In diesem Kapitel:
- *Analyse in Kurzfassung*
- *Analyse im Software Lebenszyklus*
- *Artifacts*
- *Beteiligte Personen*
- *Workflow*
  - *Architektur Analyse*
  - *Use Case Analyse*
  - *Klassenanalyse*
  - *Packageanalyse*
- *Zusammenfassuna*

14

ROP / RUP Rational Objectory / Unified Process - Analyse

Nachdem wir uns recht ausgiebig mit den Anforderungen und deren Formalisierung mit Hilfe von Use Cases und Aktivitätsdiagrammen, sowie Sequenz-Diagrammen befasst haben, wenden wir uns nun der Analyse zu.

14.1. Einführung

Die Analyse umfasst mehrere Aktivitäten, die zu einer verfeinerten Struktur der Anforderungen führt.
Anforderungen mussten vor allem in der Sprache der Benutzer erfasst werden: die Benutzer mussten in der Lage sein, das was wir "fest schreiben" auch nach vollziehen zu können. Use Cases sind dafür sicher ein geeignetes Mittel.
Auf der andern Seite müssen wir damit leben, dass die eher unpräzise Formulierung der Use Cases zu Problemen bei der Implementation führen wird.
Fassen wir noch einmal die wichtigsten Kennzeichen einer guten Use Case Analyse zusammen:

1. *Use Cases müssen so unabhängig von einander sein wie nur immer möglich*. Sonst besteht die Gefahr, dass sich der Benutzer in Details verliert und am Schluss eher alles unklar als geklärt ist.
2. *Use Cases müssen in der Sprache des Benutzers beschrieben werden.* Das wird dadurch erreicht, dass die Use Cases in Textform beschrieben werden.
3. *Use Cases müssen je so formuliert werden, dass sie ein vollständiges und intuitives Bild der funktionalen Anforderungen liefern.* Use Cases sollten also auf keinen Fall "in Hinblick auf die Implementierung" bereits unnötig fein strukturiert und in Teilfunktionen zerlegt werden.

Das Hauptziel der Analyse ist es die Anforderungen zu präzisieren, aber nicht mehr in der Sprache des Benutzers sondern *in der Sprache des Entwicklers.*

Deswegen werden wir in der Analyse vertieft über die Struktur des zu realisierenden Systems nachdenken. Wir müssen mit Hilfe einer verfeinerten Formalisierung die Anforderungen an unser zu realisierendes System präzisieren.

Zudem können wir in der Analyse Phase die Anforderungen so strukturieren, dass wir bzw. die Entwickler, diese besser verstehen können. Unser Fokus muss sich also ändern. Themen wie Wiederverwendbarkeit, Unabhängigkeit und Wartbarkeit spielen eine wichtige Rolle bei der Analyse.

Als Ergebnis werden wir Analyse Klassen und Analyse Packages erhalten, die im Design und der Implementierung jeweils "umgewandelt" werden müssen.

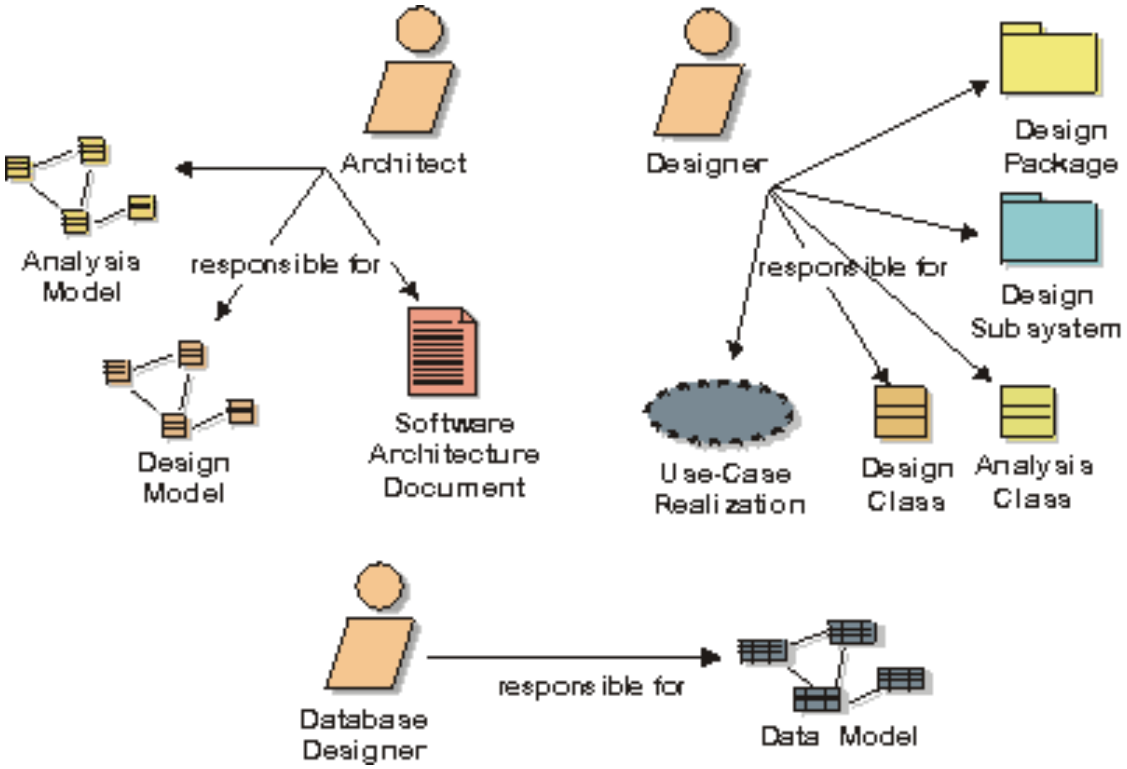
Vergleich des Use Case Modells und des Analyse Modells

Use-Case Modell	**Analyse Modell**
Beschreibung in der Sprache des Kunden	Beschreibung in der Sprache des Entwicklers
Externe Sicht des Systems	Interne Sicht des Systems
Strukturiert mit Hilfe von Use Cases: liefert Struktur für die externe Sicht	Strukturiert mit Hilfe von Klassen und Packages und Stereotypen; liefert Struktur für die interne Sicht
Wird vorallem eingesetzt als Kontrakt zwischen Kunden und Entwicklern; beschreibt das WAS und WAS NICHT	Wird vorallem von Entwicklern benutzt, um zu verstehen, wie das System ausgelegt werden soll, also designed und implementiert
Enthält eventuell Redundanzen, Inkonsistenzen u.ä. zwischen Anforderungen	Enthält möglichst keine Redundanzen, Inkonsistenzen u.ä. zwischen den Anforderungen
Hält die Funktionalität des Systems fest, auch Funktionalitäten, die für die Architektur relevant sind	Beschreibt, wie die Funktionalität im System implementiert werden soll, inklusive Architektur-Konzepten. Damit erhält man eine erste Form des Designs.
Definiert Use cases, welche im Analyse Modell verfeinert und analysiert werden	Definiert die Implementation der Use Cases, jeweils pro Use Case eine Analyse, ausgehend vom Use Case Modell

Zudem dient die Analyse als Basis für die Gesamtarchitektur des Systems. Deswegen spielen auch Faktoren wie Wiederverwendbarkeit, Wartbarkeit, Erweiterungsfähigkeit ... eine grosse Roll bei den Analyse-Tätigkeiten.

Workers und Artifacts der Analyse [und des Designs]:

(Zur Erinnerung: Artifakte / Artifacts sind Ergebnisse der einzelnen Aktivitäten)



Als Variante dieses Charts kann man folgende "Workers" und deren "Artifacts" identifizieren:

- Architekt : ist verantwortlich für die Architektur Beschreibung und das Analyse Modell
- Use-Case Engineer : untersucht die Use-Cases in Hinblick auf deren Realisierung
- Komponenten Engineer [an Stelle des Datenbank-Designers]: definiert die Analyse Packages und Analyse Klassen [an Stelle des Datenmodells]

14.2. Analyse in Kurzfassung

Analyse orientiert sich an einem konzeptionellen Modell dem *Analyse Modell*. Mit Hilfe dieses Modells wollen wir die Struktur, die Architektur des zu erstellenden Systems besser verstehen und definieren. In der Analyse werden die einzelnen Konzepte konkreter gefasst, präzisiert.

Deswegen verwenden wir auch andere Konstrukte und Visualisierungen:

- Interaktions-Diagramme helfen uns die Dynamik des Systems zu erfassen und zu beschreiben
- (Analyse) Klassen-Diagramme legen erste Strukturen auf Klassenebene fest
- Package Diagramme helfen uns die allgemeine Struktur des Systems zu visualisieren

Die Ergebnisse liefern Inputs für die nachfolgenden Aktivitäten : Design und Implementierung.

In diesem Sinne sind die Analyse Modelle erste Entwürfe, Versionen des Design Modells.

Ein wesentlicher Gesichtspunkt in der Analyse ist die Wiederverwendbarkeit und Wartbarkeit. Deswegen kommt auch der Komponenten Designer ins Spiel.

Auf der andern Seite machen wir in der Analyse Phase etliche Abstraktionen und Vereinfachungen, die erst wieder im Design und der Implementierung verfeinert und konkretisiert werden.

In Design und Implementierung kann sich unser Modell auch noch gehörig ändern:

- Die Wahl einer bestimmten Programmiersprache zwingt uns unter Umständen zu bestimmten Änderungen
- Bestehende Pakete sollen eventuell neu eingebaut werden, um die Wiederverwendung einzelner Packages zu erhöhen.
- Bestimmte Konzepte lassen sich auf einem Betriebssystem besser als auf einem andern umsetzen, beziehungsweise einige der Konzepte sind eventuell auf einer Hardware oder Systemsoftware bereits vorhanden.
- Bestehende Umsysteme können zu bestimmten Kompromissen führen
- Aus Kostengründen kann eine Implementierung anders aussehen, wenn bestimmte Rahmenbedingungen berücksichtigt werden müssen

All diese Gründe können dazu führen, dass das Analyse Modell im Design und speziell beim Implementieren zum Teil signifikant modifiziert werden muss.

14.2.1. Was unterscheidet Analyse von Design oder Implementation?

In einigen Standard-Lehrbüchern wird darauf hingewiesen, dass die Unterscheidung zwischen Analyse und Design schwierig ist.
Die selbe Ansicht wurde auch im Objectory Prozess von Rational vertreten. Dort sind Analyse und Design zusammengefasst. In der neueren Version "The Unified Software Process - The Complete Guide to the Unified Process" von Ivar Jacobson, Grady Booch und James Rumbaugh, sind die zwei Phasen wieder geteilt. Rational müsste sich vielleicht doch etwas einheitlicher und konsequenter zeigen!

Gemäss diesem (1999 erschienen) Buch gehören Anforderungen und Analyse auf der einen Seite, Design und Implementierung auf der anderen Seite dichter zu einander.

Design und Implementierung sind wesentlich mehr als Analyse (als Verfeinerung und Strukturierung der Anforderungen).
Deswegen macht eine Trennung sehr viel Sinn.

Design muss die Struktur des Systems liefern, mit seiner Architektur, in einer Form, die als stabile Plattform für spätere Erweiterungen und die Wartung dienen kann.
Design muss uns Antworten liefern auf Fragen wie :

- Performance des Systems
- Verteilung der Funktionalität

Und explizit Antworten auf die Fragen:

- Wie können wir das System so auslegen, dass die Antwortzeiten kleiner als 5 Milli-Sekunden sind?
- Wie kann diese Funktion so auf das Netzwerk verteilt werden, dass der Verkehr auf dem Netz nicht zusammen bricht.

Im Design müssen wir uns auch für eine bestimmte Datenbank entscheiden, für eine konkrete Programmiersprache, ein bestimmtes Betriebssystem und eine definierte Netzwerkumgebung.

Design und Implementierung ist wesentlich mehr als die Analyse von Anforderungen oder deren Verfeinerungen oder Strukturierung; Design und Implementierung definieren die Struktur (intern und extern) des Systems und müssen garantieren, dass alle Anforderungen an das System auch erfüllt werden können, auch die nicht-funktionalen.

Daher macht es Sinn, vor dem eigentlichen Bau des Systems das Verständnis über unser System zu vertiefen. Im Vergleich zum Design Modell ist das Analyse Modell etwa 20% so umfangreich wie das erstere.

In der Analyse werden auch allfällig vorhandene Klippen (hoffentlich) entdeckt und bereits Lösungen oder Umwege darum vorbereitet.

Vereinfacht gilt : Analyse beschreibt das WAS; Design beschreibt das WIE.

14.2.2. Was will man mit einer Analyse erreichen : Zusammenfassung

Die Analyse der Anforderungen in Form eines Analyse Modells ist aus folgenden Gründen wichtig:

- Ein Analyse-Modell liefert eine Präzisierung der Anforderungen, einschliesslich des Use Case Modells
- Ein Analyse-Modell beschreibt die Anforderungen aus Sicht des Entwicklers; es kann daher konkreter sein, formaler und abstrakter
- Ein Analyse-Modell strukturiert die Anforderungen so, dass die Struktur des zukünftigen Systems ersichtlich wird, insbesondere liefert das Modell auch Hinweise in Bezug auf Wartbarkeit und Erweiterbarkeit
- Ein Analyse-Modell kann vereinfacht gesagt als erster Entwurf für ein Design-Modell angesehen werden. Daher ist es ein Fundament, auf dem Design und implementation aufbauen.

14.2.3. Konkrete Beispiel für den Einsatz einer Analyse

Hier einige konkrete Beispiele für den praktischen Einsatz der Analyse und den Nutzen von deren Ergebnissen, dem Analyse-Modell:

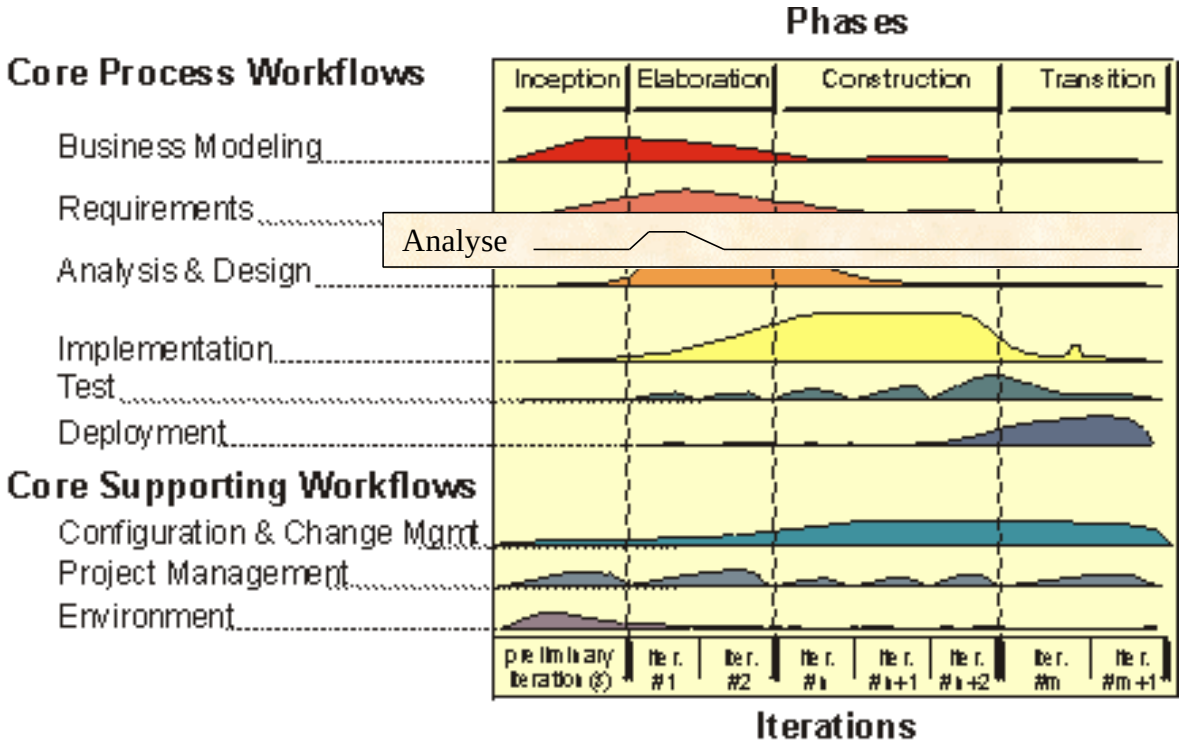
- Dadurch, dass wir die Analyse als selbständige Aktivität ansehen, können wir unabhängig vom Design und der Implementierung einen Grossteil des Systems analysieren und planen. Damit erhalten wir eine gute Basis für die Planung der Design und Implementierungs-Aktivitäten. Speziell im Falle des interativen Vorgehens müssen wir uns klar darüber sein, welche Systemblöcke wir in welcher Iteration realisieren möchten und können.
- Analyse liefert eine Gesamtsicht des neuen Systems in einer Form, die wir bei Design und Implementation (Programmcode) nicht mehr haben, auf einer Abstraktionsebene, die gerade weil viele Details weggelassen werden, leichter überschaubar ist (Design Modelle sind etwa 5 Mal komplexer als Analyse Modelle). Dieses gröbere Modell ist im Falle der Einarbeit neuer Mitarbeiter oder dem Training eine grosse Hilfe.
- Für einige Systemteile existieren unterschiedliche Designs und Implementierungen. Dies gilt speziell im Space Sektor, aber auch in andern Domänen. Ein anderes Beispiel ist unterschiedliche Standard-Software in unterschiedlichen Niederlassungen, deren Ergebnisse auf einer höheren Stufe konsolidiert werden müssen.
  In diesen Fällen werden immer Teile in einer andern Programmiersprache, auf einem anderen Betriebssystem, ... realisiert werden müssen. Wir haben also unterschiedliche Implementationen, aber zum Beispiel einen gemeinsamen Kontenplan.
- In vielen Fällen müssen alte Systeme "Reenginered" werden, um Erweiterungen zu ermöglichen, oder Teile auszutauschen. In diesen Fällen ist ein abstrakteres Modell als das Design Modell als Ausgangspunkt ideal.

14.3. Analyse im Software-Lebenszyklus

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.



Das Ziel der Analyse ist in jedem Projekt das selbe. Wie man aber konkret vorgeht, kann sich von Projekt zu Projekt ändern. Hier einige Beispiel:

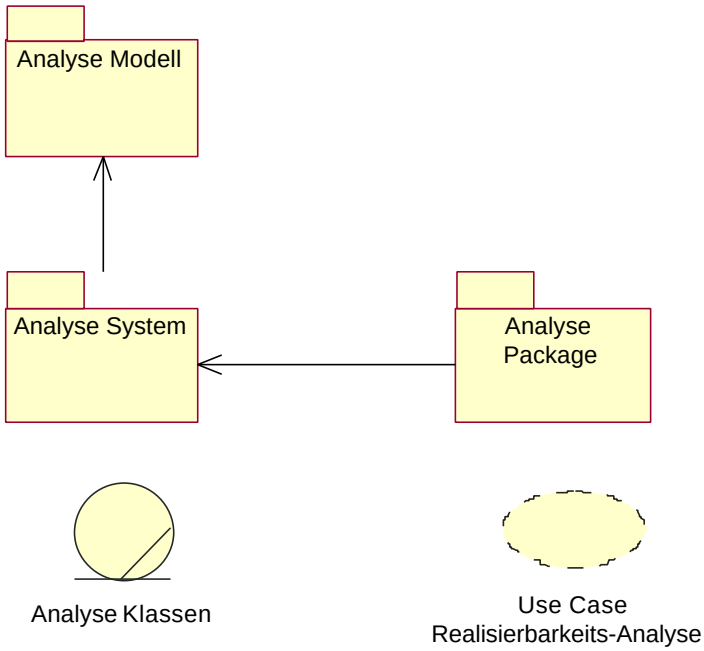
1. Im Projekt I wird das (noch im Detail zu besprechende) Analyse Modell eingesetzt, um die Ergebnisse der Analyse fest zu halten und um die Dokumentation über den gesamten Produktlebenszyklus konsistent zu erhalten. In jeder Iteration muss das Modell nach geführt werden, es dient aber gleichzeitig auch als Basis für die Planung der weiteren Phasen und Iterationen.
2. Im Projekt II wird das Analyse Modell als Zwischenschritt angesehen, als Vorbereitung auf die Erarbeitung des Design Modells und des Implementation Modells. Das Modell wird dann auch nicht mehr weiter entwickelt. Es wird als abgeschlossen angesehen.
3. Im Projekt III wird kein Analyse Modell erstellt. Die Teile, die typischerweise im Analyse Modell erfasst werden, werden in diesem Falle in das Anforderungs-Modell und / oder das Design-Modell integriert. Im ersten Fall legt der Kunde offensichtlich sehr viel Gewicht auf ein detailliertes Verständnis der Benutzeranforderungen, und geht vielleicht

fälschlicherweise davon aus, dass die Implementierung eher einfach sei. Im Falle der Integration mit dem Design geht man davon aus, dass die Anforderungen einfach oder bestens bekannt sind, also weniger zeitlichen Aufwand erfordern.

In den ersten zwei Fällen wird ein zusätzlicher Aufwand betrieben, den man im gesamten Projektablauf rechtfertigen muss.

14.4. Artifacts

14.4.1. Artifacts : Analyse Modell



Ein Analyse Modell besteht aus genau einem Analyse System (dem System zum Modell). Das System beziehungsweise sein Modell besteht aus mehreren Analyse Klassen (hier mit dem Stereotyp Icon, nicht mit dem Klassen-Diagramm dargestellt) UND aus mehreren Use-Cases, welche bereits in Hinblick auf deren Realisierbarkeit analysiert sind. Klassen und Use Cases bilden zusammen auf der einen Seite das System [dessen Modell] und auf der andern Seite das Analyse Package oder die Analyse Packages, als übergeordnete Grösse.

Analyse Klassen stellen Abstraktionen dar, aus denen unter Umständen im Design echte Klassen werden, aber unter Umständen auch ganze Subsysteme.

Use Case Realisierbarkeits-Analysen sind Use Cases, welche bereits auf den Analyse Klassen und deren Objekte aufbauen.

14.4.2. Artifact : Analyse Klasse

Artifacts :

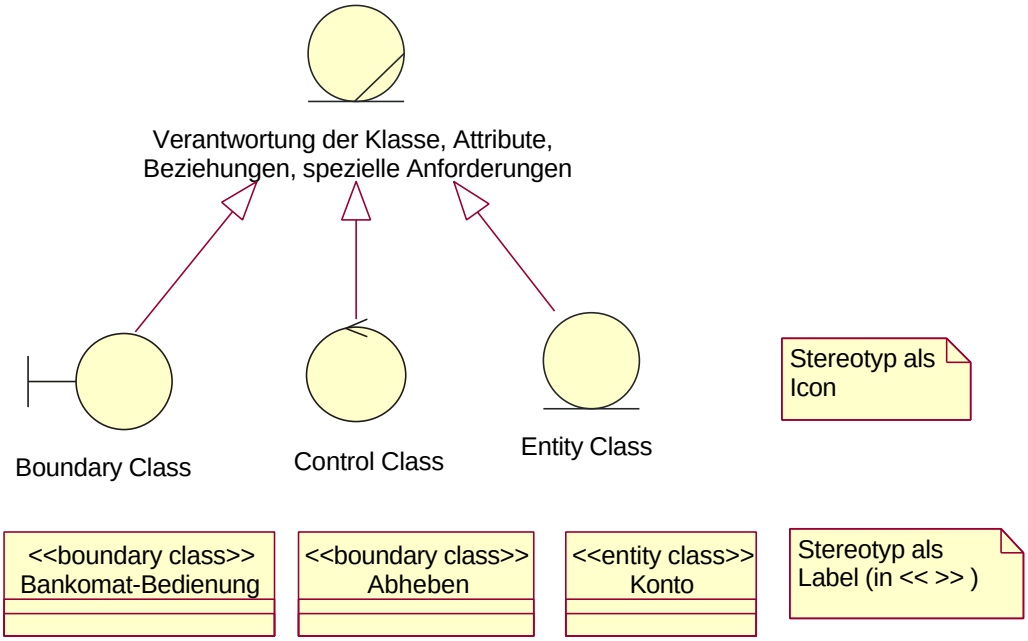
- ✓ Analyse Modell
- **Analyse Klassen**
- Use Case Umsetzung
- Analyse Packages
- Architektur Analyse

Eine Analyse Klasse ist abstrakter und allgemeiner als die Klassen, die wir schliesslich implementieren werden. Eine Analyse Klasse kann auf der Implementationsseite auf mehrere Klassen oder ein ganzes Subsystem abgebildet werden.

Wie lässt sich eine Analyse Klasse charakterisieren?

- Eine Analyse Klasse fokusiert sich auf funktionale Anforderungen und verschiebt Fragen betreffend nicht funktionale Aspekte auf die spätere Design und Implementierungs-Aktivitäten. Das Ziel ist es, die Klasse auf einer sehr hohen "konzeptionellen" Ebene darzustellen, mit entsprechender Ungenauigkeit im Vergleich zu Design und Implementations Klassen
- In der Regel spezifiziert das Analyse Modell keine Schnittstellen mit den entsprechenden Datenstrukturen und Methoden: deren Signatur. Das Zusammenspiel wird auf einer höheren Ebene informell beschrieben. Im wesentlichen wird die "Verantwortung" der Klasse beschrieben, im Sinne des CRC Modells (CRC : Class - Responsibility - Collaboration ; Ref : A Laboratory For Teaching Object-Oriented Thinking ,Kent Beck, Apple Computer, Inc.Ward Cunningham, Wyatt Software Services, Inc. auf dem Internet)
- Ein Analyse Modell definiert Attribute, aber ebenfalls auf einem sehr hohen Level, vergleichbar mit einem Datenmodell auf Entitätsmengen-Ebene. Diese Attribute ergeben sich aus dem Anwendungsgebiet, im Gegensatz zu den Attributen im Design und der Implementierung. Design und Implementierungs-Attribute ergeben sich oft aus den Möglichkeiten der konkreten Programmiersprache. Analyse Attribute werden im Design zum Teil ganze Design oder später Implementierungs-Klassen.
- Die Beziehungen der Klassen untereinander ist ebenfalls auf einer abstrakten Ebene zu verstehen. Im Extremfall wird aus einer Analyse Klasse ein Design Package. Die Bedeutung der Beziehung zweier Klassen kann sich entsprechend ändern.
- Analyse Klassen gehören zu genau drei Basis- Stereotypen:
  Systemgrenze - Boundary
  Kontroll - Controll
  Entitätsmenge - Entity

- Jede dieser Klassen besitzt ein eigenes Icon, eine eigene Darstellungsform in UML:

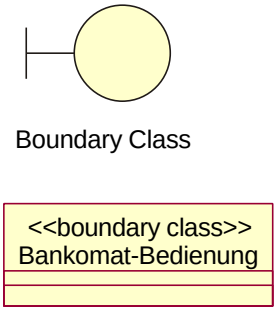


Es gibt noch eine dritte Form, bei der das Icon an Stelle des Label im Klassendiagramm auftaucht. Diese Darstellungsweise wird von Rose noch nicht unterstützt. Die obige existiert in Rose 98, nachdem man von Rational die RUP AddIns runter geladen und installiert hat.

Die Grundidee der Einführung einer limitierten Anzahl von Stereotypen besteht darin, dass dadurch der Mitarbeiter in der Analyse sich an drei Grundkonstrukten orientieren kann.

Wir wollen uns kurz mit den einzelnen Klassen befassen.

14.4.2.1. Schnittstellen/Systemgrenze-Klassen (Boundary Class)



Diese Klasse dient der Modellierung der Wechselwirkung zwischen dem System und den Actors, also dem Benutzer oder systemexternen Packages.
Die Wechselwirkung besteht in der Regel aus dem Empfang und der Präsentation von Informationen bzw. Anforderungen von und an externe Systeme.

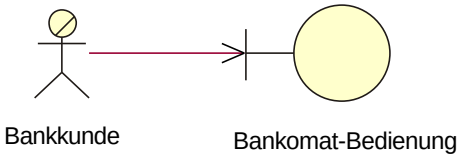
Systemgrenzen-Klassen modellieren die Teile des Systems, welche von den Actors anhängen und somit Informationen über die Systemgrenzen enthalten müssen.
Damit können Änderungen in der Benutzerschnittstelle einfach lokalisiert und lokal geändert werden, ohne grossen Einfluss auf das Gesamtsystem.

Systemgrenzen-Klassen stellen oft Abstraktionen von Windows, Druckern, Sensoren, Terminals, Kommunikations-Interfaces, ... APIs dar. Die Beschreibung der Systemgrenzen

muss allgemein gehalten werden, da es unwichtig ist, wie ein Benutzerinterface im Detail aussieht. Im Wesentlichen muss ja das WAS nicht das WIE festgehalten werden! Aus der Klasse bezw deren Beschreibung müssen wir also seine Funktion, nicht im Detail seine Funktionsweise erkennen können. Das WIE wird im Design und der Implementierung verfeinert und festgelegt.

Jede Systemgrenz-Klasse sollte mit einem Actor verbunden sein und umgekehrt.

Beispiel **Schnittstelle zu einem Zahlungssystem**

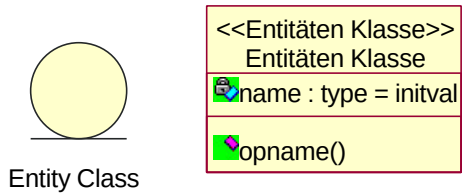


Der Bankkunde kann, nach Eingabe seiner Identifikationsnummer, seinen Kontostand abfragen, Geld abheben, Geld überweisen, eine Zahlung stornieren, ...

Als erstes reicht diese Art der Darstellung. Wir werden sie jedoch weiter verfeinern müssen und ergänzen. Dazu benötigen wir die folgenden Klassen.

14.4.2.2. Entitäten Klassen

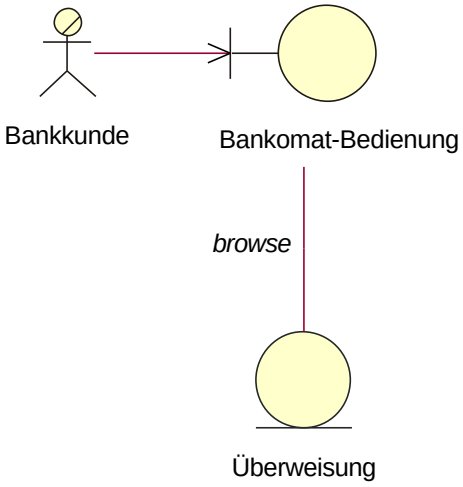
Eine Entitäten-Klasse repräsentiert Informationen, die langlebig sein sollen und oft auch gespeichert werden (persistent). Eine Entitäten Klasse beschreibt oft eine physisch fassbare Grösse, wie in der Datenbanktheorie (Entitäten im ER-Modell).



Entitäten Klassen werden aus den Business Klassen hergeleitet, oder aus einem Daten-Referenzmodell für die spezifische Applikationsdomäne. Allerdings stellt das Business Klassenmodell die Sicht des Anwenders dar, während die Entitäten-Klassen bereits aus der Sicht des Entwicklers beschrieben werden. Die Speicherung der Informationen und Implementierungs-bedingte Aspekte werden daher bei den Entitäten Klassen bereits berücksichtigt.

Zum Teil umfasst eine Entitäten Klasse auch ganze Abläufe, die bei der Implementierung genauer zerlegt werden. Das Ziel ist es jedoch, dass die Entitäten Klassen abstrakte Datentypen auf einem sehr hohen Abstraktionslevel darstellen.

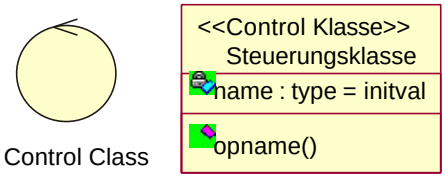
Beispiel **Schnittstelle zu einem Zahlungssystem : die Überweisung**



Der Bankkunde kann, nach Eingabe seiner Identifikationsnummer, seinen Kontostand abfragen, Geld abheben, Geld überweisen, eine Zahlung stornieren, ...
Er kann in diesem Beispiel durch seine Überweisungen scrollen und nachsehen, welche Überweisungen innerhalb einer bestimmten Periode gemacht wurden.

14.4.2.3. Control Klassen

Bleiben noch die Kontroll Klassen und deren Einbettung in das "Klassen-Portfolio", definiert durch unterschiedliche Stereotypen / Rollen.



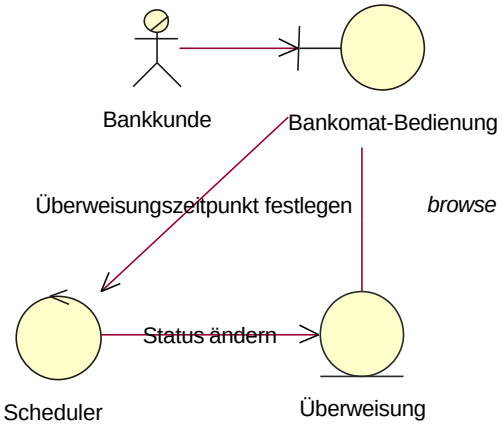
Kontroll Klassen stellen die Koordination, Sequenzen, Transaktionen und die Kontrolle anderer Objekte dar, also etwas, was wir typischerweise in den Methoden vermutet hätten, auf der Implementations-Stufe.

Kontrollklassen können auch komplexe Berechnungen zusammen fassen, also die Business Logik in einer 3 Tier Architektur (Frontend, Businesslogik, Backend).

Alle Aktivitäten, die mit dem Benutzer, dem Actor zu tun haben, werden nicht im Rahmen der Kontroll Klassen abgehandelt. Dafür gibt es die Systemgrenzen Klassen.

Beispiel **Schnittstelle zu einem Zahlungssystem : die Überweisung**

Der Bankkunde kann, nach Eingabe seiner Identifikationsnummer, seinen Kontostand abfragen, Geld abheben, Geld überweisen, eine Zahlung stornieren, ...
Er kann in diesem Beispiel durch seine Überweisungen scrollen und nachsehen, welche Überweisungen innerhalb einer bestimmten Periode gemacht wurden.



Zudem hat jetzt der Benutzer die Möglichkeit, automatische Überweisungen zu spezifizieren, die dann zu einem bestimmten Zeitpunkt automatisch ausgeführt werden (mit Hilfe eines Schedulers).

Dadurch erweitert sich unser Modell durch eine Kontroll Klasse.

Eine mögliche (grobe) Darstellung (eines 3 Tier Systems) zeigt das obige Diagramm.

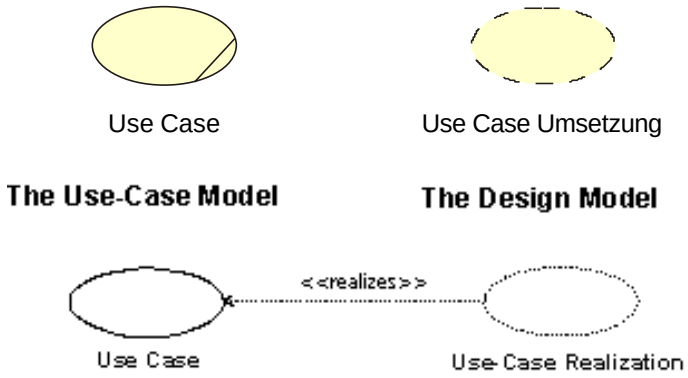
14.4.3. Artifact : Use Case Umsetzung - Analyse

Wenden wir uns nun dem dritten Artifakt zu (nach dem Analyse Modell und den Analyse Klassen) : der Use Case Umsetzung.

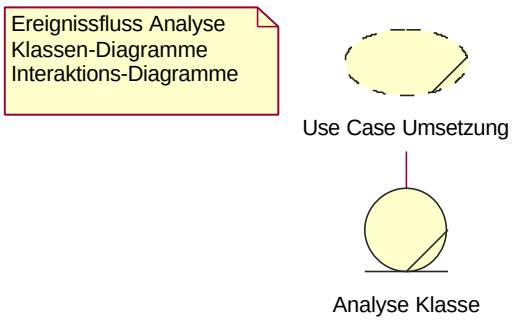
- ✓ Analyse Modell
- ✓ Analyse Klassen
- • **Use Case Umsetzung**
- • Analyse Packages
- • Architektur Analyse

Dieser Teil der Analyse dient der Abklärung, wie ein Use Case vermutlich implementiert werden kann. Im Wesentlichen versucht man, den Use Case aus der Systemspezifikation mit Hilfe von Analyse Klassen zu modellieren, und damit kann man eine erste Formalisierung erreichen.

Eine Use Case Umsetzung kann somit direkt mit einem Use Case in Verbindung gesetzt werden. Im folgenden Diagramm, aus dem RUP / ROP (file:/C|/Programme/Rational/Rational Unified Process 5.1/process/modguide/md_ucrea.htm) und wie bereits erwähnt mit der Mischung von Analyse und Design. Im Buch zur Methode steht an Stelle von "The Design Model" korrekterweise "Analysis Model".



Wichtig ist auch zu beachten, dass der Use Case durch ein ausgezeichnetes, die Use Case Umsetzung durch ein gestricheltes Oval repräsentiert wird:



Die Verbindung im englischen Diagramm dient nur der Illustration, sie stellt keine echte Verbindung im Sinne von UML (gestrichelt mit Pfeil = Instanzierung einer Klasse) dar, sondern soll lediglich die Beziehung Use Case zu Use Case Umsetzung (auch fast eine Instanzierung) verdeutlichen.

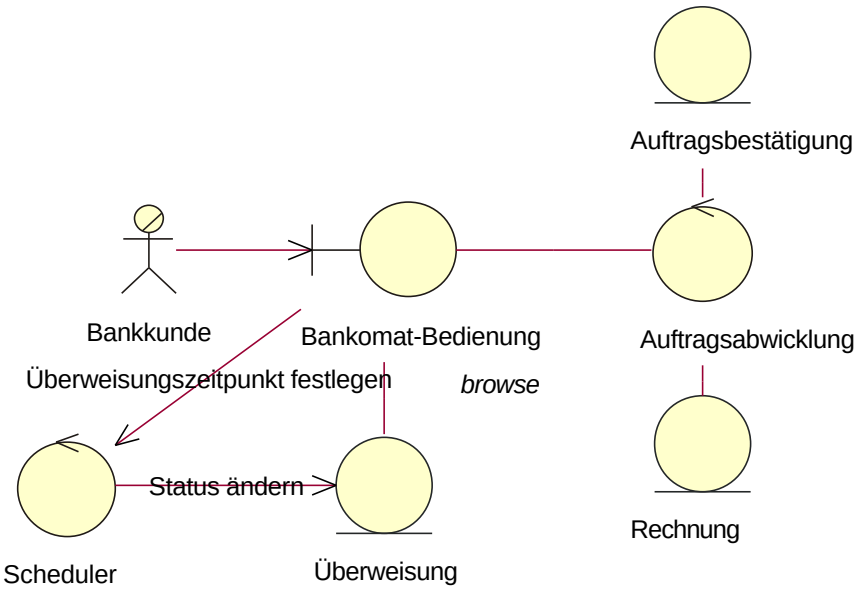
14.4.3.1. Klassen Diagramm

Eine Analyse Klasse wird in der Regel in unterschiedlichen Use Case Umsetzungen verwendet, eigentlich in möglichst vielen, damit die Wiederverwendung einzelner Komponenten möglichst hoch ist.

Einige der Attribute, Assoziationen, ... dieser Analyse Klassen können in unterschiedlichen Use Cases Umsetzungen unterschiedlich wichtig sein, oder überhaupt nicht benötigt werden.

Deswegen muss im Rahmen der Analyse genau geprüft werden, was konkret verwendet und der Klasse zugeordnet werden muss.

Die Diagramme sehen oft entsprechend verschachtelt aus. Wichtig ist daher, auf einer hohen Ebene zu modellieren. Dadurch wird es wesentlich leichter, die Übersicht zu behalten und eine brauchbare Architektur definieren zu können.



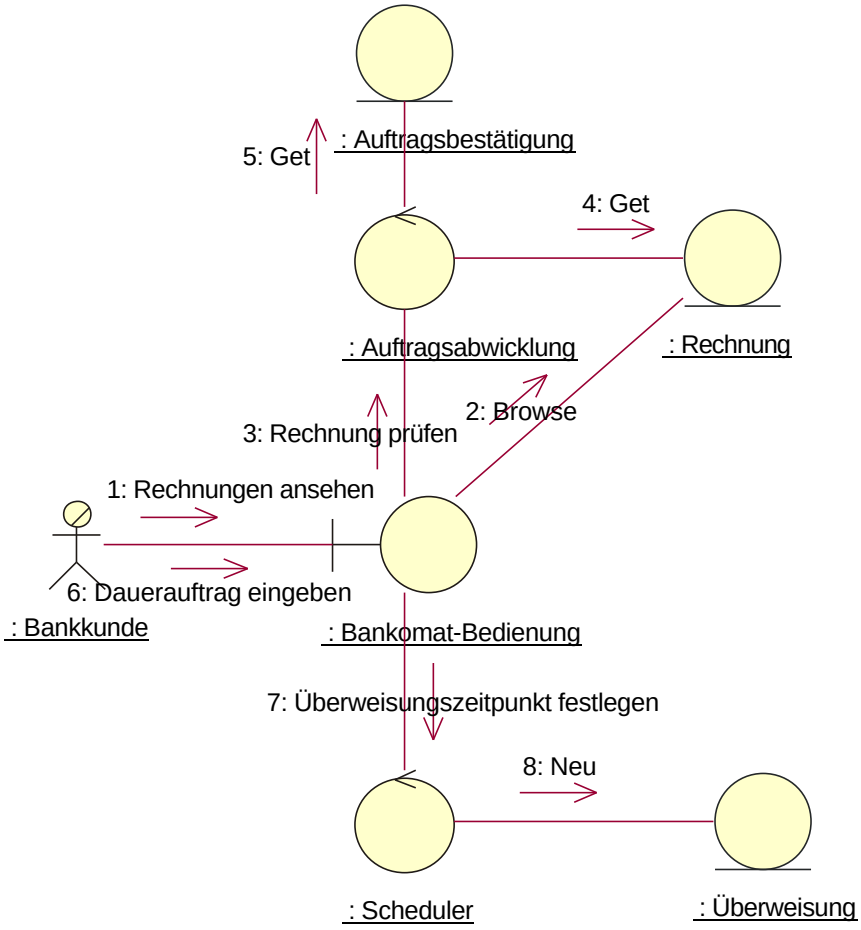
14.4.3.2. Interaktions Diagramm

Die Aktivitätensequenz mit der der Use Case beginnt, wenn der Benutzer den Use Case startet, lässt sich als Sequenz von Meldungen darstellen. Die erste Meldung stammt vom Benutzer. Nachher werden die Analyse Klassen Meldungen untereinander austauschen.

In der Analyse werden solche Aktivitäten mit Hilfe von "Kollaborations-Diagrammen" dargestellt. Der Grund liegt schlicht darin, dass die Sequence Diagramme bereits sehr spezifisch sind und damit in der Regel das Zusammenspiel einzelner (Design und Implementation) Klassen beschrieben und analysiert wird. Die passende Vergröberung stellt das Kollaborations-Diagramm dar, welches sich primär auf Aufgaben und Verantwortung (Responsibilties im CRC Modell) der Klassen beschränkt.

Das folgende Beispiel stellt ein solches Kollabroations-Diagramm dar, für die schon besprochene Situation der Banküberweisungen, mit Dauerauftrag.

Klar erkennbar ist, dass das Diagramm nicht genau beschreibt, welche Datenstrukturen ausgetauscht werden.



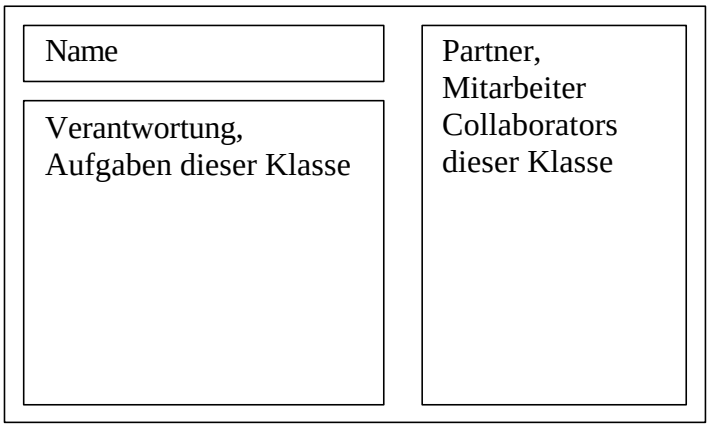
Ein weiterer Aspekt der Unterscheidung der drei Analysis Klassen zeigt sich bei einer Lebenszyklus-Analyse : wie lange leben die Objekte?

- Eine Systemgrenze Klasse können in mehreren Use Cases verwendet werden, zum Beispiel zur Beschreibung eines Benutzerinterfaces. Die Lebensdauer ist daher in der Regel so lange wie die eines Use Cases.
- Ein Entitäten Diagramm lebt oft viel länger als ein Use Case. Es muss sozusagen Use Cases überleben, da oft die Entitäten Klassen mit der Datenspeicherung zusammen hängen.
- Kontrollklassen beschreiben oft den Ablauf eines Use Cases. Allerdings haben wir bereits beim Scheduler gesehen, dass im Extremfall eine Kontrollklasse auch nach dem Use Case weiter leben kann oder dass verschiedene Kontrollklassen an einem Use Case beteiligt sind, wie ebenfalls das obige Beispiel zeigt.

14.4.3.3. Ereignisfluss - Analyse

Die Kollaborations Diagramme sind oft recht schwierig zu lesen. Man erkennt das problem bereits am einfachen Beispiel von weiter vorne.
Deswegen muss man öfters an Stelle eines unübersichtlichen Diagrammes einfach Text verwenden. Sinnvollerweise beschreibt dieser Text Objekte und Klassen, also Dinge, die wir später beim Design und der Implementierung wieder verwenden können.

Typischerweise orientiert sich ein solcher Text an der CRC Card:



Und typischerweise hat diese Karte Platz in einem Karteikasten, die Beschreibung muss also kurz und bündig sein.

Ein weiteres Beispiel einer möglichen Beschreibung in Form von Text wäre:

Der Bankkunde schaut seine Rechnungsliste nach, die ihm vom System zur Verfügung gestellt wird.
Der Dauerauftrag verwendet den Scheduler, um zu fest vorgegebenen Terminen Überweisungen zu tätigen.
... (reine Textbeschreibung der Abläufe)
Im Gegensatz zu einer Use Case Beschreibung, die aus Sicht des Anwenders geschieht, finden wir hier bereits System nähere Elemente (Scheduler , ...).

14.4.3.4. Spezielle Anforderungen

Spezielle Anforderungen sind Textbeschreibungen von Anforderungen, die sich aus Use Cases ergeben, aber die nicht den funktionalen Bereich betreffen.

Einige dieser Anforderungen wurden bereits bei der Beschreibung der Use Cases beschrieben, können also unverändert oder kaum verändert übernommen werden.

Beispiel

- Der Benutzer benötigt die Daten einer Transaktionasabfrage innerhalb von 0.5 Sekunden
- Die Rechnungen sollen mit Hilfe des SET Standards überwiesen werden

14.4.4. Artifact : Analyse Packages

Kommen wir zu den nächsten Ergebnissen / Artifacts der Analyse:

- ✓ Analyse Modell
- ✓ Analyse Klassen
- ✓ Use Case Umsetzung
- **Analyse Packages**
- Architektur Analyse

Wie sieht die Struktur eines Analyse Packages aus:
Ein Analyse Package besteht aus einer oder mehreren Analyse Klassen und einer oder mehreren Use Case Umsetzungen;
Zusätzlich kann ein Analyse Package auch weitere Analyse Packages verwenden (rekursiv)

Analyse Packages sollten kohäsiv sein, dh. sich mit einem klar umschriebenen Thema befassen und möglichst wenig Kopplung nach aussen haben.

In der Regel zeigen Analyse Packages folgende Charakterisierungen:

- Analyse Packages unterteilen das System in Analysebereiche, die mehr oder weniger unabhängig von einander behandelt werden können - unter Umständen parallel von verschiedenen Mitarbeitern mit unterschiedlichem Fachwissen
- Analyse Packages basieren in der Regel auf funktionalen Gesichtspunkten oder Problembereichen (Applikationen oder Geschäftsabläufen). Die Bereiche sollten von den Fachspezialisten als zu ihnen gehörig erkannt werden. Daher dürfen sie sich nicht auf nicht-funktionale Bereiche beziehen und schon gar nicht auf den Lösungsbereich, da die Analyse sich mit dem WAS, weniger mit dem WIE beschäftigen soll.
- Analyse Packages betreffen oft obere Layer im Design Modell also benutzernahe Bereiche. In Einzelfällen beschreibt ein Analyse Package einen oberen Design Layer.

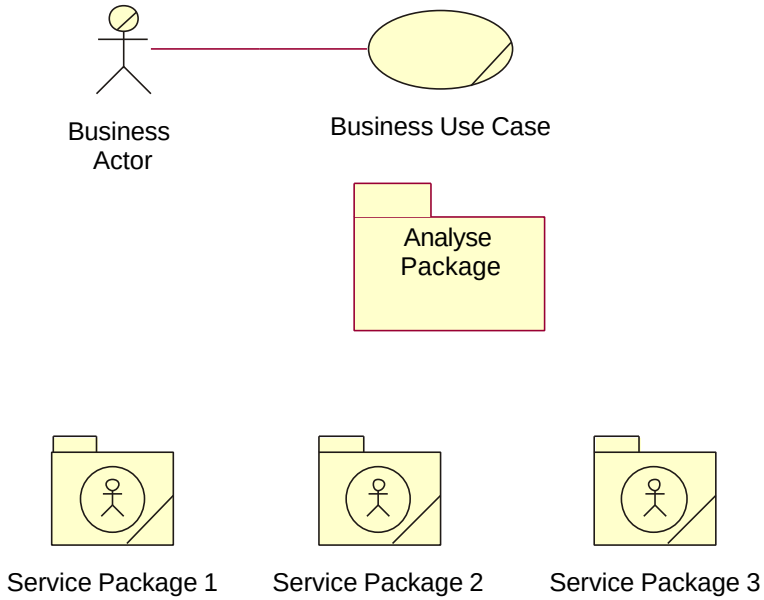
14.4.4.1. Service Packages

Damit das Konzept der Service Packages verständlich wird, müssen einige Begriffe erst geklärt werden. Auf der einen Seite haben wir den Actor, den externen Kunden unseres Systems; auf der andern Seite haben wir interne Funktionen oder Teilapplikationen, die bestimmte Funktionen oder Teilfunktionen benutzen. Diese Teilapplikationen bezeichnen wir als "Kunden" dieser Teilapplikation. Der Actor ist also der (externe) Kunde des Use Cases, beziehungsweise diverser Use Cases.

Die Hauptaufgabe des Systems ist es dem Actor Use Cases zur Verfügung stellen. Zudem stellt jedes System bestimmte Services zur Verfügung zum Beispiel die Rechtschreibprüfung in einem Textverarbeitungssystem.

Jede Teilapplikation, jeder Kunde gemäss der obigen Definition, benutzt einen bestimmten Mix von Services, mit deren Hilfe dem Actor bestimmte Use Cases zur Verfügung gestellt werden.

Anders ausgedrückt, hierarchisch dargestellt:



- Ein Use Case spezifiziert eine Sequenz von Aktionen : ein Thread wird durch einen Actor initiiert, dann folgt eine Interaktion zwischen dem Actor und dem System, und schliesslich endet der Thread, meistens in dem er bestimmte Daten an den Actor zurück gibt (Informationen, Charts, Daten,...).
  Aber Use Cases existieren nicht isoliert. Die Rechnungserstellung ist zum Beispiel nur dann möglich, wenn bestimmte Stammdaten und bestimmte Transaktionsdaten bereits vorhanden sind (Adressen, Konditionen, Auftargsdaten,...)
- ein Service besteht aus funktional kohärenten Aktionen - ein funktionales Package also - welches in verschiedenen Use Cases verwendet werden. Der Kunde (eine Teilapplikation) wählt also einen bestimmten Mix von Services aus, um dem Benutzer einen oder mehrere Use Cases zur Verfügung zu stellen.
  Ein Service ist unteilbar, er wird also entweder als Ganzes oder gar nicht ausgeführt und eingesetzt.
- Use Cases sind für Benutzer, Actors; Services sind für Kunden, Teilapplikationen. Use Cases verwenden verschiedene Services, ein Use Case verwendet also verschiedene Services.

Im Rahmen des Unified Processes (RUP) werden Services in Form von Service Packages verwendet. Service Packages sind primär Low Level Analyse Packages. Mit Hilfe von Services kann also das System auf einer tieferen Stufe strukturiert werden.

Folgende Facts über Service Packs / Packages gelten im Allgemeinen:

- Ein Service Package enthält eine bestimmte Menge funktional zusammenhängender Klassen
- Ein Service Package ist unteilbar. Jeder Kunde (Applikation) erhält entweder das gesamte Service Package oder gar nichts.
- Ein Use Case benutzt in der Regel ein oder mehrere Service Packages. Ein Service Package wird in der Regel in mehreren Use Cases eingesetzt.
- Ein Service Package hängt in der Regel nur sehr limitiert von anderen Service Packages ab.
- Ein Service Package kann auch nur für eine limitierte Anzahl Actors relevant sein.
- Die Funktionalität eines Service Packages ist in der Regel so ausgelegt, dass das Package als Einheit ausgeliefert und implementiert werden kann.
  Ein Service Package definiert in diesem Sinne ein "Add-In".
  Falls ein Service Package nicht an einem Use Case teilhaben darf, dann kann der Actor auch keinen andern Use Case verwenden, der diesen Service benutzt.
- Oft sind Services EXOR im Sinnen : Service Packages kapseln Funktionen, die sich gegenseitig nicht stören aber im Extremfall ausschliessen. Zum Beispiel ist die Rechtschreibprüfung ein Service Package. Wenn wir nun Englisch als Standard Sprache wählen, dann kann nicht gleichzeitig Deutsch geprüft werden.
- Oft spielen Service Packages eine wesentliche Rolle im anschliessenden Design und der Implementierung, da Service Packages sich mit Low Level Funktionalität befassen. Insbesondere bei der Zerlegung des Systems in sinnvolle Teilapplikationen und Subsysteme spielen Service Packages eine entscheidende Rolle.

Ein Nebeneffekt der Definition der Service Packages ist die Wartbarkeit unabhängiger Komponenten, der Austausch einzelner Komponenten ohne grosse Systemänderungen. Wir erhalten somit eine stabile, robuste Architektur unseres System.

Bleibt festzuhalten, dass es immer noch primär um die Definition der Analyse Packages geht in der Analyse; aber die Service Packages werden oft bereits in dieser Phase erkennbar.

14.4.4.1.1. Service Packages sind wiederverwendbar

Wie bereits im letzten Abschnitt abgetönt, bilden die Service Packages recht autonome Einheiten, die kaum oder nur lose gekopplet sind und separat ausgeliefert werden können. Dadurch werden die Service Packages zu Kandidaten für die Wiederverwendbarkeit.

Falls ein Service Package sich im Wesentlichen um eine Entitäten Klasse herum gruppiert, dann ist die Wahrscheinlichkeit gross, dass das Service Package auch in anderen Bereichen dieser Anwendungsdomäne einsetzbar ist.

Der Grund ist der, dass die Entitäten Klassen aus Business Klassen, also Benutzerproblemen heraus definiert werden.

Falls die Service Packages wie oben beschrieben in mehreren Use Cases eingesetzt werden, und die tieferen Layer des Systems beschreiben, dann ist fast garantiert, dass diese Packages mehrfach eingesetzt werden.

Als Konsequenz daraus ergeben sich auch wertvolle Hinweise auf die Architektur des Systems aus dem Analyse Modell.

14.4.5. Artifact : Architektur Beschreibung (aus Sicht der Analyse)

Kommen wir zu den letzten Ergebnissen / Artifacts der Analyse:

- ✓ Analyse Modell
- ✓ Analyse Klassen
- ✓ Use Case Umsetzung
- ✓ Analyse Packages
- **Architektur Analyse**

Die Architektur Beschreibung lässt sich entweder Top Down, zum Beispiel mit Hilfe von Architektur Patterns, oder Bottom Up, durch Analyse der Packages und Klassen erreichen. In der Praxis muss man gemischt vorgehen.

Die Architektur Beschreibung des Analyse Modells beschreibt in Form einer Architektur die wesentlichen Fakten des Analyse Modells.

Die folgenden Artifakte des Analyse Modells spielen in der Regel eine grosse Rolle bei der Definition der System Architektur:

- Die Zerlegung des Analyse Modells in Analyse Packages und deren gegenseitige Abhängigkeiten. Dies liefert in der Regel Hinweise auf Layer, die im Design und der Implementierung konkretisiert werden.
- Die Schlüssel Analyse Klassen, zum Beispiel die Entitätem Klassen, welche wichtige Aspekte des Systems kapseln und zusammenfassen, bilden wichtige Building Blocks für die System Architektur.
  Benutzerinterface, Bedienungsmechanismen werden in den Systemgrenzen Klassen zusammen gefasst.
  Wichtige Abläufe werden in Kontroll Klassen zusammen gefasst.
  Analyse Klassen sind normalerweise mit anderen Klassen vernetzt und zeigen somit Teile der Gesamtarchitektur.
  In der Regel reicht es, abstrakte Klassen zu betrachten, da es nur um die allgemeine Struktur geht, analog zum ER Modell in der Datenbanktheorie (dort startet man in der Regel auch mit Kernentitäten)
- Use Case Umsetzungen zeigen wichtige Funktionalitäten des Systems auf.
  Typischerweise werden dabei mehrere Analyse Klassen berücksichtigt und darunter liegende Services eingebunden.

14.5. Beteiligte Personen

Bevor wir uns den Abläufen der Analyse widmen, wollen wir noch die Personen kennzeichnen, die daran beteiligt sind.

Jede Aktivität (Requirement, Analyse, Design, Implementation, Test) kennt seine Spezialisten. In der Analyse werden , wie wir bereits am Anfang skizziert haben, folgende Personen, bei kleineren Projekten sicher in Personalunion.

- Architekt
- Use Case Engineer
- Komponenten Engineer

14.5.1. Beteiligte Person : Architekt

Der Architekt ist innerhalb des Workflows verantwortlich für die Integrität des Analyse Modells, um zu garantieren, dass das Analyse Modell korrekt, verständlich und konsistent bleibt.

Verantwortung des Architekten in der Analyse:

- Analyse Modell
- Architektur Beschreibung
  erstellen, unterhalten, weiterschreiben

Bei mehrfacher Iteration wird diese Funktion unter Umständen zur Routine und kann damit an jemand anderes delegiert werden (in Personalunion). Aber der Architekt trägt weiter die Verantwortung und muss die Arbeiten koordinieren und überwachen.

Das Architektur Modell ist korrekt, falls das System die Funktionalität beschreibt, die in den Use Cases erfasst wurde, und nichts mehr.

Der Architekt ist auch verantwortlich für das Analyse Architektur Modell, also der speziellen Sicht auf das Analyse System, eben aus der Sicht des Architekten.

Der Architekt ist nicht für die Fortschreibung des Use Case Modells und weiterer Artifacts zuständig. Diese Aufgaben werdem vom Use Case Engineer oder dem Komponent Engineer wahrgenommen.

14.5.2. Beteiligte Person : Use Case Engineer / Designer

Der Use Case Engineer ist verantwortlich für die Integrität der einzelnen (aller) Use Case Umsetzungen. Er muss dafür sorgen, dass die Anforderungen der Actors an die Use Cases auch tatsächlich berücksichtigt werden.

Ein Use Case Umsetzung muss die Situation, die im Use Case beschrieben wird, das Verhalten des Systems aus Benutzersicht, korrekt wieder geben. Alle Beschreibungen müssen verständlich und nachvollziehbar sein. Die Diagramme, falls welche eingesetzt werden, müssen klar und verständlich sein (oder sonst fehlen).

Die Verantwortung des Use Case Engineers ist somit:

- Use Case Umsetzung Analyse

Der Use Case Engineer ist nicht verantwortlich für die Analyse Klassen und den Beziehungen dieser Klassen untereinander, welche bei der Realisierung des Use Cases eingesetzt werden.

14.5.3. Beteiligte Person : Komponenten Ingenieur

Der Komponenten Engineer definiert und unterhält die Verantwortlichkeiten, Attribute, Beziehungen und spezielle Anforderungen einer oder mehrerer (aller) Analyse Klassen.

Dadurch soll garantiert werden, dass alle Anforderungen aus der Use Case Analyse auch umgesetzt werden.

Der Kompoenten Engineer ist auch verantwortlich für die Integrität der verschiedenen Analyse Packages. Das impliziert auch die Verantwortung für das korrekte Zusammenfassen und die Verantwortung für den Inhalt der Packages, die Analyse Klassen.

Die Verantwortung des Komponenten Engineers ist somit:

- Analyse Klassen
- Analyse Packages

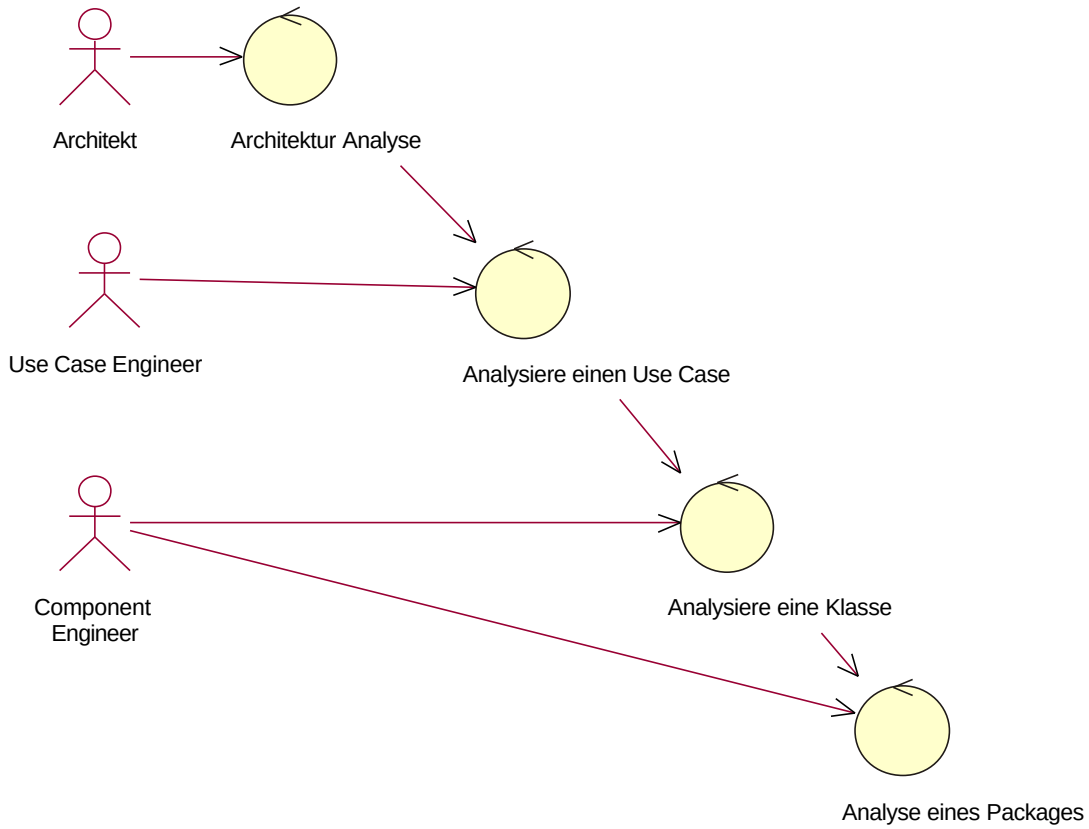
In späteren Aktivitäten (Design, Implementierung) wird die gleiche Person für die entsprechenden Konzepte innerhalb dieser Aktivitäten verantwortlich sein.

14.6. Workflow

Bisher haben wir uns erst mit den statischen Beschreibungen der Analyse beschäftigt. Jetzt wollen wir uns mit der Dynamik, den Abläufen befassen.

Beteiligte	*1. Aktivität*	*2. Aktivität*	*3. Aktivität*	*4. Aktivität*
Architekt	Architektur Analyse			
Use Case Engineer		Analyse der Use Cases		
Komponenten Engineer			Analyse der Klassen	Analyse der Packages

Das Analyse Modell wird zuerst vom Architekten skizziert, in dem er die wesentlichen Analyse Packages identifiziert oder versucht zu identifizieren, zudem die wichtigsten Entitäten Klassen und allgemeine Anforderungen. Dann versucht der Use Case Engineer Use Cases umzusetzen, also mit Hilfe der beteiligten Analyse Klassen und dem entsprechenden Verhalten zu formalisieren. Die Anforderungen der Analyse Klassen (Methoden und Datenfelder, soweit bereits formulierbar) werden vom Komponenten Engineer (oder dem DB Designer) zusammen getragen, formalisiert und entsprechende Kooperationen, Verantwortungen den Klassen, bzw. zwischen den Klassen zugeteilt bzw. definiert.
In dieser Phase muss der Architekt versuchen iterativ sein Modell zu verbessern und zu einem



stabilen Modell zu gelangen. Das Gleiche gilt für die Packages oder Komponenten, die am Anfang vage definiert sind und dann konkretisiert werden müssen.

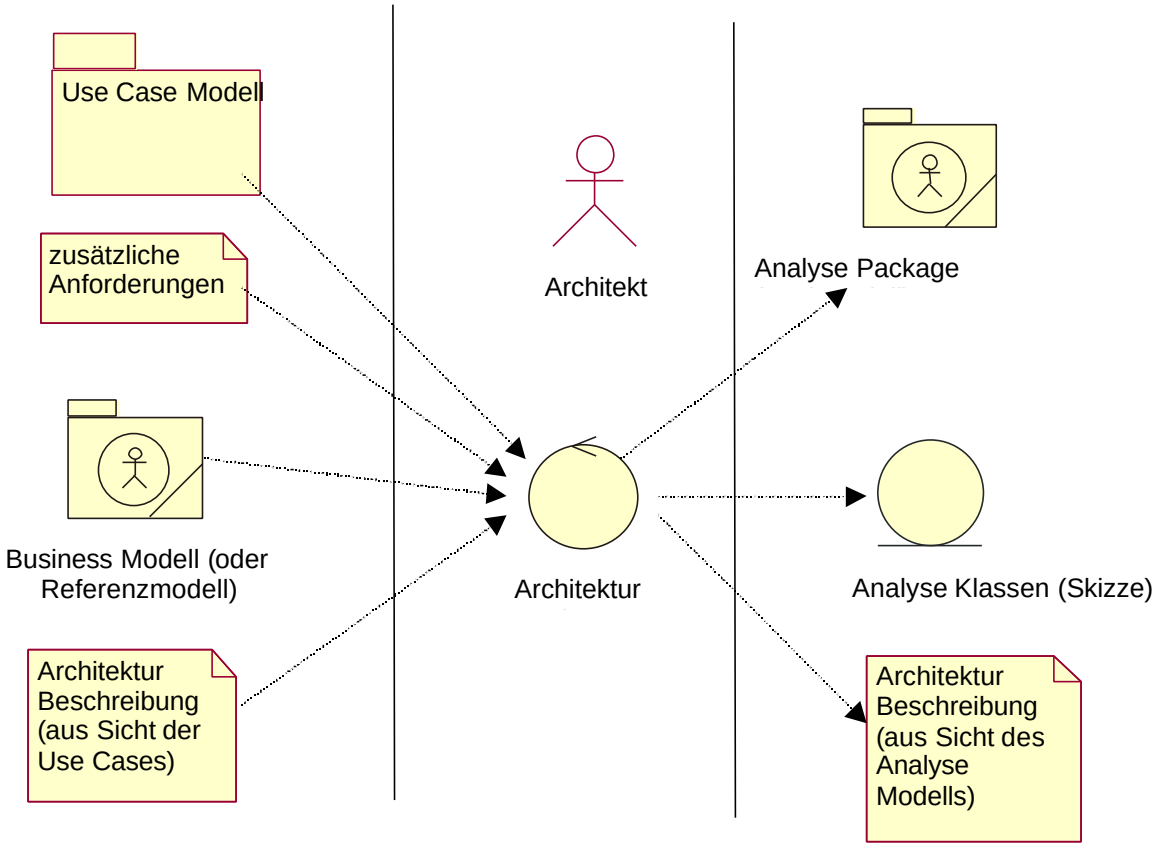
14.6.1. Aktivität : Architektur Analyse

Das Ziel der Architektur Analyse ist es das Analyse Modell (Klassen, Packages) und die (erst grobe) Architektur beschreibung zu erstellen.

Schematisch:

Input	Aktivität und Akteur	Ergebnisse / Output
Use Case Modell	Architekt	Analyse Package(s) [Gobmodell]
Zusätzliche Anforderungen		
Business Modell / Referenzmodell	Architektur Analyse	Analyse Klassen [Grobmodell]
Architekturbeschreibung aus Sicht der Use Cases		Architektur Beschreibung (aus Sicht der Analyse)

In Rose finden Sie ein Architektur Template (zum Beispiel C:\Programme\Rational\Rational Unified Process 5.1\wordtmpl**templates**\a_and_d**softwarearchitecturedocument.**doc)



14.6.1.1. Identifikation der Analyse Packages

Mit Hilfe von Analyse Packages wird das Analyse Modell in überschaubare Teile zerlegt.

Wie kommt man zu diesen Packages? Top Down auf Grund von Architektur Patterns oder anderer globaler Betrachtungen (Standard Software, funktionalen Blöcken, Middleware, ...); Bottom Up durch Analyse der Analyse Klassen und Verdichtung, Abstraktion.

Da wir die funktionalen Anforderungen mit Hilfe von Use Cases erfassen, besteht der direkte Weg der Analyse darin, zu versuchen, mehrere Use Cases zu einem Package zusammen zu fassen, entweder die Use Cases als Ganzes, oder aber viel wahrscheinlicher, durch Zusammenfassung von Services, auf denen die Use Cases aufbauen, aus denen die Use Cases bestehen:

- Use Cases, die einen bestimmten Geschäftsprozess unterstützen, werden zusammen gefasst.
- Use Cases, die einen bestimmten Actor betreffen, werden gruppiert und nach Möglichkeit zusammen gefasst.
- Use Cases, die auf Grund von "Generalisierung" und "Erweiterung" zusammen gehören, also kohärent sind, im Sinne : die Use Cases spezialisieren sich gegenseitig oder erweitern sich gegenseitig.

Durch solche Zusammenfassungen erreichen wir eine erhöhte Änderungsfreundlichkeit: da die Use Cases zusammen hängen, wirken sich Änderungen und Erweiterungen in der Regel auf alle beteiligten Use Cases aus.

Allerdings werden diese Zusammenfassungen bei einer genaueren Analyse oft wieder aufgebrochen. Use Cases nutzen in der Regel Klassen aus verschiedenen Packages. Es zeigt sich im Verlaufe der Analyse und spezieller noch im Design, dass andere Zerlegungskriterien für eine Implementierung günstiger sein können. Trotzdem ist die Definition der Analyse Packages wesentlich, da aus Benutzersicht diese Zerlegung primär von Interesse ist und nicht eine auf der Technologie basierende Zerlegung.

Beispiel **Identifikation eines Analyse Packages**

Das System habe folgende Use Cases:

1. Rechnung bezahlen
2. Mahnung versenden
3. Rechnung stellen

Alle Use Cases gehören zum gleichen Business Prozess : Verkauf (von der Bestellung bis zur Warenlieferung).

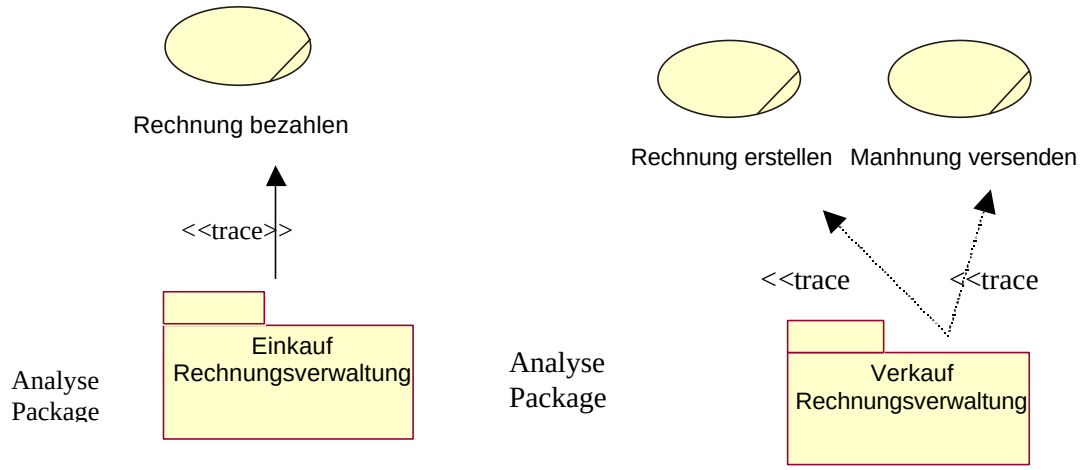
Wir könnten also EIN Analyse Package definieren.

Das Projektteam beschliesst einen anderen Weg zu gehen:

- Das System soll auch anderen Firmen angeboten werden (Mandanten)
- Diese Firmen setzen jeweils nicht das gesamte System, sondern nur Teile davon ein.
  - Einige Firmen verwenden nur den Einkausfteil der Applikation
  - Einige Firmen verwenden nur den Verkaufsteil der Applikation
  - Einige Firmen verwenden sowohl Einkauf als auch Verkauf

Die Firma beschliesst deswegen, zwei Packages zu definieren und separat zu bearbeiten:

- Ein Einkaus-System zur Bearbeitung von (eingehenden) Rechnungen
- Ein Verkaufs-System zur Bearbeitung von (ausgehenden) Rechnungen



14.6.1.1.1. Behandlung von Gemeinsamkeiten von Analyse Packages

In vielen Fällen wird man in der Analyse feststellen, dass bestimmte Gemeinsamkeiten in den Packages bestehen. Zumindest auf Analyse Klassen Ebene wird es grössere Überlappungen geben, hoffentlich!

Eine Variante, mit solchen Klassen umzugehen, besteht darin, diese in separate Packages zu stecken, also zu isolieren, da dadurch eine gemeinsame Nutzung erleichtert wird.

Solche Klassen, die gemeinsam genutzt werden, sind in der Regel für Entitäten Klassen. Sie können in der Regel auf "Business Entitäten Klassen", also Entitäten der realen Welt, zurück verfolgt werden.

Von daher lohnt es sich, im Hinblick auf eine effiziente Implementierung und Wartbarkeit der Applikation, die Anwendungs-Domäne genau zu analysieren.

Beispiel Identifikation genereller Analyse Packages

Das System habe folgende Use Cases:

4. Rechnung bezahlen
5. Mahnung versenden
6. Rechnung stellen

Als Actors wurden identifiziert

- Bankkunden (Kunden, die ausschliesslich über die Bank Geschäfte abwickeln), mit einem Bankkonto
- Normale Kunden, mit einem Kundenkonto

Es zeigt sich, dass die Verwaltung der Bankkunden und der normalen Kunden völlig analog erfolgen kann.

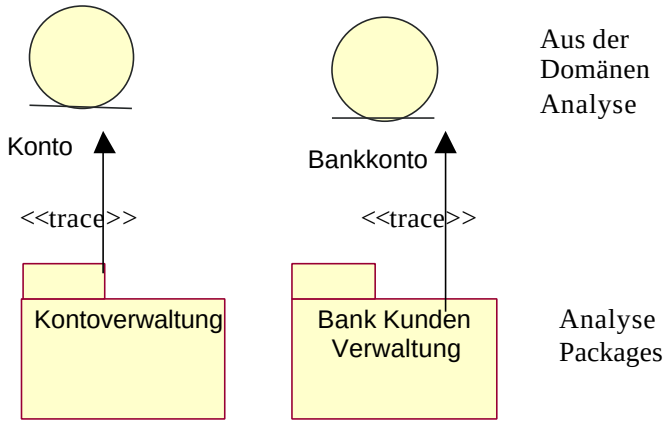
Auf der andern Seite erkannte das Projektteam, dass die Bankkontoverwaltung auch in andern Teilsystemen genutzt werden kann.

Das Projektteam beschliesst folgenden Weg zu gehen:

- Es wird ein Kontomanagement-Package für Privatkunden erstellt
- Zusätzlich wird ein Bankkunden Kontomanagement Package erstellt

Vergleichen Sie das folgende Diagramm mit dem obigen:

- Oben haben wir die Packages mit den Use Cases verbunden
- Hier leiten wir aus den Packages auf Grund der Anwendungs-Domäne Entitäten Analyse Klassen her.



Die beiden Packages werden viele Klassen gemeinsam nutzen. Das Projektteam muss also im Folgenden versuchen, diese Gemeinsamkeiten heraus zu arbeiten, um möglichst wiederverwendbaren Code zu finden.

14.6.1.1.2. Identifikation von Service Packages

Service Packages wird man in der Regel zu einem späten Zeitpunkt in der Analyse identifizieren, dann wenn man Gemeinsamkeiten erkennen kann.

Bekanntlich müssen alle Analyse Klassen eines Service Packages zum selben Service gehören, gemäss Definition eines Service Packages.

Folgendes Vorgehen hat sich als brauchbaren Weg zur Identifikation der Service Packages erwiesen:

1. Definiere für jeden Service ein unabhängiges Package, als Ausgangspunkt
2. Identifiziere alle Service Packages, welche optionalen Charakter haben. Diese können eventuell in andere Service Packages verschoben werden.
   Das wichtigste Kriterium ist Kohärenz und Kohäsion : pro Service Package darf nur EIN Dienst realisiert werden, um die Änderungsfreundlichkeit zu erhalten, oder einzubauen.
   Suche Service Packages, die voneinander abhängen.
   Beispiel:
   - Wenn Klasse A eliminiert wird, dann brauchen wir Klasse B auch nicht : Klasse B hängt von Klasse A ab
   - Wenn wir Klasse B ändern, dann müssen wir auch Klasse A ändern: wie oben

---

Beispiel Identifikation von Service Packages, welche funktional zusammen gehörige Klassen zusammen fassen.

Wir gehen vom selben Business Case aus wie oben.

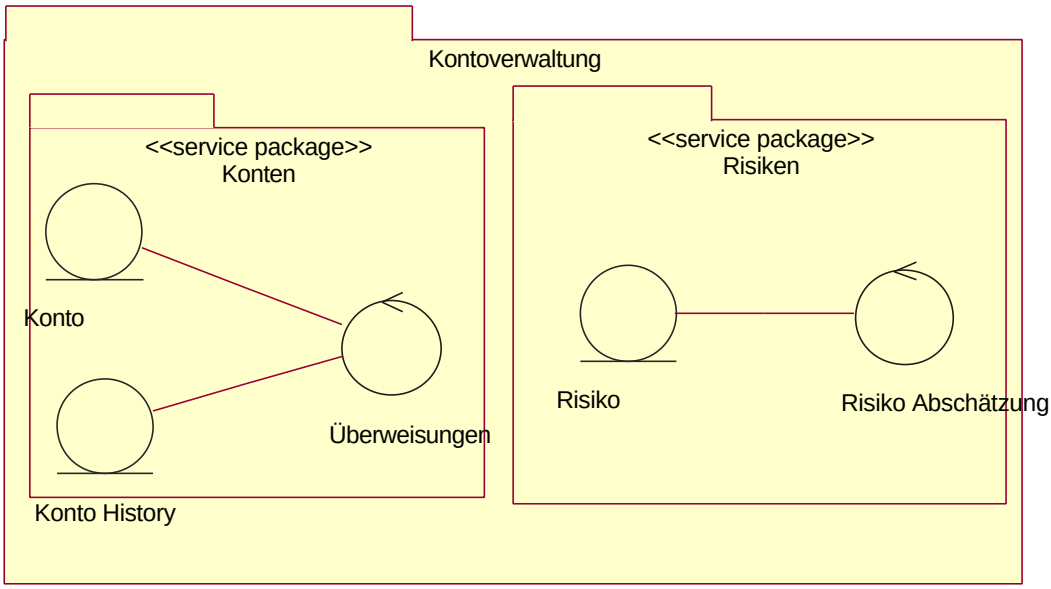
Es zeigt sich, dass die Verwaltung der Bankkunden und der normalen Kunden völlig analog erfolgen kann.

Auf der andern Seite erkannte das Projektteam, dass die Bankkontoverwaltung auch in andern Teilsystemen genutzt werden kann.

Für das Package "Kontomanagement" wird eine Detailanalyse durch geführt mit folgenden Ergebnis:
Die Kontoverwaltung besteht eigentlich aus zwei potentiellen Packages:

- Einem Konto Anteil
- Einem Risiko Management Anteil

Das Projektteam zerlegt deswegen das Kontomanagement Package wie folgt:



14.6.1.1.3. Definition von Analyse Packages Abhängigkeiten

Damit die Kohärenz und Kohäsion der Packages optimiert werden kann, müssen die Abhängigkeiten der Analyse Klassen aufgezeigt werden. Das Ziel ist es, möglichst in sich abgeschlossene Packages zu finden, also mit möglichst nur internen engen Kopplungen.

Diese Analyse wird in der Regel durch ein schichtenweises Vorgehen erleichtert:
Die Applikation wird mit Hilfe des Layer Architektur Patterns in mehrere logisch sinnvolle Ebenen zerlegt. Kohäsion und Kopplung wird auf jeder Ebene separat untersucht.

Typischerweise orientiert sich ein Ebenen Modell mindestens an den drei Layern :

- Benutzerebene
- Middleware / Support Ebene
- Systemebene

Je nach Applikation (vergl. ISO OSI Modell) sind weitere Layer zwingend.

Beispiel **Abhängigkeit der Analyse Packages und Ebenen**

Wir gehen vom selben Business Case aus wie oben.

Das Projektteam identifiziert im Paket Kontomanagement einige Klassen, wie zum Beispiel das Konto Management, Klassen aus anderen Paketen benutzt.

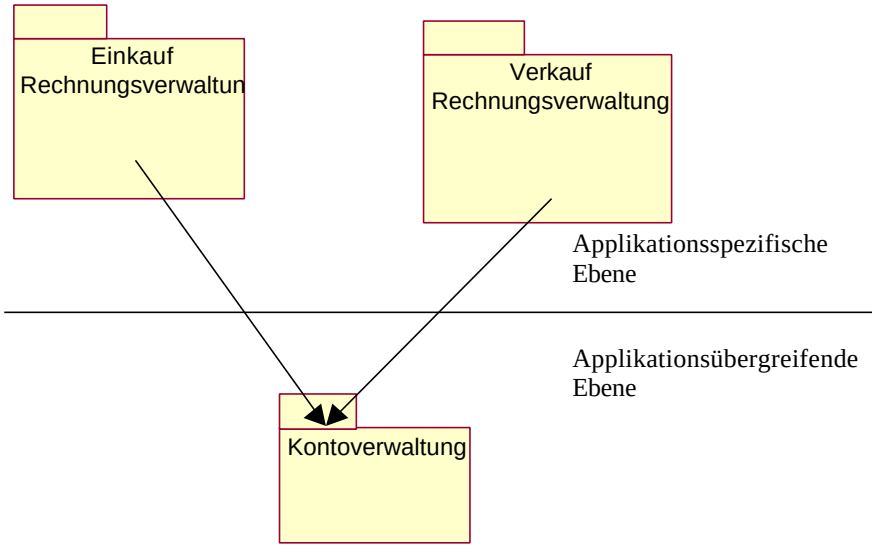
Beispiel:

- Die Kontoklasse wird auch von der Rechnungsverwaltung und der Verkaufsverwaltung benutzt

Das Projektteam definiert deswegen zwei Ebenen:

1. Applikatorische Ebene mit spezifischen Ausprägungen
2. Applikatorische Ebene mit genereller Ausprägung

Das resultierende Paket-Diagramm sieht wie folgt aus:



Diese Ebenen sind erst vorläufig und werden im Laufe des Designs und der Implementation weiter verfeinert.

14.6.1.2. Identifikation von Entitäten Klassen

Der besser Übersicht halber sollte man sich bei der Identifikation der Entitäten Klassen auf Kernentitäten im Sinne der Datenbanktheorie beschränken (siehe Datenbank Theorie, Details mit vielen Beispielen sind im Buch von Max Vetter zu finden).

Dadurch wird das gesamte Modell überschaubar gehalten. Oft muss man dafür iterativ vorgehen: zuerst werden zu viele Entitäten (Entitäten Klassen) gefunden; diese müssen anschliessend sinnvoll zusammen gefasst werden, in der Regel durch Abstraktion.

Im Rahmen des Designs und der Implementierung muss man dann wieder auf die Details zurück kommen; im Rahmen der Analyse bereits bei der Umsetzungung der Use Cases, aus der in der Regel die meisten Details erkennbar werden (müssen).

Die Aggregation und Assoziation der Klassen aus dem Domänen Modell (Referenzmodell, Kernentitäten Modell, ...) dient als Richtschnur bei der Definition der Assoziationen und Aggregationen der Analyse Klassen (und später der Design und der Implementierungs Klassen).

Beispiel **Abhängigkeit der Analyse Packages und Ebenen**
Wir gehen vom selben Business Case aus wie oben.

Aus dem Anwendungsgebiet ergibt sich, dass das Konto sicher eine der Entitäten ist, die alle Analyse und Design Aktivitäten (Vereinfachungen, Abstraktionen, Verdichtungen, Verallgemeinerungen, ...) überleben wird.

Einige der Attribute ergeben sich auch recht natürlich:
- Kontostand
- Datum der letzten Transaktion
- Währung
- Konto-Nummer

Auch die Beziehung zwischen Kunde und Konto wird alle weiteren Aktivitäten überleben, da sie sich direkt aus der Realitätsbeobachtung ergibt und das Informationssystem diese spiegeln muss.

14.6.1.3. Identifikation von allgemeinen speziellen Anforderungen

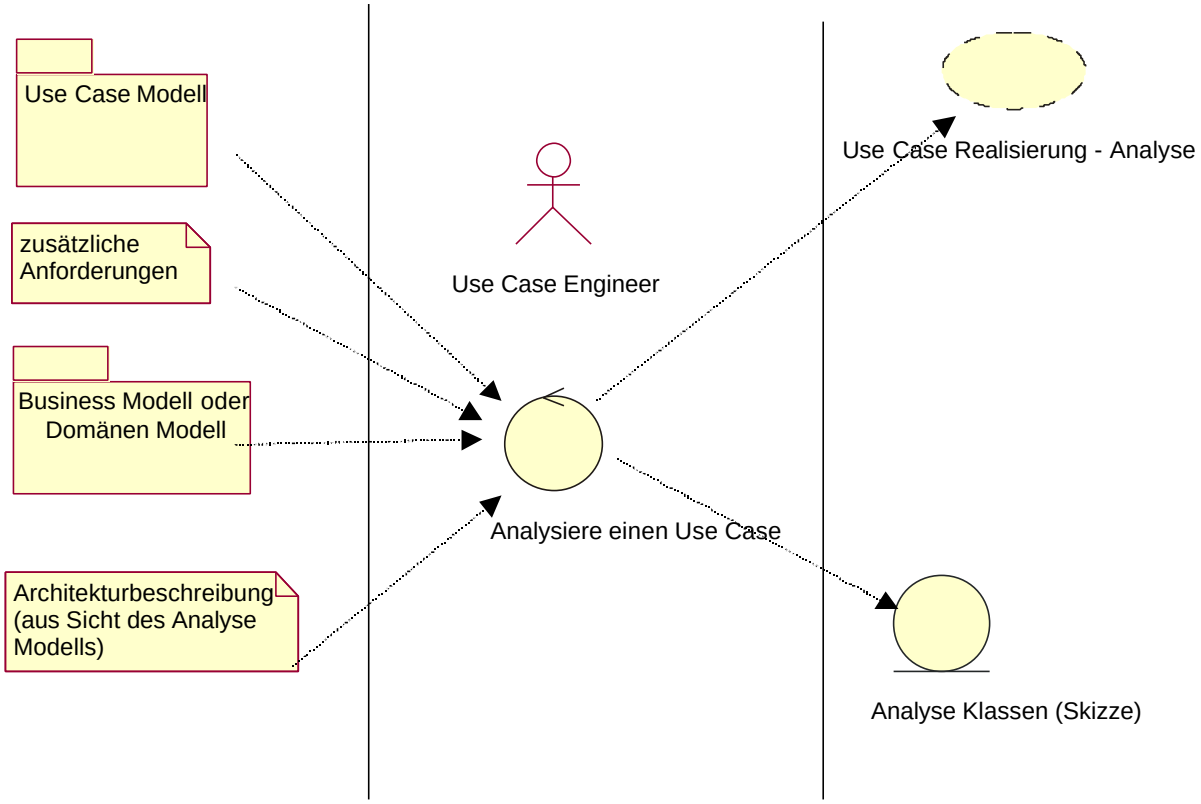
Spezielle Anforderungen sind Anforderungen, die im Laufe der Analyse deutlich werden, wie zum Beispiel:
- Persistenz : die dauerhafte Speicherung eines Zustandes / von Daten (Zustand = Wertebelegung der Attribute eines Objektes)
- Applikationsverteilung und Concurrency
- Security Features
- Fehler Toleranz, Ausfallsicherheit
- Transaktionsmanagement

Der Architekt muss diese Anforderungen möglichst vollständig erfassen, da sie bei der Realisierung zwingend berücksichtigt werden müssen, oft aber beim Modellieren eher störend die Modelle überladen würden.

Spezielle Anforderungen ergeben sich oft aus der Use Case Analyse, sofern die Use Cases umfassend erfasst wurden und vollständig beschrieben sind.

Beispiel **Identifikation der Schlüsselmerkmale einer speziellen Anforderung**
Für die Persistenz ergeben sich typischerweise folgende Anforderungen:
- *Grösse:* wie umfangreich sind die Objekte, die gespeichert werden sollen
- *Volumen :* wie viele Objekte / Daten werden gespeichert werden müssen
- *Persistenz Periode :* das Zeitintervall, während dem ein Objekt typischerweise persistent gehalten werden muss
- *Mutationsintervall :* Häufigkeit der Veränderung der Objekte
- *Zuverlässigkeit :* wie sicher soll ein Objekt gespeichert werden? was soll für den Fall eines Crashes vorgesehen werden?

14.6.2. Aktivität : Analyse der Use Cases



Die Use Cases müssen analysiert werden, um:

- Die Analyse Klassen zu finden, mit deren Hilfe (bzw. deren Objekte) die Use Cases als Abläufe dargestellt werden können
- Das Verhalte, welches im Use Case beschrieben wurde, auf die Objekte zu verteilen und deren Wechselwirkung aufzuzeigen
- Spezielle Anforderungen, die sich nicht sinnvoll in Form von Objekten und / oder Klassen ausdrücken lassen, fest zu halten

An Stelle von Use Case Analyse wird in der Fachliteratur auch der Begriff "Use Case Verfeinerung" verwendet.:

- Wir verfeinern den Use Case als eine Kollaboration, ein Zusammenarbeiten, von Analyse Klassen.

14.6.2.1. Identifikation von Analyse Klassen

Die Identifikation der Analyse Klassen, als einer der Schritte im Workflow "Analyse" dient dazu, die Entitäten, Kontroll und Boundary Klassen zu finden, um dann den Use Case realisieren zu können:

- Im Sinne der CRC Methode:

1. C : Class / Namensgebung für die Klasse
2. R : Responsibility / wozu dient die Klasse
3. C : Collaboration / Beziehungen der Klasse zu anderen Klassen

- Finden der Attribute

Die Use Cases werden in der ursprünglichen Form oft zu ungenau erfasst (Requirements sind oft auch nicht im Detail bekannt : es kann sein, dass der Benutzer gar nicht über die Möglichkeiten der Technologie informiert ist).

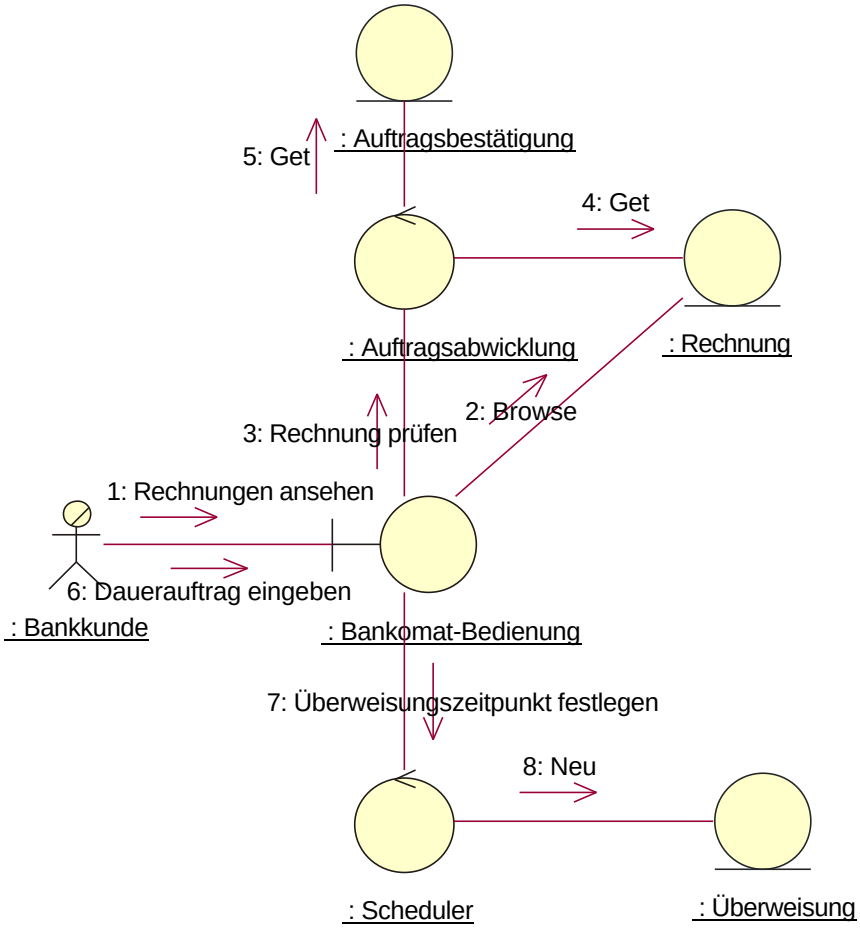
Die Struktur des Systems selber ist in der Beschreibung des Use Cases völlig nebensächlich, da die Beschreibung stur aus Sicht des Anwenders geschieht.

Die Verfeinerung muss nun die nötigen Details liefern.

Allgemein hat sich folgendes Vorgehen im Sinne einer generellen Vorgehensweise bewährt:

- Identifiziere die Entitäten Klassen, indem die Use Cases analysiert werden. Zusätzlich sollte ein Domänen-Modell, falls vorhanden (in Form eines Referenzmodells, in Form eines Industriestandards, in Form von Empfehlungen), beigezogen werden..
  Hauptproblem:
  was kann als Entität, was als Attribut, was als Kontroll Klasse, was als Systemgrenz Klasse verwendet werden?
  Es gibt in der Regel mehrere Lösungen; man kann unterschiedliche Modelle eines bestimmten Systems erstellen, je nach Gesichtspunkt, je nach der Wichtigkeit einzelner Kenngrössen (Entitäten bzw. Attribute).
- Pro Actor muss eine Systemgrenzklasse definiert werden, die Schnittstelle mit der es der Actor, der Nutzer unseres Systems zu tun haben wird : das User Interface.
  Diese Klasse sollte nach Möglichkeit :
  wiederverwendbar sein
  gemäss internen oder generellen Standards erstellt werden (GUI Guidelines)
  die Firmenstandards berücksichtigen (diese sind oft wesentlich detaillierter als die allgemeinen GUI Richtlinien der Hersteller [Microsoft, Motif und ähnliches]).
  Zum Teil lassen sich solche Boundary Classes aus weiteren Bauteilen zusammen setzen (Framesets im Web, Widget Libraries,...).
- Pro Entitäten Klasse sollte eine Systemgrenzklasse gesucht werden:
  wir wollen ja irgendwie die Klasse nutzen können!
  Diese Grundklassen (Boundary Classes) müssen weiter verfeinert und detailliert werden, dienen aber als sinnvoller Ausgangspunkt für eine detaillierte Beschreibung der System Schnittstelle.
- Pro Actor sollte eine zentrale Schnittstellen Klasse definiert werden:
  die direkt vom Benutzer, dem Actor, eingesetzte Schnittstelle als Kommunikations Schnittstelle.
  System Actors (Sensoren, andere Applikationen, Terminals, Alarm Einrichtungen,...) werden gleich wie die "menschlichen" Actors behandelt.
  Falls die Kommunikation auf mehreren Ebenen erfolgt (Beipiel ISO OSI Modell), dann muss pro Ebene eine Schnittstellen Klasse definiert werden, sofern die Kommunikation auf mehreren Ebenen möglich sein soll.
- Identifiziere eine Kontrollklasse, die den Ablauf des Use Cases regelt. Die Kontrollklasse muss also die Dynamik des Use Cases beschreiben und ermöglichen.
  In einigen Fällen ist es vorteilhaft, das Verhalten in Schnittstellen Klassen zu kapseln, die Kontroll Klasse wird dann überflüssig.
  In anderen Fällen kann die Kontroll Klasse so umfangreich sein, dass sie aus mehreren Klassen zusammen gesetzt werden muss.

Die Klassen, die man im Analyse Klassenmodell bereits identifiziert hat, müssen möglichst wieder verwendet werden. Das bestärkt die Richtigkeit des Designs. Am Schluss resultiert ein



Klassenmodell der Use Case Umsetzung, welches möglichst alle Klassen aus der Analyse umfassen sollte. Wichtiger als Vollständigkeit ist aber die Übersichtlichkeit!

14.6.2.2. Beschreibung der Wechselwirkung zwischen Analyse Objekten

Sobald wir die Analyse Klassen identifiziert haben, müssen wir beschreiben, wie die Objekte dieser Klassen miteinander wechselwirken. Dazu werden Collaboration / Kollaborations Diagramme eingesetzt. Wir haben weiter vorne bereits ein Beispiel dafür gesehen : Kollaborations Diagramme beschreiben die Wechelwirkung der Objekte auf eher informelle Art und Weise, im Gegensatz zu Sequence Diagrammen, die explizit Methoden der teilnehmenden Objekte zeigen.

Ein Kollaborations Diagramm beschreibt den Actor, die beteiligten Analyse Klassen und deren Wechselwirkungen.

Falls der Use Case komplex ist, dann lohnt es sich der Übersicht halber die einzelnen Abläufe separat aufzuzeichnen.

Praktisches Vorgehen zum Erstellen eines Kollaborations Diagrammes:

1. Ausgangspunkt ist der Use Case: man beginnt also beim Actor.
   Der Actor sendet eine Message zum Systemgrenze-Objekt.
2. Jedes Analyse Objekt des Use Cases sollte mindestens einmal verwendet werden. Das Kollaborations Diagramm muss also diese Analyse Objekte miteinander verbinden.

Sofern eine Klasse nicht verwendet wird, muss untersucht werden, ob diese nicht ganz aus dem System eliminiert werden kann.

3. Die Meldungen / Messages, die zwischen den Objekten hin und her geschickt werden, sind nicht identisch mit den Methoden auf Java Ebene.
   Aus der Meldung sollte hervor gehen, warum dieses Objekt eingesetzt wird, nicht technische Details. Es muss klar werden, welche Rolle das Zielobjekt im Gesamtsystem spielt, nicht wie die Methode im Details aussieht. Die Analyse findet hier also durchaus auf dem CRC Level statt.
4. Die Links zwischen den Objekten stellen oft Instanzen von Assoziationen zweier Klassen dar. Diese Assoziationen sind entweder schon definiert, oder es werden spezielle Rollenverhalten dieser Assoziationen beschrieben.
5. Die (zeitliche) Sequenz des Ablaufes ist eher nebensächlich in folgendem Sinne:
   wir wollen die Sequenzen darstellen, keine Echtzeit. Die Beziehungen der Objekte untereinander stehen im Vordergrund.
6. Kollaborationsdiagramme müssen alle Beziehungen der Analyse Objekte untereinander visualisieren, aber auch die Beziehung der Use Cases untereinander.
   Wenn zum Beispiel Use Case B eine Spezialisierung des Use Cases A ist, dann muss dies auch auf Stufe Collaboration Diagramm erkennbar sein.

Unter Umständen macht es Sinn, Kollaborationsdiagramme durch Text zu ergänzen, speziell bei komplexen Diagrammen. Diese Texte sollten nach Möglichkeit direkt im Diagramm eingetragen werden, nicht separiert in einem speziellen Dokument.

14.6.2.3. Festhalten spezieller Anforderungen

In Textform werden allfällige spezielle Anforderungen erfasst und in sinnvollem Detail festgehalten.

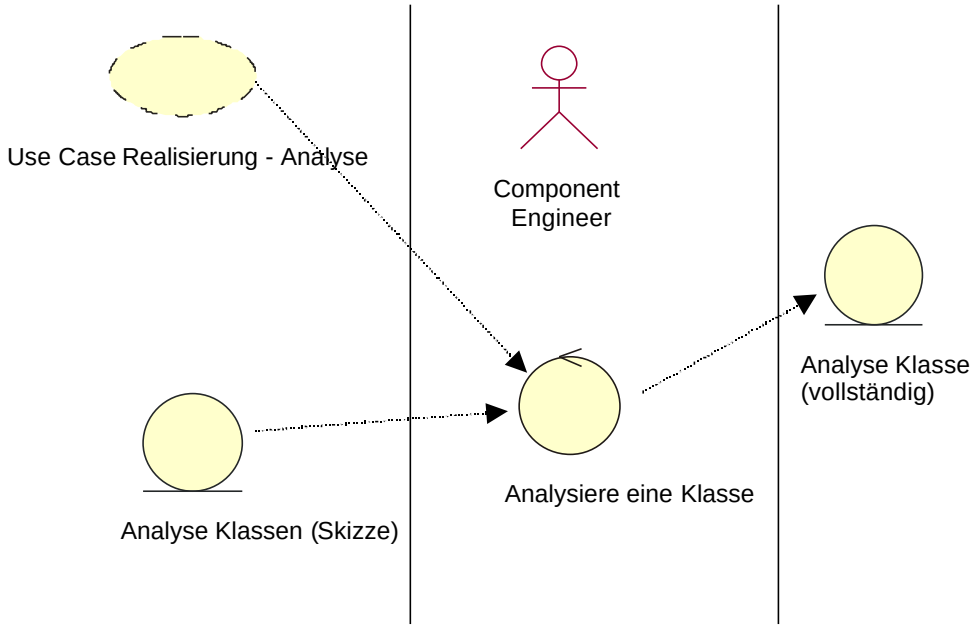
Beispiel Spezielle Anforderungen einer Use Case Umsetzung

Für das Kontenverwaltungsystem gelten folgende speziellen Anforderungen:

- Die Rechnungsklasse muss persistent sein
- die Auftragsabwicklungsklasse muss in der Lage sein, bis zu 10'000 Transaktionen pro Stunde abwickeln zu können.

14.6.3. Aktivität :Analyse einer Klasse

Die Aktivität lässt sich wie folgt symbolisch festhalten:



Der Ablauf besteht aus folgenden Schritten:

1. Identifikation der Zuständigkeiten der Analyse Klasse, basierend auf der Funktion / Rolle, die sie in der Use Case Umsetzung spielt.
2. Festhalten der Attribute der Analyse Klasse
3. Festhalten der Beziehungen zwischen den Analyse Klassen : Assoziationen und Aggregationen
4. Falls möglich und nötig : Identifikation möglicher Generalisierungen
5. Festhalten spezieller Anforderungen betreffend der Realisierung der Analyse Klasse

14.6.3.1. Festhalten der Zuständigkeiten / Responsibilities

Die Zuständigkeiten einer Klasse lassen sich dadurch ermitteln, indem man sich ansieht, welche Rolle diese Klasse in allen Use Cases spielt.

Durch die Analyse der Abläufe und Interaktionen der Klassen, wird ersichtlich, welche Methoden von einer bestimmten Klasse erwartet werden. Dazu muss auch die Beschreibung der speziellen Anforderungen heran gezogen werden, da diese bei der Realisierung, der Implementierung kritisch werden.

Beispiel Klassenrollen
Rechnungsobjekte werden im Use Case "Rechnungserstellung" erstellt.
Der Verkäufer führt diesen Use Case aus, um den Käufer anzuweisen, die Bestellung zu bezahlen (der Use Cas zur Erstellung der Bestellung kann zum Beispiel "Warenbestellung" oder "Servicebestellung" sein.
Im Verlaufe dieses Use Cases (Rechnungsstellung) muss die Rechnung zum Käufer gelangen.
Der Käufer muss die Rechnung anschliessend bezahlen.
Die Bezahlung wird im Use Case "Rechnung begleichen" beschrieben. Dort wird mit Hilfe eines Schedulers eine Überweisung ausgelöst. Dadurch wird die Rechnung bezahlt und das Rechnungsobjekt geschlossen.
Das selbe Objekt muss somit am Use Case "Rechnungserstellung" und "Rechnungbegleichung" teilnehmen.

Die Zuständigkeiten der Klassen lassen sich ähnlich bestimmen:
Eine mögliche einfache Methode besteht darin, die Zuständigkeiten aus den Rollen herzuleiten. Unter Umständen müssen dabei bestimmte bereits bestehende Zuständigkeiten ergänzt, überschrieben oder modifiziert werden.

Beispiel Klassen Zuständigkeiten
Der Überweisungsscheduler ist für folgende Aktivitäten zuständig:
- Kreiere eine Zahlungsaufforderung
- Verfolge die Zahlungen, die vorgesehen sind, und sende eine Mitteilung, sobald die Überweisung aktiviert / durch geführt wurde.
- Initialisiere den Geldtransfer zum vorgesehenen Zeitpunkt
- Sende eine Mitteilung, falls eine Zahlung ausgeführt wurde oder fällig ist.

14.6.3.2. Identifikation der Attribute

Ein Attribut spezifiziert eine Eigenschaft einer Analyse Klasse. Oft wird ein Attribut durch die Zuständigkeiten der Klasse vorgegeben oder deutlich gemacht.

Die Hinweise helfen die Attribute sinnvoll zu dfefinieren:
1. Attribute sind Nomen, Dingwörter
2. Der Attributtyp darf nicht durch die Implementation mit Hilfe eines bestimmten Datenbanksystems oder sonst was vorgegeben werden:
   es genügt zu fixieren, dass der Kontostand zur Entität Konto gehört. Es muss spezifiziert werden, dass der Kontostand vom Typ Integer ist.
3. Die Anzahl Attributtypen sollte möglichst gering gehalten werden.
4. Falls ein Attribut von mehreren Klassen benötigt wird, dann spricht vieles dafür, dass dieses "Attribut" verselbständigt werden sollte.
5. Falls eine Klasse zu kompliziert wird, weil sie zu viele Attribute umfasst, dann muss man sich überlegen, ob es nicht sinnvoll und praxisnaher wäre, die Klasse in mehrere Klassen zu unterteilen.
6. Attribute einer Entitäten Klasse sind in der Regel leicht zu finden. In der Regel ergeben sich diese aus dem Anwendungsfall (Use Case)
7. Attribute einer Boundary Class sind häufig Datenelemente, die vom Benutzer manipuliert werden können.

8. Control Klassen haben selten Attribute : ihre Hauptfunktion ist die Ablaufkontrolle. Typische Attribute einer Kontrollklasse dienen der Speicherung von Zwischenwerten.
9. Unter Umständen kann man eine formale Definition von Attributen in der Analyse durch eine einfache Beschreibung der Funktion der Klasse ersetzen.
10. Klassendiagramme lassen sich in fast jedem Tool in unterschiedlichem Detail darstellen:
    mit allen Attributen
    mit physikalischen Darstellungen der Attribute
    ohne Attribute, zur Erreichung einer übersichtlicheren Darstellung.

14.6.3.3. Identifikation von Assoziationen und Aggregationen

Analyse Objekte arbeiten zusammen, indem sie Meldungen austauschen. Im Kollaborationsdiagramm wird dies durch Verbindungen der Klassen bzw. der Objekte dargestellt.

Wie bereits erwähnt sind diese Verbindungen im Kollaborationsdiagramm oft Instanzen einer Aggregation oder Assoziation der Klassen.

Der Komponenten Designer muss daher diese Kollaborationsdiagramme im Detail untersuchen.

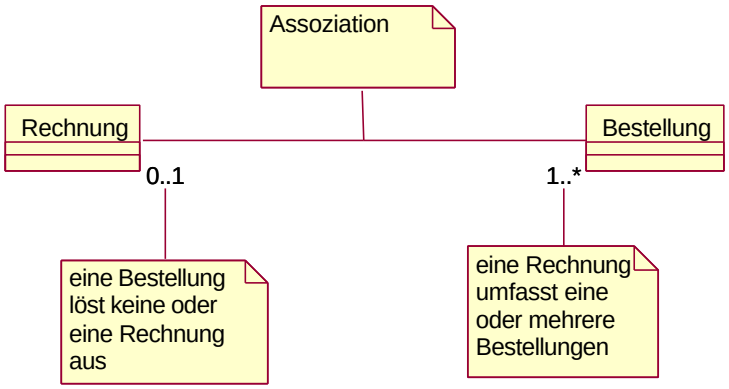
Klassen sollten so selbständig wie möglich sein, also möglichst wenige Assoziationen und Assoziationen umfassen (wegen Wartbarkeit, Fehlerfortpflanzung, Erweiterbarkeit, ...).

Die volle Ausgestaltung der Assoziationen und Aggregationen erfolgt erst im Design, dh. einfach vor der Implementierung.

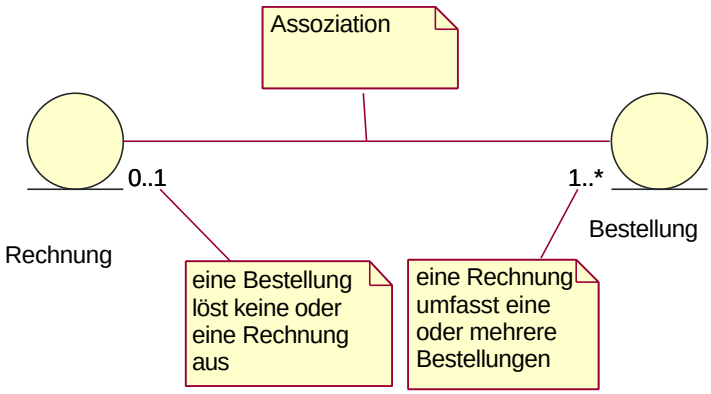
Der Komponenten Engineer muss auch die Mächtigkeiten der Beziehungen analysieren. Daher kann die Funktion "Komponenteningenieur" und "Datenbankdesigner" auch zusammen gelegt werden. Die Aktivitäten sind ähnlich, konzeptionell etwa auf dem gleichen Level.

Beispiel Eine Assoziation zwischen Analyse Klassen

Eine Rechnung wird jeweils für mindestens eine Bestellung erstellt: "1..*" (mindestens eine)
Eine Lieferung umfasst jeweils eine oder mehrere Bestellungen : wieder 1..*. Visualisierung:



Das gleiche Diagramm aber mit Icons / Stereotypen:



14.6.3.4. Identifikation von Generalisierungen

In der Analyse muss alles daran gesetzt werden, zu Generalisierungen zu gelangen. Dadurch werden die Modelle überschaubar und die Implementierung einfacher.

Beispie Ein Beispiel aus einem Grossprojekt

Die Firma beschliesst ein Client Server Paket zu entwickeln. Je ein Team befasst sich mit den Geschäftsprozessen :

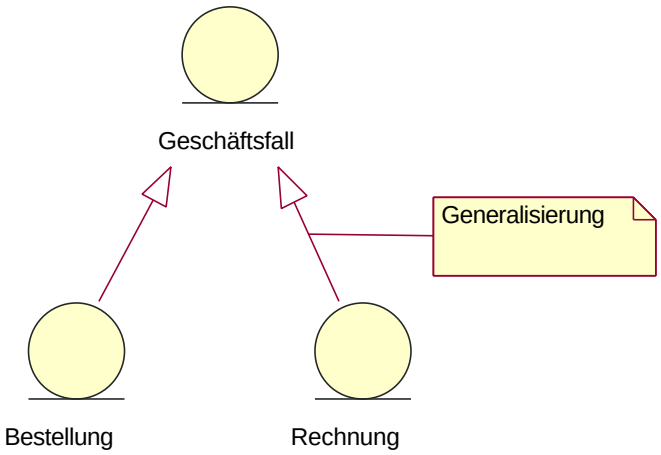
- Offertwesen,
- Auftrag erfassen,
- Ware liefern

Nach der Implementierung wird in einem Review des Datenmodells festgestellt, dass viele Funktionen mehrfach entwickelt wurden.
Grund:
Es wurde nicht berücksichtigt, dass aus einer Offerte durch eine reine Statusänderung ein Auftrag werden kann, aus dem durch eine weitere Statusänderung ein Lieferschein wird und schliesslich eine Rechnung.
Nach dem Review war klar, dass das Paket mit diesen Erkenntnissen in etwa einem Viertel der Zeit hätte entwickelt werden können und erst noch leichter zu erweitern und zu warten wäre.

Wie findet man mögliche Generalisierungen?

- Falls Klassen ähnliche Zuständigkeiten haben, kann man versuchen die Klassen zusammen zu fassen und zu einer abstrakteren Beschreibung gelangen.
- Oft sind es alleine sprachliche Konstrukte die einem behilflich sind:
  der Kunde und der Lieferant sind beides Geschäftspartner, können also auf einer höheren Abstraktionsebene zusammen gefasst werden.

Beispiel Identifikation von Generalisierungen
Rechnungen und Bestellungen haben ähnliche Zuständigkeiten. Beide sind Spezialfälle der allgemeineren Klasse Geschäftsfall



Im Design und der Implementierung muss man sich genauer überlegen, wie im Einzelnen Verallgemeinerungen implementiert werden können. Ob allgemeine Konzepte überhaupt implementierbar sind, hängt von der Programmiersprache, Datenbanken, Betriebsystemen,... ab.

14.6.3.5. Festhalten spezieller Anforderungen

Dieser Schritt ist immer für Fälle gedacht, in denen eine Detailmodellierung nicht angebracht ist. Im Speziellen werden nichtfunktionale Anforderungen durch Text erfasst.

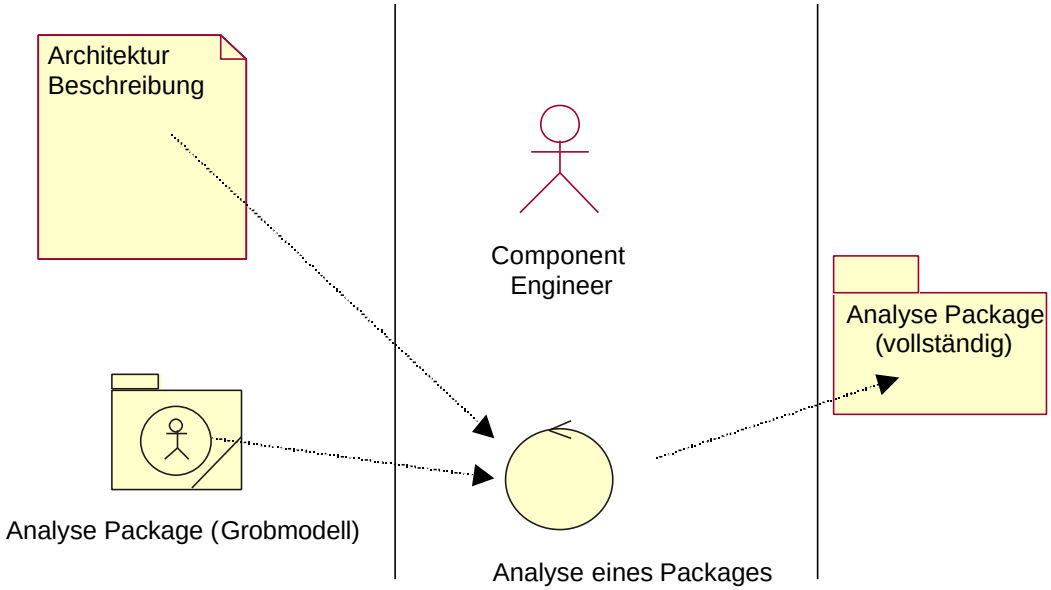
Beispiel Festhalten von speziellen Anforderungen über eine Analyse Klasse
Die Datenspeicherung für die Rechnungsklasse kann wie folgt quantifiziert werden:

- *Speicherbedarf pro Objekt:* 2-24 KB pro Objekt
- *Datenvolumen:* bis 100'000 Objekte
- *Mutationshäufigkeit :*

1.	Neue Objekte / Löschen von Objekten	:	1'000 pro Tag
2.	Mutationen	:	bis zu 30 pro Stunde
3.	Lesen von Objekten	:	mindestens ein Zugriff pro Minute

14.6.4. Aktivität : Analyse eines Packages

Zuerst die Aktivitäten in der Übersicht:



Das Ziel der Package Analyse ist es:

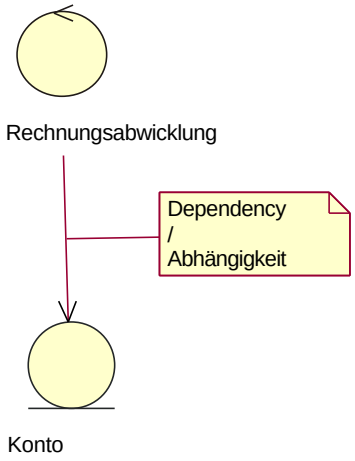
- Sicher zu sein, dass das Package unabhängig von anderen Packages ist
- Sicher zu sein, dass das Analyse Package das, was von ihm gefordert wird, auch erfüllen kann (Anwendungsfälle)
- Die Abhängigkeiten der Packages untereinander vollständig wieder zu geben.

Folgendes Vorgehen führt in der Regel zum Ziel:

1. Definiere und untersuche alle Abhängigkeiten von andern Paketen oder Abhängigkeiten von Klassen im Paket von Klassen in anderen Paketen.
2. Stelle sicher, dass das Paket die richtigen Klassen enthält. Das Paket soll so kohäsiv / zusammen hängend sein, wie nur irgend wie möglich.
3. Reduziere die Abhängigkeit von anderen Paketen, indem Klassen probeweise von einem ins andere Paket verschoben wird, um zu testen, ob dadurch einfachere Beziehungen entstehen.

Betrachten wir zur Illustration ein Beispiel aus dem Geschäftsbereich "Rechnungswesen", "Auftragsabwicklung".

Beispiel Paket Abhängigkeit
Gegeben ist die folgende Abhängigkeit:



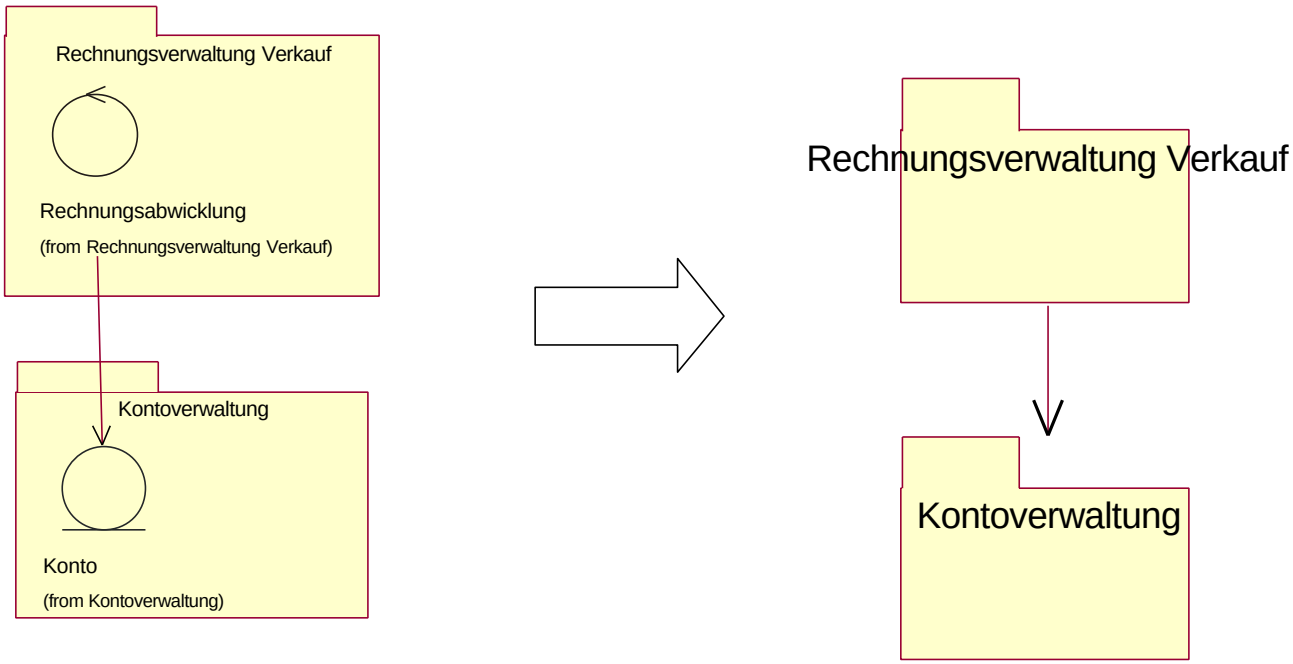
Auf Grund der Anwendungsfälle ergibt sich die Abhängigkeit zwischen dem Konto Objekt / der Konto Klasse und der Rechnungsabwicklung.

Aus dieser Abhängigkeit ergibt sich übergeordnet eine Abhängigkeit auf Paketebene.

Die Kontrollklasse "Rechnungsabwicklung" gehört zum Paket "Rechnungsabwicklung Verkauf".

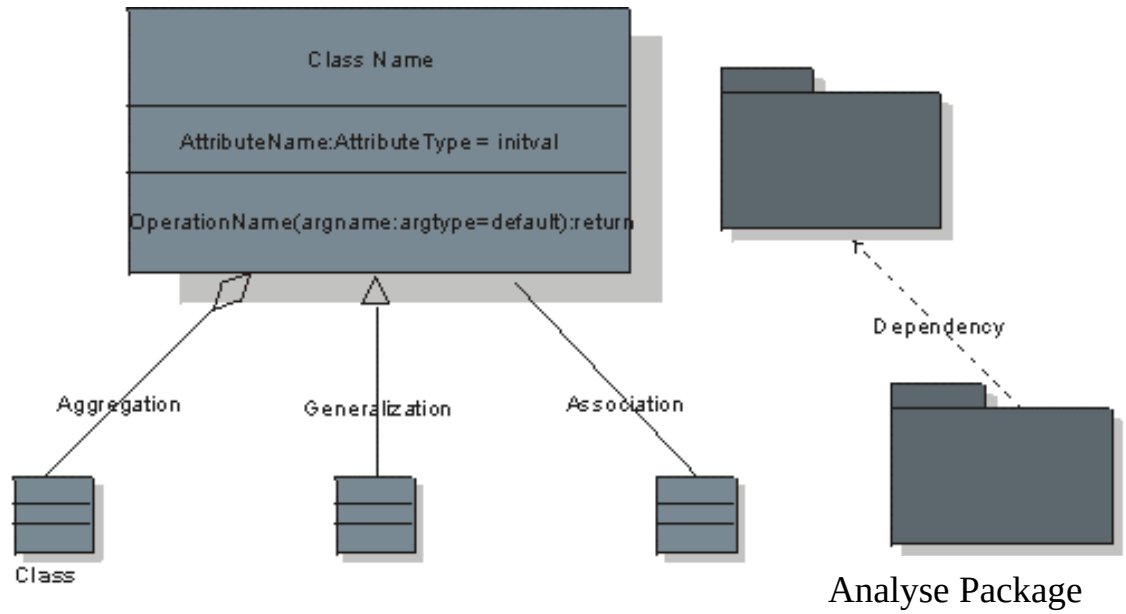
Die Entitäten Klasse "Konto" gehört zum Paket "Kontoverwaltung".

Somit ergibt sich folgendes (vorläufiges) Paketdiagramm:



14.7. Zusammenfassung der Analyse

Schematisch lässt sich das Ergebnis der Analyse wie folgt zusammen fassen:



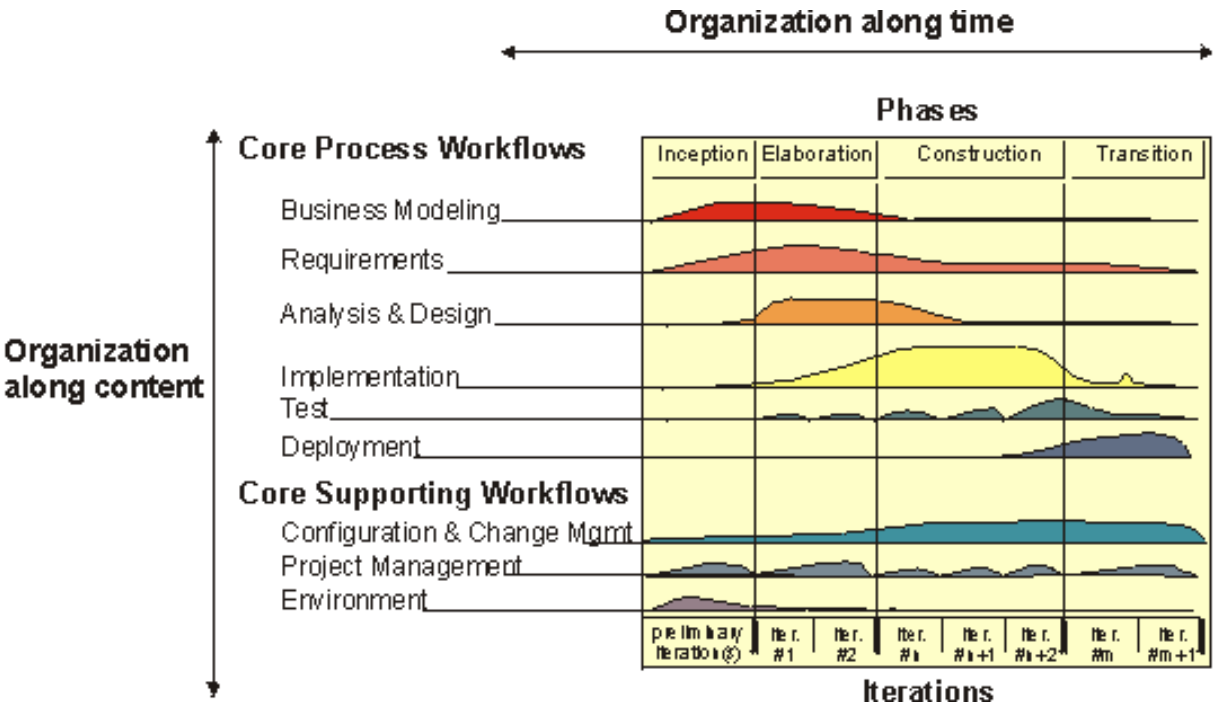
Das Analyse Modell umfasst folgende Elemente:

- **Analyse Packages**
  und Service Packages und ihre Abhängigkeiten und Inhalte.
  Analyse Packages fassen das Verhalten von Anwendungsfällen (Use Cases), das Verhalten der Benutzer (Actors) und Geschäftsfälle zusammen fassen.
  Service Packages fassen Service Klassen zusammen, also Bausteine, aus denen wieder verwendbare Subsysteme aufgebaut werden.
- **Analyse Klassen**
  inkl deren Zuständigkeiten, Attribute und Beziehungen (CRC), sowie speziellen Anforderungen.
  Jede der Entitäten, Systemgranzen und Kontroll Klassen beschreibt ein bestimmtes Verhalten des Systems. Änderungen können in der Regel dadurch lokal begrenzt werden.
  Eine Änderung im Bereich "User Interface" wirkt sich dadurch in der Regel nur auf Systemgrenz-Klassen aus, kann also eingegrenzt werden.
  Änderungen von persistenten Information werden in der Regel durch Entitäten Klassen beschrieben.
  Komplexe Geschäftslogiken werden in der Regel durch Kontrollklassen beschrieben.
- **Use Case / Anwendungsfall Umsetzungs Analyse**
  Anwendungsfälle werden mit Hilfe von Kollaborationsdiagrammen analysiert.
  Spezielle Anforderungen werden in Form von Text erfasst und festgehalten.
- **Architektur**
  eine vorläufige Version der Systemarchitektur dient als Basis für die eigentliche Systemkonstruktion in der Design und der Implementation.

Wie geht es weiter?
Im Design werden wir sehen wie aus den Artifakten der Analyse detailliertere Modellle hergeleitet werden.

Konkret:

- **Analyse Packages**
  bilden die Basis für die Packages und Subsysteme, mit deren Hilfe das System schliesslich gebaut werden
- **Analyse Klassen**
  bilden die Basis für die zu implementierenden Klassen. Je nach Technologie ergeben sich Änderungen oder Ergänzungen..
  Speziell im Bereich der Benutzerschnittstellen werden sich grössere Änderungen ergeben, da beim Implementieren in der Regel Standards berücksichtigt werden und zum Beispiel Widgets eingesetzt werden müssen.



14.8. Anhang : Kollaborations-Diagramme

In UML unterscheidet man zwischen:

- Diagrammen für die statische Strukturbeschreibung:
  Klassendiagramme mit Klassennamen, Attributen, Methoden, Stereotypen, Mächtigkeiten, Beziehungen (Assoziationen und Aggregationen, Verallgemeinerungen, Impleementierungen)
- Use Case Sicht
  Actors, Use Cases, Beziehungen, Verallgemeinerungen
- Zustandssicht
  Zustandsgraphen (Übergänge, Ereignisse und Zustände)
- Aktivitätensicht
  Aktivitätendiagramm (Ereignisse, Zusände, Swimmlanes)
- Ineraktionssicht:
  Collaborations Diagramme
  Sequence Diagramme
- Physische Sicht:
  Komponenten, Implementation

14.8.1. Kollaborations-Diagramme

Ein Kollaborationsdiagramm ist ein Klassendiagramm., welches die Interaktion auf der Ebene der Rollen (Klassifikation und Assoziation) auf der Ebene der Instanzen, der Objekte.

Die folgende Skizze beschreibt die Kollaboration auf der Stufe der Objekte, der Verbindungen (Links) und Stimuli

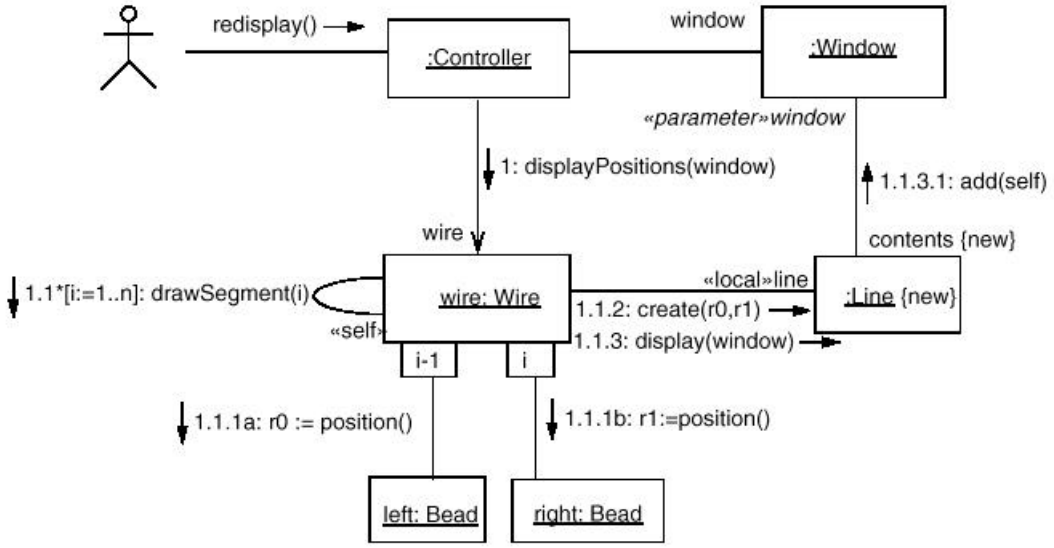


Figure 3-38 Collaboration Diagram at instance level, presenting Objects, Links, and Stimuli.

Daneben gibt es eine leicht unterschiedliche Darstellung der Kollaborationsdiagramme auf der Stufe der Spezifikation. Hier werden die Rollen in den Vordergrund gestellt.

Beide Figuren stammen aus dem Reference Manual (PDF bei WWW.OMG.ORG):

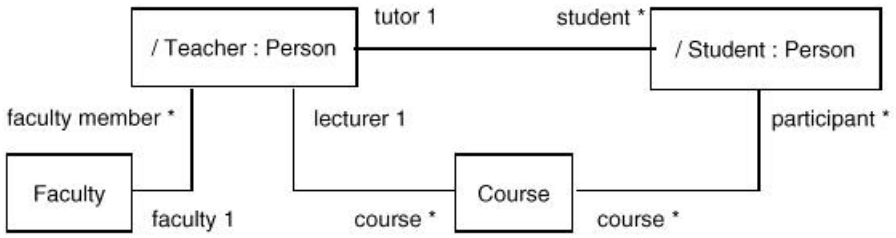


Figure 3-39 Collaboration Diagram at specification level, presenting Classifier Roles and Association Roles.

Der wesentliche Unterschied zwischen Sequenzdiagrammen und Kollaborationsdiagrammen besteht in der Berücksichtigung der Zeit in den Sequenzdiagrammen. In den Kollaborationsdiagrammen wird die Reihenfolge durch die Nummerierung deutlich gemacht.