kreiert, liest, manipuliert und löscht Information?
- Was für Informationen muss der Akteur liefern?
- Über was für Ereignisse muss das System den Akteur informieren?

Use-Case Diagramm

Use-Cases stellen nicht alle Anforderungen grafisch dar, nur als Modell, vereinfacht wie die Landkarte Italien. Auf der Karte ist Italien klein dargestellt, doch in Wirklichkeit ist sie riesig. So sieht es auch mit der Erstellung eines Systems aus.

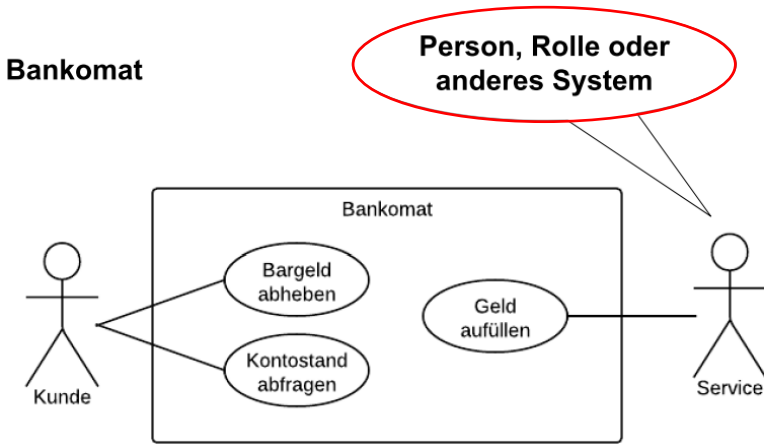
 Actor	**Actor (Akteur):** Der Akteur modelliert eine Person oder eine Rolle, die ein externer Benutzer oder ein externes System während der Interaktion mit dem System einnimmt. Akteure immer ausserhalb des Systems!!!
 Use Case	**Use-Case (Anwendungsfall):** Der Use-Case spezifiziert eine abgeschlossene Menge von Aktionen, die von einem System bereitgestellt werden und einen erkennbaren Nutzen für einen oder mehrere Akteure erbringen.
 System	**System Boundary (Systemgrenze):** Die System-grenze umfasst ein System, das die benötigten Use-Cases bereitstellt und mit dem die Anwender interagieren. Akteure befinden sich immer ausserhalb der Systemgrenze.

Konnektoren (1/2)

———	**Association (Assoziation):** Eine Assoziation modelliert eine Beziehung zwischen Akteuren und Use-Cases. Hat ein Use-Case Assoziationen mit mehreren Akteuren, werden für die Ausführung des Use-Cases alle Akteure benötigt. Die (ungerichtete) Assoziation definiert eine bidirektionale Kommunkation: Der Akteur kann sowohl den Use-Case aufrufen (initiieren), als auch vom Use-Case Informationen erhalten (teilnehmen). bidirektional
——→	**Directed Association (Gerichtete Assoziation):** Die gerichtete Assoziation unterscheidet sich von der ungerichteten Assoziation dadurch, dass eine Kommunikationsrichtung definiert ist. Der Akteur kann entweder den Use-Case aufrufen, oder vom Use-Case Informationen erhalten. unidirektional

nur zw. Akteuren & Use-Cases!

Pfeil auf mich gerichtet: Ich erhalte Infos. Pfeil von mir auf jdm. / etw. anderem gerichtet: Ich rufe den Use Case auf. ENTWEDER ODER!



≙ initiates

≙ participates

Konnektoren (2/2)

NIE zw. Akteur & Use-Case (meist ausserhalb System)

←——	**Generalization (Generalisierung):** Eine Generalisierung kann Akteure oder Use-Cases verbinden und modelliert eine Beziehung zwischen einem spezifischen und einem allgemeinen Element.
<<include>> ⟵	**Include Relationship (Include-Beziehung):** Die Include-Beziehung modelliert eine unbedingte Einbindung der Funktionalität eines Use-Cases eines anderen Use-Cases.
<<extend>> ⟵	**Extend Relationship (Extend-Beziehung):** Die Extend-Beziehung modelliert die bedingte Einbindung der Funktionalität eines Use-Cases in einen weiteren Use-Case.

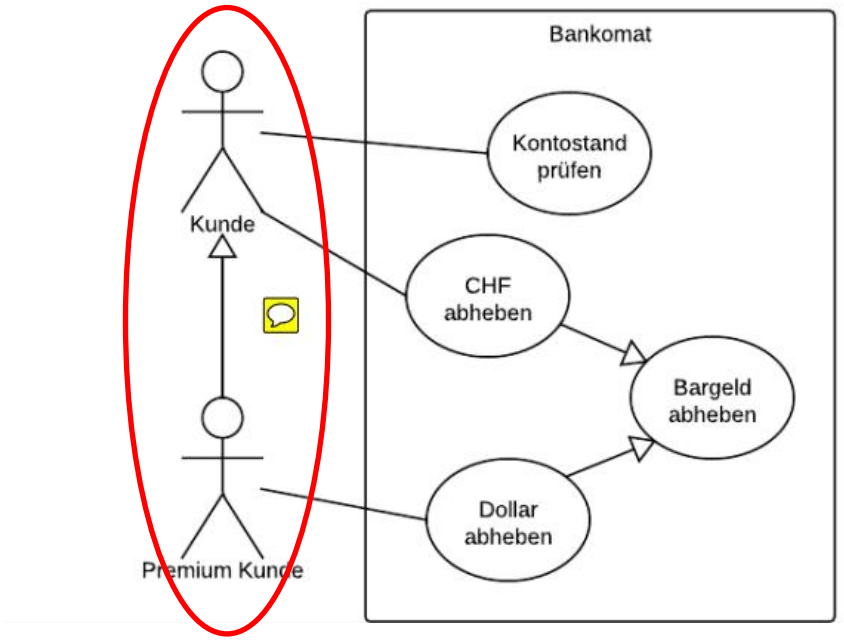
meist Use-Cases (innerhalb System)

Generalisierung

Spezifischen: Premium-Kunde

Allgemeinen: Kunde

Beispiel Bankomat korrigiert und erweitert



Premium Kunde kann alles was Kunde kann und noch zusätzlich Dollar abheben!

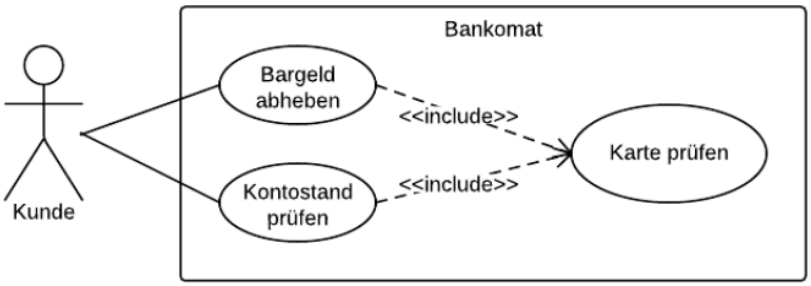
Unterschied ←<<include>> und <<extend>>→

Include Relationship: Wenn ein Use-Case ausgelöst wird, wird der andere auch ausgelöst (Essen -> include -> Essen zubereiten). `zwingend auch ausgeführt! (bei Mehrfachverwendung ODER hierarchischer Gliederung)`

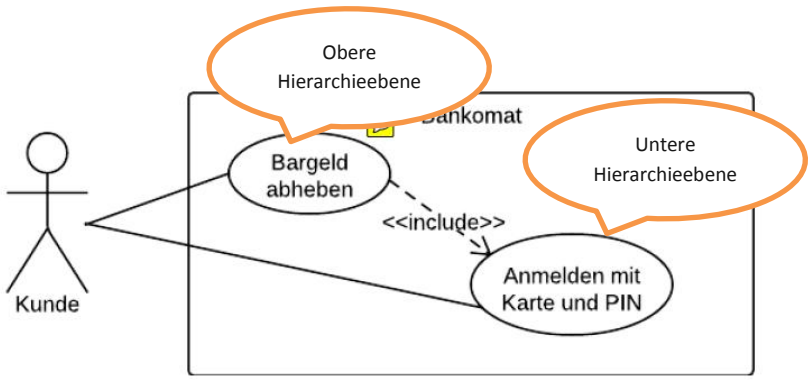
Extend Relationship: Trinken/Extension Points:*Happy hour* <<extend>>--> Gratis Getränk. Hier ist eine Bedingung dabei, die lautet: *Zwischen 9:00 und 10:00 Uhr gibt es ein Gratisgetränk* also nur wenn Happy Hour ist!! `Erweiterung mit Bedingung!`

Include- Wann macht es Sinn?

Mehrfachverwendung



Hierarchische Gliederung

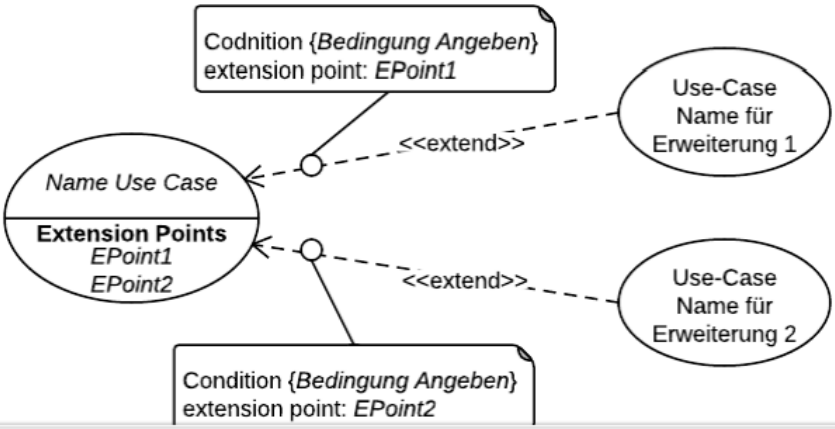


Extend

Pfeil zeigt zum erweiterten Use-Case, also zu demjenigen Use-Case, der die Erweiterung benutzt.

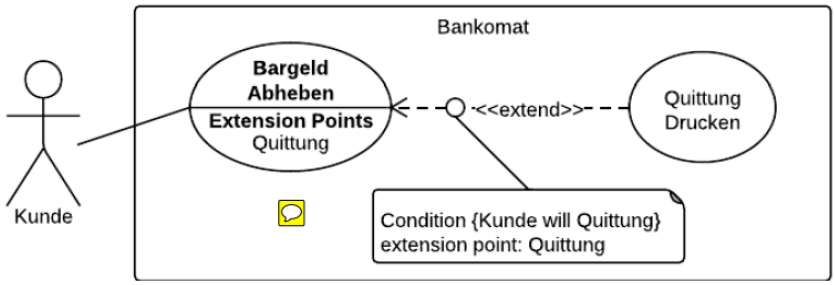
Vollständige Notation:

Bedingung nicht vergessen!!!



Extend Beispiel

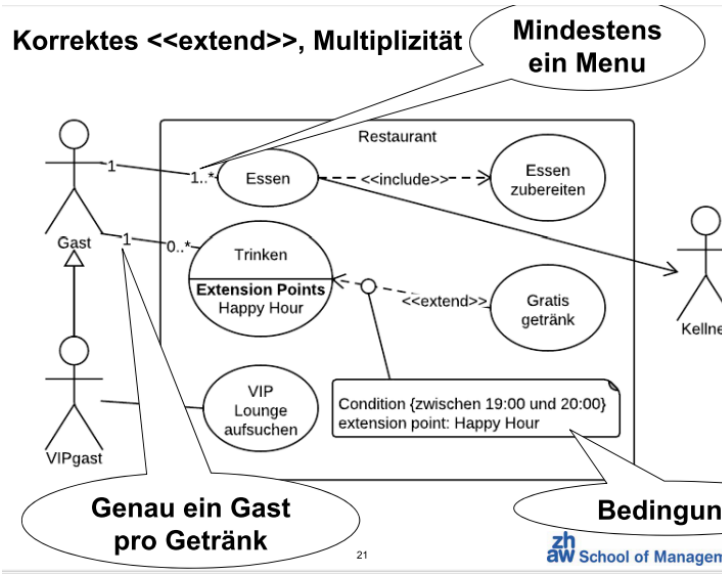
Bankomat mit Quittung



Multiplizität

Kann bei der Assoziation verwendet werden `--> %`

Zeigt, dass ein Akteur mehrmals den Use-Case anstoßen kann, aber nicht muss.



Multiplizität	Beschreibung
1	Genau einmal
1..3	Zwischen einmal und dreimal
0..*	Zwischen keinmal und unendlich oft

Dos and Don'ts Use Case

- Use-Case nicht für die detaillierte Beschreibung der Funktion verwenden.
- Use-Case immer aus der Sicht des Anwenders und nicht
- des Entwicklers erstellen (Use-Case).
- Nicht die interne Struktur beschreiben, sondern nur was gegen aussen sichtbar ist. Für interne Strukturen gibt es andere Diagramme.
- Keine nicht-funktionale Anforderungen darstellen.
- Keine Akteure innerhalb des Systems.
- Keine Generalisierung zwischen Akteur und Use-Case.
- Arten von Anforderungen

Use-Case Beschreibung Allgemein

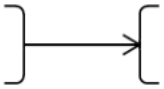
1. Name des Use-Cases
2. Actors (Beschreibung der Akteuren, welche im Use-Case involviert sind)
3. Eintrittsbedingung ("Dieser Use-Case beginnt, wenn …")
4. Ereignisssequenz (Freie Form, informelle, natürliche Sprache)
5. Austrittsbedingung ("Dieser Use-Case ist beendet, wenn …")
6. Ausnahmen (Was geschieht, wenn etwas schief geht?)
7. Besondere Anforderungen (Nicht-funktionale Anforderungen)

Activity Diagramm (SW04)

Activity Diagramme dienen dazu, Abläufe jeglicher Art sowie deren Regeln darzustellen. Activity Diagramme sind die in der Praxis am häufigsten eingesetzten Diagramme.

SZENARIO – USE-CASE DIAGRAMM– ACTIVITY DIAGRAMM

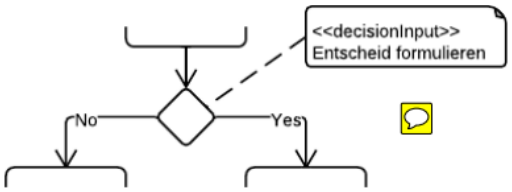
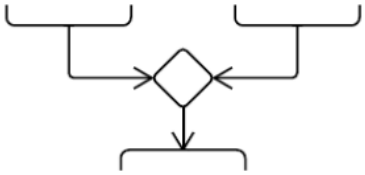
Elemente und Konnektoren

	**Action (Aktion):** Ist eine ausführbare Funktionalität, die im Modell nicht weiter aufgeteilt werden soll. Die Aktion ist atomar.
	**Control Flow (Kontrollfluss):** Ist eine gerichtete Verbindung zwischen Aktionen und definiert die Ausführungsreihenfolge.
	**Initial Node (Startknoten):** Definiert den Startpunkt des Kontrollflusses.
	**Final Node (Endknoten):** Definiert das Ende des Kontrollflusses.

Atomar = nicht aufteilbar !!

Entscheidungs- und Verbindungsknoten

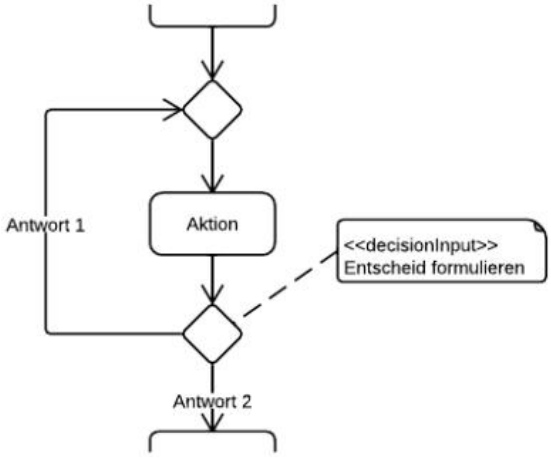
Mit Entscheidungs- und Verbindungsknoten können Entscheide und Schleifen dargestellt werden.

	**Decision Node (Entscheidungsknoten):** Definiert eine Verzweigung des Kontrollflusses, bei der genau einer der möglichen Kontrollflüsse ausgeführt wird.
	**Merge Node (Verbindungsknoten):** fasst mehrere alternative Kontrollflüsse zusammen.

Token geht entweder in die rechte oder linke Richtung. Nie werden beide Aktionen gleichzeitig ausgeführt. `==> Entscheidung (wahr / falsch)!`

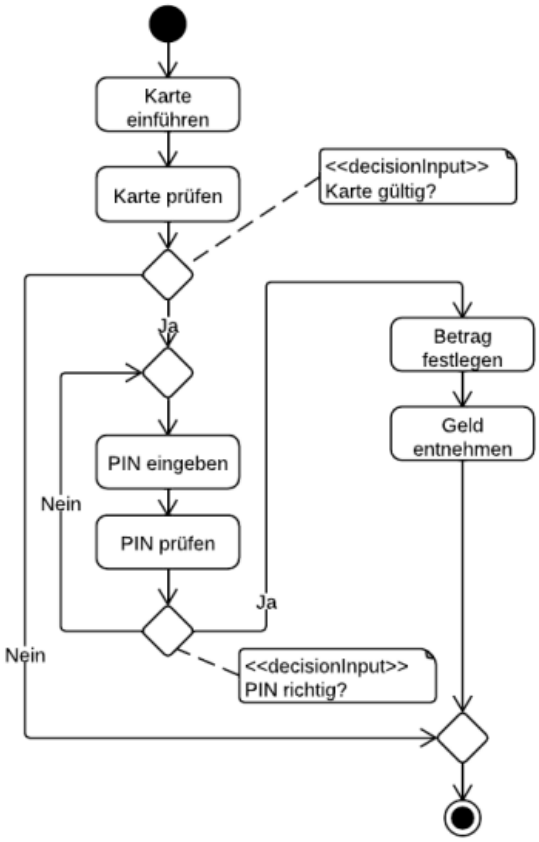
Schleife

Die Aktion wird wiederholt, bis die Entscheidung mit Antwort 2 beantwortet wird.



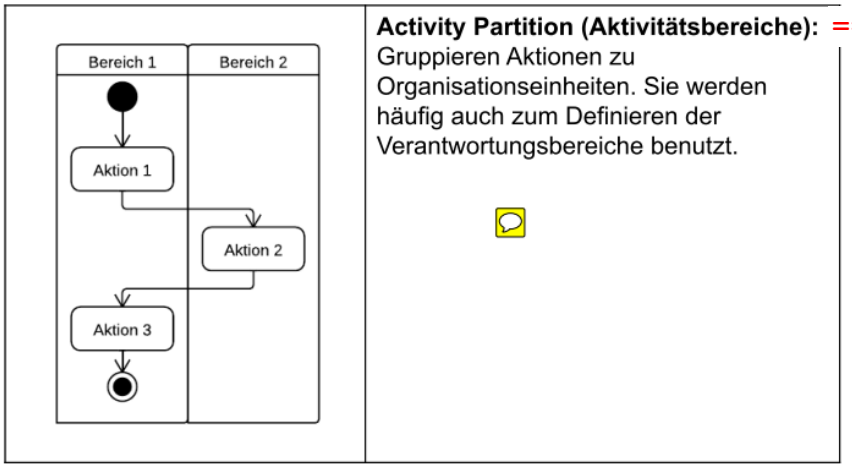
Schleife Beispiel

Bankomat

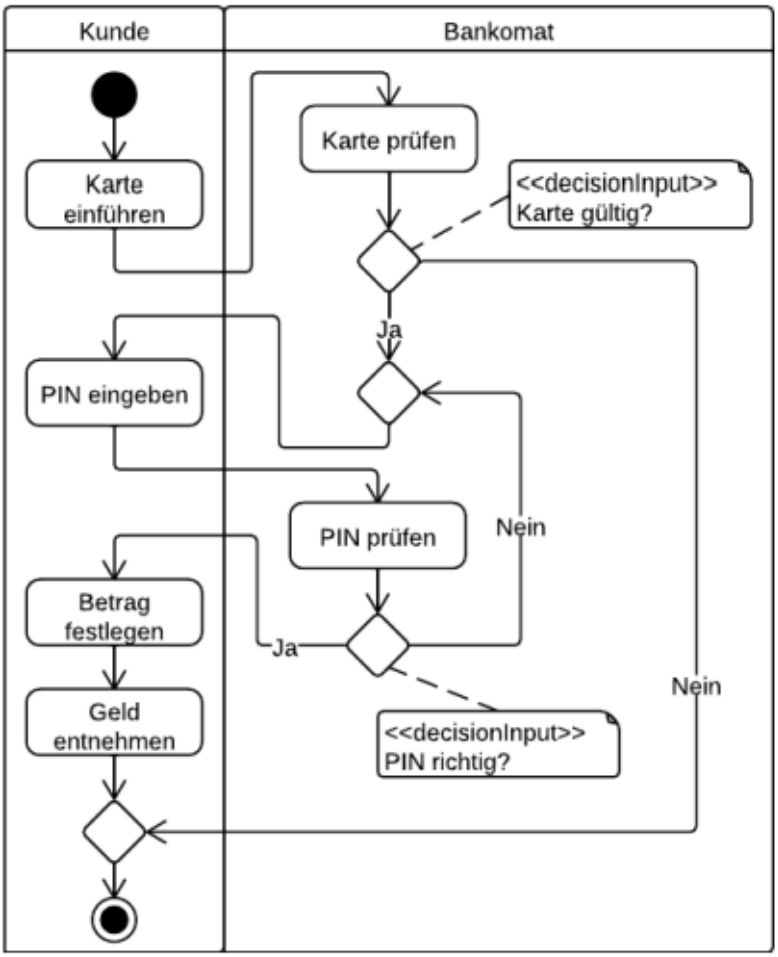


Aktivitätsbereiche (Activity Partition)

Definieren wer in welchen Bereichen für was zuständig ist!



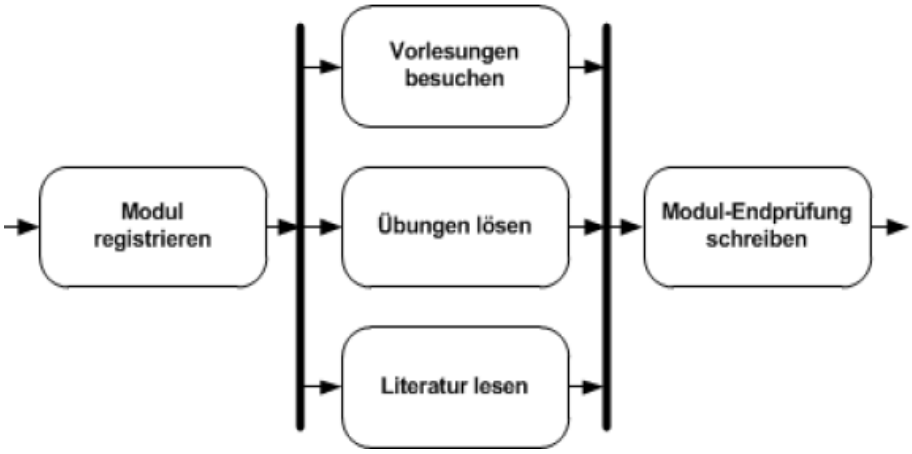
Activity Partition (Aktivitätsbereiche): ==> Swim Lanes
Gruppieren Aktionen zu Organisationseinheiten. Sie werden häufig auch zum Definieren der Verantwortungsbereiche benutzt.



Parallelisierungs- und Synchronisationsknoten

Anfang paralleler Aktionen und deren Zusammenkommen (UND Verknüpfung). Das Token wird beim Parallelisierungsknoten vervielfacht und die Aktion nach dem Synchronisationsknoten wird ausgeführt, wenn alle Tokens angekommen sind.

Mehrere Token, die gleichzeitig Aktionen durchlaufen, wobei keine Entscheidung dazwischen läuft. Erst wenn all diese Aktionen durchgeführt werden, kann weiter zur "Modul-Endprüfung schreiben" gegangen werden



Von Use-Case zu Activity Diagramm

Vorgehen:

- Für jeden Use-Case ein Activity Diagramm erstellen.
- Aus jedem Schritt der “normalen” Ereignissequenz wird
- ein Activity Diagramm gemacht.
- Die Ausnahmefälle als zusätzliche Aktionen einbinden.

Dokumentation und Validierung

1. Anforderungsdokumentation
2. Anforderungspriorisierung
3. Anforderungsvalidierung
4. Projektübereinkommen.

Anforderungsdokumentation

1. Einführung (Text verfassen: Was, warum, wie, Szenario)
    a. Zweck, Anwendungsbereich und Abgrenzung des Systems
    b. Ziele und Erfolgskriterien des Systems
    c. Definitionen, Abkürzungen, Referenzen, Übersicht
2. Gegenwärtiges System
3. Vorgeschlagenes System
    a. Übersicht
    b. Funktionale Anforderungen (Szenarien, Use-Cases, Activities, Sequenzdiagramme)
    c. Nicht-funktionale Anforderungen
4. Glossar

Prioritäten von Anforderungen

Hohe Priorität (Kernanforderungen):
- Müssen während der Analyse, des Designs und der Realisierung angegangen werden.
- Müssen bei der Kundenabnahme erfolgreich demonstriert werden.

Mittlere Priorität (Optionale Anforderungen)
- Müssen während der Analyse und des Designs angegangen werden.
- Normalerweise in einer zweiten Runde implementiert und demonstriert.

Niedrige Priorität (Nice To Have)
- Müssen während der Analyse angegangen werden.

Anforderungsvalidierung

Wenn die Anforderungen erhoben wurden, sollen diese überprüft und vom Kunden und den Entwicklern abgenommen werden.

Überprüfung durch Kunde und Entwickler:
- Prüfung auf Qualitätskriterien
- Review, Walkthrough, …: Gesunder Menschenverstand!

Validierung kann auch mittels eines Prototyps vorgenommen werden:
- Kunde kann erfahrungsgemäss besser Feedback geben
- Erster Machbarkeitsnachweis
- Fokus auf Benutzerschnittstellen

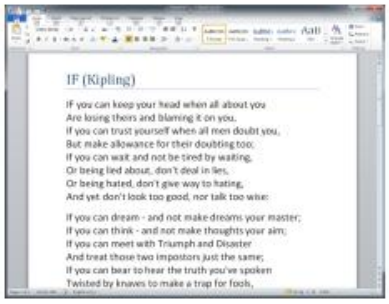
Projektübereinkommen

Der Kunde akzeptiert die Anforderungsdokumentation und das Analysemodel. Der Kunde und die Entwickler stimmen überein betreffend der Funktionen und Features des Systems, sowie:
• Einer Liste von priorisierten Anforderungen
• Eines Revisierungsprozesses
• Kriterien welche benutzt werden, um das System zu akzeptieren oder verwerfen
• Einen Zeitplan (Milestones) • Ein Budget (der Preis)

2. Architektur festlegen

Was haben Programme gemeinsam?
Optionen die Daten kreieren, lesen, modifizieren, updaten und löschen können.

FB Eigenschaften: Benutzerprofil lesen, ändern, deaktivieren.
Kollegen lesen erstelltes Profil.



Eigenschaften: Titel, Inhalt

Unterschied zwischen allen Applikationen: Unterschiedliche Daten Grafisch verschieden

Eigenschaft:
Produkt und Produktnummer.
Auftrag: KUnde gibt auftrag (kreiert) und Besitzer "liest" Auftrag.

Beschrieb von Komponenten mit bestimmten Funktionen und wie sie miteinander interagieren. Z. B. ein Haus: Stuhl, tisch, Badezimmer, Esszimmer, etc.

Gemeinsamkeiten von Software Systemen

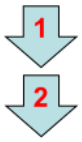
- Alle Systeme verwalten irgendeine Form von Daten.
- Daten werden kreiert, gelesen, verändert und gelöscht. ==> CRUD
- Auf den Daten werden Berechnungen ausgeführt.
- Operationen auf den Daten werden durch Benutzer oder andere Systeme ausgeführt, welche mit dem System interagieren.
- Alle Systeme brauchen Hardware.

Unterschiede von Software Systemen

- Systeme unterscheiden sich darin, dass sie unterschiedliche Daten in unterschiedlichen Formen verwalten.
- Verschiedene Systeme führen unterschiedliche Berechnungen auf die Daten aus.
- Systeme haben bieten unterschiedliche Interaktionsformen an.
- Systeme benutzen unterschiedliche Hardware.

Aufteilung eines Systems Variante 1

	Beispiele
Interaktion	User Interface (UI) Programmatisches Interface (API)
Operationen, Berechnungen	CRUD, Algorithmen
Daten Datenmodel	Java: Objekte und Klassen Relationale Datenbank: Relationen und Tuple



1: Interaktion besteht darin, Operationen und Berechnungen auszuführen, sowie deren Resultate zu "konsumieren".

2: Operationen und Berechnungen werden auf Daten ausgeführt.

Aufteilung eines Systems Variante 2

	Contacts	Camera	Tagging
Interaktion			
Operationen, Berechnungen	CRUD, Suchen	Fotographieren, Anschauen, Löschen	Personen mit Bildern Assoziieren
Daten	Personendaten	Bilder	Personen-Bilder-Verknüpfungen

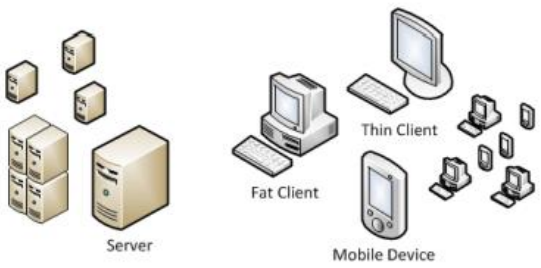
Contacts und Camera sind zwei unabhängige Anwendungskomponenten, Tagging ist eine dritte, welche auf die ersten zwei aufbaut.

Weitere Aufteilungsvarianten von Systemen

Funktional

dito Variante 1 oben!

Hardware- und Netzwerk-Orientiert



Technologien

Komponentenschnittstellen

- Ein Softwaresystem besteht aus Komponenten, die jeweils bestimmte Aufgaben erfüllen.
- Einige Komponenten bieten ihre Funktionalität anderen Komponenten oder der Systemaussenwelt an.
- Einige Komponenten benutzen die Funktionalität anderer Komponenten.
- Folglich definieren Komponenten sogenannte Schnittstellen, welche die gebotenen oder erwünschten Funktionalitäten definieren, sowie auch die Form deren Benützung.

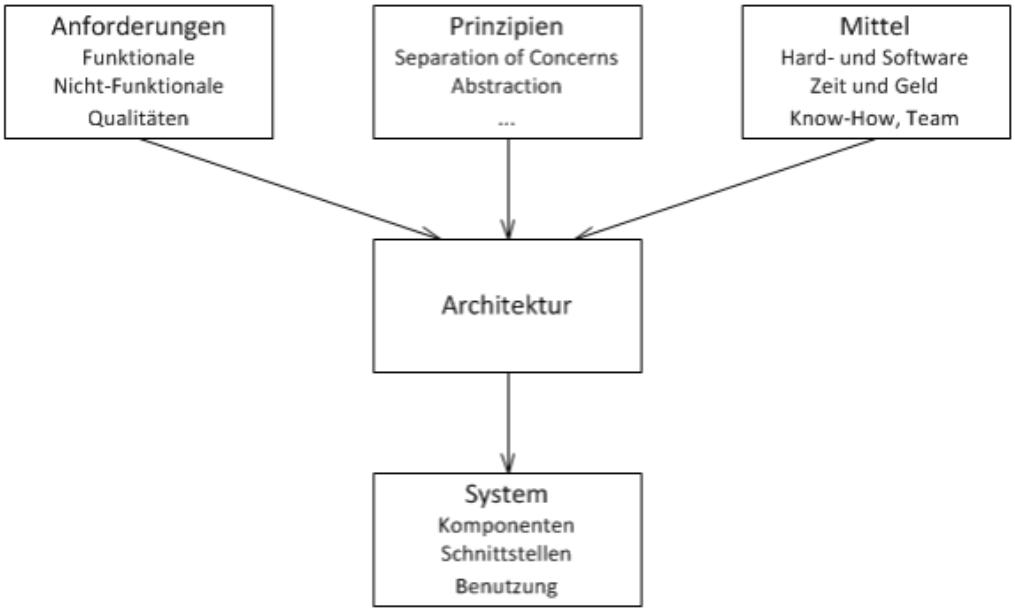
Architektur (IT)

Komponentenmodell = Architekturen

Definition von Hardware- und Software-Komponenten des Systems, die im Zusammenspiel (Aufgaben, Interaktionen) als System die Anforderungen erfüllen.

Spannungsfelder

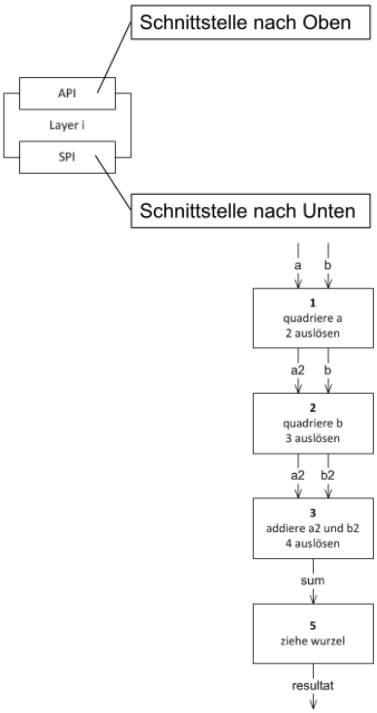
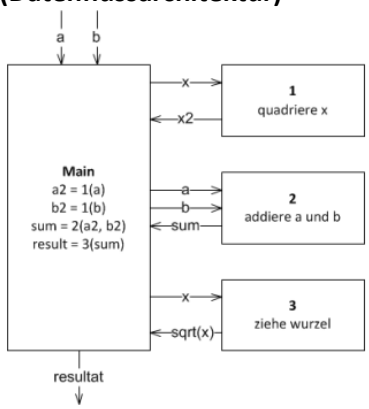
- Interessenvertreter (Kunde, Entwickler, Endbenutzer)
- Anforderungen (Beständigkeit, Zweckmässigkeit, Eleganz)
- Produkteanforderungen, Machbarkeit, Kosten usw.



SwE Prinzipien (damit System besser wird):

- Modularität = System ist in Komponenten unterteilt
- Separation of Concerns = Unterteilung von Zwecken/Aspekten
- Abstraktion = Vereinfachung gegen aussen (im Innern komplexer)

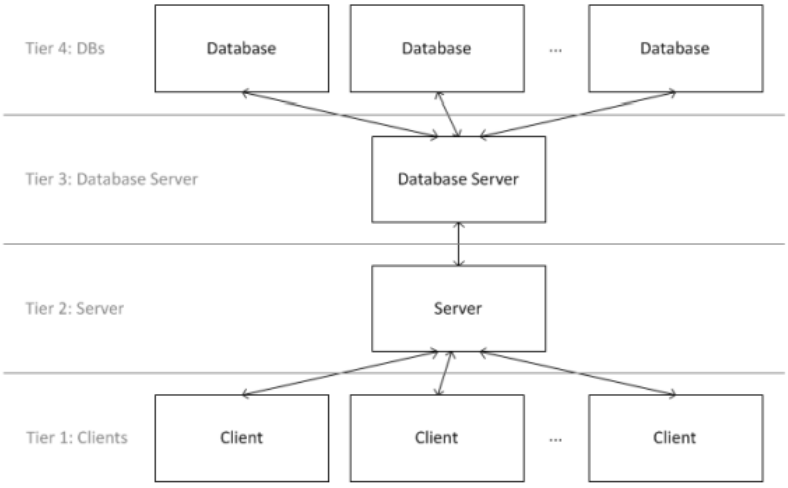
Basisarchitekturen

Architektur	Eigenschaft	SE Prinzipien	Qualitätskriterien
Monolith	Keine Aufteilung in Komponenten, System als einen einzigen Baustein.	Keine Separation of Concerns, Keine Modularität, Keine Abstraktion	Wartbarkeit, Wiederverwendbarkeit, Erweiterbarkeit
Schichtenarchitekturen 	Anwendung wenn es eine hierarchische Abhängigkeitsstruktur von Komponenten gibt. Jede Schicht kommuniziert ausschliesslich mit seiner darunterliegenden Schicht. Jede Schicht hat eine spezifische Teilaufgabe des Gesamtsystems. Bei jedem Übergang von einer zur darunterliegenden Schicht wird (nach unten) etwas hinzugefügt oder (nach oben) wieder entfernt. Jede Schicht bietet nach oben eine wohldefinierte Schnittstelle an, und nach unten erfüllt sie die Schnittstelle der darunterliegenden Schicht.	Abstraktion (Schnittstellen) Separation of Concerns (Jede Schicht hat eine Aufgabe) Modularität (Jede Schicht bildet ein Modul) Erfüllte Qualitätskriterien Wartbarkeit Erweiterbarkeit Robustheit Überprüfbarkeit	
Batch Sequential **(Datenflussarchitektur)**	Jede Komponente nimmt eine Eingabe entgegen, führt einen Teilschritt (z.B. Berechnung) aus, und leitet mit der Ausgabe des Teilschrittes die Ausführung der folgenden Komponente aus.	Formality Separation of Concerns Modularity	Korrektheit Verständlichkeit Robustheit *Schlechte Wiederverwendbarkeit*
Verbesserung Batch Sequential **(Datenflussarchitektur)** 	Eine zusätzliche Komponente (Main) codiert die Reihenfolge der Ausführung der bisherigen Komponenten. Komponenten wurden generalisiert (z.B. Komponente 1: eine Eingabe, Quadrat als Ausgabe) und konnten so wiederverwendet werden.	Formality Separation of Concerns Modularity Abstraction Bessere Separation of Concerns (Ausführung der Berechnungen und Operationen ist getrennt von der Ausführungsreihenfolge)	Korrektheit Verständlichkeit Robustheit Wiederverwend-barkeit Wartbarkeit Erweiterbarkeit

Architektur	Eigenschaft	SE Prinzipien	Qualitätskriterien
N-Tier Architekturen * 2-Tier Architektur z.B.	Aufteilung von Komponenten auf mehere Rechner. Leistungsfähigkeit (Rechner hat nur eine Aufgabe) Client-Server: einmal auf Server aufsetzen, mehrfach und von überall benutzbar. Rich-client vs. Thin-client	Abstraction Modularity Separation of Concerns	Skalierbarkeit Sicherheit Wartbarkeit Robustheit, Performanz, Verlässlichkeit
Peer-to-Peer Architekturen *	Dezentrale Netzwerke werden angewendet, wenn ein System sehr Robust, Skalierbar und mit einfachen Mitteln betrieben werden muss. Dezentrale Algorithmen erzielen erstaunliche Effekte und ergeben sehr Anpassungsfähige Systeme. Es ist schwierig, aufgrund von erwünschtem globalen Verhalten die lokalen Regeln herzuleiten.	Modularität Abstraktion	Robustheit Skalierbarkeit
Ereignisorientierte Architekturen *	Komponenten, welche über Ereignisse benachrichtigen (Publishers), wissen nichts von den Komponenten, welche dafür registriert sind (Subscribers). Eignet sich, wenn eine lose Kopplung der Komponenten gefragt ist, z.B. für verteilte Systeme (z.B. P2P).	Separation of Concerns Abstraktion Modularitaet	Skalierbarkeit
SOA Service-Oriented Architecture *	Typischerweise wird SOA eingesetzt, wenn heterogene Systeme integriert werden sollen. Des Weiteren gibt es die Idee von «**Software as Service**», nach dem für die Konsumation von Services bezahlt werden muss.	Abstraktion Modularität Separation of Concerns	Erfüllte Qualitätskriterien Interoperabilität Wiederverwendbarkeit Skalierbarkeit

2-TierArchitektur
Browser ist der Client, der Computer den Sie anfragen und der das Resultat zurückgibt ist der Server. Der Client übernimmt die Interaktion (Eingabe, Darstellung), der Server übernimmt die Berechnung/Operation und Datenhaltung.

4-Tier-Architektur



Peer-to-Peer Architekturen

Ein Peer ist eine Einheit des Systems, welche mit anderen Peers vernetzt ist, und welche sich nach bestimmten Regeln verhält. Die anderen Peers, mit denen ein Peer direkt verbunden ist bilden seine Nachbarschaft.

Die Regeln seines Verhaltens halten fest, wie ein Peer seine Nachbarn wählt, sowie was und wie er mit ihnen kommuniziert.

Beispiel: File Sharing

Ein Peer kreiert eine Anfrage nach einem File mit bestimmten Namen und gibt sich selber diese Anfrage.
Jeder Peer, welcher eine Anfrage erhält, bearbeitet diese wie folgt: Nachschauen, ob Peer im Besitz des angefragten Files ist. Falls ja, wird seine Identifikationsnummer (ID) demjenigen Nachbarn zurückgeschickt, von dem die Anfrage als erstes erhalten wurde. Falls nein, wird die Anfrage an alle Nachbarn ausser demjenigen, von dem sie erhalten wurde, weitergeleitet. Der Peer merkt sich, von wem er die Anfrage als erstes erhalten hat. Sobald er eine Antwort (ID) eines Nachbarn kriegt, leitet er sie diesem Nachbarn weiter.

Weitere Beispiele

Internet (z.B. Routing Protocol)
File Sharing (eMule, Gnutella, Napster, Bit Torrent)
Content Sharing (Freenet)
Mobile Anwendungen (eher Forschungsprojekte)
Sensornetzwerke
Geteilte Berechnungen (SETI@HOME, …)
Finden eines kürzesten Weges (Ameisenalgorithmus)

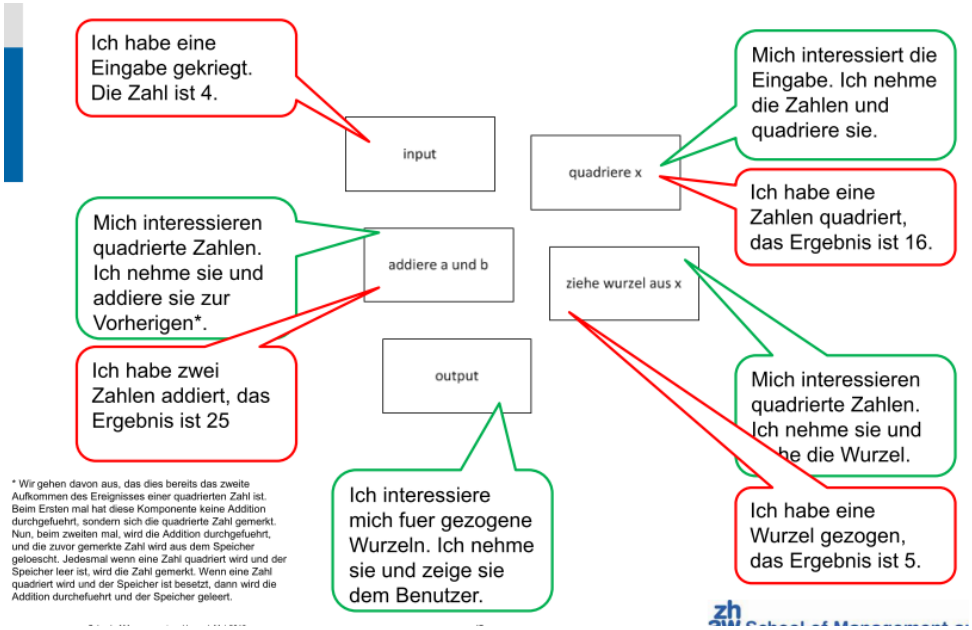
Aspekte der Zentralität (P2P)

In gewissen Peer-to-peer Anwendungen haben ein Teil der Peers (Superpeers) erweiterte Aufgaben (z.B. Erster Nachbar für Neuankömmlinge, Verteilung von Daten).
In diesen Anwendungen bilden die Superpeers, welche miteinander verbunden sind, ein eigenes Netzwerk. Alle anderen Peers sind jeweils nur an einem Superpeer angeschlossen.
In anderen Peer-to-peer Anwendungen existiert ein Server, welcher eine Anlaufstelle für Neuankömmlinge bietet. In noch anderen Anwendungen wird die Suche nach einem File auf solch einem Server durchgeführt, und das Resultat erlaubt es, zur Übertragung eine Peer-to-peer Verbindung herzustellen.

mehrere zentrale Server (Superpeers)
vs.
ein Zentralserver

Ereignisorientierte Architekturen

In Schichten- und Datenflussarchitekturen werden Berechnungen explizit durch eine Eingabe ausgelöst, was die Ausführung eines gesamten Prozesses mit sich führt. In den Peer-to-peer Netzwerken werden Lokale Regeln entweder periodisch oder aufgrund von Ereignissen aus der Nachbarschaft ausgelöst. In Ereignisorientierten Architekturen benachrichtigen die einen Komponenten die Allgemeinheit über das Stattfinden von Ereignissen, während andere Komponente für die Benachrichtigung gewisser Ereignisse registriert sind und bei deren Auftreten darauf reagieren.



Service-Oriented Architecture (SOA)

In Service-Oriented Architekturen (SOA), werden benötigte Funktionalitäten als separate, plattformunabhängige Services angeboten. Für jeden Service gibt es eine Spezifikation und mehrere Anbieter, welche einen Service anbieten, der dieser Spezifikation genügt. Wenn eine Komponente einen Service braucht, wird dieser aufgrund der benötigten Spezifikation gesucht und konsumiert. Die Entwicklung einer Applikation besteht darin, Services zusammenzustellen – man spricht von Orchestrierung.

Daten

Was wird mit Daten gemacht?
- Repräsentiert (Textfile, Binärfile, Tabelle, Datenbank, Objekte)
  Fragen wie Verarbeitbarkeit? Persistent? Grösse? Lesbarkeit?
- Komponenten geben/nehmen Daten aus/auf (Input/Output, Einlesen aus externen Quellen)
- Datenverarbeitung (kreieren, Lesen, Suchen, Manipulation, Löschen, Analyse)
- Datenübertragung

Daten werden:
- kreiert
- gelesen
- verändert
- gelöscht

==> CRUD

Information = Daten und Datenmodell (beschreibt Daten)

Objekt-Orientierte Datenmodelle

Einfache Daten werden durch einfache Typen wie z. B. integer beschrieben (z. B. Zahlen, welche einzelne Werte beinhalten).

Komplexe Daten Behälter für mehrere, unterschiedliche Werte. Beschrieb dieser Daten: Festlegen wie viele und was für Werte sie beinhalten und wie diese Werte genannt werden (Beispiel: Benutzerprofil = Behälter für Vornamen, Namen, Emailadresse und Passwort).

Objekte beinhalten einfache Daten wie Zahlen und Zeichenketten oder auch andere Objekte

Klassendiagramm (UML)

Menge aller Klassen, welche alle Objekte eines Systems beschreiben, werden in einem Klassendiagramm festgehalten.

Wie entwerfen wir ein Klassendiagramm?

Domänenanalyse

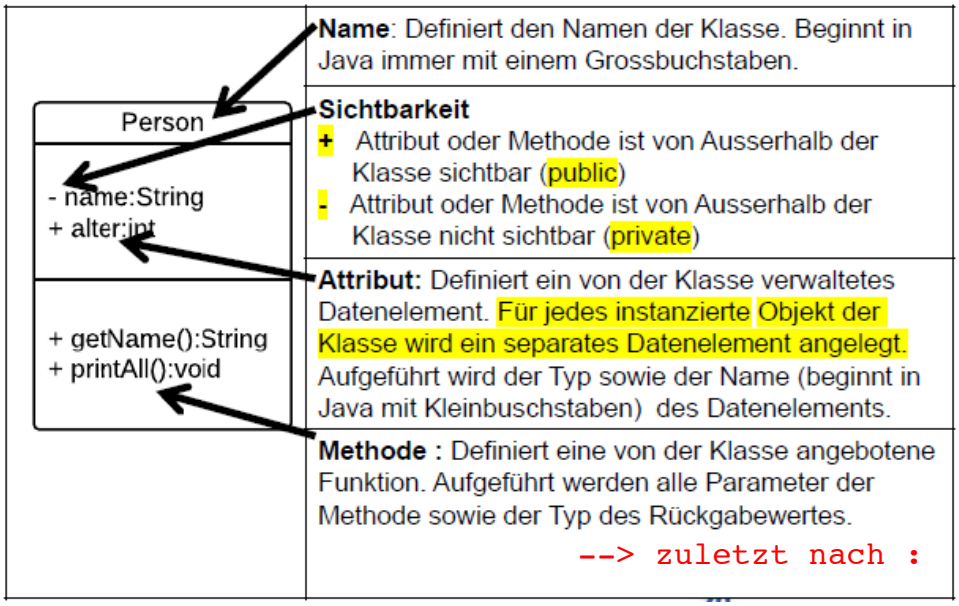
Entitäten: Menschen, Firmen

Eigenschaften: Name, Vorname

Beziehungen: Menschen arbeiten für Firmen

Applikationsdomäne Eine Applikationsdomäne umfasst verschiedene, zusammenhängende Funktionen eines Geschäftes. Eine Domäne ist der Ausschnitt aus der realen Welt, welche in der Applikation abgebildet wird.

Elemente



Name: Definiert den Namen der Klasse. Beginnt in Java immer mit einem Grossbuchstaben.

Sichtbarkeit
+ Attribut oder Methode ist von Ausserhalb der Klasse sichtbar (public)
- Attribut oder Methode ist von Ausserhalb der Klasse nicht sichtbar (private)

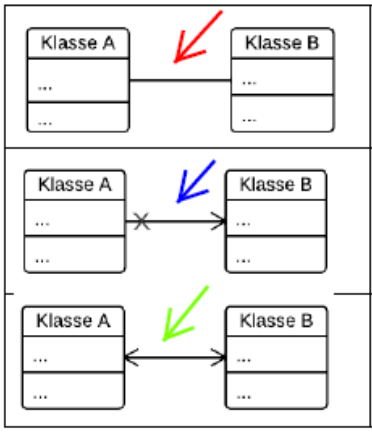
Attribut: Definiert ein von der Klasse verwaltetes Datenelement. Für jedes instanzierte Objekt der Klasse wird ein separates Datenelement angelegt. Aufgeführt wird der Typ sowie der Name (beginnt in Java mit Kleinbuschstaben) des Datenelements.

Methode : Definiert eine von der Klasse angebotene Funktion. Aufgeführt werden alle Parameter der Methode sowie der Typ des Rückgabewertes.

--> zuletzt nach :

Navigierbarkeit

Zeigt an ob sich die Klassen gegenseitig kennen oder nicht.



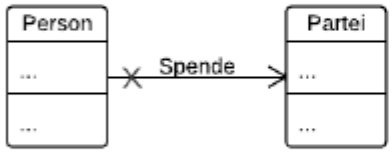
Unspezifizierte Navigierbarkeit: Obwohl die Navigierbarkeit nicht festgelegt ist, bedeuten diese Darstellung meist eine bidirektionale Navigierbarkeit.

Unidirektionale Navigierbarkeit: Klasse A kennt Klasse B, Klasse B kennt A jedoch nicht.

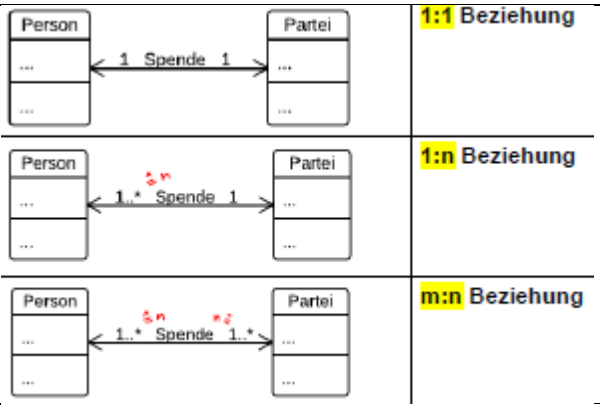
Bidirektionale Navigierbarkeit: Die Klassen kennen sich gegenseitig.

Anonyme Parteispende

Klassendiagramm	Implementation in Java

	`class Person`
	`{`
	`  String name;`
	`  Partei unterstuetzePartei;`
	`}`
	
	`class Partei` // Partei kennt die Spender nicht!
	`{`
	`  String parteiname;`
	`}`

Multiplizität (Kardinalität)

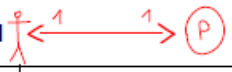


Person spendet nur einer Partei. Eine Partei hat ein Spender.

1 Partei hat mind. 1 Spender. Eine Person spendet nur einer Partei.

Mind. 1 Person hat mind. 1 Partei.

1:1 Beziehung - Bidirektional



Klasse Person	Klasse Partei

class Person
{
String name;
Partei meineParteiPartei;
}
	class Partei
	{
	String parteiname;
	Person spender;
	}

1:1 Beziehung – Unidirektional (Person zu Partei)

Klasse Person	Klasse Partei

class Person
{
String name;
Partei meineParteiPartei;
}
	class Partei
	{
	String parteiname;
	}

1:n Beziehung - Bidirektional

Klasse Person	Klasse Partei

class Person
{
String name;
Partei meineParteiPartei;
}
	class Partei
	{
	String parteiname;
	ArrayList<Person> alleSpender;
	}

1:n Beziehung – Unidirektional (identisch!)

Klasse Person	Klasse Partei

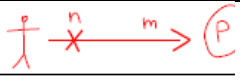
class Person
{
String name;
Partei meineParteiPartei;
}
	class Partei
	{
	String parteiname;
	}

m:n Beziehung - Bidirektional

Klasse Person	Klasse Partei

class Person
{
String name;
ArrayList<Partei> allePart;
}
	class Partei
	{
	String parteiname;
	ArrayList<Person> alleSpender;
	}

n:m Beziehung - Unidirektional



Klasse Person	Klasse Partei

class Person
{
String name;
ArrayList<Partei> allePart;
}
	class Partei
	{
	String parteiname;
	}

So können die Daten anhand folgender Datenmodelle beschrieben werden…

Unwichtig

Relational Model (RM)

Menschen(Name, Email, Work_For), Company(Name, …)

Entity Relationship (ER)

UML

Java

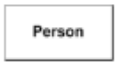
```
class Person {
   String name;
   String email;
   Company workFor;
}
class Company {
   String name;
   …
}
```

Allgemeines Verfahren zur Erstellung eines Datenmodells:

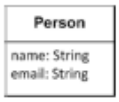
1. Entitäten
2. Eigenschaften von Entitäten
3. Behaviour (Methoden)
4. Beziehungen (uni- oder bidirektional)
5. Constraints (Kardinalität)
6. Klassifizierungen = Unterteilen, Gruppen bilden (Menge aller Instanzen durch 2 ArrayLists (Personen & ihre Freunde) unterteilen)

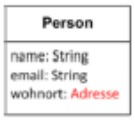
UML Klassendiagramme ?!? (UML)

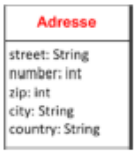
Klasse



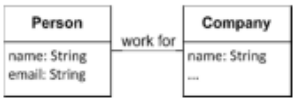
Klasseneigenschaften





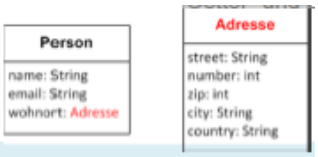


Beziehungen




```
class Person {

   String name;
   String email;
   Company workFor;

}
```

```
class Company {

   String name;
   private Set<Person> employees;

}
```




```
class Person {

   String name;
   String email;
   Address address;

}
```

```
class Address {

   String street;
   int umber;
   int zip;
   String city;

}
```

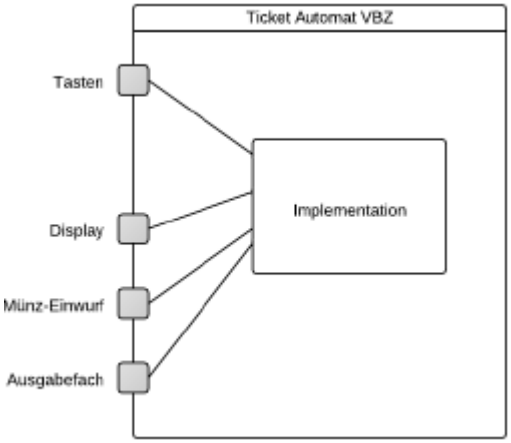
Kompositionsstrukturdiagramm (UML)

Elemente

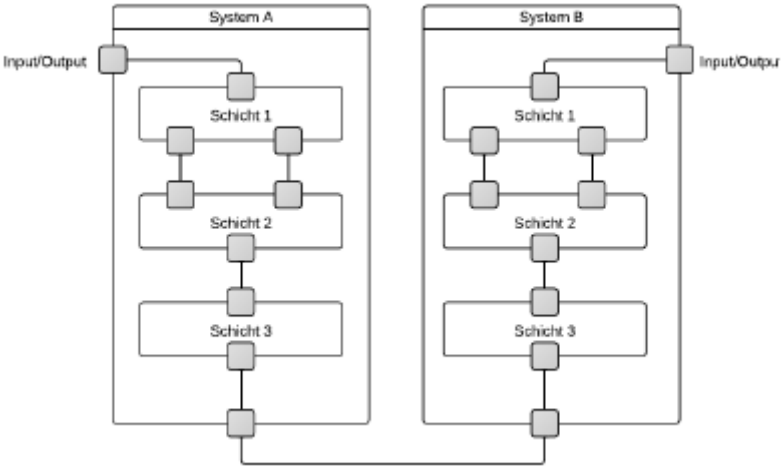
Classifier	**Classifier**: Der Classifier wird benutzt, um das dargestellte System zu benennen, oder um das System in mehrere Subsysteme zu unterteilen.
Part	**Part (Part):** Der Part ist eine Komponente innerhalb des Systems oder Subsystems.
Port	**Port (Port):** Der Port definiert eine Schnittstelle von einem Part oder einem Classifier.
Konnektor	**Konnektor (Ckonnektor) :** Der Konnektor kann die Elemente Classifier, Part und Port verbinden. Er wird benutzt um zu zeigen, dass die verbundenen Elemente miteinander kommunizieren. Der Konnektor kann Elemente mit unterschiedlichen Typen verbinden (z.B. ein Part mit einem Port).

Architekturen in Kompositionsstrukturdiagrammen erfasst

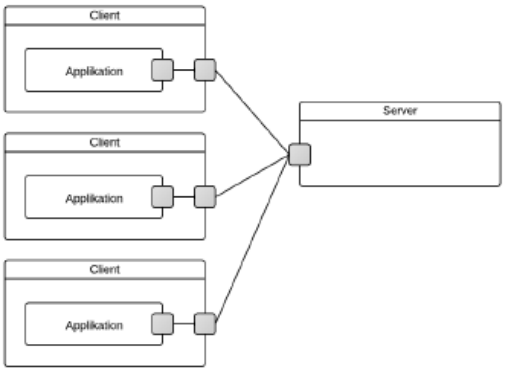
Monolitharchitektur



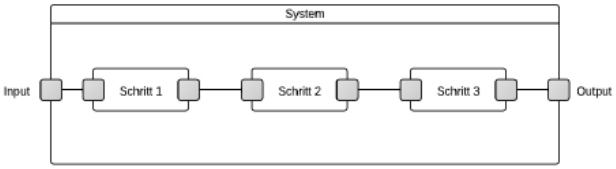
Schichtenarchitektur



2-Tier Architektur



Datenflussarchitektur (Batch-Sequential)



3. System Entwicklung

In einem System werden Operationen ausgeführt. Objekte werden kreiert, gelesen, gesucht, manipuliert und gelöscht (CRUD). Objekte können mit Klassen beschrieben werden.

Kreieren	Person damian = new Person();
Werte setzen	damian.email = „damian@marley.com“;
Werte aus Objekten lesen	system.out.println(damian.email);

Entwicklungs-Methode

Domänenanalyse (Entitäten, Eigenschaften, Beziehungen) → UML-Klassendiagramm

Abbildung auf Java-Klassen → Domänenklasse
Architektur (Komponenten) → Komponentenmodell
Abbildung auf Java Klassen (Komponenten) → Komponentenklasse

Entwicklungs-Methode: Prinzipien

- Abkapselung (private Attribute, getter-/setter-Methoden) Encapsulation
- Abstraktion (Methoden zur höheren Abstraktion, Konsistenzerhalt) ==> Vereinfachung gegen aussen (im Innern komplexer)

- Stepwise Refinement (zuerst Klassen mit leeren Methoden, dann dazu benötigte Eigenschaften und Methodenkörper, Aufteilung in weiteren (privaten) Methoden)
  Modularität
- Separation of Concerns = Unterteilung von Zwecken/Aspekten (Trennung der Modelle)

Domänenanalyse von UML nach Java


```
class Person {

   String name;
   String email;
   Company workFor;

}
class Company {

   String name;
   private Set<Person> employees;

}
```

Sequenzdiagramm

Festhalten von Interaktionen zwischen den Akteuren, Komponenten, Klassen und/oder Objekte.
Interaktion besteht aus Methodenaufrufe, Übergabeparameter und Rückgabewerte.

Akteur, Komponente, Klasse, Objekt

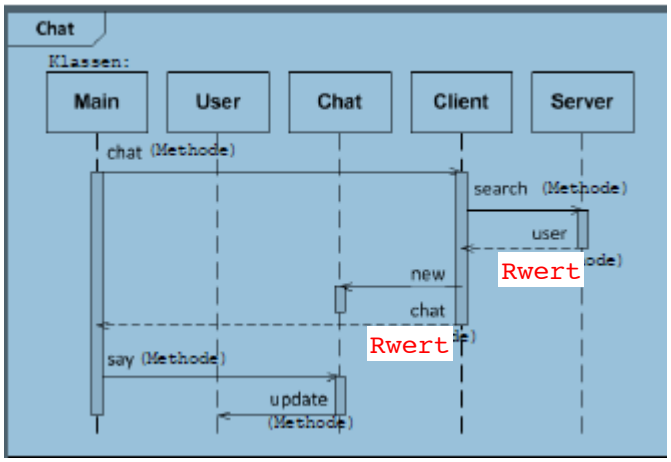
Unterstrichen: Instanz (Objekt) einer Klasse

Lebenslinie (Lifeline), Dauer einer Interaktion

Linien immer gleichlang

Interaktion

Methodenaufruf
< - - - - Rückgabewert



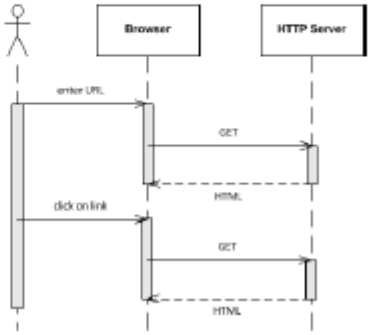
Akteur und Komponenten

Sequensdiagramm

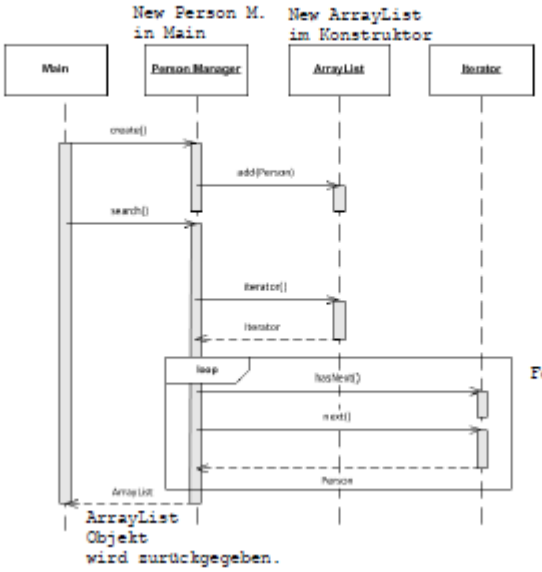
Aufgabe: Wie Komponenten benutzt werden und sich gegenseitig benutzen.
Vergleich mit Activity Diagramm

Nach enter URL
hat der Akteur nichts
zu tun bis "click on link"

Von oben nach unten:
zeitlicher Ablauf
(gestrichelte Linie
= Zeitlinie)
Balken = Aktion



Instanzen, Loop



Unterstrichen bedeutet:
Instanz (Objekt) einer Klasse.
Beispiel Instanz von Klasse
Person Manager

For oder While Schleife

ArrayList
Objekt
wird zurückgegeben.

Zusammenfassend:

1. **Requirements**
   Use Case Diagramme (z.B. Benutzer kreiert Person, Benutzer sucht Person) & Szenario
   Activity Diagramme (genauer Ablauf)
2. **Design**

Kompositionsstrukturdiagramm

Klassendiagramm aus Domänenanalyse (Person) (--> Beginn: Klassen- & Sequenzdiagramm)

Architektur (Client-Server)

3. **Implementierung**

Java Klassen aus Domänenmodell

Java Klassen aus Architekturdesign

- Klassendiagramm
- Sequenzdiagramm

Anhang Code zu den jeweiligen Architekturen

Client-Server

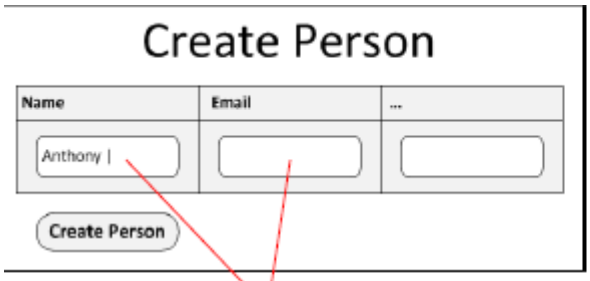

```
class Server {

   ArrayList<Person> persons;

   Person create(String name,…) {
      Person result = new Person();
      this.persons.add(result);
      return result;
   }

   …
}
```

```
class Client {

   void main(String[] args) {
       Server s = new Server();
       s.create("Damian", …);
       …
   }
}
```

CLIENT

1. Benutzerinterface (GUI) mit Formularfelder (Textfelder, Knöpfe).
Eingabe Attributwert – Knopfdruck – Neues Objekt mit den eingegebenen Attributen wird erstellt (Create Person).

2. Benutzerinterface. Eingabe Attributwert – Knopfdruck – bisher erstellte Objekte mit dem jeweiligen Attributwert anzeigen (Search Person)




```
Person p = new Person();
p.setName("Anthony …");
p.setEmail(…);
```

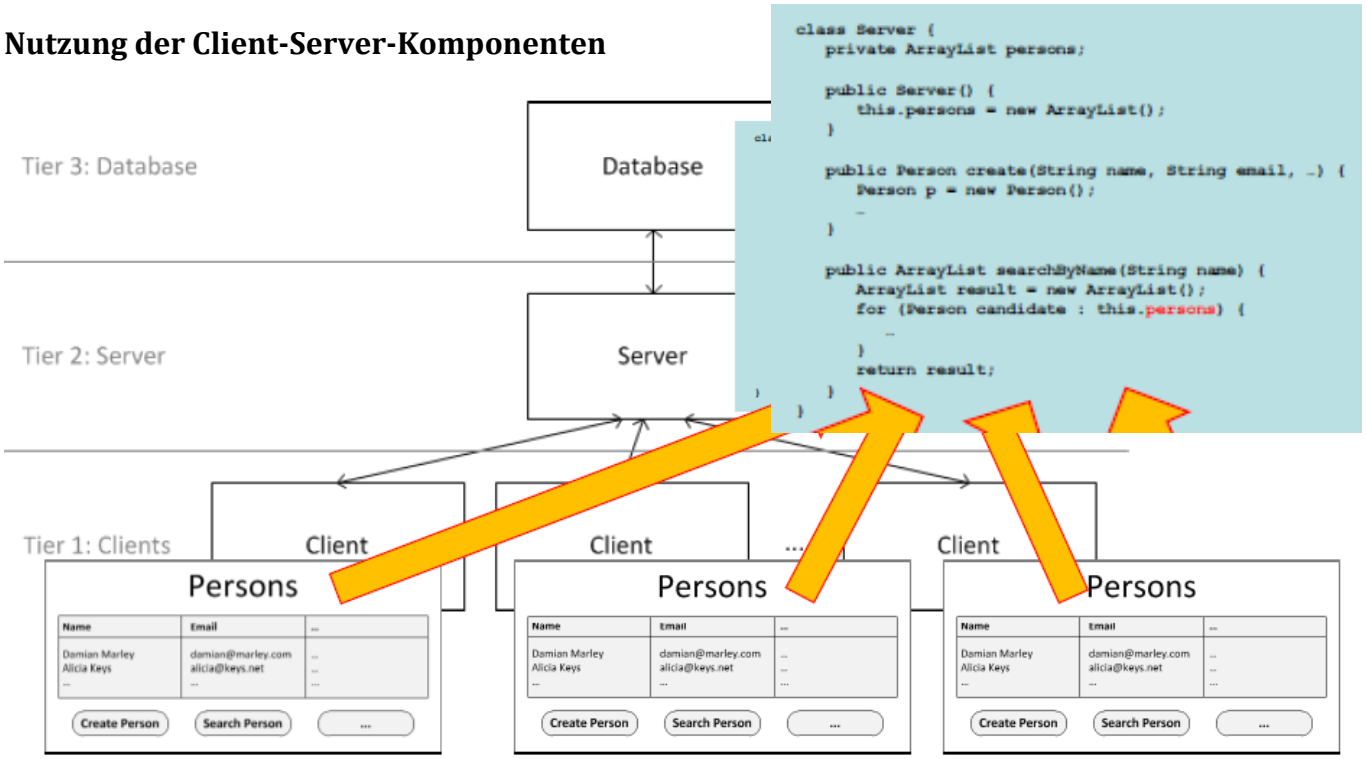

```
ArrayList result = new ArrayList();
for (Person candidate : persons) {
   if (candidate.getName() == …) {
      result.add(candidate);
   }
}
return result;
```

Eine neue Instanz sollte automatisch in eine ArrayList eingefügt werden, die von der Suchfunktion benutzt werden kann. Beide Code-Ausschnitte als Methoden einer Klasse zu haben, welche diese gemeinsame ArrayList als Attribut hat.

```
class Server {
   private ArrayList persons;

   public Server() {
      this.persons = new ArrayList();
   }

   public Person create(String name, String email, …) {
      Person p = new Person();
      p.setName(name);
      p.setEmail(email);
      p.set…(…);
      this.persons.add(p);
      return result;
   }

   public ArrayList searchByName(String name) {
      ArrayList result = new ArrayList();
      for (Person candidate : this.persons) {
         if (candidate.getName() == name) {
            result.add(candidate);
         }
      }
      return result;
   }
}
```

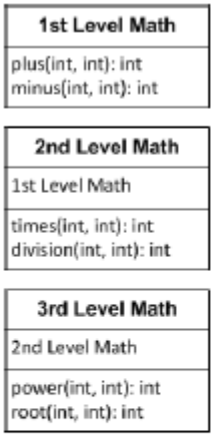
Server braucht eine Methode, die Attributwerte entgegennimmt und die gesuchten Instanzen der Domänenklasse zurückgibt.

Nutzung der Client-Server-Komponenten



Batch Sequential

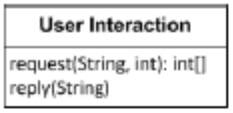
Komponenten führen Operationen aus. Wir haben eine Anwendung, für die diese Operationen mit bestimmten Werten und in einer bestimmten Reihenfolge ausgeführt werden sollen.




```
class FirstLevelMath {
   public int plus(int lhs, int rhs) {
      return lhs + rhs;
   }
   public int minus(int lhs, int rhs) {
      return lhs - rhs;
   }
}
```

```
class SecondLevelMath {
   private FirstLevelMath math;
   public SecondLevelMath() {
      this.math = new FirstLevelMath();
   }
   public int times(int lhs, int rhs) {
      int result = 0;
      for (int i = 0; i < rhs; i++) {
         result += lhs;
      }
      return result;
   }
   …
}
```

```
class ThirdLevelMath {
   …
}
```




```
class Main {
   public static void main(String[] args) {

      // instanziierungen
      UserInteraction ui = new UserInteraction();
      FirstLevelMath math1 = new FirstLevelMath();
      ThirdLevelMath math3 = new ThirdLevelMath();

      // user input
      int[] userArguments = ui.request("Please enter
                              the lengths of both
                               shorter sides of a
                                triangle", 2);
      // berechnung
      int aSquare = math3.power(userArguments[0], 2);
      int bSquare = math3.power(userArguments[1], 2);
      int sum = math1.plus(aSquare, bSquare);
      int root = math3.root(sum, 2);

      // output zum user
      ui.reply("The length of the third side is " +
                                                  root);
   }
}
```

Wir haben ein einfaches Batch Sequential bei der Implementierung von plus/divide und power/root.

In beiden Fällen wird eine Operation einer anderen Komponente in Serie aufgerufen.

Beachte: die 1st Level Komponente ist von der 3rd Level Komponente vollständig versteckt

(= Encapsulation, führt zu höherer Modularität, Austauschbarkeit, besserer Wartbarkeit und Erweiterbarkeit)

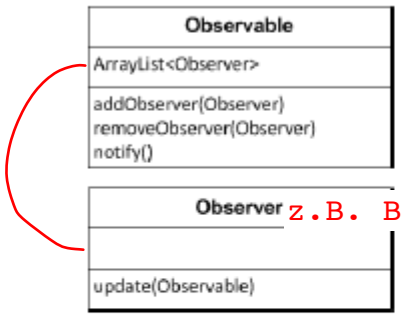
Der Code der Main-Klasse implementiert eine Batch Sequential Architektur wie wir es in den Basisarchitekturen gelernt hatten. Zusätzlich wird eine User Interaction Komponente benutzt.

Wir nehmen die UI-Komponente als gegeben, wissen nicht wie ihre Methoden implementiert sind, können diese Komponente trotzdem benutzen.

Ereignisorientierte Architektur

Observer-Observable WICHTIG

z.B. Bankkunde




```
class Observable {
   ArrayList<Observer> observers;

   public Observable() {
      this.observers = new ArrayList<Observer>();
   }

   publiv void addObserver(Observer observer) {
      this.observers.add(observer);
   }

   public void notify() {
      for (Observer current : this.observers) {
         current.update(this);
      }
   }
}
```

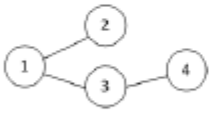
```
class Observer {

   public void update(Observable source) {
      // react on the event
      System.out.println("I got notified!");
   }
}
```

```
class Main{
   public static void main(String[] args) {
      Observable alicia = new Observable();
      Observer dadOfAlicia = new Observer();

      alicia.addObserver(dadOfAlicia);
      alicia.notify(); // yields output in System.out
   }
}
```

Peer-to-Peer Architektur

Beispiel

Peer
id: long neighbours: ArrayList<Peer>
addNeighbour(Peer) removeNeighbour(Peer)




```
class Peer {
   private long id;
   private ArrayList<Peer> neighbours;

   public Peer(long id) {
      this.id = id;
      this.neighbours = new ArrayList<Peer>();
   }
   public void addNeighbour(Peer peer) {
      if (this.neighbours.contains(peer)) {
         return;
      }
      this.neighbours.add(peer);
   }
   …
}
```

```
class Main {
   public static void main(String[] args) {
      Peer p1 = new Peer(1);
      Peer p2 = new Peer(2);
      Peer p3 = new Peer(3);
      Peer p4 = new Peer(4);

      p1.addNeighbour(p2);
      p1.addNeighbour(p3);
      p2.addNeighbour(p1);
      p2.addNeighbour(p3);
      p3.addNeighbour(p1);
      p3.addNeighbour(p2);
      p3.addNeighbour(p4);
      p4.addNeighbour(p3);
   }
}
```

Beispiel …

```
class Request {
   private String request;

   public Request(String request) {
      this.request = request;
   }
   … // getter/setter methods
}
```

Peer
id: long neighbours: ArrayList<Peer>
addNeighbour(Peer) removeNeighbour(Peer) broadcast(Request)

Request
request: String

```
class Peer {
   …
   public void broadcast(Request request) {
      for (Peer current : this.neighbours) {
         current.broadcast(request);
      }
   }
}
```

```
class Main {
   public static void main(String[] args) {
      …
      Request song = new Request("The Strokes - Heart in a Cage");
      p1.broadcast(song);
   }
}
```

Beispiel …

Peer
id: long neighbours: ArrayList<Peer>
addNeighbour(Peer) removeNeighbour(Peer) broadcast(Request)

Request
request: String path: ArrayList<Peer>

```
class Request {
   …
   private ArrayList<Peer> path;

   public Request(…) {
      …
      this.path = new ArrayList<Peer>();
   }

   public void addToPath(Peer hop) {
      this.path.add(hop);
   }

   public boolean isOnPath(Peer hop) {
      return this.path.contains(hop);
   }
}
```

```
class Peer {
   …
   public void broadcast(Request request) {
      request.addToPath(this);
      for (Peer current : this.neighbours) {
         if (request.isOnPath(current) {
            // do nothing (avoid cycles)
         } else {
            current.broadcast(request);
         }
      }
   }
}
```


```
Peer[1] rcvd req for 'Eliane Muller - Nothing Else Matters'
Peer[2] rcvd req for 'Eliane Muller - Nothing Else Matters'
Peer[3] rcvd req for 'Eliane Muller - Nothing Else Matters'
Peer[4] rcvd req for 'Eliane Muller - Nothing Else Matters'
```

Peer
id: long neighbours: ArrayList<Peer>
addNeighbour(Peer) removeNeighbour(Peer) broadcast(Request) receive(Response)

Request
request: String path: ArrayList<Peer>

Response
responder: Peer response: String path: ArrayList<Peer>

Sequenzdiagramm

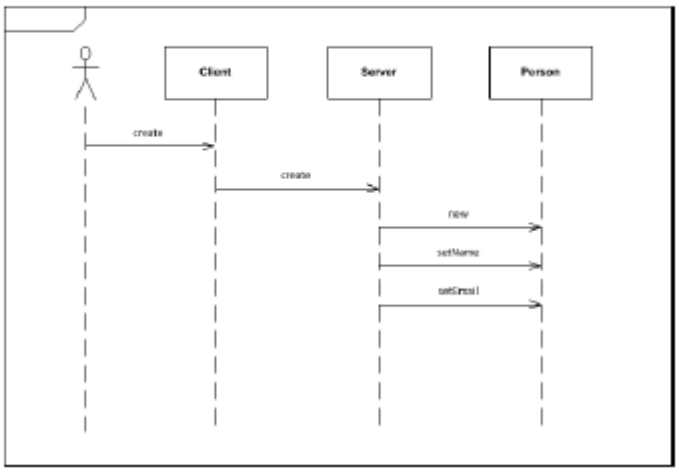
- Welche Klasse ruft Methoden von welchen anderen Klassen auf?
- Welche Methoden werden durch welche Klassen aufgerufen?
- Im Kontext welcher Methode wird eine andere Methode aufgerufen?
- Verknüpfung mit Use-Case-Diagramme / Activity-Diagramme

```
class Person {

   String name;
   String email;
   Company workFor;

}
```

```
class Server {

   ArrayList<Person> persons;

   Person create(String name,…) {
      Person result = new Person();
      this.persons.add(result);
      return result;
   }

   …
}
```

```
class Client {

   void main(String[] args) {
       Server s = new Server();
       s.create("Damian", …);
       …
   }
}
```

Sequenzdiagramme: Beispiel Client/Server



Spezialfall: Wert wird nicht überschrieben

Methode gibt primitiven Typ zurück.

```
TestClass1
{
    private int wert1;

    public TestClass1()
    {  this.wert1 = 123;  }

    public int getPrimitveType()
    {
       return wert1;
    }

    public void print()
    {  System.out.println("Wert1:"+wert1); }
}
```

Methode benutzen:

```
TestClass1 testClass = new TestClass1();
int tempPrimitive;
testClass.print();

tempPrimitive = testClass.getPrimitveType();
tempPrimitive = 456;
testClass.print();
```

Output:

```
Wert1:123
Wert1:123
```

Referenzen: Wert überschreiben

```
class WertClass{
private int wert2;
public WertClass()
{ this.wert2 = 123; }
public void setWert(int wert)
{ this.wert2 = wert; }
public void print()
{ System.out.println("Wert2:"+wert2); }}

class TestClass2{
private WertClass meinRtyp;
public TestClass2()
{ this.meinRtyp = new WertClass();}
public WertClass getRefereneType(){
return meinRtyp;}
public void print(){
meinRtyp.print(); }}
TestClass2 testClass = new TestClass2();
WertClass tempReference;
testClass.print();
tempReference = testClass.getReferenceType();
tempReference.setWert(456);
testClass.print();
```

Output:
Wert2: 123
Wert2: 456

Glossar

Separation of Concerns

Übersetzt mit Trennung der Belange bedeutet dieses Prinzip, dass man nicht mehrere Belange in einer Klasse zusammenfassen soll. Was sind Belange? Belange sind "komplett verschiedene" Zwecke.

Das Akronym **CRUD** [kɹʌd] umfasst die grundlegenden Datenbankoperationen **C**reate (Datensatz anlegen), **R**ead oder **R**etrieve (Datensatz lesen), **U**pdate (Datensatz aktualisieren), und **D**elete oder **D**estroy (Datensatz löschen).

High Level Languages (HLL) sind problemorientierte Programmiersprachen. Sie orientieren sich speziell an den zu lösenden Problemen und sind im Wesentlichen unabhängig von der Struktur der Computer (z. B. C++, PASCAL, Ada, FORTRAN, BASIC, COBO, Java).

Libaries (Programmbibliothek): Sammlung von Unterprogrammen, die Lösungswege für thematisch zusammengehörende Problemstellungen anbieten.

Skalierbarkeit: Nimmt der Ressourcenbedarf etwa im gleichen Mass(proportional) zu wie die Inputmenge, ist das System (die Software) noch gut skalierbar. Nimmt der Ressourcenbedarf überproportional zu, ist das System schlecht skalierbar (Mehr verkaufte/abgesetzte Menge, Grössere Daten am Input des Systems/Höhere Auflösung).

HCI (Human computer interaction)*:* Die Mensch-Computer-Interaktion (englisch Human-Computer Interaction, HCI) als Teilgebiet der Informatik beschäftigt sich mit der benutzergerechten Gestaltung von interaktiven Systemen und ihren Mensch-Maschine-Schnittstellen.

Adaptive Interfaces: Anpassungsfähige Schnittstellen

Gut dokumentierte Schnittstellen Dateiformat http: Hier kann man sicher sein dasses noch in weiter Zukunft verwendet wird. Unterstützung für diesen Standard muss hoch sein (also viele Leute sollten diese Schnittstelle verwenden). Wofür das alles: Damit zukünftige Updates für die Software, mit der Software übereinstimmen.

XML (Extensible Markup Language (engl. für „erweiterbare Auszeichnungssprache“)), abgekürzt. Metasprache, auf deren Basis durch strukturelle und inhaltliche Einschränkungen anwendungsspezifische Sprachen definiert werden. Ein XML-Dokument besteht aus Textzeichen, im einfachsten Fall in ASCII-Kodierung, und ist damit menschenlesbar. XML wird u. a. für den plattform- und implementationsunabhängigen Austausch von Daten zwischen Computersystemen eingesetzt, insbesondere über das Internet.

graphical user interface (GUI), auch human machine interface (HMI) genannt: Gestattet eine Interaktion der Komponente mit dem Benutzer durch eine grafische Benutzeroberfläche. Sie wird beispielsweise über die Maus oder den Bildschirm bedient.

command line interface (CLI): Insbesondere dann von Interesse, wenn Komponenten ohne Zutun des Benutzers durch das System aufgerufen werden sollen, beispielsweise, um in periodischen

Abständen immer wiederkehrende Aufgaben abzuarbeiten. Eine solche Schnittstelle wird durch Eingabe von Befehlen in eine Kommandozeile angesprochen.

data interface: Erlauben das Ein- und Auslesen von Daten der Komponente. Auf diese Schnittstelle wird programmintern zugegriffen.

application programming interface (API): Durch diese Schnittstelle ist es dem Programmierer und anderen Komponenten möglich, die von der Komponente angebotenen Funktionalitäten und Dienste durch Programmierbefehle anzusprechen. Soweit nicht anders angegeben wird mit Interface im Folgenden immer eine API gemeint sein.

graphical user interface (GUI), auch human machine interface (HMI) genannt: Gestattet eine Interaktion der Komponente mit dem Benutzer durch eine grafische Benutzeroberfläche. Sie wird beispielsweise über die Maus oder den Bildschirm bedient.

command line interface (CLI): Insbesondere dann von Interesse, wenn Komponenten ohne Zutun des Benutzers durch das System aufgerufen werden sollen, beispielsweise, um in periodischen Abständen immer wiederkehrende Aufgaben abzuarbeiten. Eine solche Schnittstelle wird durch Eingabe von Befehlen in eine Kommandozeile angesprochen.

data interface: Erlauben das Ein- und Auslesen von Daten der Komponente. Auf diese Schnittstelle wird programmintern zugegriffen.

application programming interface (API): Durch diese Schnittstelle ist es dem Programmierer und anderen Komponenten möglich, die von der Komponente angebotenen Funktionalitäten und Dienste durch Programmierbefehle anzusprechen. Soweit nicht anders angegeben wird mit Interface im Folgenden immer eine API gemeint sein.