In diesem Kapitel:
- *Design im Software Lebenszyklus*
- *Artifacts*
- *Beteiligte Personen*
- *Workflow*
  - *Architektur Design*
  - *Design eines Anwendungsfalles*
  - *Klassendesign*
  - *Subsystemdesign*
- *Zusammenfassung*

15
ROP / RUP Rational Objectory / Unified Process - Design

Nach der Analyse folgt die Konkretisierung in der Design Phase.

15.1. Einführung

Im Design definieren wir die Struktur unseres zukünftigen Systems - inklusive der Architektur, die hoffentlich möglichst lange erhalten bleiben wird - inklusive den nichtfunktionalen Anforderungen und anderer Einschränkungen.

Sie erinnern sich an die Phasen im ROP:
ROP kennt vier Phasen zur Darstellung der *zeitlichen* Entwicklung eines Software Systems:
- Startphase (inception phase)
- Entwurfsphase (elaboration phase)
- Konstruktionsphase (construction phase)
- Übergangsphase (transition phase)

Analyse tritt schwerpunktmässig in der Entwurfs-Phase auf. Analyse trägt ganz Wesentliches bei zu einer soliden Architektur und liefert ein vertieftes Verständnis der Anforderungen. Später in der Konstruktionsphase, auf der Basis einer stabilen Architektur, mit klaren Anforderungen, ändert sich der Fokus und Design und Implementation werden dominant.

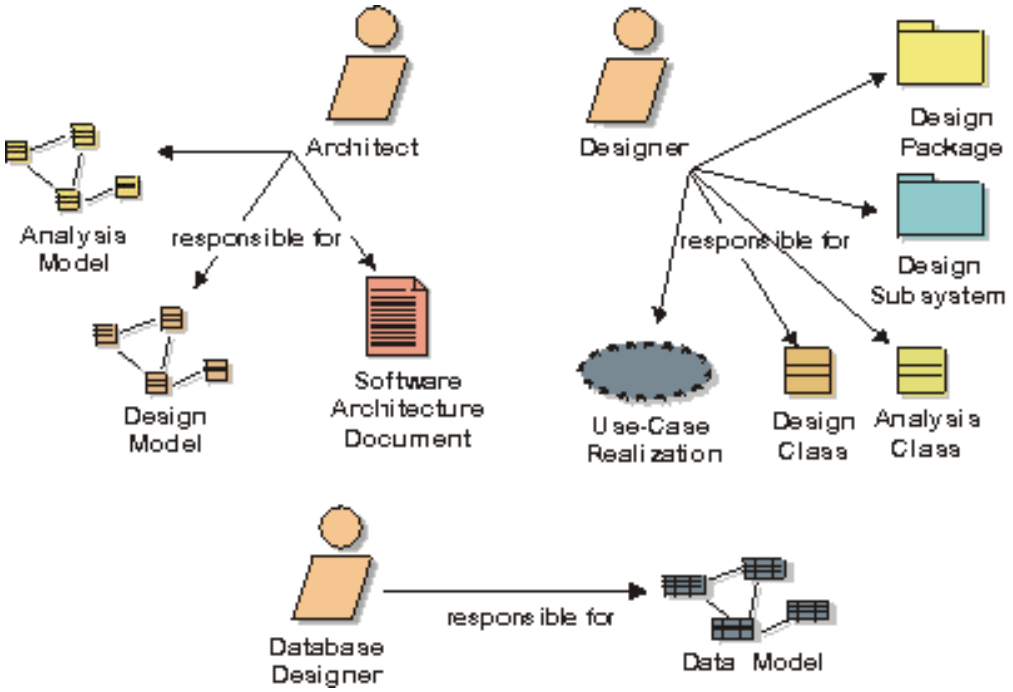
Als wesentlicher Input für den Design wird das Analyse Modell verwendet. Das Analyse Modell liefert die vertiefte Darstellung aller Anforderungen an das zu kreierende System.

Wesentliche Aufgaben im Design sind:
- Ein vertieftes Verständnis der nichtfunktionalen Anforderungen zu gewinnen;
  Einschränkungen, die sich aus der Wahl oder Vorgabe der Programmiersprache ergeben;
  Wiederverwendung bestehender Komponenten;
  Wahl oder Berücksichtigung des Betriebssystems;
  Entscheide bezüglich der Verteilung der Applikation;
  Entscheidungen bezüglich der parallelen Ausführung bestimmter Aufgaben (Threads);
  Entscheidungen betreffend der zu verwendenden Datenbank;
  Entscheidungen betreffend der Benutzerschnittstellen;
  Transaktionsmanagement Technologien;
  und ....
- Schaffen einer Basis für die anschliessende Implementierung:
  Festlegen der Aufgaben der einzelnen Subsysteme;

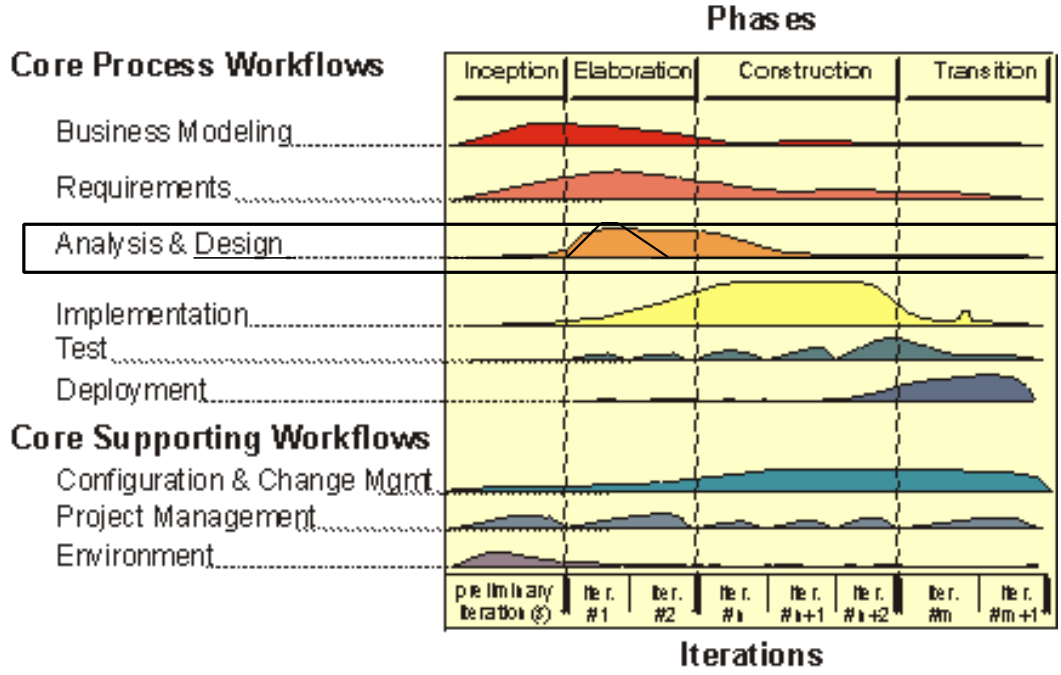
Beschreibung der Interfaces;
Spezifikation der Klassen

- Zerlegung des Systems in handlichere Teilprojekte, welche von unterschiedlichen Entwicklungsteams bearbeitet werden können, möglichst parallel.
  Hinweise für eine mögliche Zerlegung ergeben sich in der Regel bereits aus den Anforderungen, der Analyse der Anwendungsfälle und der Analyse des Systems.
- Festlegen der wichtigsten Schnittstellen zwischen den einzelnen Subsystemen.
  Dadurch wird die Synchronisation der Teilprojekte vereinfacht, da die Schnittstelle fix definiert ist und möglichst unveränderlich bleibt.
- Verwenden einer einheitlichen Notation, damit die einzelnen Mitarbeiter einander auch verstehen können und das Design verstehen und sich damit auseinander setzen können.
- Kreieren einer sinnvollen Abstraktion des Systems, so dass die Implementation sich natürlich daraus herleiten lässt (Definition einer Architektur, die auch implementierbar ist). Dadurch werden auch "Round-Trip" Engineering Techniken einsetzbar.

Die folgenden Abschnitte befassen sich mit dem WIE. Wie lassen sich die obigen Ziele erreichen?



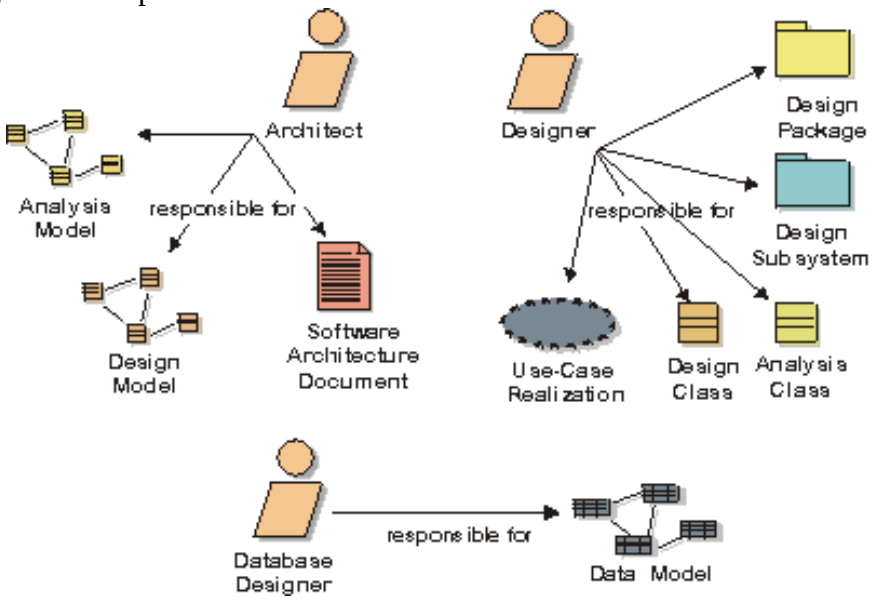
15.2. Design im Software Lebenszyklus



Zur Wiederholung : die einzelnen Phasen der zeitlichen Entwicklung

- Startphase (inception phase)
- Entwurfsphase (elaboration phase)
- Konstruktionsphase (construction phase)
- Übergangsphase (transition phase)

Design ist die Hauptaktivität in der Entwurfsphase und zu Beginn der Konstruktionsphase und soll eine stabile Architektur als Basis für die Implementation liefern. In der Implementationsphase liegt der Schwerpunkt bei der Konstruktion, also der Implementierung. Da Design und Implementation dicht beieinander sind, ist es das Ziel, das Design Modell so zu gestalten, dass es als Basis für die weitere Entwicklung des Systems dienen kann, auch als Basis für ein eventuelles "Round-Trip" Engineering, mit graphischer Unterstützung. Personen und Ergebnisse des Designs sind ähnlich wie in der Analyse, lediglich die Detaillierung und die Sprache ändern sich wesentlich.



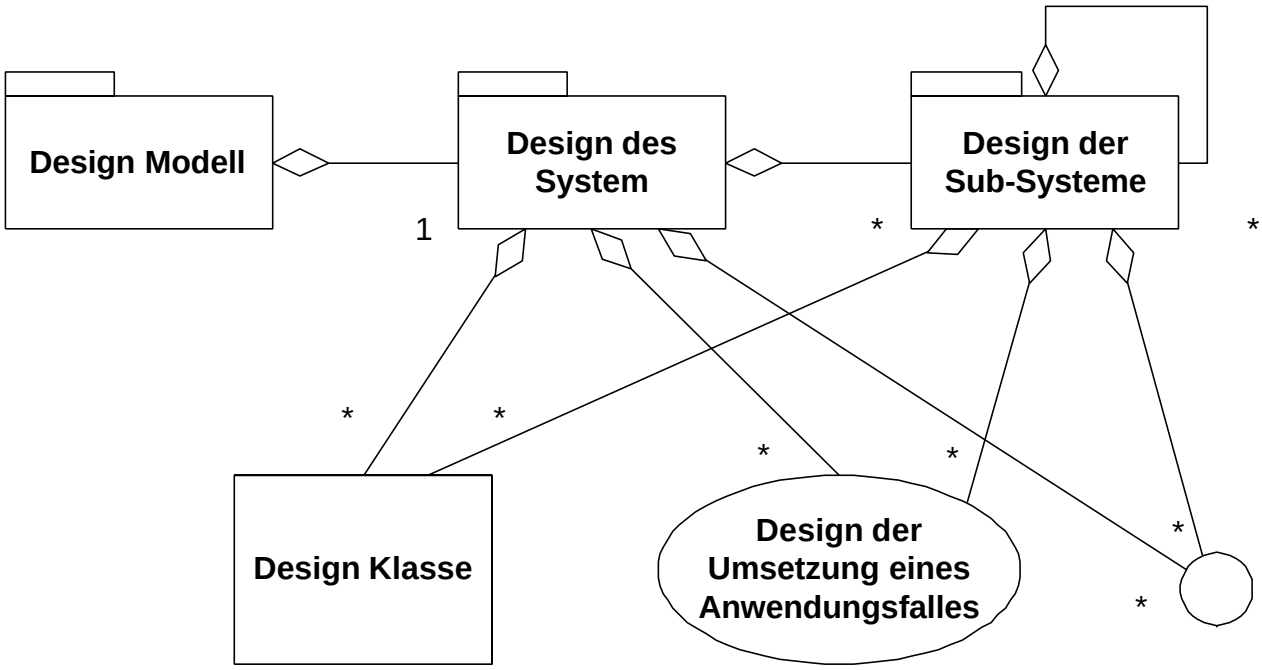
15.3. Artifakten

Artifakten sind Unterlagen, die auf Dauer gepflegt werden.

15.3.1. Artifakt : Design Modell

Das Design Modell ist ein Objekt Modell, welches die physikalische Realisierung der Anwendungsfälle beschreibt, unter Berücksichtigung der funktionalen und nichtfunktionalen Anforderungen und weiterer Einschränkungen, die sich für die Implementierung ergeben. Zudem dient das Design Modell als Abstraktion der System Implementation und die Basis dafür.

Die Artifakten des Designs lassen sich wie folgt zusammenfassen.



Das Design Modell stellt also eine Hierarchie dar.
Das Design Modell stellt ein System Modell dar, welches die oberste Ebene des Systems, das oberste Subsystem, beschreibt.

Die weitere Beschreibung des Systems besteht in einer rekursiven Beschreibung der Subsysteme, also einer Zerlegung des Systems in handhabbare Teile.

Design Subsysteme und Design Klassen stellen **Abstraktionen** von Subsystemen des Modells und der Komponenten der System Implementation dar. Ziel ist es, eine möglichst direkte Verbindung zwischen Design Modellen und deren Implementierung herzustellen.

Innerhalb des Design Modells werden die Anwendungsfälle durch Design Klassen und deren Objekte dargestellt. Dies bezeichnen wir als "Design der Umsetzung eines Anwendungsfalles" oder auch "*Anwendungsfall Realisierung - Design"*. Der Unterschied zwischen einem "Anwendungsfall Realisierung - Analyse" und einem "Anwendungsfall Realisierung - Design" besteht in den Objekten bzw. Klassen: im Falle des Designs werden

Design Klassen und Design Objekte und deren Wechselwirkung modelliert; im Falle der Analyse stammen Klassen und Objekte aus dem Analyse Modell, sind also weniger formal und benutzernahe formuliert.

15.3.2. Artifakt : Design Klassen

Design Klassen sind Abstraktionen der Klassen, die implementiert werden. Die Abstraktion ist implementationsnah in folgendem Sinne:

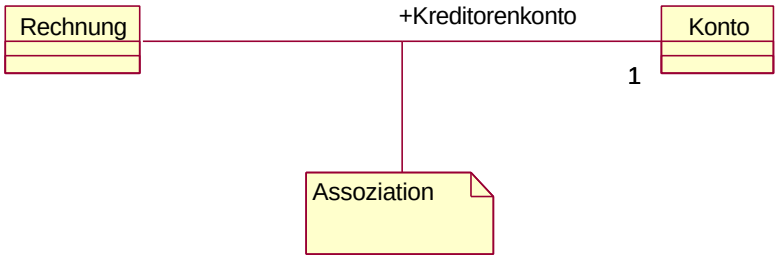
- Die Spezifikation der Design Klassen geschieht in der Programmiersprache, in der später implementiert wird. Das heisst, dass Parameter, Rollen, ... mit den Möglichkeiten der Programmiersprache ausgedrückt werden müssen.
- Die Sichtbarkeit der Attribute und Methoden einer Design Klasse spielt in der Regel eine grosse Rolle in der Implementierung. *Public, Protected, Private* ... werden typischerweise in Java und C++ eingesetzt und müssen dementsprechend im Design verwendet werden.
- Die Beziehungen zwischen Design Klassen muss mit den Beziehungen in der Implementierung korrespondieren. Aggregation und Assoziation müssen entsprechend den möglichen Sprachkonzepten verwendet werden.
- Die Methoden der Design Klasse und deren Implementierung müssen aufeinander abgestimmt sein. Im Design kann an Stelle der Formulierung mit Hilfe der Programmiersprache auch Pseudocode verwendet werden. Allerdings muss dies so geschehen, dass ein Übergang zur Implementation problemlos möglich ist.
- Falls möglich sollte die gleiche Person die Klassen designen UND implementieren.
- Eine Design Klasse kann auf einzelnen Details verzichten, wenn deren Umsetzung in der Implementierung offensichtlich ist.
- Stereotypen wie <<form>> und ähnliche, sollten im Design nur dann verwendet werden, falls sie auch implementiert werden können d.h. falls die verwendete Programmiersprache diese Konzepte auch unterstützt.
- Interfaces sollten im Design dann verwendet werden, falls die Programmiersprache oder übergeordnete Konzepte dies unterstützen. In Java sind Interfaces definiert, also werden sie auch sinnvollerweise im Design verwendet.
  Eine Variante besteht darin, IDL (Interface Definition Language) der OMG als Beschreibung der Interfaces zu verwenden. Dies impliziert nicht, dass deswegen auch CORBA (Common Object Request Broker), RMI (Remote Methode, Invocation) oder ähnliche Middleware Konzepte für die Objekt-Kommunikation eingesetzt werden.
- Die Definition unabhängiger Prozesse und Threads gehört als wesentliche Aufgabe zum Design. Hier gilt analog zu oben, dass die verfügbaren Konzepte der Programmier-Sprache und des Betriebssystems berücksichtigt werden müssen.

Kurzer Vergleich des Analyse und des Design Modells:

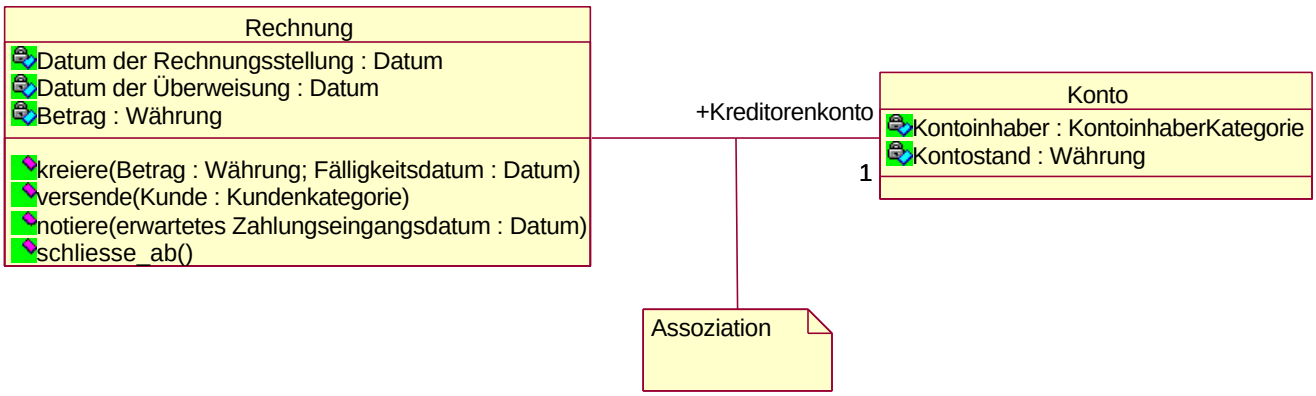
Analyse Modell	**Design Modell**
Konzeptionelles Modell Stellt eine Abstraktion des Systems dar und verzichtet auf Fragen der Implementation	Physikalisches Modell, weil es eine Vorlage darstellt für die Implementation
Generisch, auf mögliche Designs ausgerichtet	Spezifisch, auf die Implementation ausgerichtet
Enthält drei (conzeptionelle) Stereotypen für die Klassen : <<controll>>, <<entity>>, <<boundary>>	Stereotypen orientieren sich am (physikalisch) möglichen der in der Implementierung zu verwendenden Programmiersprache
Wenige formale Beschreibung	Eher formale Beschreibung
Kosten für die Entwicklung eines Analyse Modells halten sich in Grenzen (ca. 20% des Design Modells)	Kosten sind ein signifikanter Anteil des Projektbudgets (etwa das Fünffache der Analyse Kosten)
Enthält wenige Ebenen	Enthält möglichst die Ebenen, die in der Implementierung auch vorhanden sein werden
Dynamisches Modell; aber durch die Verwendung der Kollaborations-Diagramme eher informell	Dynamisches Modell; durch die Verwendung von Sequenz-Diagrammen eher formal.
Beschreibt den Design des Systems in groben Zügen, inklusive seiner Architektur	Beschreibt den Design des Systems inklusive der Architektur (als eine der Sichten)
Wird in Form von Workshops und ähnlichen Techniken erarbeitet	Wird eher mit Hilfe von "Round-Trip" Engineering Tools erstellt und visualisiert.
Wird eventuell nicht über den gesamten Software-Lebenszyklus gewartet	Sollte aktuell gehalten werden während des gesamten Software Lebenszyklus
Liefert Input, mit dessen Hilfe die Struktur des Systems festgelegt werden kann	Legt die Struktur des Systems fest, unter Berücksichtigung der Strukturen, die im Analyse Modell erarbeitet wurden

Beispiel: Design Klasse für eine Rechnung (Auszug)

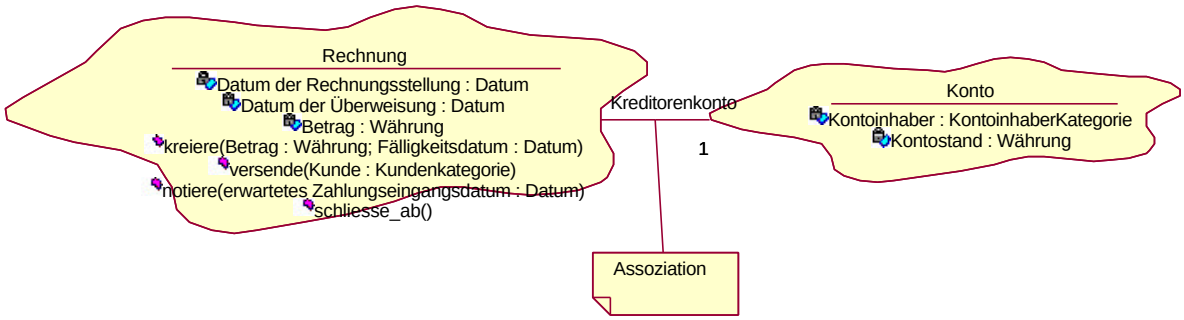
Das folgende Klassendiagramm zeigt die Beziehung zwischen einer Rechnung und einem Konto. Das Modell ergibt sich auf einer abstrakten Stufe direkt aus dem Datenmodell (eine Rechnung betrifft genau ein Konto; die andere Richtung lassen wir der Einfachheit halber weg : einem Konto werden mehrere Rechnungen gutgeschrieben (Lieferantenseite), bzw. belastet (Kundenseite)).



Zusätzlich zur Datenstruktur werden auch einige Methoden spezifiziert (der Klasse und deren Instanzen, den Objekten).



Das gleiche Diagramm in der überholten Booch Notation der 80er Jahre:

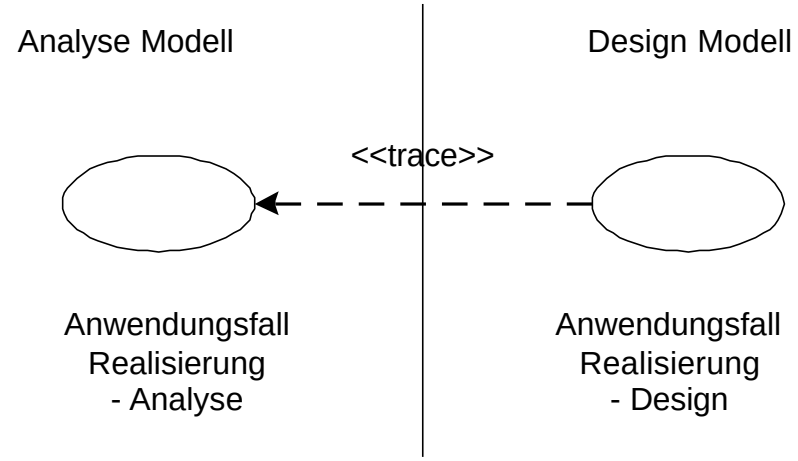


15.3.3. Artifakt : Anwendugsfall Realisierungs – Design

Anwendungsfall Realisierungs-Design als Teil des Design Modells, wie ein spezifischer Anwendungsfall umgesetzt wird, ausgedrückt mit Hilfe von Design Klassen und deren Objekte.

Zwischen der *Anwendungsfall Realisierungs-Analyse* (des Analyse Modells) und dem *Anwendungsfall Realisierungs-Design* (des Design Modells) besteht ein direkter Zusammenhang. Dadurch besteht auch eine (indirekte) Beziehung zum Anwendungsfall Modell über das Analyse Modell des Anwendungsfalles.

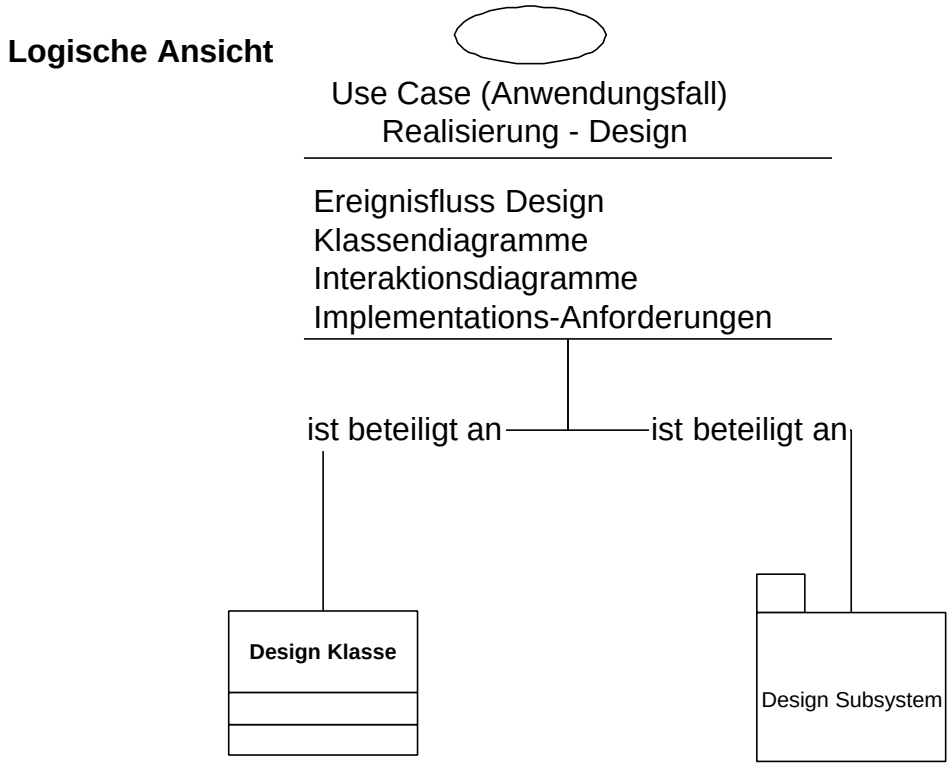
Falls das Analyse Modell im Laufe des Lebenszyklus der Software nicht weiter gepflegt wird, dann muss auf jeden Fall die Beziehung des Design Modells zur Anforderung festgehalten werden. Das Anwendungsfall Modell des Designs ist hierfür die Basis.



Eine Realisierung eines Anwendungsfalles besteht aus:

- textlicher Beschreibung des Ereignisflusses
- Klassendiagrammen, bestehend aus Design-Klassen und deren Beziehungen untereinander
- Interaktions-Diagrammen, aus denen die Szenarios ersichtlich sind, als Interaktion der verschiedenen Objekte.
- unter Umständen werden auch Subsysteme und deren Grenzen dargestellt

Die Realisierung eines Anwendungsfalles beschreibt die physische Realisierung des Anwendungsfalles! Aber wie bei den Design Klassen können einzelne Implementations-Details auf später verschoben werden.



15.3.3.1. Klassen Diagramme

Eine Design Klasse und ihre Objekte und damit auch ihre Subsysteme, sind in der Regel Teil einer Realisierung eines Anwendungsfalles. Unter Umständen sind einzelne Methoden und Attribute nur für bestimmte Implementierungen und bestimmte Anwendungsfälle relevant.

Wir können, der besserer Sichtbarkeit wegen, beim Zeichnen bestimmte Klassen speziell kennzeichnen (fett, rot, ...), um deren Zusammengehörigkeit zu unterstreichen.

15.3.3.2. Interaktions-Diagramme

Die Abläufe beginnen in der Regel in allen Anwendungsfällen gleich: der Aktor sendet eine Meldung an das System und erhält vermutlich eine Antwort. Innerhalb des Systems muss also ein Objekt, oder abstrakter eine Klasse, diese Meldung empfangen und verarbeiten können. Das Objekt kann zur Bearbeitung dieser Meldung auch andere Objekte beiziehen. Die wechselwirkenden Objekte machen den Anwendungsfall schlussendlich aus. In der Analyse haben wir bereits informelle Hilfsmittel, wie etwa die Kollaborations-Diagramme kennen gelernt, um dieses Zusammenspiel zu beschreiben. Im Design sind wir formaler : das entsprechende Hilfsmittel sind die Sequence oder Sequenz Diagramme. Diese stellen den Ablauf der Kommunikation im Detail dar, inklusive dem zeitlichen Aspekt.

Sequenz Diagramme können auch nur Auszüge, Subsysteme beschreiben. Dadurch bleiben sie übersichtlicher und lesbarer. Wenn wir auch noch die Anwendungsfälle den Subsystemen zuordnen, dann gelangen wir zu einer übersichtlichen, vollständigen Darstellung unseres Systems.

Wir werden später in den Beispielen die Details der Sequenz Diagramme noch erklären.

15.3.3.3. Ereignisfluss-Design

Interaktions-Diagramme sind oft schwer zu lesen. Als Ausweg bietet sich eine Beschreibung in Textform an, aber in der Regel nicht auf allen Ebenen, da Text nicht so präzise kompakt ausdrückt was aus einem Diagramm heraus zu lesen ist.

Bei der Beschreibung der Dynamik mit Hilfe von Text ist darauf zu achten, dass die Sprache angepasst ist, das heisst, es müssen Objekte, Methoden ... verwendet werden, da wir das System mit deren Hilfe realisieren möchten. Objekt Attribute auf der andern Seite sind Details, auf die man bei der Textbeschreibung verzichten sollte, da sie zuschnell ändern und der Text kaum sehr wartungsfreundlich sein wird.

Text kann also in der Regel nur als Ergänzung verwendet werden. Es ist allerdings zu beachten:

- Ereignisfluss-Design Dokumente beschreiben in der Regel einen oder sogar mehrere Anwendungsfall bzw. Anwendungsfälle. Ein Sequenz-Diagramm beschreibt dagegen oft nur einen Ausschnitt aus einem Anwendungsfall. Der Text erlaubt es somit das Zusammenspiel mehrerer Diagramme zu beschreiben.
- Labels (Zeitmarken, Zeitdauern) der Sequenz-Diagramme beschreiben lokale Zeitbeziehungen. Falls viele Labels verwendet werden müssten, wird das Diagramm unübersichtlich. Die Textbeschreibung kann hier helfen, das Chaos zu reduzieren.

Wenn man beide zusammen verwendet, Diagramme und Text, sollten sich beide also sinnvoll ergänzen.

Beispiele für Textbeschreibungen und Sequenz-Diagramme lernen wir in Abschnitt 15.5.2.2 kennen.

15.3.3.4. Implementations-Anforderungen

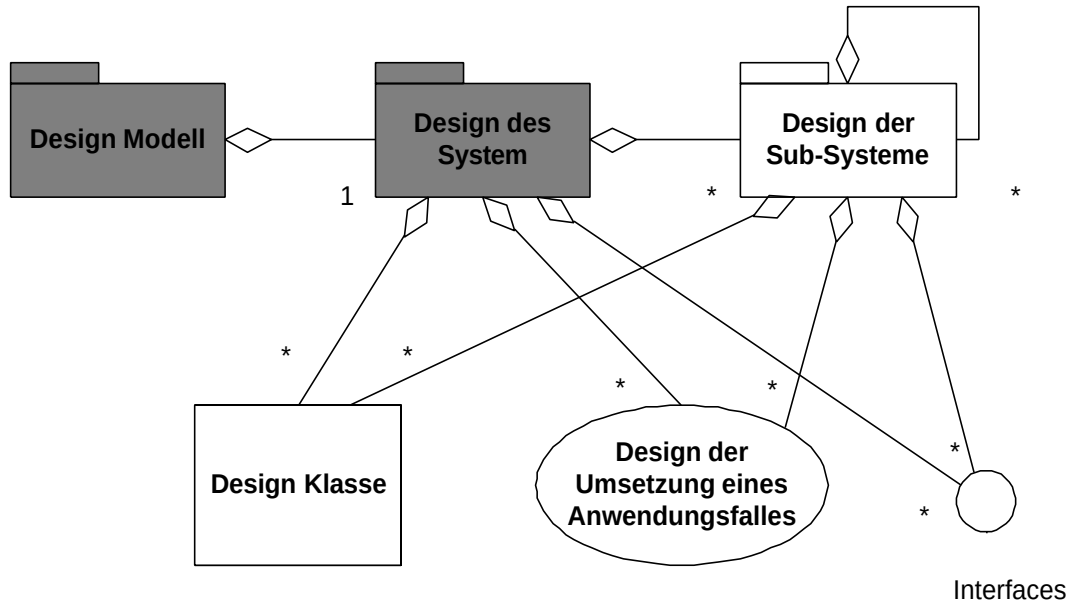
Implementations-Anforderungen halten die Anforderungen, die im Design sichtbar werden, aber die Implementation betreffen, fest.

Typischerweise handelt es sich um Weiterschreiben der (textlich) erfassten Zusatzanforderungen an einen Anwendungsfall.

Beispiele dafür lernen wir im Abschnitt 15.5.2.5 kennen.

15.3.4. Artifakt : Design Subsystem

Design Subsysteme ermöglichen es, das System in überschaubare Teile zu zerlegen, die wir möglichst unabhängig implementieren möchten. Ein Design Subsystem kann selber wieder aus weiteren Subsystemen bestehen. Sicher besteht es aber aus Design Klassen, Anwendungsfall Realisierungs-Designs und Interfaces.



Subsysteme sollten kohäsiv, stark zusammen hängend sein. Subsysteme untereinander sollten lose gekoppelt sein. Dadurch lassen sich Änderungen lokal am Besten eingrenzen.

Design Subsysteme sollten folgende Charakteristika haben:

- Subsysteme stellen Design Schwerpunkte dar. Dies ermöglicht es anschliessend recht unabhängig einzelne Subsystem parallel zu bearbeiten und zu implementieren.
- oft besteht ein direkter Zusammenhang zwischen den obersten zwei Layern des System-Designs und den Anwendungsfällen. Subsysteme lassen sich also oft auf Grund der Anwendungsfälle bereits gruppieren.
- Subsysteme präsentieren grob bereits die Komponenten, aus denen das System in Sinne der Komponenten basierten Software-Entwicklung erstellt werden kann. Komponenten, die später Implementations-Klassen werden, sind in der Regel im Design bereits erkennbar.
- Subsysteme können auch wieder verwendbare Komponenten beschreiben, bzw. Hinweise auf solche Komponenten und Klassen liefern.
- ein Subsystem kann ein bereits bestehendes System darstellen, die mit Hilfe von "Wrappern" unserem neuen System zugänglich gemacht werden.

15.3.4.1. Service Subsysteme

Service Subsysteme beschreiben bestimmte Funktionen, die wir bereits in der Analyse als Service Pakete identifiziert haben. Die Identifikation der Service Subsysteme basiert somit auf den Service Paketen.

Service Pakete trifft man in der Regel auf Layer 1 und Layer 2 (applikations-spezifisch und applikations-übergreifend). Die Service Subsysteme unterscheiden sich jedoch von Service Paketen in folgenden Aspekten:

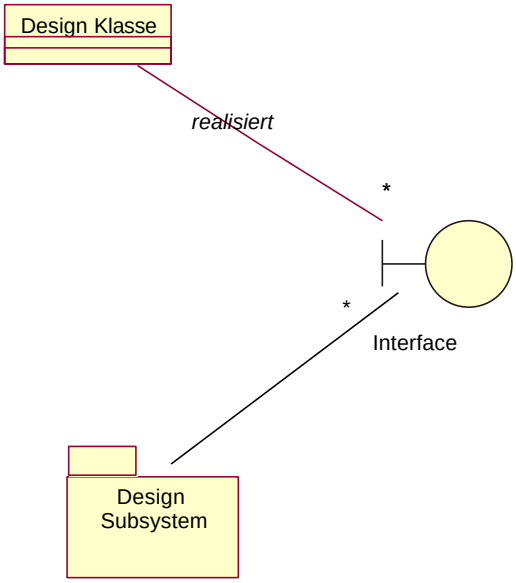
- Service Subsysteme müssen ihre Dienste in Form von Schnittstellen und Methoden andern zur Verfügung stellen.
- Service Subsysteme enthalten Design Klassen, nicht Analyse Klassen wie die Service Pakete.
  Service Subsysteme enthalten in der Regel mehr Klassen als Service Packages. Service Subsysteme müssen eventuell auch noch weiter zerlegt werden, damit sie handhabbar werden.
- ein Service Subsystem wird oft als ausführbares Programm realisiert.

Beispiele besprechen wir in Abschnitt 15.5.1.2.1

Service Subsysteme fehlen in unserem Artifakten Diagramm, sie stellen einen Spezialfall der Design Subsysteme dar, wir können (damit die Diagramme lesbarer werden) einen speziellen Stereotyp dafür einführen.

15.3.5. Artifakt : Interface

Mit Hilfe von Interfaces werden die Methoden der Design Klassen bekannt gegeben.



Die Design Klasse, die das Interface implementiert, muss auch die entsprechenden Methoden zur Verfügung stellen, analog zu den Interfaces in Java.

Interfaces erlauben es, in Systemen die Spezifikation von der Implementation zu trennen. Implementationen des Interfaces werden austauschbar, zum Beispiel beim Verfügbarwerden neuer, schnellerer Algorithmen.

Interfaces dienen auch dem Design und dem Implementierungs-Team möglichst unabhängig voneinander, aber doch kontrolliert, einzelne Systemteile zu designen und zu implementieren: alles was fix ist, sind die Interfaces, sinnvoll beschrieben.

In Hinblick auf die Implementierung mit Hilfe einer Objekt Orientierten Programmiersprache kann aus einer Interface Beschreibung ein IDL (Interface Definition Language) Programm werden, welches in CORBA, RPC, RMI oder ähnlichen Konzepten für verteilte Objekte einsetzbar ist.

Beispiele finden Sie in Abschnitt 15.5.1.24.

15.3.6. Artifakt : Architekturbeschreibung (Sicht des Design Modells)

Die Architekturbeschreibung enthält eine **Architektursicht des Design Modells.** Daraus müssen die architektonisch relevanten Artifakten erkennbar werden.

Typischerweise sind in einer Architekturbeschreibung folgende Artifakten enthalten:

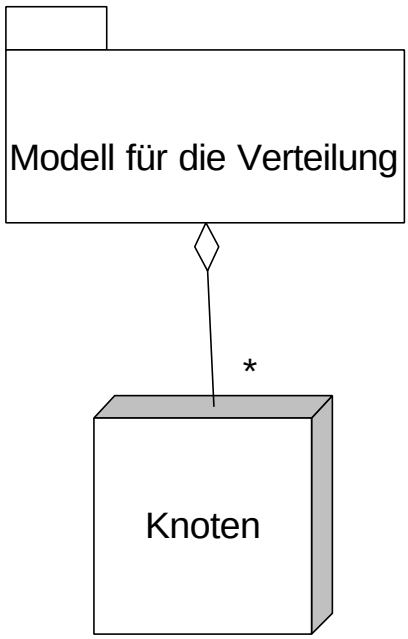
- die Dekomposition des Design Modells in Subsysteme, deren Interfaces und deren Abhängigkeiten untereinander. Diese Zerlegung ist sehr wesentlich für die Architektur des Systems generell und der Subsysteme im Speziellen. Daraus wird die wesentliche Struktur des Systems, seine Architektur, sichtbar.
- Schlüssel-Design-Klassen, jene die direkten Einfluss auf die Architektur haben und in Beziehung mit vielen andern Klassen stehen.
- Anwendungsfall Realisierungs-Designs, welche kritische und wichtige Funktionen enthalten.

15.3.7. Artifakt : Verteilungs-Modell (Deployment Modell)

Das Verteilungsmodell ist ein Objektmodell, welches beschreibt, wie das System physisch verteilt wird und wie die Funktionen auf einzelne Knoten eines Netzwerkes verteilt werden.

Dieses Modell hat einen sehr wesentlichen Einfluss auf die konkrete Verteilung der Applikationen und auf deren Design.

Abstrakt lässt sich das Verteilungs-Modell wie folgt formalisieren:



Die Applikation wird auf mehrere „Knoten“, bei Booch „Prozessoren“ genannt, verteilt.

Es geht also um die physische Verteilung der Applikation!
Folgende Punkte sind wesentlich beim Distributions-Modell:

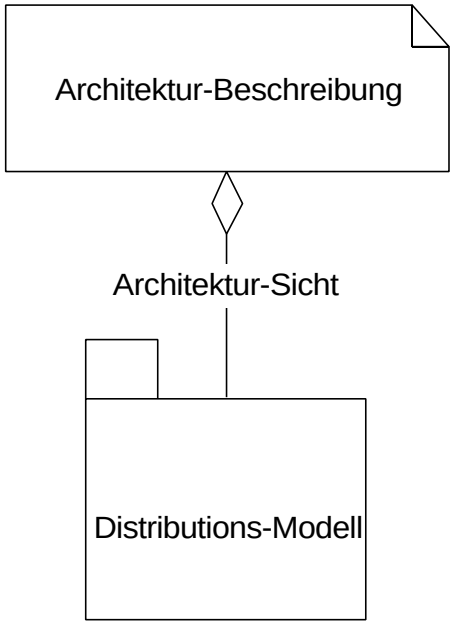
- Jeder Knoten repäsentiert Rechnerresourcen, einen Prozessor oder ähnliche Hardware
- Knoten, die miteinander verbunden sind, werden auch im realen Modell mit einander kommunizieren. Die Verbindung stellt eine reale Kommunikation der „Rechner“ dar, im Sinne eines *Internets*, *Intranets* oder eines *Bus*
- Das Distributions-Modell kann auch verschiedene Aspekte enthalten, wie die Verteilung der Daten, die Verteilung der Applikationen, ...
- Die Funktionalität eines Knotens ergibt sich aus dessen Beschreibung, ist also nicht unbedingt Teil des Distributions-Modells
- Das Distributionsmodell stellt auch ein Bindeglied dar, zwischen der Hardware, der Software und des gesamten Systemmodells.

In Abschnitt 15.5.1.1 finden Sie Beispiele für die Verteilung von Applikationen.

15.3.8. Artifakt : Architekturbeschreibung (Sicht der Applikations-Verteilung)

Die Architekturbeschreibung muss auch eine **Architektur des Distributions-Modell** enthalten, aus dem die relevanten Artifakte hervor gehen.

Abstrakt lässt sich die Architekturbeschreibung, aus Sicht des Distributions-Modells, wie folgt schematisch beschreiben:



Da die Applikations-Verteilung ein wesentliches Merkmal jeder Applikation ist, muss dieser Aspekt in einer Architektur-Beschreibung zwingend enthalten sein.

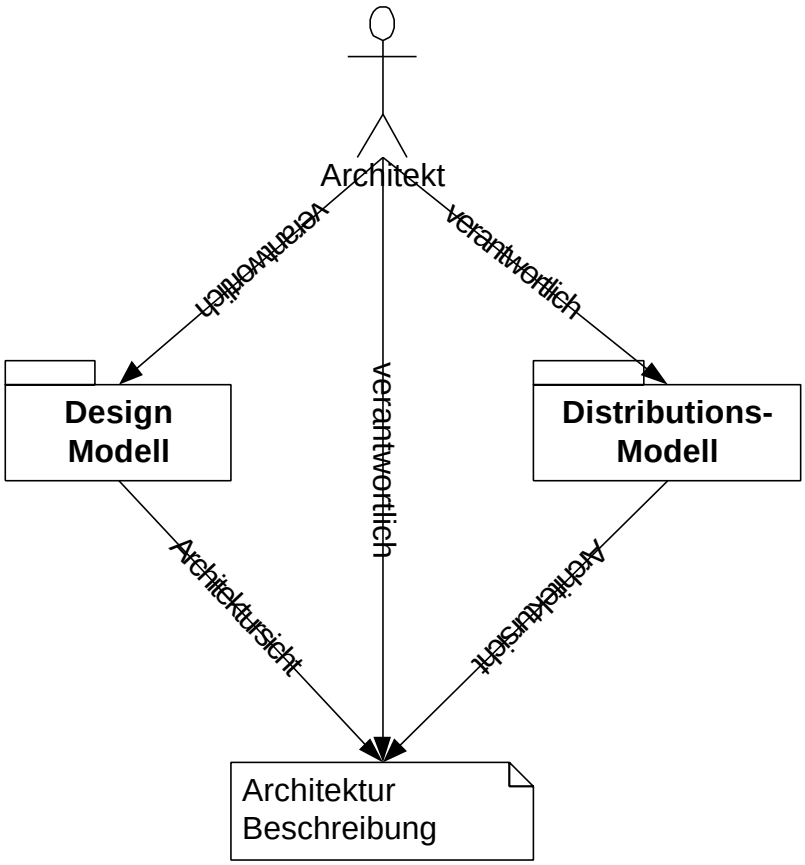
In Abschnitt 15.5.1.1 finden Sie Beispiele für eine Distributions-Architektur.

15.4. Mitarbeiter / Beteiligte

Wer ist am Design beteiligt? Eigentlich die gleichen Mitarbeiter wie in der Analyse, da diese sich mit Hilfe der Analyse ein vertieftes Bild des Systems verschafft haben.

15.4.1. Mitarbeiter : der Architekt

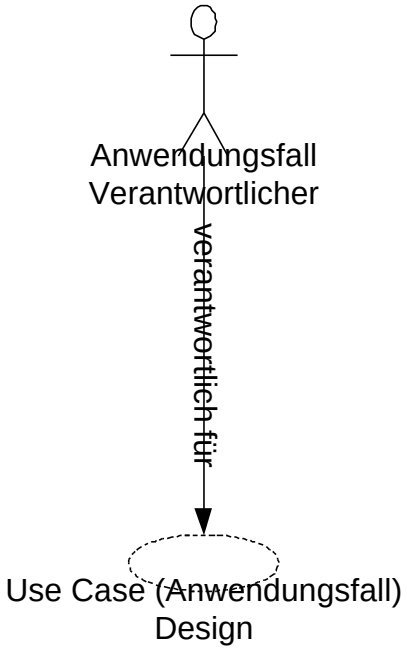
Im Design ist der Architekt verantwortlich für die Integration des Design und Distributions-Modells, damit sichergestellt ist, dass das Modell als Ganzes korrekt und konsistent und lesbar ist. Bei komplexen Systemen kann jemand für jeweils ein Submodell verantwortlich gemacht werden.



Ein Modell gilt dann als korrekt, falls es die Funktionalität, die in den Use Cases verlangt wurde, implementiert. Das impliziert auch, dass das Analyse Modell und allfällige zusätzliche Anforderungen, die in Textdokumenten berücksichtigt wurden, im Design Modell berücksichtigt werden.

Der Architekt trägt die Verantwortung für das Design und das Distributionsmodell. Er muss die unterschiedlichen Artifakten jedoch nur bedingt über den ganzen Lebenszyklus pflegen und unterhalten. Dafür sind auch der Anwendungsfall-Verantwortliche und der Komponenten Verantwortlichen zuständig.

15.4.2. Mitarbeiter : Der Anwendungsfall – Verantwortliche



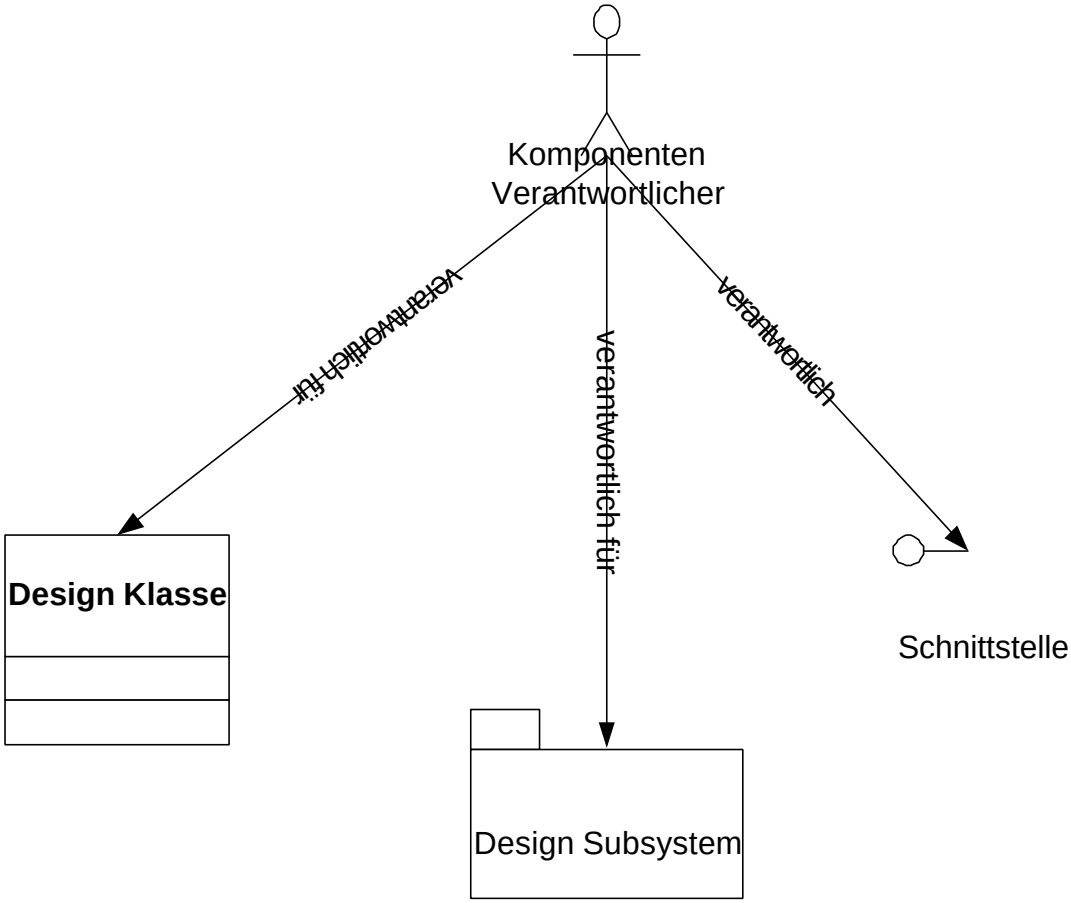
Der Verantwortliche für die Anwendungsfälle ist verantwortlich für die Integrität der Umsetzung eines oder mehrerer Anwendungsfälle und deren Realisierung. Designs eines Anwendungsfalles müssen den ursprünglichen Anwendungsfall wieder geben, dessen Funktionalität implementieren.

Alle vorhandenen Diagramme aus der Anwendungsfall-Analyse müssen berücksichtigt werden.

Der Anwendungsfall-Verantwortliche ist jedoch nicht für Design Klassen, Subsysteme, Schnittstellen und den Beziehungen zwischen Klassen und Objekten zuständig. Dafür trägt der Komponenten Designer die Verantwortung.

15.4.3. Mitarbeiter : der Komponenten Verantwortliche

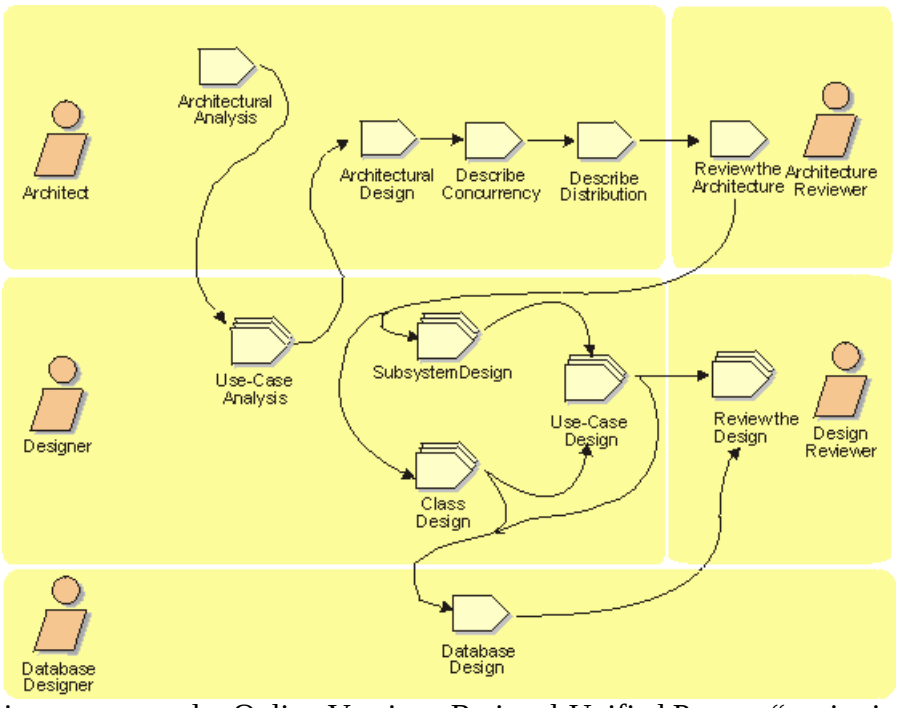
Die Person oder das Team, welches für die Definition und die Wartung der Methoden, Attribute, Beziehungen und Implementation der Benutzer-Anforderungen zuständig ist, muss auch dafür sorgen, dass alle Anforderungen der Anwendungsfälle in den Design Klassen berücksichtigt werden.



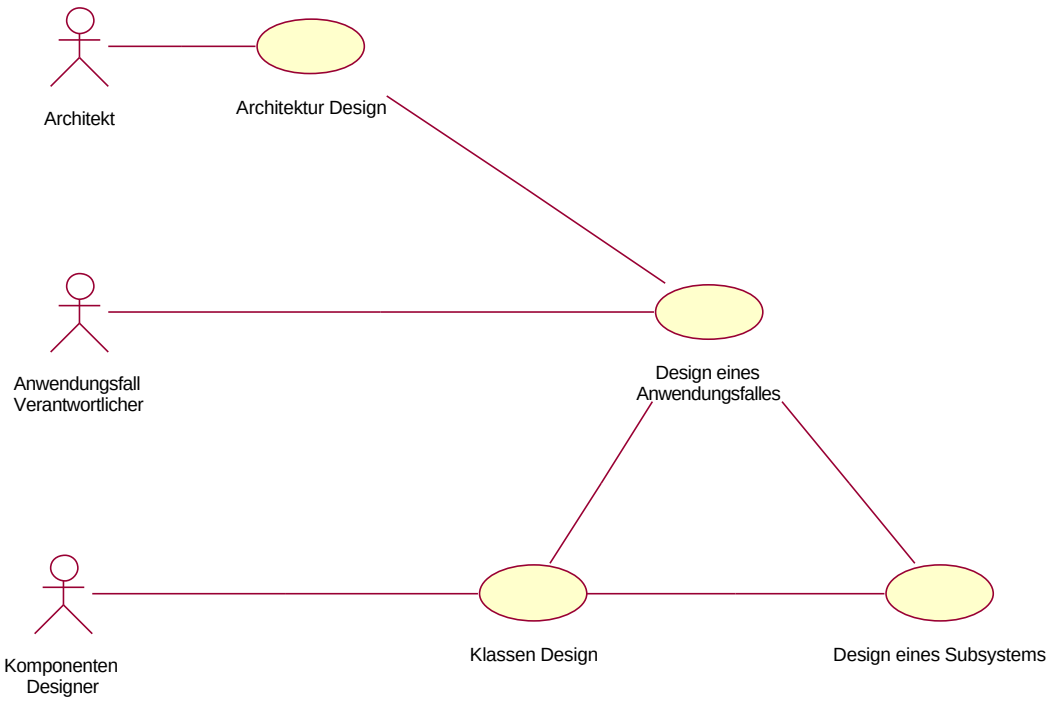
Damit der Verantwortliche für die Komponenten sinnvoll arbeiten kann, sollte er auch für die Implementierungs-Phase beigezogen werden. Er sollte zudem für Komponenten UND deren Subsysteme verantwortlich gemacht werden.

15.5. Workflow

Bisher haben wir die Design Aktivitäten einfach als Ansammlung von Design Ergebnissen beschrieben. Jetzt wollen wir Design Abläufe aufzeichnen.



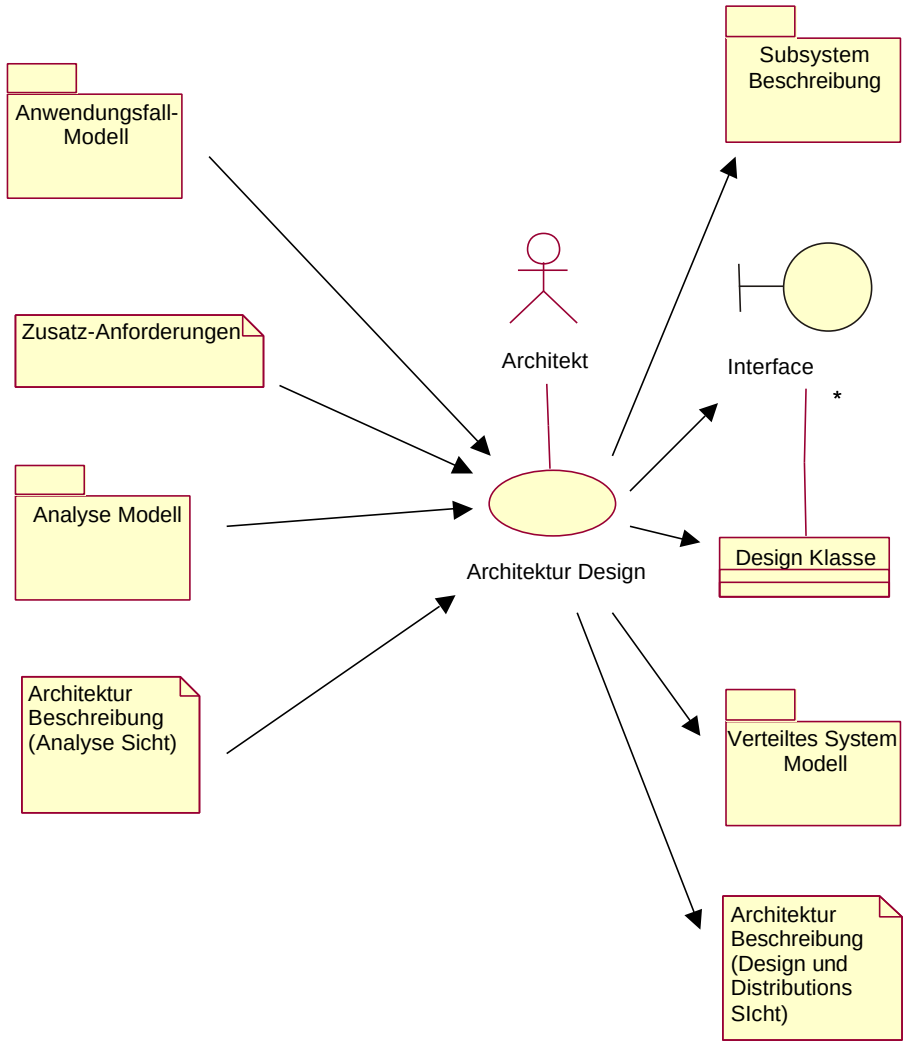
Das obige Diagramm, aus der Online Version „Rational Unified Process“, zeigt in etwa wer was zu tun hat und wie die Aktivitäten voneinander abhängen.



15.5.1. Aktivität : Architektur Design

Ziel des Architektur-Designs ist es, das Design Modell und das Verteilungs-Modell und deren Architektur zusammen zu fassen und dabei folgendes zu identifizieren:

- Knoten und deren Netzwerk Konfiguration
- Subsysteme und deren Schnittstellen
- Architektur-relevante Design Klassen
- Allgemeine Design Aspekte, wie zum Beispiel die Datenspeicherung, Datenverteilung, Leistungsanforderungen, ..., die in der Analyse oder sogar schon in der Anwendungsfall Beschreibung erfasst wurden.



Im Verlaufe dieser Aktivitäten muss der Architekt bestrebt sein, die Wiederverwendbarkeit der gefundenen Klassen zu prüfen und Gemeinsamkeiten einzelner Subsysteme zu erkennen.

Der Architekt muss dabei natürlich auch die Architekturbeschreibung nachführen.

15.5.1.1. Identifikation der Knoten und Netzwerk-Konfiguration

Das physische Netzwerk hat oft einen einschneidenden Einfluss auf die Software Architektur: die Verteilung der Funktionen und Daten muss in der Regel auf Grund der physischen Gegebenheiten durchgeführt werden.

Typische Netzwerk-Konfigurationen sind die 3-Tier-Architektur : Client/User-Interface – Geschäftslogik – Datenspeicherung. Das Client Server Modell ist ein Spezialfall dieser Architektur: die Business-Logik wird einem Knoten zugeteilt.

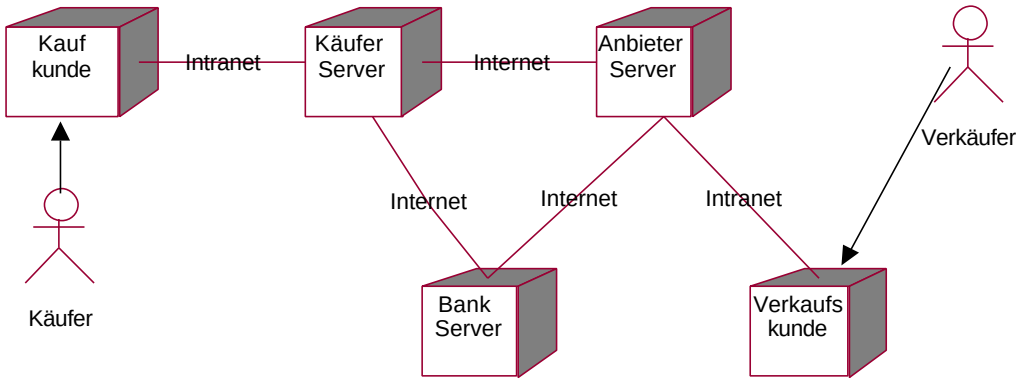
Wie findet man eine sinnvolle Verteilung der Applikation?
Mögliche Aspekte, die man berücksichtigen muss, sind:

- Welche Knoten sind zu berücksichtigen?
  Welche Rechnerleistung haben sie?
  Wieviel Speicher benötigen wir in den Knoten?
- Welche Art Verbindungen existieren?
  Welche Kommunikations-Protokolle werden eingesetzt?
- Welche Chararakteristiken bezüglich Bandbreite, Verfügbarkeit, Qualität ... benötigen wird?
- Benötigen wir redundante Auslegung für einzelne Funktionen, Routings,...?

Daraus lassen sich mögliche Software Architekturen, wie zum Beispiel Objekt Broker (CORBA), RMI, RPC, ... realistisch einplanen oder verwerfen.

Beispiel **Netzwerk Konfiguration für das Interbank System**

Das Interbank-System soll auf drei Server-Knoten und je mehreren Client-Knoten laufen.



Die Kunden (Käufer und Verkäufer) haben über je einen eigenen Knoten Zugang zum Gesamtsystem. Die Knoten kommunizieren untereinander mit Hilfe von TCP/IP. Der Zugriff der Käufer und Verkäufer erfolgt ausschliesslich über dedizierte Knoten. Die Bank hat ein Intranet, an das auch deren interne Systeme angeschlossen sind (zur Verbuchung der Gebühren, ...). Jeder Knoten verfügt über spezielle Authentisierungs-Unterstützung; zudem ist die Verbindung zwischen dem Anbieter-Server und dem Käufer-Server mit dem Bank-Server so ausgelegt, dass ein einfacher Verbindungsausfall zu einem Rerouting führt.

15.5.1.2. Identifikation der Subsysteme und deren Schnittstellen

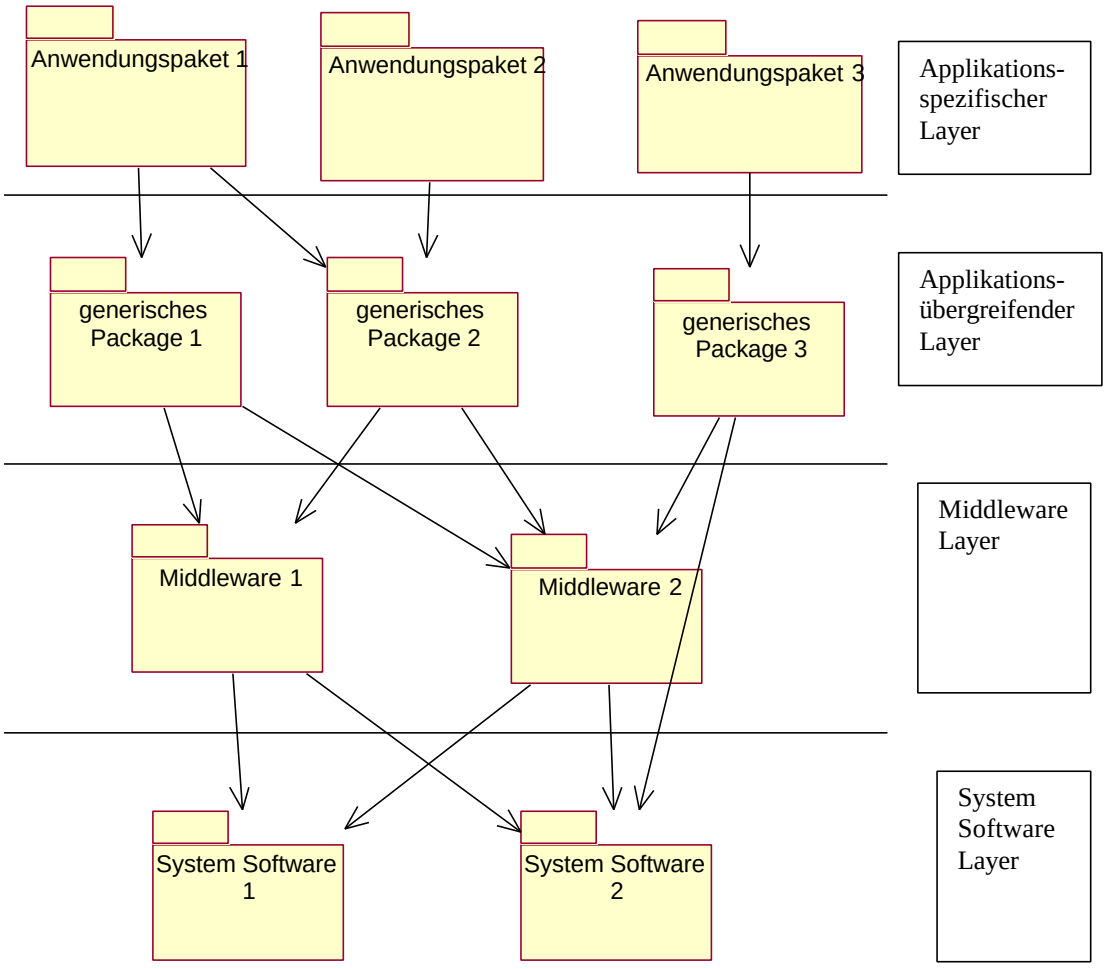
Mit Hilfe der Subsysteme wird das Design Modell in überschaubare Teile zerlegt. Die Subsysteme werden entweder auf Grund der Anwendungsfälle bereits vordefiniert, oder aber erst später, eventuell im Rahmen des Designs erkennbar.

Einige Subsysteme können auch ausser Haus entwickelt werden oder entwickelt worden sein, wie das Betriebssystem, die Datenbanksoftware, die Netzwerksoftware, ...

Damit wir zum Beispiel in Java oder Visual C++ sauber designen können, hat man in Rational Rose die Standard-Packages (java.*. bzw. MFCs von Microsoft für VC++) implementiert. Wir können die Klassen dieser Packages gleich einsetzen und erkennen sofort alle Methoden, Datenfelder und allfällige Vererbungseigenschaften (final, static, public, protected im Falle von Java).

15.5.1.2.1. Identifikation der Anwendungs-Subsysteme

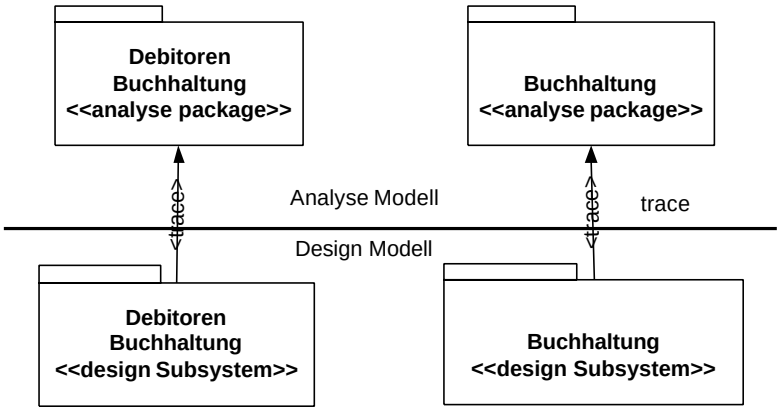
In diesem Schritt müssen wir die grobe Architektur des Systems festlegen. Typischerweise werden wir ein Layer Pattern als Architektur Pattern verwenden.:



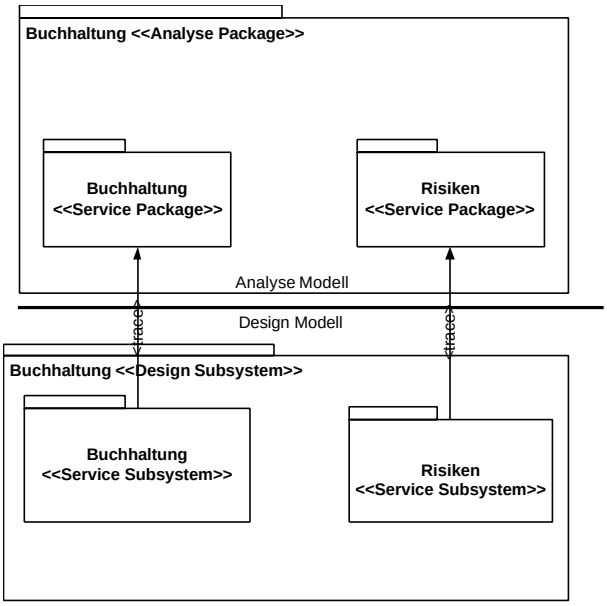
Die Zerlegung des Systems in Packages stammt in der Regel bereits aus der Analyse oder den Anwendungsfällen. Spezielle Beachtung muss man den Service Packages widmen, da diese nach Möglichkeit universell, mindestens mehrfach einsetzbar sein sollten.

Beispiel **Identifikation der Design Subsysteme auf Grund der Analyse Packages**

In unserem Beispiel mit den Kunden-Konten und den Bankkonten können



Wir können eine direkte Beziehung zwischen Analyse Packages und Design Subsystemen herstellen. Zudem können wir auch die Service Pakete, wie zum Beispiel die Risiko-Berechnung, in unser Design Modell übernehmen.



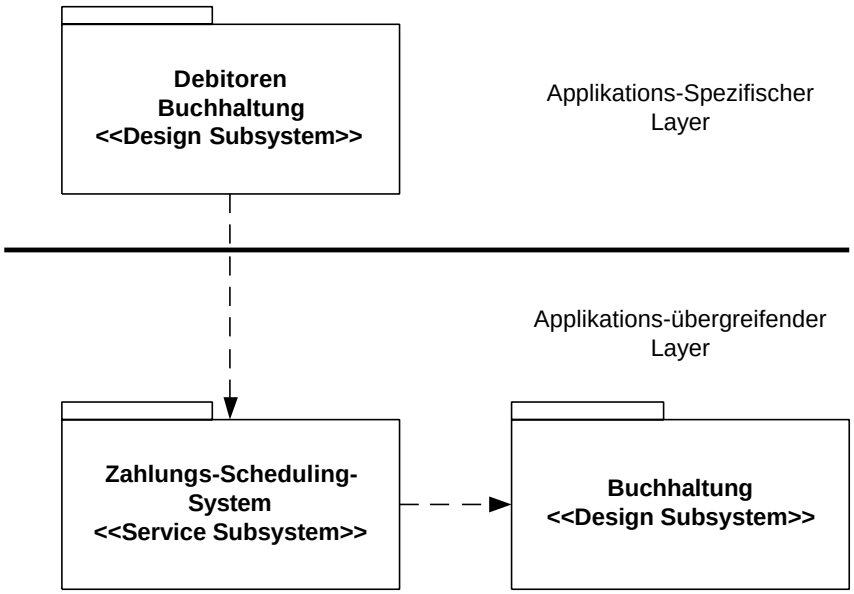
Das resultierende Systemmodell sehen wir oben. Unter Umständen müssen wir, zum Beispiel auf Grund der Anwendungsfälle, das Modell umgestalten und verfeinern.

Beispiel **Verfeinerung der Subsysteme auf Grund des Designs**

Die Interbank-Software versucht alle Service Pakete für die Bezahlung von Rechnungen in die Debitoren-Buchhaltung zu stecken.

Die Entwickler erkennen aber im Laufe der Analyse, dass verschiedene Funktionen auch in andern Teilen der Buchhaltung benötigt werden, beziehungsweise genutzt werden könnten.

Aus diesen Gründen wird die Zerlegung des Systems neu überdacht und neu definiert:



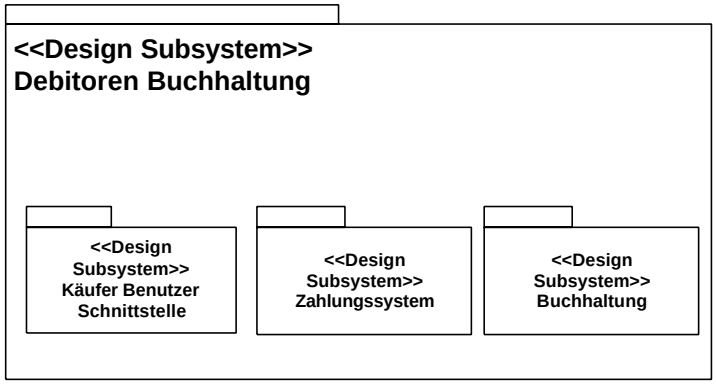
Es zeigt sich, dass es sinnvoll ist, die Bezahlungen jederzeit eingeben zu können, die Ausführung der Überweisungen aber tagesgenau ausführen zu lassen, wie im eBanking und eCommerce üblich.

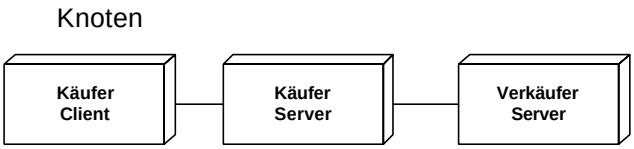
Eine solche Scheduling Funktion ist universell einsetzbar, also sinnvollerweise in einem eigenen Subsystem zu implementieren.

- Einige Teile des Analyse Paketes lassen sich mit Hilfe käuflicher bzw. existierender Software realisieren. Diese Teile der Applikation können wir dem Middleware Layer zuordnen.
- Die vorher definierten Analyse Pakete wurden auf Grund einer eher Analyse der Anwendungsfälle definiert. Im Design wurden weitere mögliche zukünftige Anwendungsfälle definiert, die zur neuen Architektur führten.
- Die Analyse Pakete berücksichtigen die existierenden Softwarepakete nicht. Ein bestehendes Anwendungspaket oder Teile davon, können als separates Subsystem ins neue Systemmodell einbezogen werden.
- Das Analyse Modell berücksichtigte keinerlei Verteilung der Applikation. Die Definition der physischen Knoten und deren Funktion führen zu einer weiteren Verfeinerung des Systemmodells.

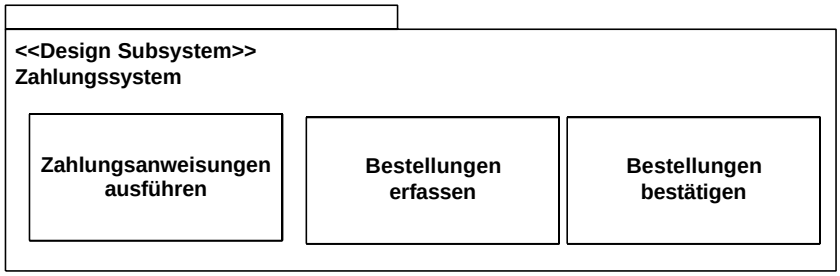
Beispiel **Verteilung eines Subsystems auf mehrere Knoten**

Als erste grobe Version erhalten wir folgende Aufteilung des Gesamtsystems auf die Knoten:





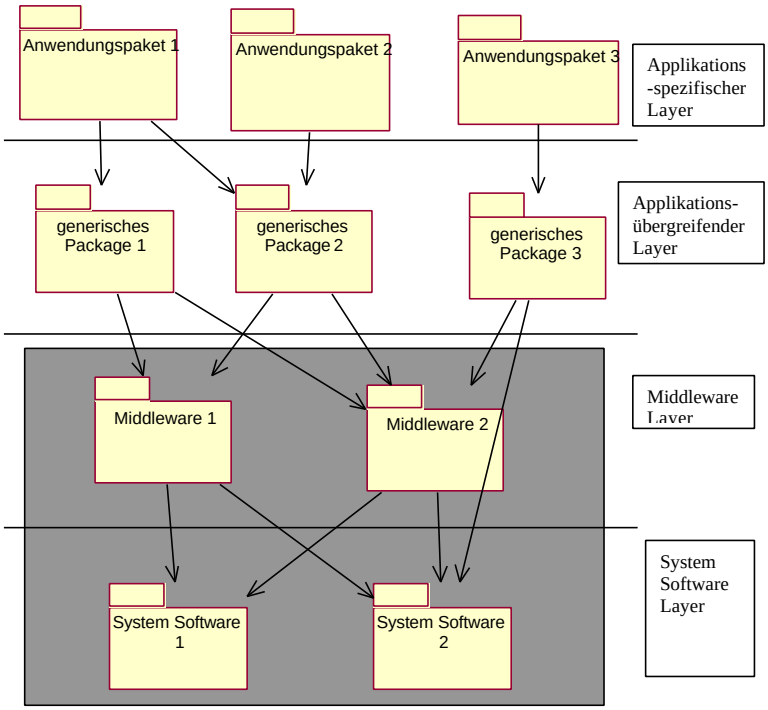
Die Subsysteme lassen sich weiter aufteilen, zum Beispiel:



Wir werden darauf zurück kommen.

15.5.1.2.2. Identifikation der Middleware und System-Software Subsysteme

Middleware und Systemsoftware ist die Basis, auf der alle Software Systeme basieren. Wir müssen also entscheiden, ob wir die Systemsoftware (Betriebssystem, Kommunikationssoftware) und die Middleware (Datenbanken, Broker, Namensdienste) mit in unser Systemmodell einbezogen werden soll oder nicht.



Welche Punkte gibt es zu beachten, falls wir unsere Middleware vollständig einkaufen?

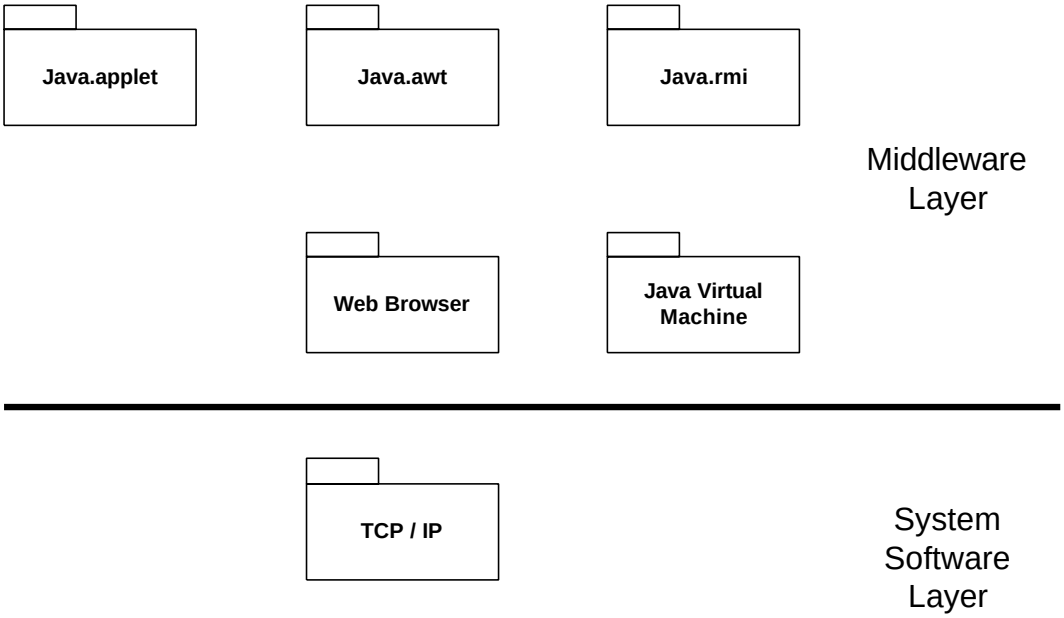
- Die Release-Fähigkeit der Gesamtsoftware, also aller Layer, muss gewährleistet sein. Dies ist oft nicht der Fall, da zum Beispiel Datenbank Hersteller hinter den Betriebssystemen nach hinken.
- Schnittstellen zwischen den Middleware Paketen müssen sorgfältig definiert werden:
  ist es sinnvoll das Betriebsystem als „gemeinsame“ Plattform anzusehen (zum Beispiel Unix oder Windows NT)
  oder
  macht es mehr Sinn die Schnittstelle eine Stufe höher anzusetzen (zum Beispiel Oracle als Datenbanksystem)
  oder
  macht es mehr Sinn die Schnittstelle noch eine Stufe höher anzusetzen und ODBC + SQL als „Schnittstelle“ zu definieren?

Beispiel **Aufbau eines Middleware layers mit Hilfe von Java**

Unsere Interbank Anwendung muss wie heute Stand der Technik ist, mit Hilfe von Java realisiert werden. Typischerweise wird die Applikation mit Hilfe des AWT, des Abstract Windowing Toolkit's von Java implementiert. Zusätzlich soll RMI, Remote Methode Invocation und Applet Packages eingesetzt werden.

Diese Java Pakete stellen wir als Subsysteme dar. Die Java Subsysteme verwenden die Java Virtual Machine (JVM), den Java Interpreter. Zudem wird ein Browser als Standard-Basissoftware für die Applets eingesetzt.

Schauen wir uns einmal an, wie die Architektur unseres Systems in diesem Falle aussieht:



Auf der System Software Ebene müssen wir TCP / IP berücksichtigen. Das wissen wir bereits aus der Distributionsplanung (Internet – Intranet Aufteilung).

Java.applet, Java.rmi und Java.awt sind Standard Java Packages, die jeweils in den Applets, bzw. Server Applikationen importiert und erweitert werden.

Java RMI ist ein recht mächtiger Mechanismus, mit dessen Hilfe verteilte Applikationen aufgebaut werden können. RMI kann auch mit CORBA und ActiveX/DCOM Applikationen kombiniert werden und bildet eines der Fundamente der Jini Technologie.

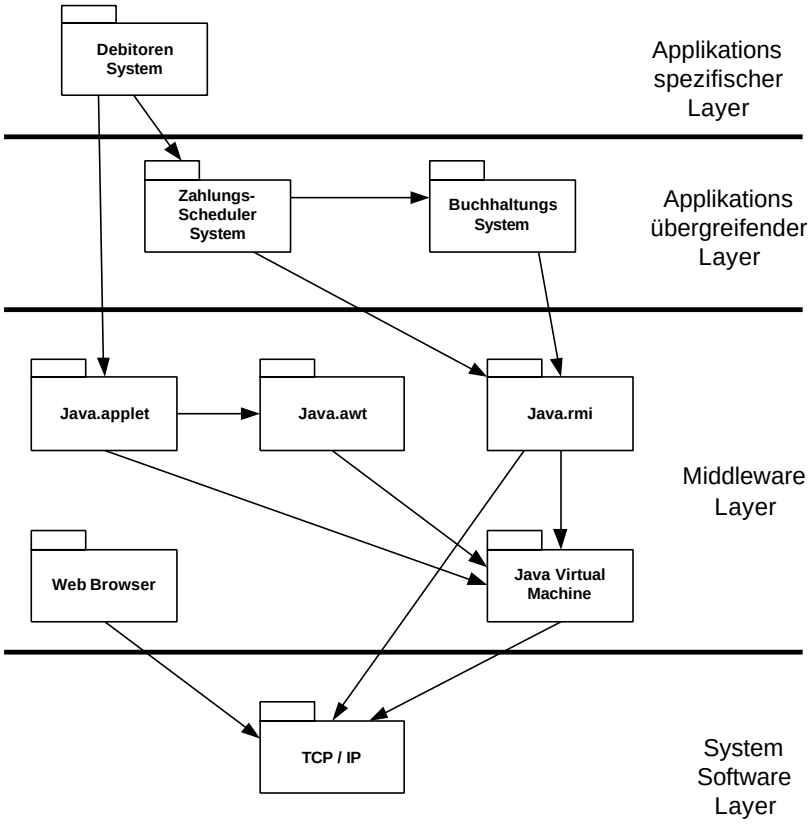
15.5.1.2.3. Definition der Abhängigkeiten von Subsystemen

Die Abhängigkeiten der Subsysteme ergeben sich zum Teil bereits aus der Analyse der Anwendungsfälle. Die Pfeile der Verbindungen sollen jeweils logisch angeordnet / gezeichnet werden, also so, wie das System typischerweise genutzt wird.

Unter Umständen sind die Details dieser Packages noch nicht endgültig. Die Verbindungen ermöglichen aber unter Umständen, einen Anwendungsfall durch zu spielen, und die Details des Packages / der Packages zu erkennen.

Beispiel **Abhängigkeit der Layer**

Die Abhängigkeiten in unserem Interbanken System ergeben sich recht einfach. Das Ergebnis ist eine Architektur, die wie folgt aussieht:



Die Middlerware Ebene ist in diesem Falle recht einfach, die Abhängigkeiten ergeben sich aus der Java Paket Hierarchie und der Abhängigkeit von Java und den Web Browsern vom darunter liegenden Kommunikations-Protokoll.

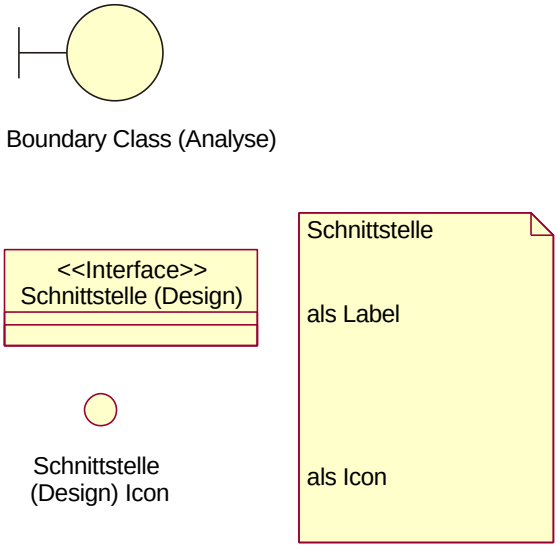
15.5.1.2.4. Identifikation von Subsystem Schnittstellen

Schnittstellen spezifizieren Fähigkeiten des Systems, so wie sie von aussen sichtbar sind. Bevor wir ein Interface spezifizieren, muss das Beziehungsdiagramm der Subsysteme aus dem Design bzw. der Packages aus der Analyse vorliegen. Immer dort wo ein Pfeil endet, ist vermutlich eine Schnittstelle zu implementieren.

Die Notation ist in der Analyse und im Design unterschiedlich:

- In der Analyse sprechen wir von Packages
- Im Design sprechen wir von Subsystemen

- In der Analyse verwenden wir „Boundary Klassen“ als Systemgrenzen
- Im Design verwenden wird das „Interface“ Symbol:

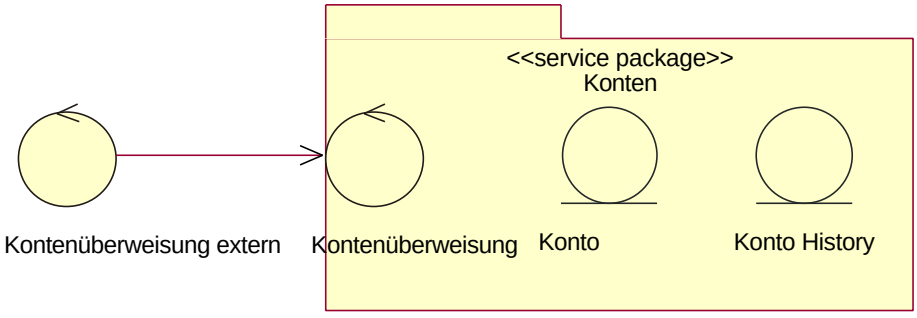


Betrachten wir ein Beispiel, wie aus der Analyse ein Design wird:

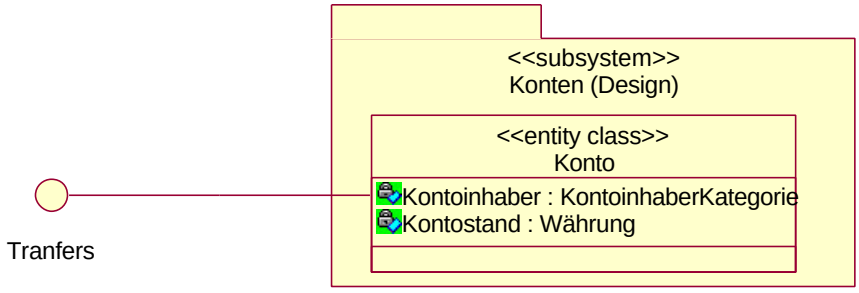
Beispiel **Schnittstellen Kandidaten im Design, auf Grund des Analyse Modells**

Betrachten wir unser Interbanken System. In diesem System müssen wir Konten verwalten, Kontenbewegungen erfassen und Kontenüberweisungen ausführen.

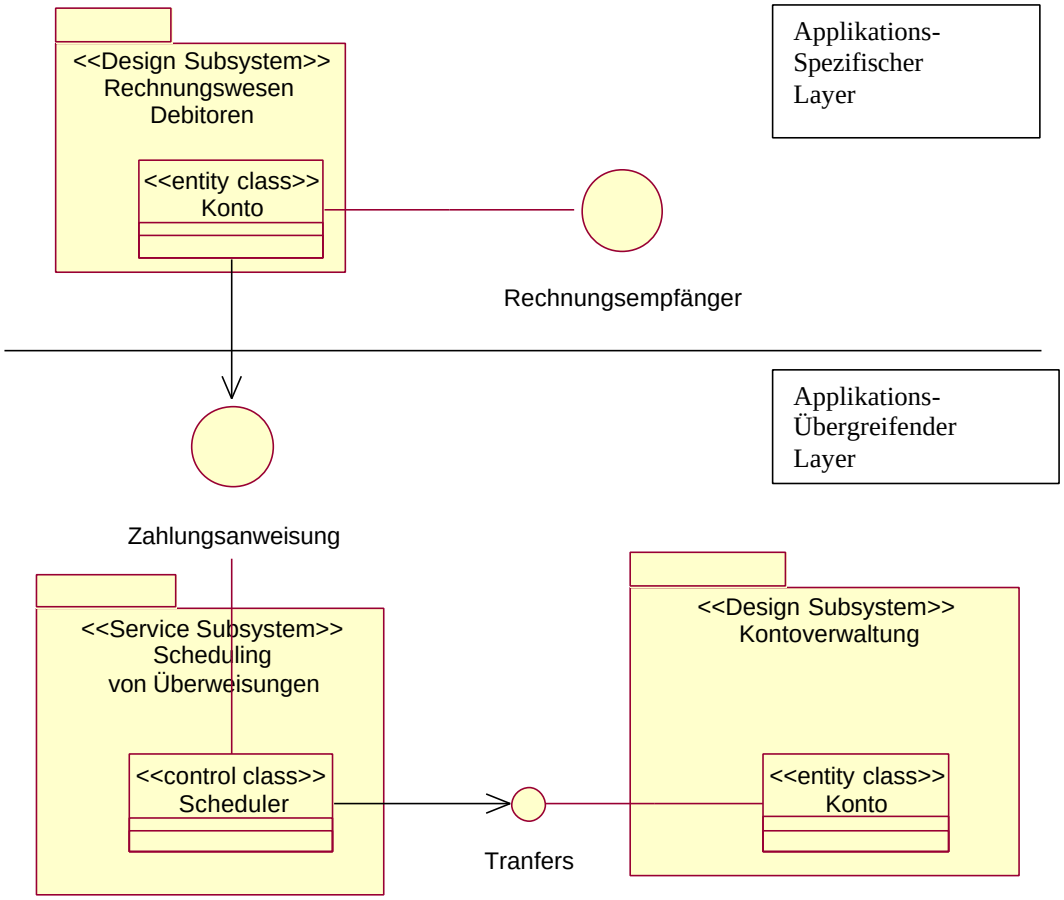
Als Analyse Modell könnten wir folgendes Package und folgende Klassen erhalten:



Daraus leiten wir folgendes Design Modell her:



Damit können wir auch die Schnittstellen auf den unterschiedlichen Layern definieren:



Das Transfer Interface wird für den „Geld“ Transport benötigt, also für die Überweisung von einem Konto auf das andere.

Zahlungsanweisungen können mit Hilfe der Schnittstelle „Zahlungsanweisung“ ins System eingespiesen werden.

Das Beispiel zeigt, wie man durch den Einsatz einer Schnittstelle, oder in Java eines Interfaces, eine einfache Entkopplung von Systemen erreichen kann.

In den unteren Layern ergeben sich die Schnittstellen in der Regel einfacher, da diese Ebenen fast immer eingekauft werden und somit auch die Schnittstellen offen gelegt und bekannt sein müssen.

Die Bezeichnung der Schnittstelle reicht aber nicht aus: wie benötigen auch die Methoden oder Operationen, die jedes Interface definiert. Dazu könnten wir einen Anwendungsfall definieren. In der Regel ist die Definition jedoch offensichtlich.

Als Beispiel aus Java:

```
public interface Tuere {              // Name des Interfaces
  public void oeffnen();              // Methoden
  public void schliessen();     //
}
public class AutoTuere implements Tuere {
  public void oeffne() {
    System.out.println( “In das Auto einsteigen" );
      //...
  }
  public void schliesse() {
    System.out.println( “Achtung, die Tuere wird geschlossen!");
      // ...
  }
}
```

15.5.1.3. Identifikation der Architektur relevanten Design Klassen

Falls möglich, sollte man die Architektur relevanten Design Klassen so früh wie möglich identifizieren. Auf der andern Seite ändert sich oft am Design einer Klasse noch Einiges bei der Programmierung. Es ist also eher ungeschickt früh zu viele Klassen als Architektur relevant zu deklarieren.

Auch hier müssen wir iterativ vorgehen:

- wir können Klassen als Architektur relevant einstufen
- dann arbeiten wir damit und gewinnen Erkenntnisse aus der „Praxis“ sprich Programmierung
- unter Umständen müssen wir dann unsere Architektur relevanten Klassen neu designen oder erweitern
- im schlimmsten Falle müssen wir allerdings vorne anfangen, weil sich unsere Klassenstruktur nicht bewährt.

Die OO Programmierung bietet hier eigentlich keinen sauberen Ausweg :

- sobald wir Klassenhierarchien definiert haben, wird jede Änderung schwierig
- Klassenerweiterungen blähen unter Umständen unser System unnötig auf

Interfaces stellen oft eine bessere Basis dar, weil sie leichter modifiziert werden können und so oder so implementiert werden müssen (siehe Java Beispiel oben).

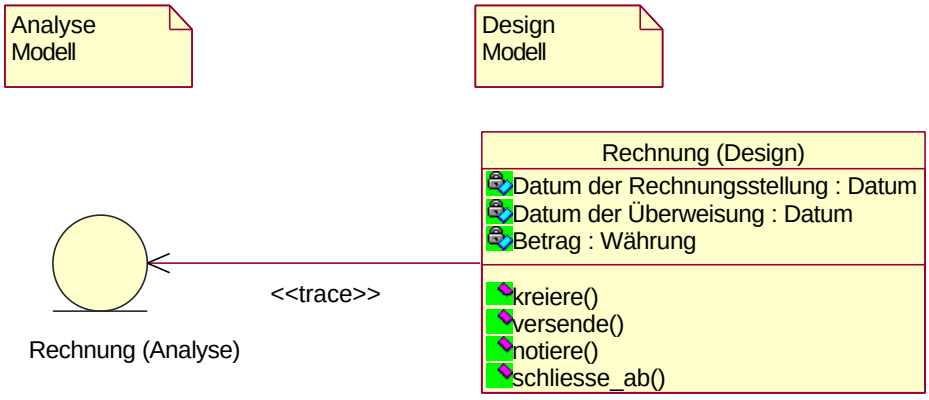
15.5.1.3.1. Identifikation von Design Klassen auf Grund der Analyse Klasse

Im Verlaufe des Designs zeigt es sich in der Regel fast von selbst, welche Klassen aus der Analyse zu Design Klassen werden. Unter Umständen können verschiedene Analyse Klassen zu einer Design Klasse zusammen gefasst werden.

Beispiel **Skizze einer Design Klasse auf Grund einer Analyse Klasse**

Im Interbanken System haben wir die Rechnungen als „Entitäten Klassen“ gefunden und definiert.

Im Design wird daraus natürlich eine Design „Rechnungs-Klasse“



Wir finden die Rechnungsklasse im Design durch Verfeinerung der Rechnungsklasse (Entitäten Klasse) der Analyse.

15.5.1.3.2. Identifikation von aktiven Klassen

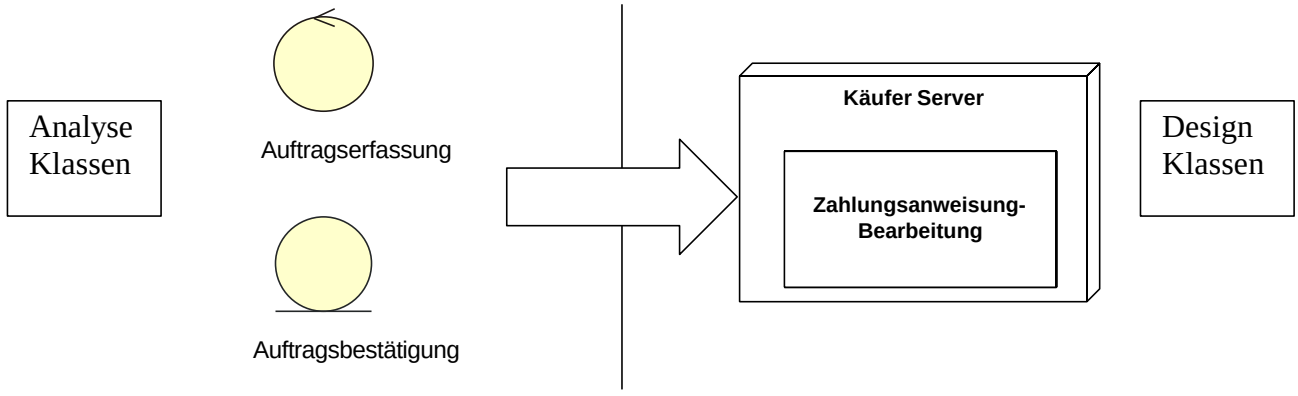
Aktive Klassen müssen vom Architekten aus folgenden Gründen erfasst und beschrieben werden:

- Performance:
  Die Leistungsfähigkeit unseres Systems, die Verfügbarkeit des Systems für verschiedene Aktoren stellen Anforderungen an unser System dar.
  Unter Umständen müssen hohe Performance Anforderungen mit Hilfe von spezialisierten Objekten erreicht werden (WinNT ist ein Beispiel dafür : der Windows on Windows Teil war ursprünglich eine Ebene höher angesiedelt).
- physische Verteilung auf die Knoten:
  Aktive Objekte müssen unter Umständen aus Verfügbarkeitsanforderungen auf mehr als einem Knoten vorhanden sein.
  Dies führt zu Interkommunikation, für die wieder neue Objekte geschaffen werden müssen.
- Deadlock, Starvation, Startup, Shutdown
  Diese grundlegenden Operationen bedingen in der Regel zusätzliche Klassen. Je nach Programmiersprache können Teile dieser Probleme mit Hilfe von Synchronisations-Mechanismen „erschlagen“ werden.

Aktive werden auf Grund ihres Lebenszyklus definiert und beschrieben. Performance und Antwortszeitverhalten kann gravierende Auswirkungen auf die Architektur eines Informationssystems haben.

Beispiel **Herleiten von Aktiven Klassen aus Analyse Klassen**

Im Interbanken System haben wir die Funktionen auf mehrere Knoten verteilen müssen, aus Sicherheitsgründen und auch aus naheliegenden Performance Gründen.
Jetzt sei die Situation die folgende:
der Käufer wäre interessiert an den Funktionen „Auftragserfassung“ und „Auftragsbestätigung“.
Er erkennt aber, dass sein Knoten nicht leistungsfähig genug ist.
Als Lösung bietet sich eine Verschiebung dieser Funktionen auf einen dedizierten Server an, den Käufer Server.
Wir fassen diese Funktionen zusammen und erhalten folgenden Lösungsvorschlag:



15.5.1.4. Identifikation von generischen Design Mechanismen

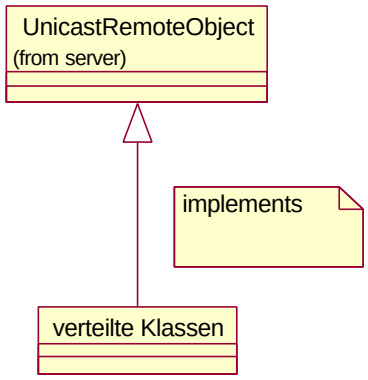
In diesem Design Schritt versuchen wir jene Teile des Systems zu finden, die in den vorgelagerten Aktivitäten unter „spezielle Anforderungen“ abgetan wurden.

Hauptsächlich geht es um folgende Punkte:

- Datenspeicherung
- Objektverteilung
- Sicherheitsanforderungen
- Fehler-Entdeckung und Behebung
- Transaktions-Management

Beispiel **Objekt Verteilung**

Beim Interbanking System müssen gleichzeitig Kunden aus verschiedenen geographischen Gegenden Zugriff auf ein und die selbe Rechnung haben.
Interbank Software entscheidet daher, dass solche Objekte mit Hilfe des Java RMI Modells verteilt werden. Diese Klassen werden damit Subklassen der abstract class java.rmi.UnicastRemoteObject, mit dessen Hilfe RMI implementiert werden kann.



Beispiel **Objekt Speicherung**

Beim Interbanking System müssen verschiedene Daten langfristig abgespeichert werden. Zum Beispiel Bestellungen. Dafür kann zum Beispiel eine Objekt-Datenbank, eine relationale Datenbank oder auch einfach nur Dateien eingesetzt werden.

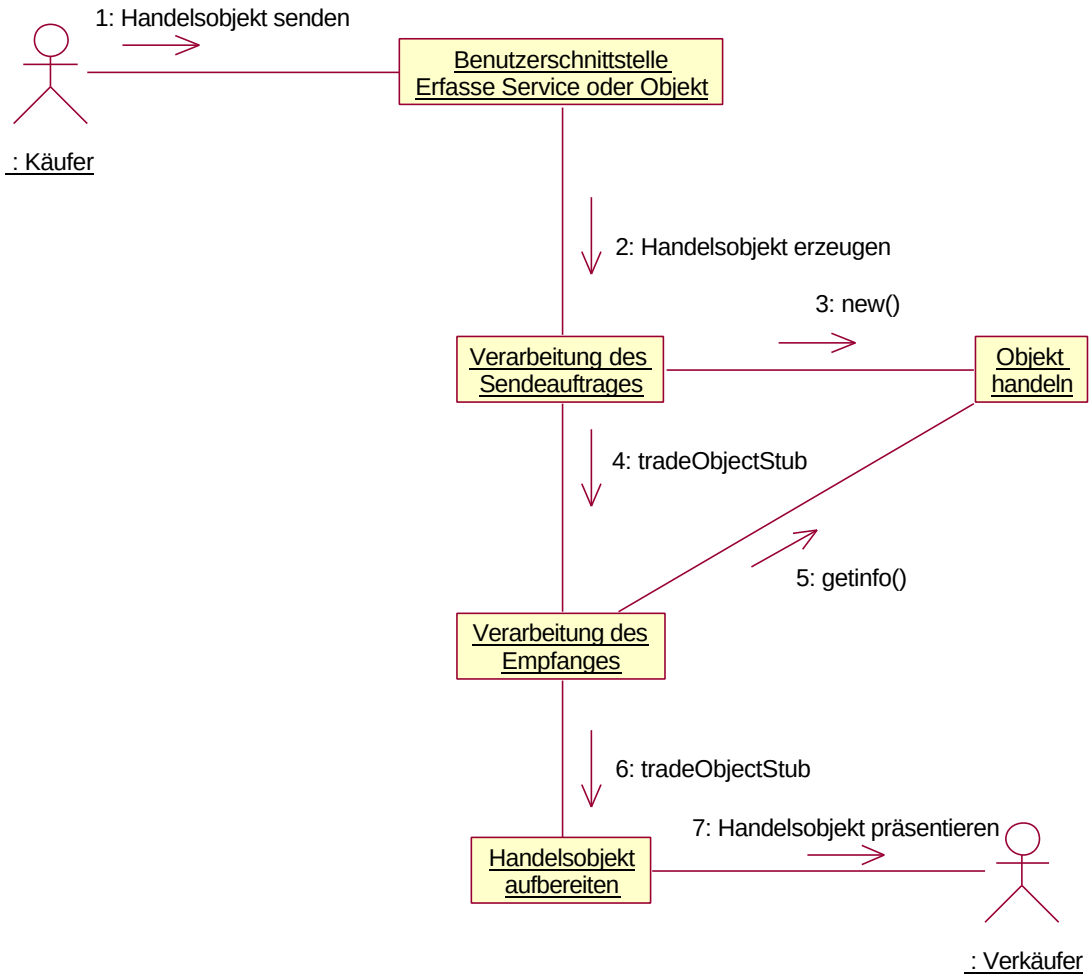
Welcher Datenbanktypus konkret eingesetzt werden soll hängt vom konkreten Problem ab.

Beispiel **Kollaboration in der Anwendungsfall-Umsetzung**

Im Interbanking geschieht es öfters, dass ein Handelsobjekt, zum Beispiel eine Bestellung oder eine Rechnung, von einem Aktor geschaffen wird und zu einem andern Aktor gesandt wird.

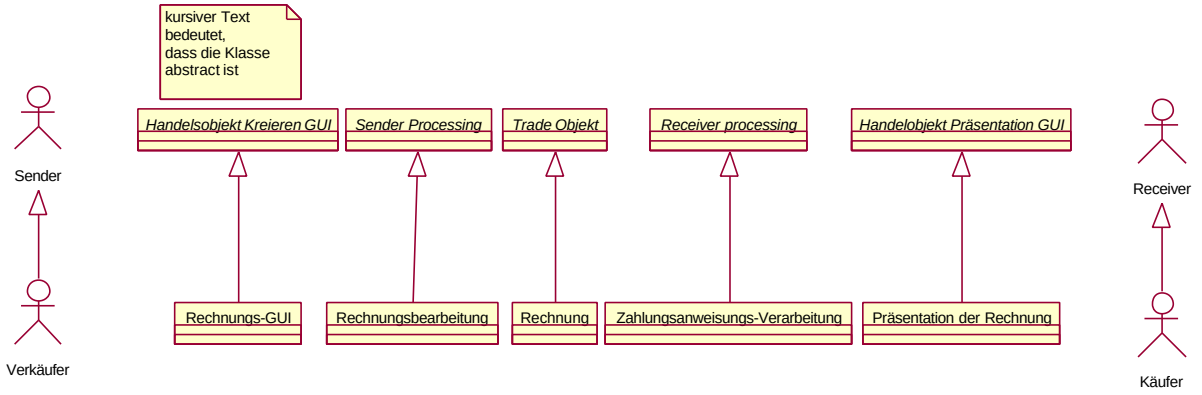
- wenn der Käufer entscheidet, Ware oder Dienstleistungen vom Verkäufer zu kaufen, dann startet er den Anwendungsfall „Auftragserfassung (Waren und Services)“. Damit kann der Käufer alle relevanten Daten erfassen und an den Verkäufer übermitteln.
- falls der Verkäufer beschliesst, dem Käufer eine Rechnung zu senden, dann startet er den Anwendungsfall „Rechnungserstellung“ und sendet die Rechnung an den Käufer.

Grob gesprochen erhalten wir folgendes Kollaborations-Diagramm:



Die Stubs sind die „Stecker“ mit deren Hilfe die Objekte beim Broker angemeldet und im Netz bekannt gegeben werden.

Wennn wir nun aus dem obigen Diagramm versuchen ein Klassen-Diagramm zu erhalten, so kann dies zum Beispiel im ersten Ansatz wie folgt aussehen:



In der oberen Ebene haben wir generische „Klassen“ als abstract class definiert, in der unteren Hälfte sehen wir mögliche konkrete Realisierungen.

Damit haben wir auch gleichzeitig die Nützlichkeit verschiedener Konzepte aus Java angedeutet (Interfaces, abstract classes).

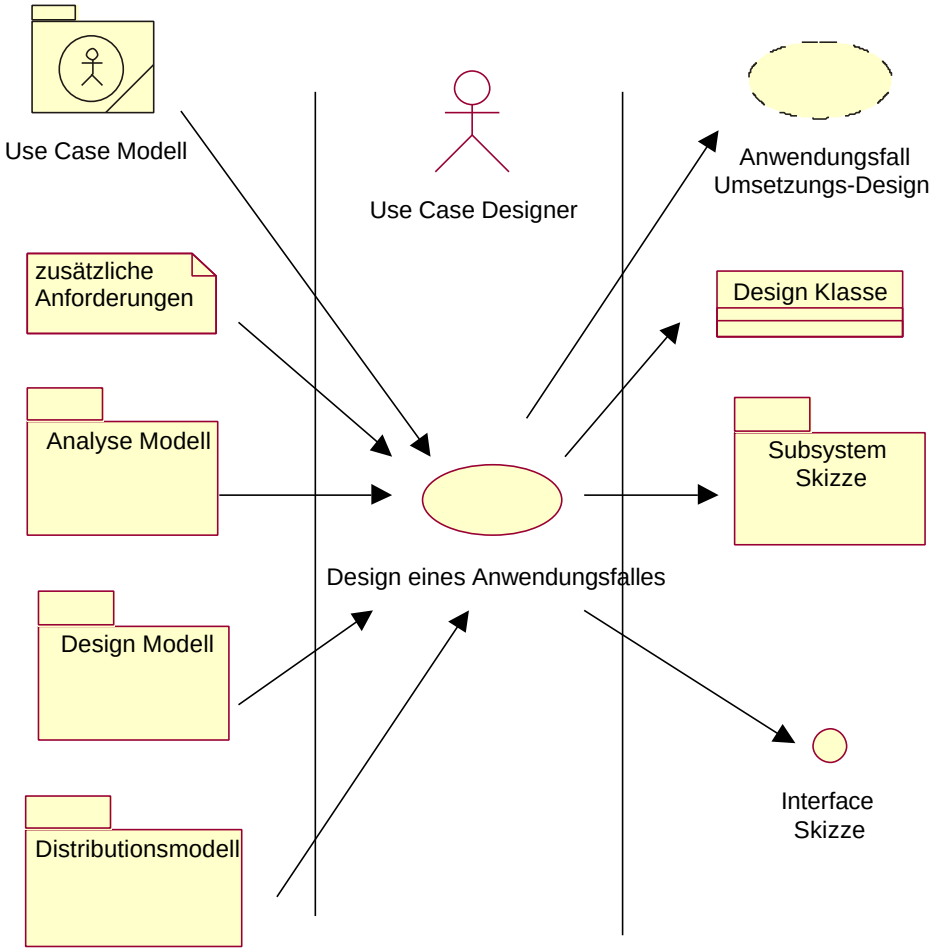
15.5.2. Aktivität : Design eines Anwendungsfalles

Anwendungsfälle müssen aus folgenden Gründen designed werden:

- Identifikation aller Design Klassen, mit deren Hilfe die Anwendungsfälle implementiert werden sollen
- Verteilen des Verhaltens des Anwendungsfalles auf interagierende Design Objekte und beteiligte Subsysteme.
- Festhalten von Anforderungen, die zum „Betrieb“ der Objekte wesentlich und nötig sind.
- Festhalten von Anforderungen für die Realisierung der Anwendungsfälle

15.5.2.1. Identifikation der partizipierenden Design Klassen

Die Aktivitäten als Ganzes lassen sich wie folgt visualisieren:

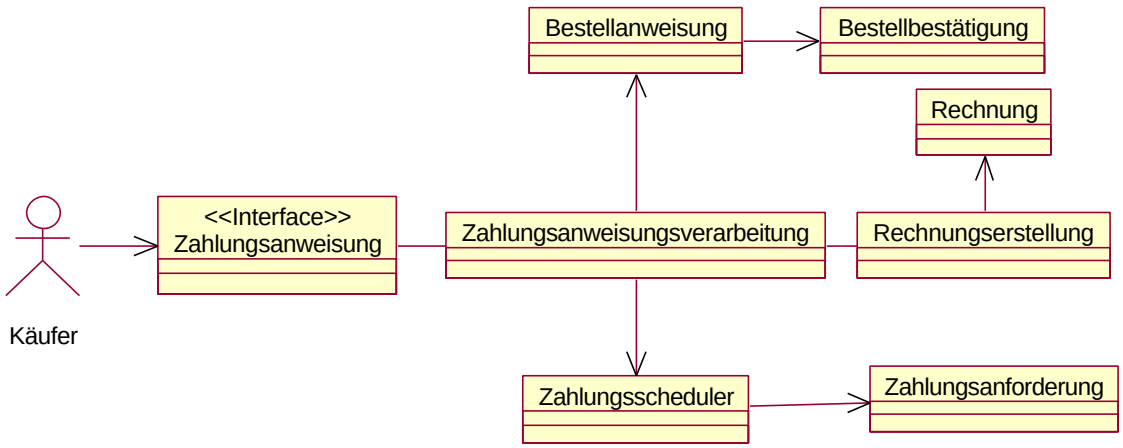


Im Einzelnen geht es um folgende Aktivitäten:

- untersuchen aller Analyse Klassen, die an einem Anwendungsfall (Analyse) beteiligt sind. Untersuche alle Klassen, die mit diesen Analyse Klassen in Verbindung stehen. Diese Klassen werden vom Komponenten Designer definiert, im Zuge der Definition der Architektur.
- Untersuche „Spezielle Anforderungen“, die zusätzlich erfasst wurden und halte alle Klassen fest, die an solchen Anforderungen beteiligt sind. Diese können sich zum Teil erst aus dem Architektur-Design ergeben.
- alle resultierenden Klassen sollten als Ganzes in irgend einer Form, zum Beispiel einer Komponente, zusammen gefasst werden. Dafür idt der Komponenten Designer zuständig.
- sollten noch einige Design Klassen fehlen, dann muss der Komponenten Designer, der Architekt und der Anwendungs-Designer zusammen bestimmen, wo noch Klassen fehlen, oder wem Klassen zugeordnet werden können.

Beispiel **Teilnahme von Klassen an einer Anwendungsfall Umsetzung**

Aus der Skizze unten ist zu erkennen, wer und wie am Anwendungsfall „Rechnung begleichen“ beteiligt ist.



15.5.2.2. Beschreibung der Wechselwirkung von Design Objekten

Nachdem wir eine Skizze der Design Klassen, die an unserem Anwendungsfall beteiligt sind, haben, müssen wir beschreiben, wie diese miteinander wechselwirken.

Dies geschieht im Design mit Hilfe von „Sequenz-Diagrammen“. Ein Sequenz-Diagramm besteht aus den partizipierenden Aktoren, Design Objekten und Meldungen, die zwischen diesen Objekten ausgetauscht werden. Die Meldungen sind, wie man leicht aus Rational Rose erkennen kann, METHODEN.

Bei komplexen Abläufen muss man in der Regel mehrere Sequenz Diagramme erstellen, jeweils eines pro Anwendungsfall kann bereits zu komplex sein.

Das Ziel der Sequenz-Diagramm Darstellung ist es, möglichst universell einsetzbare Teile zu finden.

Als Ausgangspunkt kann ein Kollaborations-Diagramm dienen. Oft kann man aber auch direkt eine Analyse der Objekt-Kommunikation machen.

Im Falle der Anwendungsfälle speilt man einfach die Anwendungsfälle durch und hofft, Sequenz-Diagramme zu erhalten, die möglichst universell gültig sind.

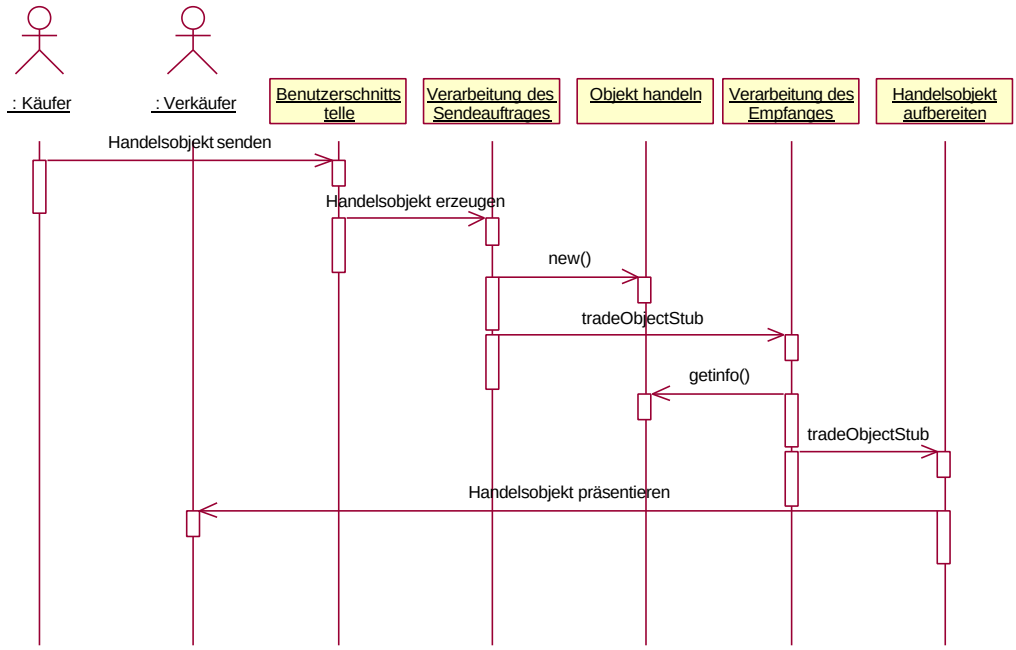
Bezüglich Sequenz-Diagrammen muss man folgende Punkte beachten:

- ein Anwendungsfall beginnt beim Aktor, der ein Objekt aufruft
- jedes Design Objekt sollte in mindestens einem Sequenz Diagramm vorkommen.
- Meldungen werden zwischen Objektlebenslinien gezeichnet (vertikale Linie, die beim Objekt startet; die Zeit schreitet nach unten fort)
- Kernpunkt der Sequenz-Diagramme sind, wie der Name andeutet, die Sequenzen. Die Sequenz der Meldungen, der Ereignisse ist wesentlich. Weitere Details können zum Teil mit Hilfe von Zusatztext erläuternd hinzu gefügt werden.
- Beziehungen zwischen Anwendungsfällen (<<extends>> zum Beispiel) müssen auch in den Sequenz-Diagrammen vermerkt werden.

Beispiel **Sequenzdiagramm für den ersten Teil der Zahlungsüberweisung**

Der folgende Ausschnitt aus einem komplexen Sequenz Diagramm zeigt wie ein solches aussehen kann, ohne dass wir auf alle Details eingehen, da dies nur mit Hilfe eines CASE Tools, wie Rational Rose möglich ist.

Falls wir bereits ein Kollaborations-Diagramm haben, dann erstellt Rose automatisch ein Sequenz-Diagramm. Wir brauchen das Kollaborations-Diagramm anzuwählen und dann im Menü „Browse“ die Option „Create Sequence Diagram“ zu wählen. Das Ergebnis sieht wie folgt aus, und kann als Ausgangspunkt für ein verfeinertes Sequenz-Diagramm dienen.



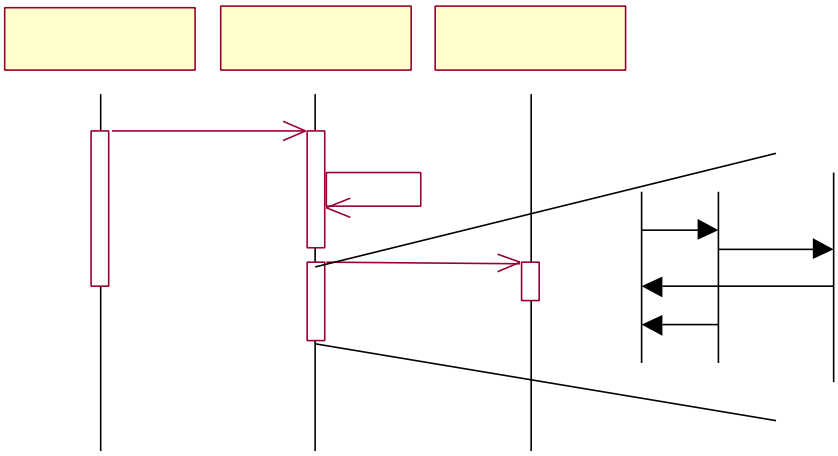
In der Regel wird man das Sequenz Diagramm, welches auf diese Art erzeugt wurde, verfeinern. Beispielsweise:

- die „Meldungen“ werden verfeinert, da im Kollaborations-Diagramm eher eine informelle Darstellung verwendet wird. Die Meldungen werden am Schluss zu Methoden der Objekte und Klassen, bei denen die Pfeile enden.
- mit Text wird das Diagramm verständlicher gemacht und allfällige kritische Teile besonders hervor gehoben.
- in der Regel werden, weil wir mehr ins Detail gehen (müssen), zusätzliche Ausnahmen, alternative Pfade erkannt. Diese müssen auch eingetragen und speziell gekennzeichnet werden.
  - Zeitüberschreitungen bei der verteilten Verarbeitung
  - Eingabefehler, die durch das Programm erkannt und evtl. automatisch korrigiert werden
  - Fehlermeldungen aus der Middleware, dem Betriebssystem, dem Netzwerk oder der Hardware

15.5.2.3. Identifikation der Subsysteme und Schnittstellen

Bis jetzt haben wir den Anwendungsfall als Zusammenarbeit verschiedener Klassen und deren Objekte dargestellt. Damit wir uns nicht in den Details verlieren, ist es in der Regel wesentlich einfacher, den Anwendungsfall auch auf einer verdichteten Stufe, der Stufe „Subsysteme“ und deren Zusammenarbeit darzustellen. Dies ist typischerweise der Fall, falls wir Top Down designen.

Auf der Stufe der Subsysteme haben wir bereits verschiedene Beispiele gesehen. Auf der Stufe Sequenz-Diagramme kann dies zum Beispiel wie folgt aussehen:



Der Balken in der Mitte wird mit Hilfe eines weiteren Sequenz-Diagrammes verfeinert. Wichtig ist, dass wir die Übersicht nicht verlieren! Wir müssen also irgendwie, in der Regel mit Hilfe eines Textlabels, die Verbindung der zwei Diagramme festhalten.

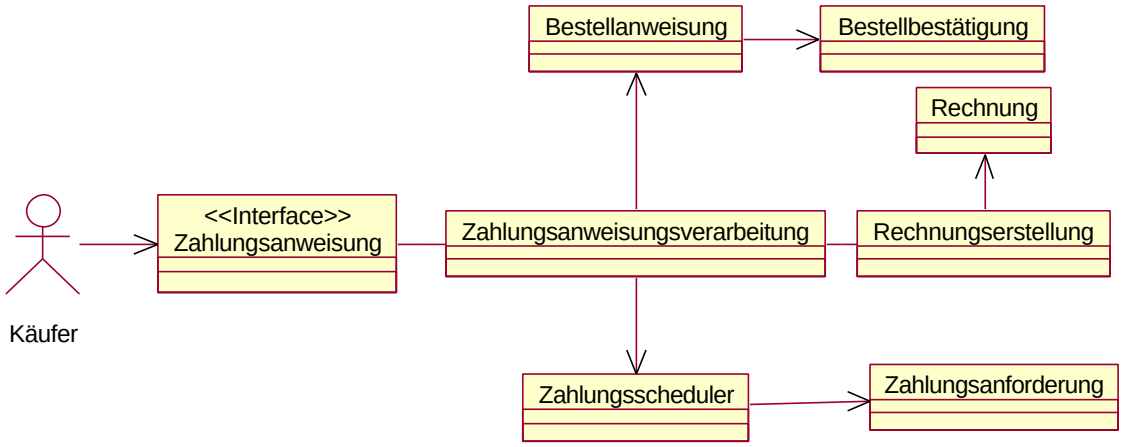
Wie können wir systematisch vorgehen?

- wir wählen die Analyse und Design Klassen aus, die an unserem Anwendungsfall beteiligt sind
- dann suchen wir die Packages, zu denen diese Klassen gehören (Analyse) bzw. die Subsysteme (Design)

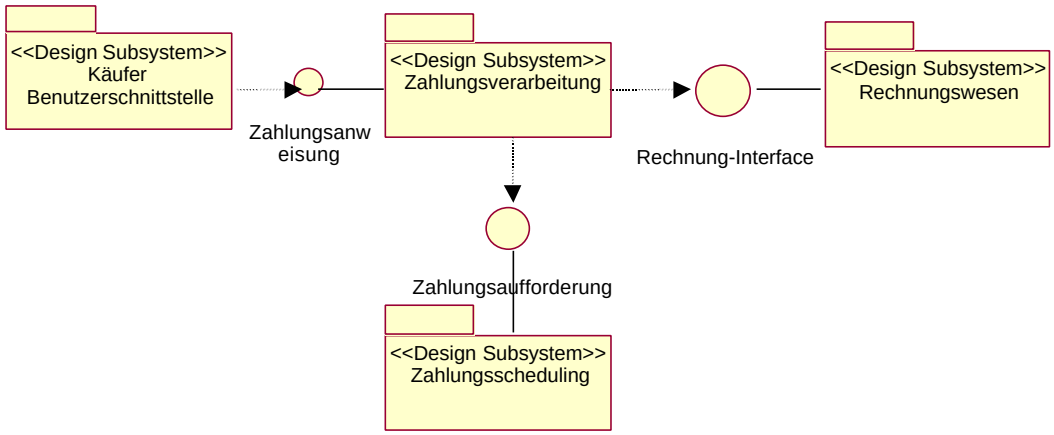
Daraus lässt sich dann eine verdichtete Darstellung herleiten.

Beispiel **Subsysteme und Schnittstellen eines Anwendungsfalles**

Betrachten wir noch einmal den folgenden Anwendungsfall:



Wenn wir die Klassen passend zusammen fassen, erhalten wir folgendes „Subsystem-Diagramm“:



15.5.2.4. Beschreibung der Wechselwirkung zwischen Subsystemen

Um den Anwendungsfall auf der Stufe Subsystem zu beschreiben, können wir analog zur Beschreibung der Wechselwirkung der beteiligten Klassen vorgehen. Wir verwenden dazu ein Sequenz-Diagramm, allerdings nicht mit Objekten und Klassen zuoberst, sondern mit ganzen Subsystemen.

- Die senkrechten Linien stellen dann die „Lebenslinien“ dieser Subsysteme dar.
- Jedes beteiligte Subsystem hat mindestens eine Lebenslinie
- Schnittstellen sollten speziell gekennzeichnet werden, damit klar ist, welche Schnittstelle zu welchem (sub-) System gehört.

15.5.2.5. Festhalten von Implementations-Anforderungen

Diese Aktivität ist wieder als Ergänzung gedacht:

- falls zusätzliche wesentliche Merkmale, die die Implementation betreffen, festgehalten werden sollen, dann geschieht dies am Einfachsten mit Hilfe von Text

---

Beispiel **Implementations-Details für den Anwendungsfall „Zahlungsanweisung“**

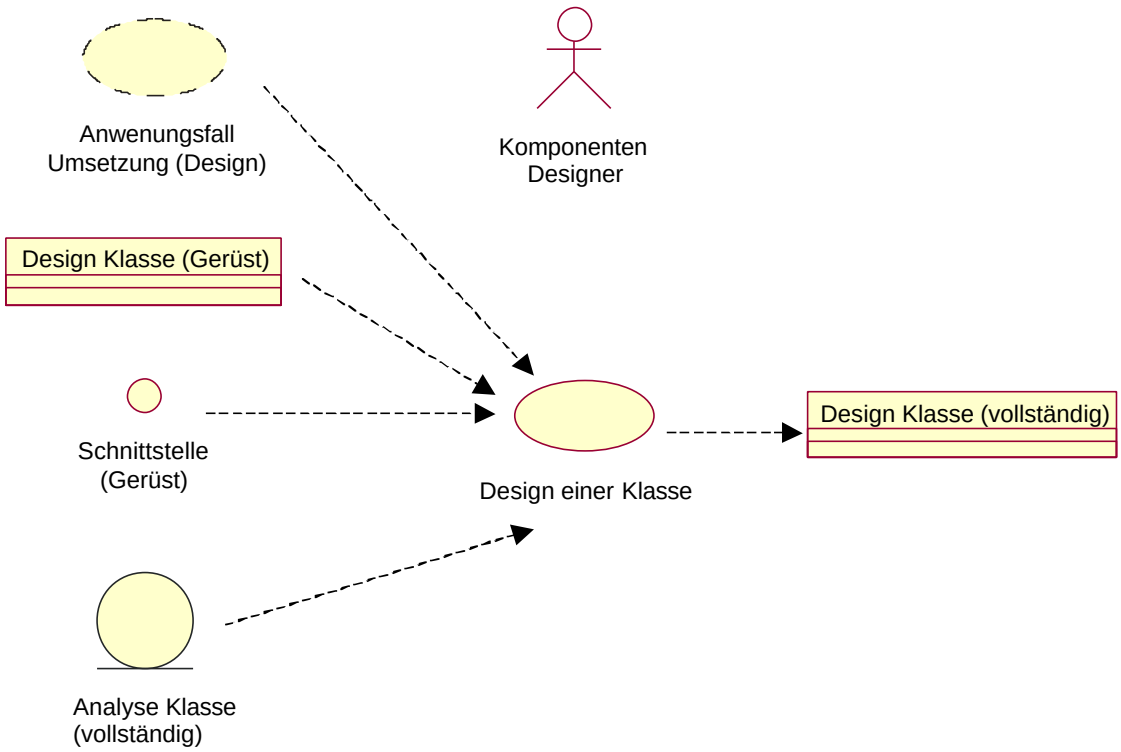
Die Objekte der Klasse „Zahlungsanweisung“ sollten in der Lage sein, quasi gleichzeitig bis zu zehn Kunden zu bedienen, ohne dass die Kunden eine signifikante Verlangsamung der Verarbeitung feststellen.

15.5.3. Aktivität : Design einer Klasse

Eines Tages müssen wir uns mit den *Details* der an einem Anwendungsfall beteiligten Klassen und Objekte beschäftigen.

Dies umfasst folgende Aktivitäten:

- die Methoden spezifizieren
- die Attribute, die Datenfelder festhalten
- die Beziehungen ausspezifizieren (Komposition, Vererbung : *has-a, is-a* Unterscheidung)
- Abhängigkeiten festhalten, die sich aus dem Umfeld ergeben, zum Beispiel der Verwendung von MFCs, der Verwendung von JFCs oder Template Libraries
- Anforderungen berücksichtigen, die für die fertige Lösung relevant sind (Antwortzeiten)
- Schnittstellen beschreiben
- Schnittstellen implementieren



Betrachten wir nun die einzelnen Schritte, die nötig sind, um zu einer vollständig spezifizierten Klasse zu gelangen.

15.5.3.1. Skizzieren der Design Klasse

Als erstes müssen wir ein grobes Bild der Klassen erhalten und das Zusammenspiel der Klassen beschreiben, also deren Schnittstellen skizzieren.

Je nach Stereotyp der Analyse Klasse müssen wir unterschiedlich vorgehen:

- im Falle von Systemgrenzen (Boundary Classes) müssen wir die speziellen Techniken berücksichtigen, die für die Schnittstellen-Implementation relevant sind.
  Beispiele:
  IDL (Interface Definition Language)
  ActiveX
  Zudem kann ein Werkzeug für die Erstellung der Benutzerschnittstelle gravierende Auswirkungen auf unsere Boundary Klassen haben (wir müssen die Benutzerschnittstelle sinnvoll und effizient in unser System integrieren).
- im Falle einer Entitätenklasse, welche gespeicherte Informationen darstellt, sind wir in der Regel auf die Spezifika der verwendeten Datenbank oder des Dateisystems angewiesen. Im Falle der relationalen Datenbanken müssen wir uns auch Gedanken zur Performance machen, da wir in der Regel nicht alle Relationen in die dritte Normalform bringen dürfen, sofern wir akzeptable Antwortzeiten wünschen.
- im Falle der Kontrollklassen müssen wir speziell vorsichtig vorgehen. Grund: Kontrollklassen beschreiben Abläufe, und diese können verteilt werden auf verschiedene Knoten. Wir haben also mindestens folgende Punkte zu überdenken:
  - Distribution:
    wie soll die Kommunikation aussehen?
    welche Teile der Kontrollklasse müssen auf welchem Knoten implementiert werden?
    was passiert in Ausnahmefällen (Ausfall eines Knotens, ...)?
  - Leistung:
    Leistungsüberlegungen können dazu führen, dass wir Teile der Business Logik zum Client schieben, zum Beispiel Datenvalidationen. Diese Aufteilung muss sorgfältig geplant werden, da wir sonst unnötig viele Abstimmungen vornehmen müssen und in der Regel Probleme mit der Bandbreite erhalten.
  - Transaktionsmanagement:
    Kontrollklassen beinhalten oft Transaktionen. Wir müssen uns überlegen, wie wir die Transaktionen atomar ausführen können, was im Fehlerfall, zum Beispiel dem Abbruch einer Transaktion geschieht, wie unterbrochene Transaktionen wieder gestartet werden können und wie wir bestehende Transaktionsmonitore sinnvoll einbinden könnten.

15.5.3.2. Identifikation der Operationen

Diese Aktivität umfasst die Spezifikation der Operationen, die unser Objekt, beziehungsweise präziser ausgedrückt unsere Klasse, ausführen muss. Diese Spezifikation muss die Spezifika der in der Implementierung zu verwendenden Programmiersprache berücksichtigen.
Als Beispiel im Falle von C++ und Java :

- Zugriffsattribute : *public, protected, private*

Im Einzelnen benötigen wir folgende Inputs:

- wofür ist die (Analyse) Klasse zuständig. Zuständigkeiten implizieren in der Regel Aktivitäten, also Operationen. Die Beschreibung der Zuständigkeiten umfasst oft auch die Beschreibung von Eingaben und Ausgaben, bzw. Rückgabewerte.
- spezielle Anforderungen, die bei der Beschreibung der Analyse Klasse festgehalten wurden. Dies kann auch Anforderungen betreffen, die sich aus der Verwendung einer speziellen Datenbank, einer speziellen Netzwerktechnologie, Web-Technologie oder ähnliches ergeben.
- die Schnittstellen und deren Operationen. Typischerweise in Java oder IDL werden in der Schnittstellenbeschreibung die Operationen mit Eingabe- und Ausgabe-Parameter und deren Typen festgehalten.
- die Anwendungsfall-Umsetzung, in der wir Klassen und deren Kommunikation beschrieben haben.

Wichtig ist zu verifizieren, dass die Klasse am Schluss über die Funktionalität verfügt, die wir von ihr erwarten. Wir müssen für diese Verifikation zwangsweise die Anwendungsfälle zu Hilfe nehmen.

Beispiel Operationen der Rechnungsklasse

Die Rechnungsklasse partizipiert an verschiedenen Anwendungsfällen:

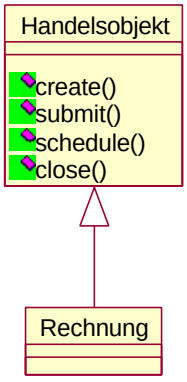
- Rechnung bezahlen
- Mahnung erstellen und versenden
- Rechnung stellen

Eine Variante, die unterschiedlichen Rollen einer Rechnung im Geschäftsablauf zu erfassen, wäre mit Hilfe eines „Status“: zum Beispiel „setzeStatus(Zahlung_erhalten)“.

Eine genauere Betrachten der beteiligten Klassen zeigt, dass ein analoges verhalten auch für andere Klassen gilt. Wir fassen diese zusammen zu einer Klasse „Handelobjekt“ und definieren dafür die Operationen:

- kreiere()
- schedule()
- schliesse()
- verschicke()

Damit erhalten wir folgendes Klassendiagramm für die Rechnung:



15.5.3.3. Identifikation der Attribute

Als nächstes müssen wir die Attribute der Klassen (und deren Objekte) festlegen. Speziell im Falle von (Analyse) Entitätenklassen müssen wir dabei die Spezialitäten der Datenbank, mit deren Hilfe wir das System implementieren möchten, berücksichtigen.

Generell können wir wie folgt vorgehen:

- als erstes berücksichtigen wir die Attribute der Analyse Klasse, auf der die Design Klasse beruht
- die Attribute müssen die Spezialitäten der Datenbank berücksichtigen, falls wir die Klasse als Relation implementieren möchten
- die Attribute müssen die Spezialitäten der Programmiersprache berücksichtigen, falls wir die Klasse mit Hilfe einer Programmiersprache implementieren möchten.
- die Attributtypen sollten nicht inflationär erweitert werden: neben den Basistypen sollten möglichst wenig anwendungsspezifische Datentypen neu definiert werden
- falls ein Attribut in mehreren Instanzen verwendet wird, müssen wir je nach Programmiersprache das Attribut in eine eigene Klasse auslagern oder das Attribut als Klassenvariable (in Java *static)* deklarieren
- falls die Klasse zu komplex wird, müssen wir vermutlich die Klasse aufteilen und mehrere Klassen daraus machen.
- falls die Klasse viele Attribute hat, sollte im Klassendiagramm eventuell eine Darstellung gewählt werden, bei der die Anzeige der Attribute unterdrückt wird.

15.5.3.4. Identifizieren der Assoziationen und Aggregationen

Design Objekte interagieren wie im Sequenz-Diagramm beschrieben. Solche Interaktionen implizieren oft eine Assoziation der entsprechenden Klassen. Der Komponenten-Designer muss also auch die Sequenz-Diagramme analysieren, um fest zu stellen, ob er in seinem Komponenten Design diese Beziehungen berücksichtigt hat.

Die Anzahl Beziehungen zwischen den Klassen sollte minimiert werden. Das kennen wir bereits zu Genüge: wir möchten kohärente und möglichst entkoppelte Klassen definieren und implementieren, damit wir diese auch möglichst unabhängig weiter entwickeln können.

Auch hier bilden die Anwendungsfälle die Basis zur Realisierung bzw. zur Definition von Assoziationen und Aggregationen (als starke Form der Assoziation).

Die folgenden generellen Richtlinien sollten beachtet werden:

- wir beginnen mit dem Analyse Modell : Assoziationen im Analyse Modell werden vermutlich auch im Design Modell überleben.
- Assoziationen müssen vollständig ausspezifiziert werden:
  - Mächtigkeiten (1:n, n:m, c:n)
  - Rollennamen
  - spezielle Anforderungen und Möglichkeiten der zu Grunde liegenden Programmiersprache.
  - Richtung der Assoziation (is-a, bzw. has-a Beziehung)

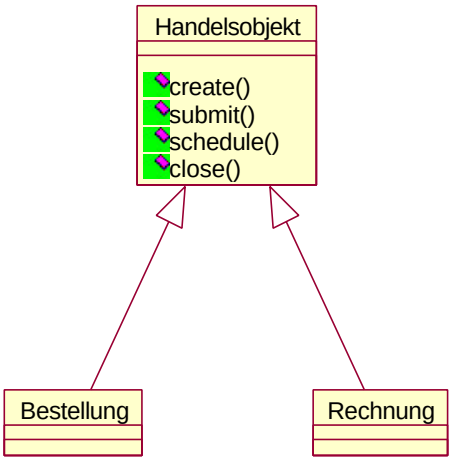
15.5.3.5. Identifikation von (möglichen) Verallgemeinerungen

Verallgemeinerungen, also Abstraktionen, können wir nur insofern berücksichtigen wie sie in der Programmiersprache für die Implementation benötigt und ermöglicht werden.

Im Extremfall haben wir eine antike Programmiersprache vor uns, die keine Vererbung ermöglicht (C, Fortran) oder zu Seiteneffekten führen kann (C++). In diesen Fällen müss(t)en wir entsprechend vorsichtig vorgehen (die Praxis zeigt das Gegenteil: in unpräzise definierten Programmiersprachen, wie zum Beispiel C/C++ ,versucht jeder Programmierer zu zeigen, wie toll er ist und wie gut er Schrott verwerten und produzieren kann und wie kompliziert er programmieren kann).

Eine mögliche Generalisierung haben wir bereits kennen gelernt:

Beispiel **Generalisierung im Design Modell**



15.5.3.6. Beschreibung der Methoden

Methoden spezifizieren, wie die Operationen implementiert werden. Die Methode enthält somit den expliziten Algorithmus, mit dessen Hilfe eine Operation implementiert werden soll.

Oft lässt man diese Details im Design weg, un verschiebt diese Entscheide auf die Implementation. ALLERDINGS: es bietet sich an, sich bereits im Design Gedanken zu machen, wie wir eine Methode implementieren möchten, damit wir nicht komplexe Entscheide einfach vor uns herschieben.

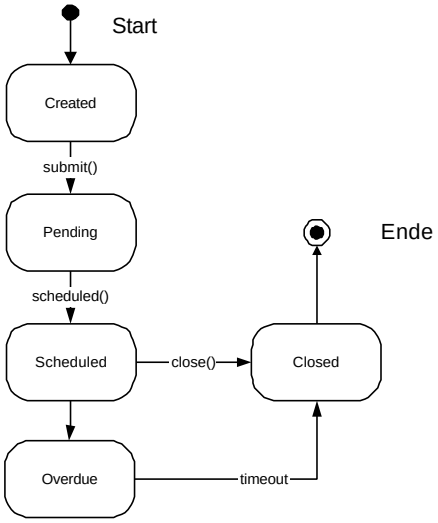
Wir können im Design auch gleich mit Hilfe der Programmiersprache oder Pseudocode den Algorithmus ausspezifizieren. Unter Umständen hilft uns dann das CASE Tool, mit dem wir den Design durch führen, bei der Generation des Programmcodes.

15.5.3.7. Beschreibung der Zustände

Zustände ergeben sich durch die konkrete Belegung der Datenfelder mit Werten. Die Werte ändern sich auf Grund des Klassenverhaltens. Wir können diese Übergänge mit Hilfe von Zustandsdiagrammen (und Petri-Netzen) beschreiben. Eine solche Beschreibung bildet eine wertvolle Basis für die Beschreibung des Algorithmus, mit dessen Hilfe die Operation implementiert wird.

Beispiel **Zustandsdiagramm für die Rechnungsklasse**

Das folgende Diagramm ist selbsterklärend:



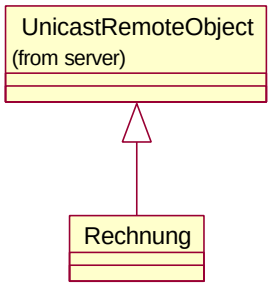
Eine Rechnung wird erzeugt (geschrieben) und versendet (submit). Damit ist sie „Pending“, wir warten auf den Eingang der Überweisung. Falls die Rechnung nicht pünktlich bezahlt wird („overdue“), müssen wir spezielle Aktionen starten.

15.5.3.8. Das Handhaben von Spezialanforderungen

Wir haben an unterschiedlicher Stelle bereits mehrfach Zusatzanforderungen mit Hilfe von Text erfasst. Jetzt müssen wir diese Texte auswerten!

Beispiel **Zusatzanforderung im Design**

Wir beschliessen, unsere Applikation zu verteilen und den Java RMI Mechanismus (wie er auch in Jini verwendet wird) dafür einzusetzen.



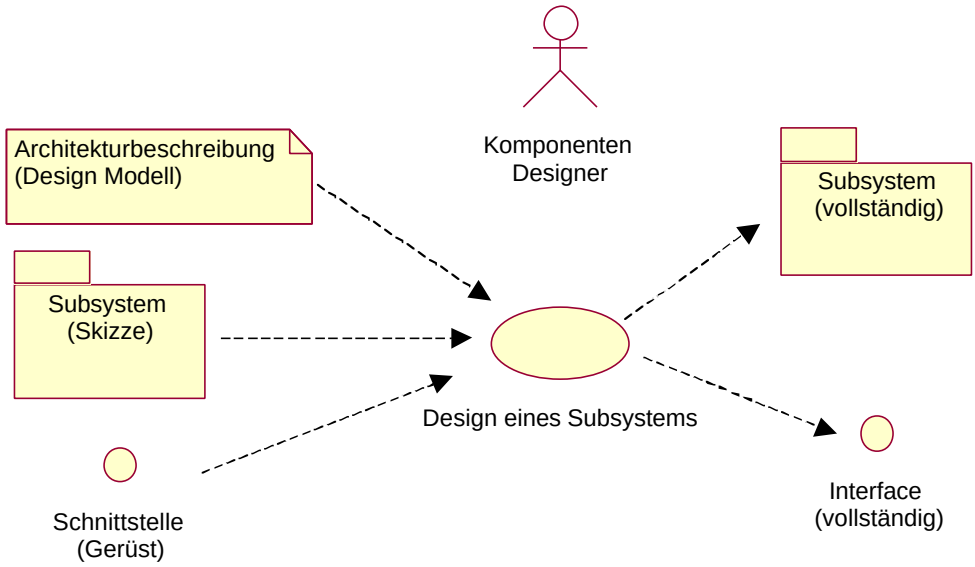
Da Sie RMI vermutlich noch nicht kennen, gehen wir nicht auf weitere Details ein. RMI (Remote Methode Invocation) ist analog zu RPC (Remote Procedure Call), aber im Gegensatz zu RPC ausschliesslich in Java implementiert.

Alle speziellen Anforderungen, die wir nicht im Design berücksichtigen möchten oder können, können in der Implementation unter Umständen zu Risiken führen. Wir sollten also nicht zu viel auf die Implementierung verschieben!

Beispiel **Zusatzanforderung für dieImplementation**

Die Implementation der RMI basierten Rechnungsstellung muss gleichzeitig zehn Benutzern eine sinnvolle Antwortszeit garantieren.

15.5.4. Aktivität : Design eines Subsystems



Der Schwerpunkt dieser Aktivität ist:

- möglichst die Subsysteme entkoppeln
- prüfen, ob die Subsysteme auch die korrekten Schnittstellen zur Verfügung stellen
- prüfen, ob die Subsysteme die benötigte Funktionalität aufweisen

15.5.4.1. Wartung der Abhängigkeiten der Subsysteme untereinander

Die Abhängigkeit der einzelnen Subsysteme muss sorgfältig geplant werden. Sie sollte sich auch zeitlich gesehen wenig bis nicht verändern.

Wie in Java ist es in der Regel geschickter Abhängigkeiten auf Schnittstellen zuu schieben, statt auf Klassen oder ganze Subsysteme. Schnittstellen lassen sich leichter modifizieren als Klassen oder ganze Subsysteme.

15.5.4.2. Wartung der Schnittstellen eines Subsystems

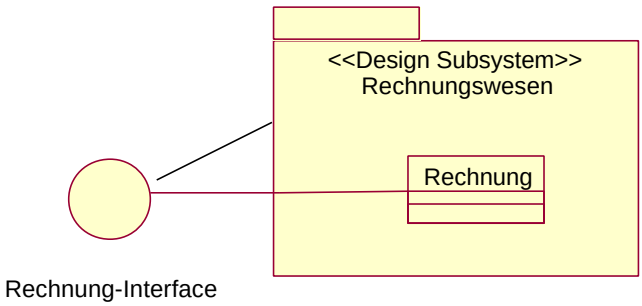
Speziell in Java lassen sich Schnittstellen relativ leicht spezifizieren und auch modifizieren. In andern Fällen, zum Beispiel im Falle von IDL (Interface Definition Language), welche für RMI, CORBA, RPC, jeweils in leicht unterschiedlicher Form eingesetzt wird, sind Änderungen zwar leicht möglich, haben aber gravierende Auswirkungen, da diese Systeme inhärent verteilt sind.

15.5.4.3. Wartung des Inhaltes eines Subsystems

Subsysteme haben eine bestimmte Aufgabe und Verantwortung. Diese sollte sich im Verlaufe des Projektes möglichst nicht ändern.

- für jedes Interface sollte eine entsprechende Design Klasse existieren, die die Funktionalität des Interfaces implementiert (wie in Java üblich)
- die Bezeichnung der Schnittstellen sollte so gewählt werden, dass daraus auf die Funktionalität des Subsystems geschlossen werden kann.

Beispiel **Subsystem „Rechnungswesen“**



Das Rechnungsinterface wird mit Hilfe der Rechnungsklasse implementiert, im Subsystem Rechnungswesen.

15.6. Zusammenfassung : Design

Das Hauptergebnis des Designs ist das Design Modell. Wichtig ist, dass das Design Modell sich aus dem Analyse Modell herleitet und auf diesem beruht. Die beiden Modell dürften also nicht allzu verschieden sein.

Das Design Modell ist die Grundlage, der „Blueprint“, für die Implementation.

Das Design Modell umfasst folgende Elemente:

- Design Subsysteme und Service Subsysteme und deren Abhängigkeiten, Schnittstellen und Inhalte.
  Design Subsysteme der oberen zwei Layer (applikationsspezifisch und applikationsübergreifend) beruhen auf den Analyse Paketen.
  Andere Teile des Design Modells können aus der Verteilung der Applikation resultieren.
- Design Klassen beschreiben Operationen, Attribute und Beziehungen, sowie spezielle Anforderungen an die Implementation.
  In der Regel beruhen die Design Klassen auf Analyse Klassen d.h. die Modelle werden nicht völlig neu entwickelt.
- Anwendungsfall-Umsetzungen im Design beschreiben, wie einzelne Klassen zusammen arbeiten, zum Beispiel mit Hilfe eines Sequenz-Diagrammes.
- Die Architektursicht des Design Modells beschreibt im Detail die Architektur des zu implementierenden Designs.

Als Zusatzergebnis erhalten wir im Design auch ein Distributionsmodell:
- dieses zeigt, wie die Applikation auf mehrere Knoten verteilt wird
- dieses zeigt, auf welche Knoten welche Klassen oder Subsysteme verteilt werden
- die Architektur des Gesamtsystems wir dem Distributionsmodell überlagert:
  eine 3-Tier-Architektur wird mit Hilfe verschiedener Knoten realisiert. Das Distributionsmodell zeigt, auf welchem Knoten welche Subsysteme anzusiedeln sind.

Als nächstes müssen wir die gefundenen Klassen implementieren:
- Design Subsysteme und Design Service Subsysteme werden als Subsysteme implementiert
- Design Klassen werden idealerweise mit Hilfe einer Objekt Orientierten Programmiersprache implementiert.
- Anwendungsfälle werden aus Klassen und Subsystemen zusammen gesetzt und „released“, freigegeben.
- das Distributionsmodell dient der Planung des Netzwerkes und der Kommunikationsmechanismen, die implementiert werden müssen.