Rational Objectory Process - Implementation

16.1. Einführung

Die Implementation startet mit den Ergebnissen des Designs und implementiert Komponenten, Subsysteme, Schnittstellen...,in Form von Skripts, Programm-Sourcen und binären Programmen.

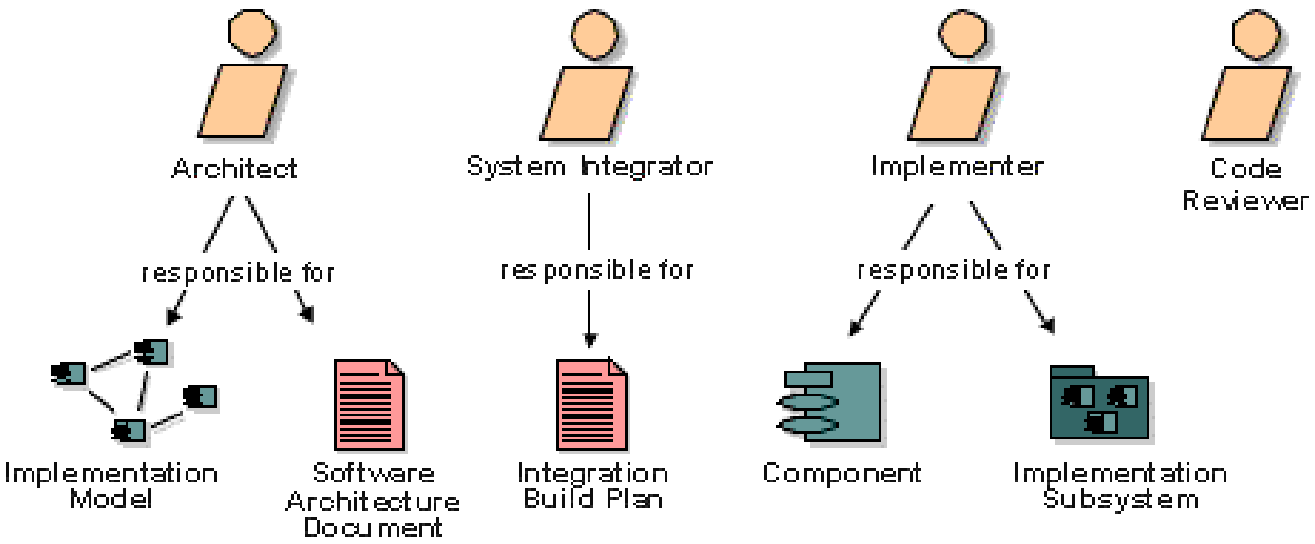
Sie erinnern sich an die Phasen im ROP:
ROP kennt vier Phasen zur Darstellung der *zeitlichen* Entwicklung eines Software Systems:

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

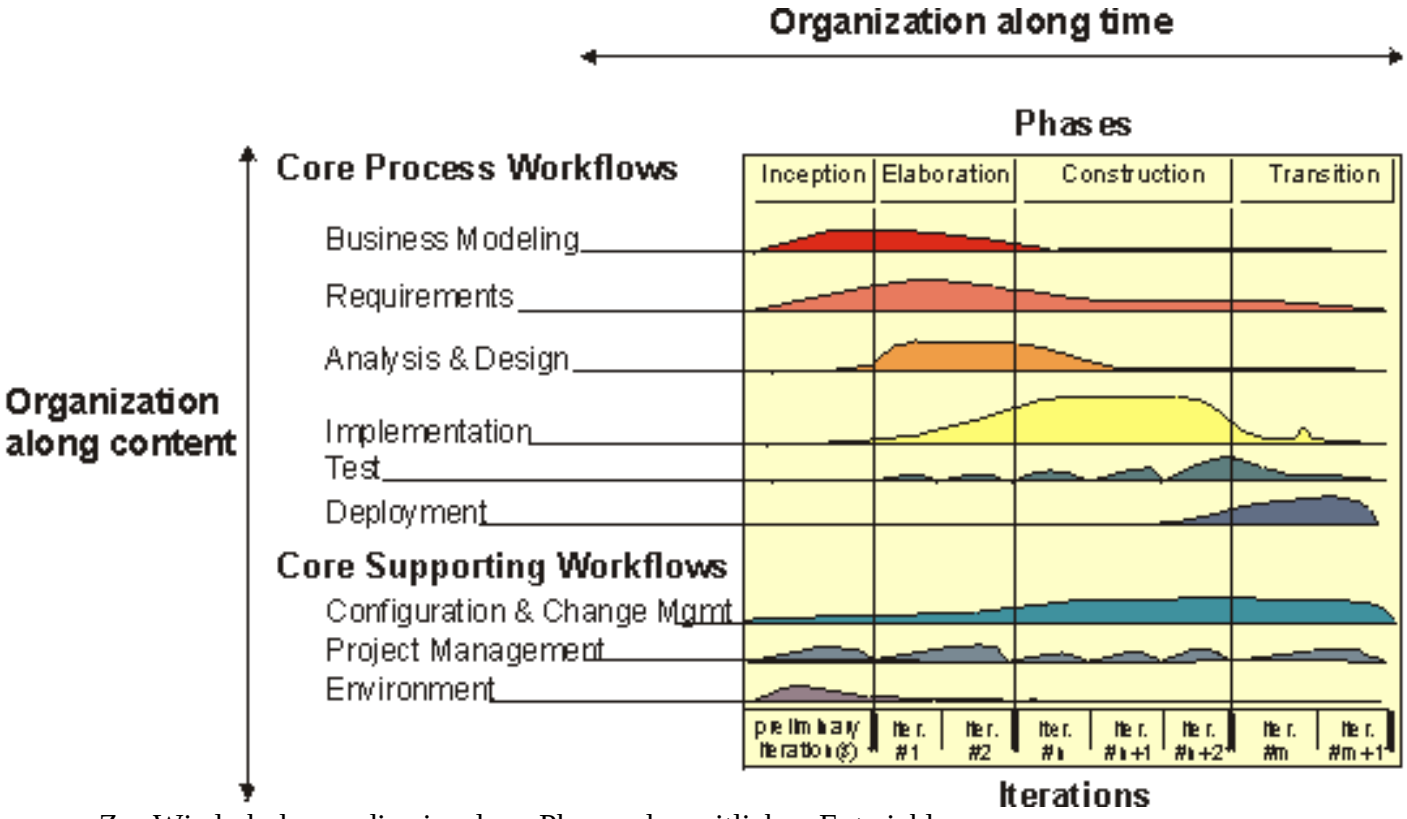
Die wesentlichen Teile der Architektur werden im Design festgelegt; in der Implementation geht es darum diese umzusetzen. In der Regel muss die Realisierung eines komplexen Systems in Schritten geschehen.

Wesentliche Aufgaben der Implementation:

- Planung der System-Integration, speziell bei einem iterativen Vorgehen. Unser Ziel ist es, das System schrittweise und in überschaubaren Teilen einzuführen.
- verteilen des Systems:
  ausführbare Komponenten können auf mehrere Knoten verteilt werden
- Implementation der Design Klassen und Subsysteme, die im Design identifiziert wurden
- Testen der testbaren Einheiten und Integration



16.2. Implementation im Software Lebenszyklus



Zur Wiederholung : die einzelnen Phasen der zeitlichen Entwicklung

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

Implementation ist die Hauptaktivität in der Konstruktionsphase und am Ende der Entwurfsphase und soll eine stabile Architektur als Basis für den restlichen Lebenszyklus 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 möglichst „original“ zu implementieren, so dass ein eventuelles "Round-Trip" Engineering, mit graphischer Unterstützung ermöglicht wird.

16.3. Artifakten

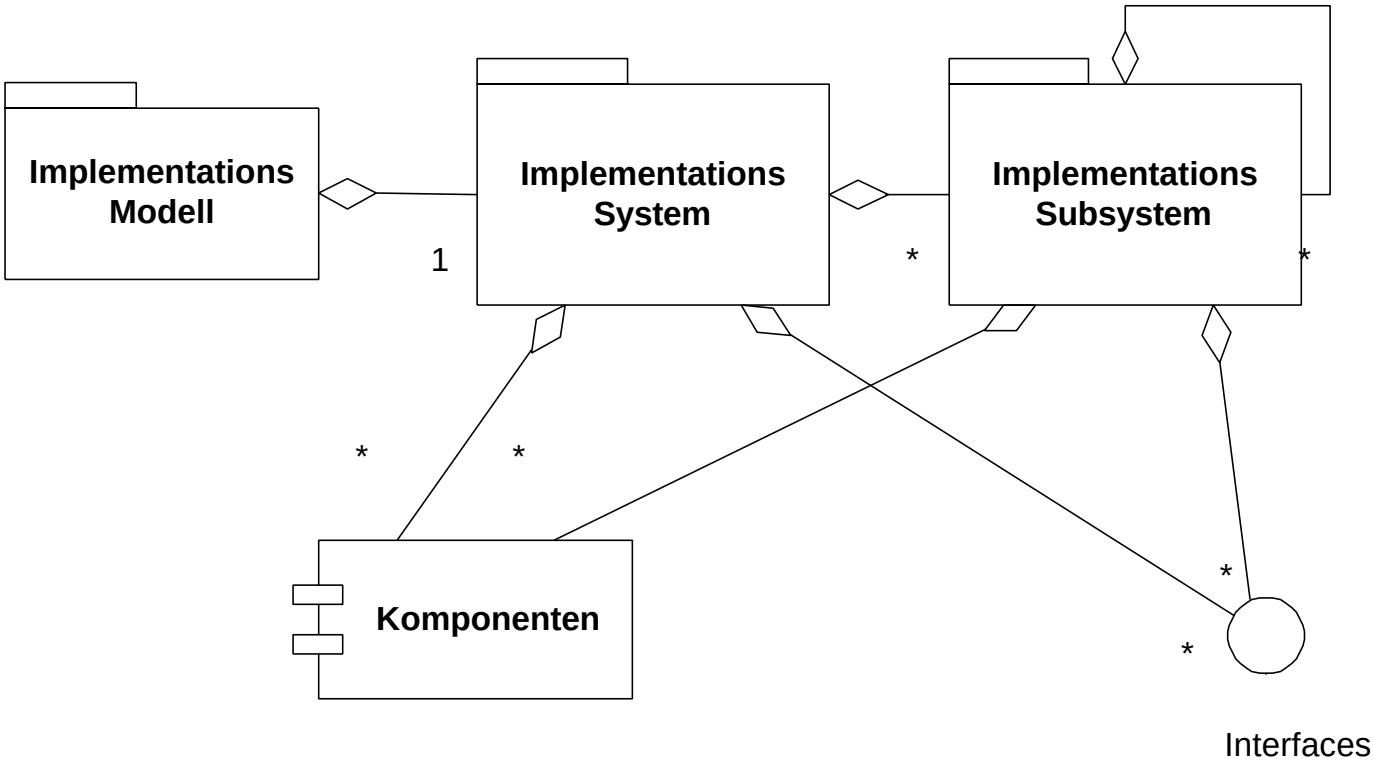
Artifakten sind Unterlagen, die auf Dauer gepflegt werden.

16.3.1. Artifakt : Implementations-Modell

Das Implementions-Modell beschreibt, wie die Elemente des Design-Modells, zum Beispiel die Design-Klassen, mit Hilfe von Komponenten implementiert werden.

Das Implementations-Modell beschreibt, wie die Komponenten organisiert sind, wie sie strukturiert werden müssen, immer unter Berücksichtigung der konkreten Programmiersprache, mit der implementiert werden soll.

Ein formales Modell des Implementations-Modells sieht wie folgt aus:



Das Implementations-Modell stellt also eine Hierarchie von zu implementierenden Subsystemen dar, die aus Komponenten und Schnittstellen bestehen.

16.3.2. Artifakt : Komponenten

Eine Komponente ist die physische Verpackung von Modellelementen, wie Design-Klassen im Design-Modell.

Die Standard Stereotypen der Komponenten sind:

- <<executable>> oder <<ausführbar>> : ein Programm, welches auf einem Knoten ausführbar ist
- <<file>> : eine Datei, welche Quellcode oder Daten enthält
- <<library>> : ist eine statische oder dynamische Bibliothek
- <<table>> : ist eine Datenbank Tabelle
- <<document>> : ist ein Dokument

Weitere Stereotypen können je nach Anwendungsgebiet nötig werden. UML ist aber in dieser Beziehung flexibel:

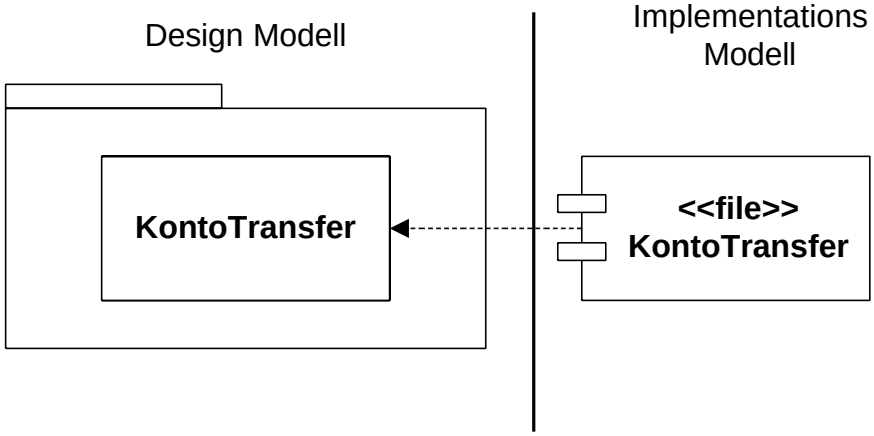
- es ist Ihnen überlassen neue Stereotypen zu definieren.
- Rose erlaubt es Ihnen beliebig viele eigene Stereotypen zu definieren

Komponenten lassen sich wie folgt charakterisieren:

- Komponenten lassen sich in Beziehung setzen zu Design Elementen
- in der Regel implementiert eine Komponente verschiedene Elemente, zum Beispiel verschiedene Design Klassen. Die genaue Beziehung zwischen den Design Elementen und den Komponenten ergibt sich jedoch abhängig von den praktischen Gegebenheiten und der verwendeten Programmiersprache.

Beispiel: Beziehung Komponente – Design Klasse

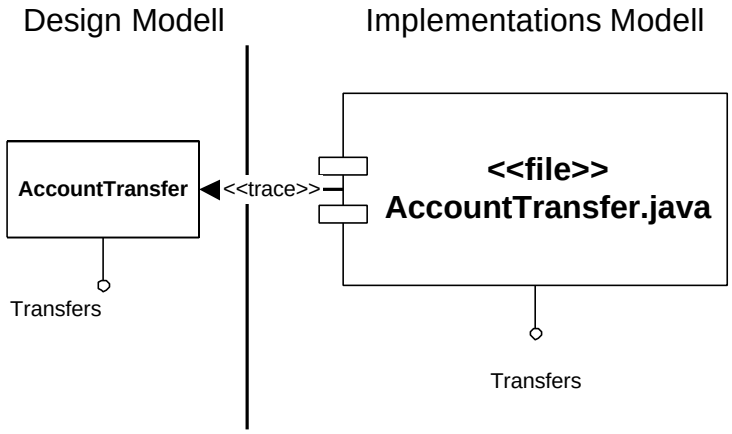
In unserem Interbanken-Beispiel kann die Kontotransfer-Klasse mit Hilfe einer Java Klasse AccountTransfer.java implementiert werden.



- Komponenten haben die gleichen Schnittstellen, wie die Modellelemente, die sie implementieren

---

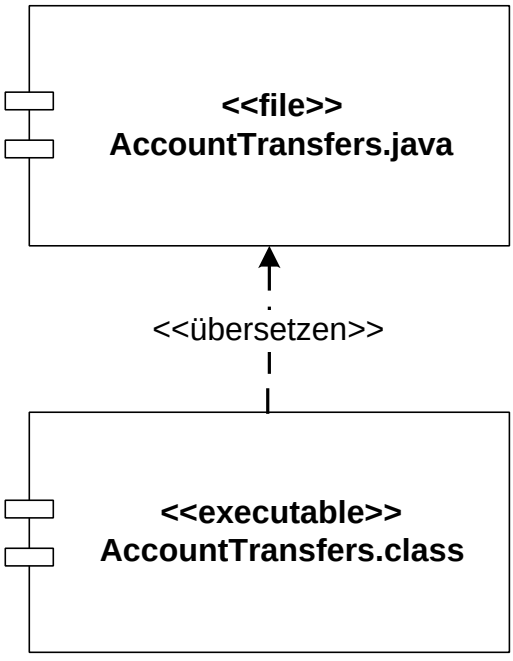
Beispiel: Schnittstelle im Design und der Implementation
In unserem Interbanken Beispiel hat die Konto-Transfer Klasse ein Transfer-Interface. Diese Schnittstelle wird auch von der Java Klasse AccountTransfer implementiert.



Komponente und Design Klasse (müssen) stellen die gleiche Schnittstelle zur Verfügung.

- unter Umständen gibt es Übersetzungsabhängigkeiten zwischen Komponenten, dh. unter Umständen hängen mehrere Komponenten so voneinander ab, dass die einzelne Komponente nicht übersetzbar, aber mehrere Komponenten zusammen.

Beispiel: Übersetzungs-Abhängigkeiten zwischen Komponenten
Die Interbanken Java Datei wird zurClass-Datei AccountTransfers.class übersetzt.



16.3.2.1. Stubs

Ein Stub ist eine Komponente mit einer speziellen Implementation, die es erlaubt, andere Komponenten zu entwickeln oder zu testen, die von diesem Stub abhängen. Die Notation wird vor allem im Zusammenhang mit RPC, CORBA, RMI, ... also verteilten Applikationen eingesetzt.

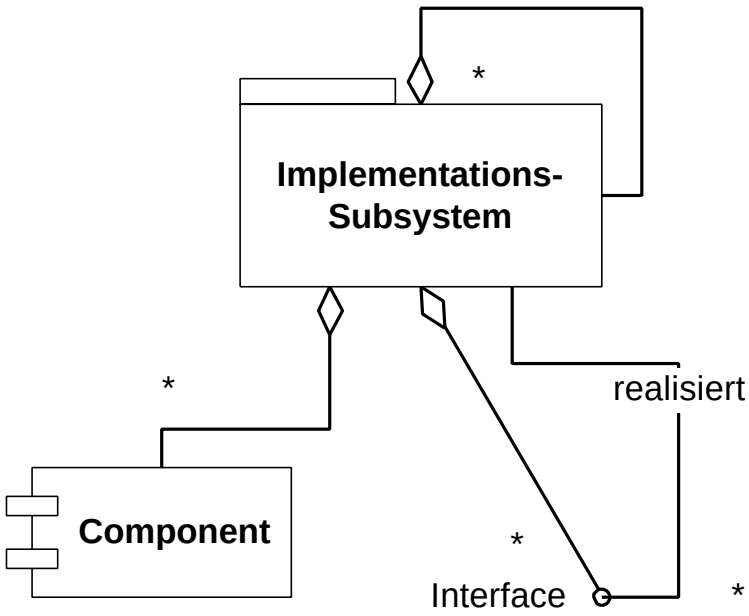
Mit Hilfe von Stubs kann in der Regel die Anzahl neuer Komponenten in einem neuen Release reduziert werden: die Stubs bilden eine klar definierte „Schnittstelle“ (Stubs können selber komplexe Programme sein). Sie lernen diese Art Systeme zu implementieren noch kennen (RMI, Jini, .. siehe oben).

Dadurch wird die Systemintegration erleichtert und die Releasefähigkeit erhöht.

16.3.3. Artifakt : Implementations-Subsystem

Implementations-Subsysteme haben zum Ziel, das zu implementierende System in handlichere Teil- oder Sub-Systeme zu zerlegen. Ein Subsystem kann aus Komponenten, Schnittstellen oder / und anderen Subsystemen (rekursiv) bestehen.

Formal sieht unser Modell also wie folgt aus:



Ein Subsystem kann auch Schnittstellen implementieren, um die Funktionalität die es implementiert nach aussen bekannt zu machen.

Das Implementations-Subsystem korrespondiert Konstrukten in realen Programmiersprachen:

- einem Package in Java
- einem Project in Visual Basic
- einem Directory in C++ Projekten
- einem Component View Package in Rational Rose

Die Bedeutung eines *Implementations-Subsystems* wird also sematisch verfeinert entsprechend der spezifischen Implementationsumgebung.

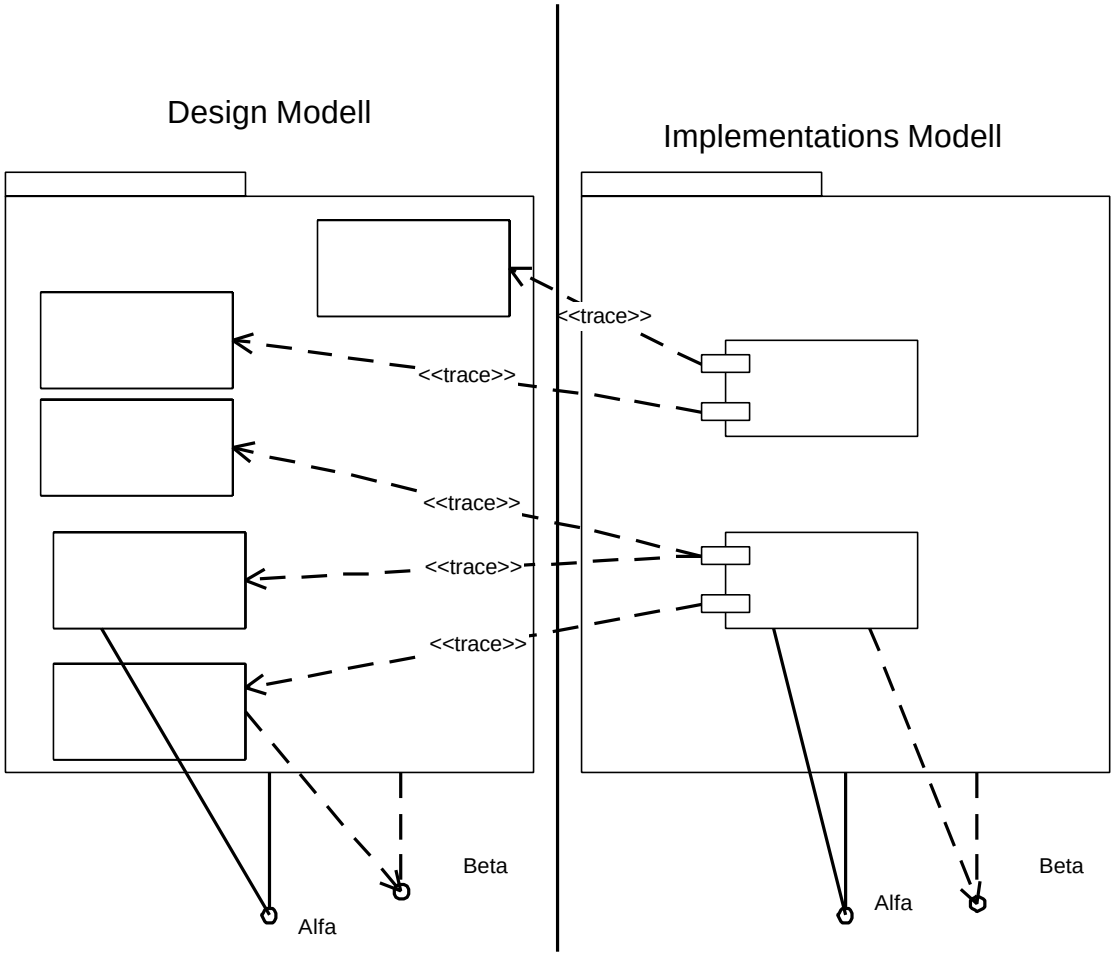
Natürlich gibt es eine sehr direkte Beziehung zwischen Implementations-Subsystemen und Design Subsystemen.

In den Design Subsysteme haben wir bereits definiert:

- Abhängigkeiten in und zwischen den Subsystemen, Schnittstellen und innerhalb eines Subsystems
- was eine Schnittstelle einem Subsystem liefert
- welche Design Klassen oder rekursiv, welche andern Design Subsysteme innerhalb eines Subsystems anern Subsystemen Schnittstellen anbieten

Für die Implementation eines Subsystems sind diese Design Aspekte aus folgenden Gründen wichtig:

- das Implementations-Subsystem wird und sollte analoge Abhängigkeiten verwenden
- das Implementations-Subsystem sollte die gleichen Schnittstellen anbieten
- Das Implementations-Subsystem sollte festlegen, welche Subsysteme die Schnittstellen, die vom Subsystem zur Verfügung gestellt werden müssen, implementiert.



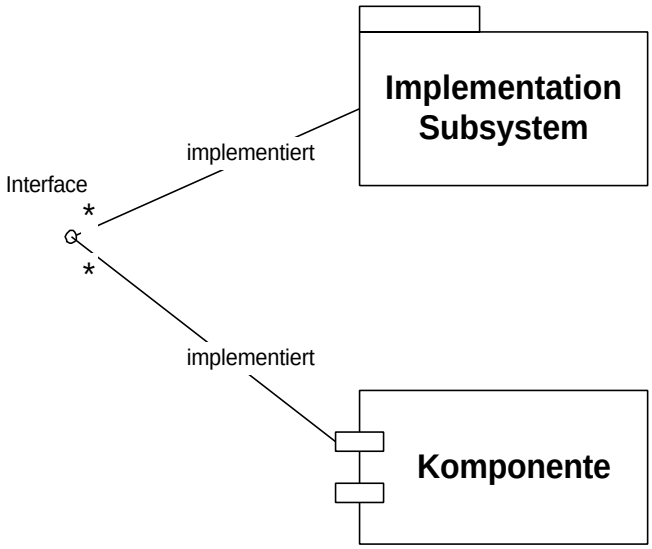
Die Design Klassen stellen die Schnittstelle Alfa zur Verfügung; die Implementation auch; die Design Klassen hängen von der Schnittstelle Beta ab; die Implementation auch.

Änderungen bei der Beschreibung der Schnittstellen im Design Modell müssen also auch zu Änderungen bei den Schnittstellen der Implementation führen.

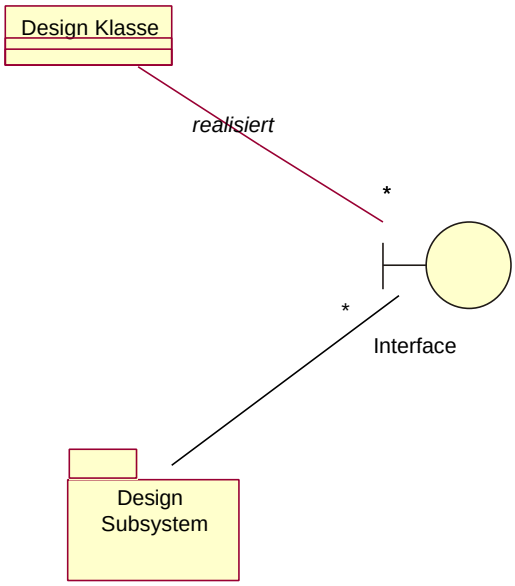
16.3.4. Artifakt : Interface

Schnittstellen haben wir im letzten Kapitel beschrieben. Die Schnittstellen im Implementationsmodell müssen die Funktionalität, die im Design beschrieben wurde, implementieren.

Beziehungen in der Implementierung:



Beziehungen im Design:



Eine Komponente, die eine Schnittstelle implementiert, muss auch alle Methoden, die die Schnittstelle zur Verfügung stellt implementieren.

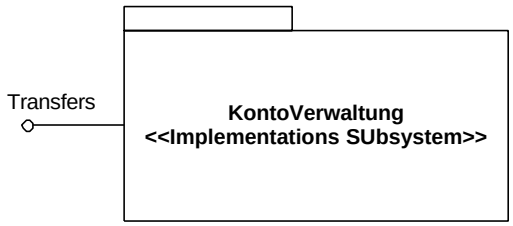
Falls die Schnittstelle nicht direkt implementiert wird, also die Services delegiert werden, dann ist das Subsystem auch dafür verantwortlich, dass die Subsysteme des Subsystems, an die delegiert wurde, implementiert werden und die entsprechenden Dienste zur Verfügung stellt.

Beispiel: Ein Implementations Subsystem, welches ein Interface anbietet

In unserem Interbanken-Beispiel können wir ein Implementations Subsystem AccountManagement (KontoVerwaltung) als Java Package implementieren.

.

Schematisch sieht diese Schnittstelle wie folgt aus:



Und als Java Code:

```
package AccountManagement;

// stellt Schnittstellen zur Verfügung

public interface Transfers {
        public Account create(Customer owner, Money balance, AccountNumber account_id);

        public void Deposit(Money amount, String reason);

        public void Withdraw(Money amount, String reason);
}
```

also eine einfache und überschaubare Angelegenheit.

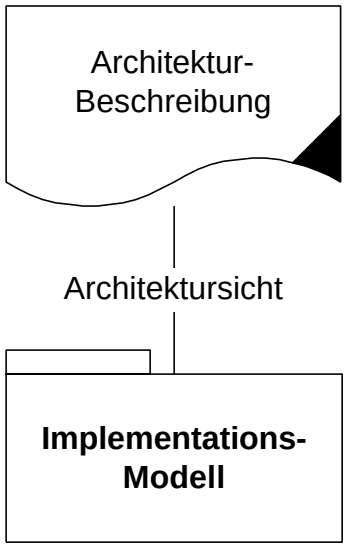
Rational Rose bietet die Möglichkeit den Programmcode automatisch zu generieren.

16.3.5. Artifakt : Architekturbeschreibung (Sicht der Implementation)

Die Architekturbeschreibung enthält eine **Implementationssicht** und beschreibt die Architektur relevanten Elemente der Implementation.

Folgende Artifakte sind in der Regel Architektur relevant:

- die Zerlegung des Implementationsmodells in Submodelle, Schnittstellen und gegenseitigen Abhängigkeiten.
  Da die Implementation die natürliche Fortsetzung des Design ist, kann man diese Elemente der Architektur (die Zerlegung) in der Regel bereits in der Design Sicht der Architektur erkennen.
- Schlüsselkomponenten, welche sich aus Architektur relevanten Design Klassen ergeben, ausführbare Komponenten, allgemein einsetzbare Komponenten sind in der Regel die Komponenten, die einen grossen Einfluss auf die Gesamtarchitektur oder die Architektur der Subsysteme haben.



16.3.6. Artifakt : Integrationsplan

Da komplexe Software fast nur iterativ entwickelt werden kann, muss ein Integrationsplan entwickelt werden, aus dem hervor geht, wie und wann welche Komponenten zusammen gefügt werden müssen und können. Das Ergebnis dieses Planes ist ein „build“ also eine ausführbare Version des Systems oder eines Subsystems.

Es muss auch der Worst Case geplant werden: im schlimmsten Fall muss auf die alte Version zurück gegriffen werden.

Der Nutzen eines solchen Vorgehens:

- es steht möglichst früh bereits eine ausführbare Version zur Verfügung. Der Integrationstest kann sehr früh beginnen und mit der ausführbaren Version steht dem Benutzer und dem Auftraggeber oder dem Geldgeber bereits etwas „Brauchbares“ zur Verfügung.
- Schwächen werden recht schnell erkennbar, da der Integrationstest in klar definierten Schritten abläuft. Fehler werden hoffentlich auch nur mit jeder Iteration neu eingeführt und sind somit lokalisierbar.
- Integrationstests sind in der Regel gründlicher als Systemtests. Daher kann man davon ausgehen, dass das Endergebnis besser getestet wurde als bei einem Systemtest.

Inkrementelle Integration ist analog zur iterativen Entwicklung der Software. Beide haben das Ziel, das System in kleinen und überschaubaren Schritten wachsen zu lassen.

Bei jeder Iteration muss ein neues „build“ (oder Release) resultieren. Mit jedem Release wächst also die Funktionalität.

Ein *Integrations Bauplan / Releaseplan* beschreibt in welcher Folge die einzelnen neuen Baustufen freigegeben werden.Der Plan umfasst folgende Information:

- die Funktionalität, welche mit jedem Release erwartet werden kann.
- welche Teile des Systems von neuen Teilsystemen betroffen sind.

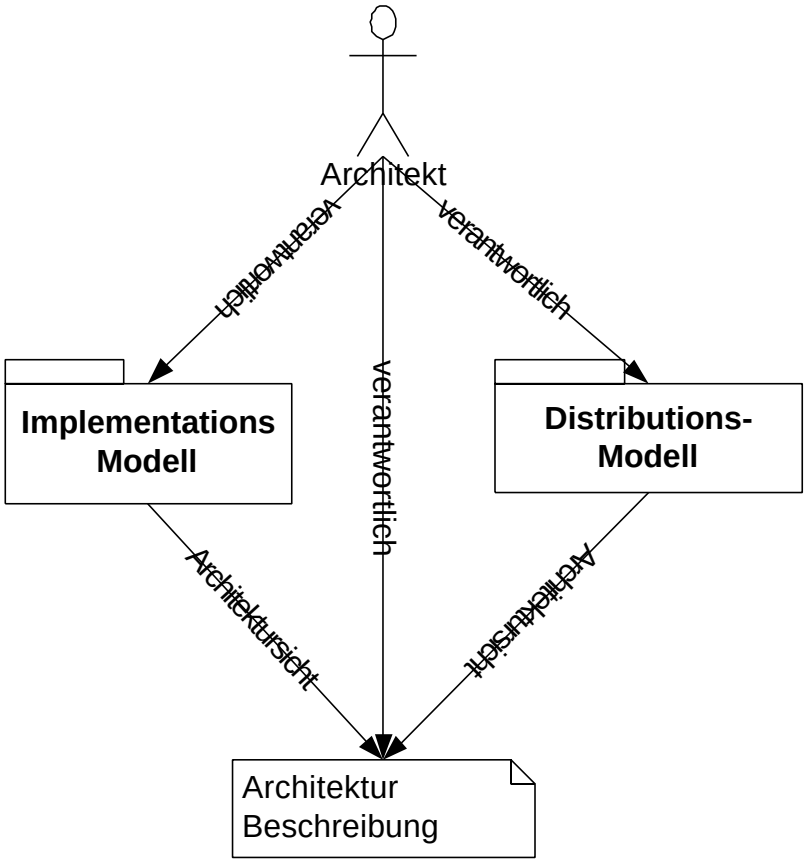
16.4. Mitarbeiter / Beteiligte

Wer ist an der Implementation beteiligt? Eigentlich die gleichen Mitarbeiter wie in der Design Phase plus Programmierer, da diese sich mit Hilfe des Design ein vertieftes Bild des Systems verschafft haben.

16.4.1. Mitarbeiter : der Architekt

Der Architekt muss die Integration der Subsysteme garantieren. Er ist dafür verantwortlich, dass bei allen Zerlegungen des Systems in Teilsysteme und Komponenten, die Integration des Gesamtsystems gewährleistet wird.

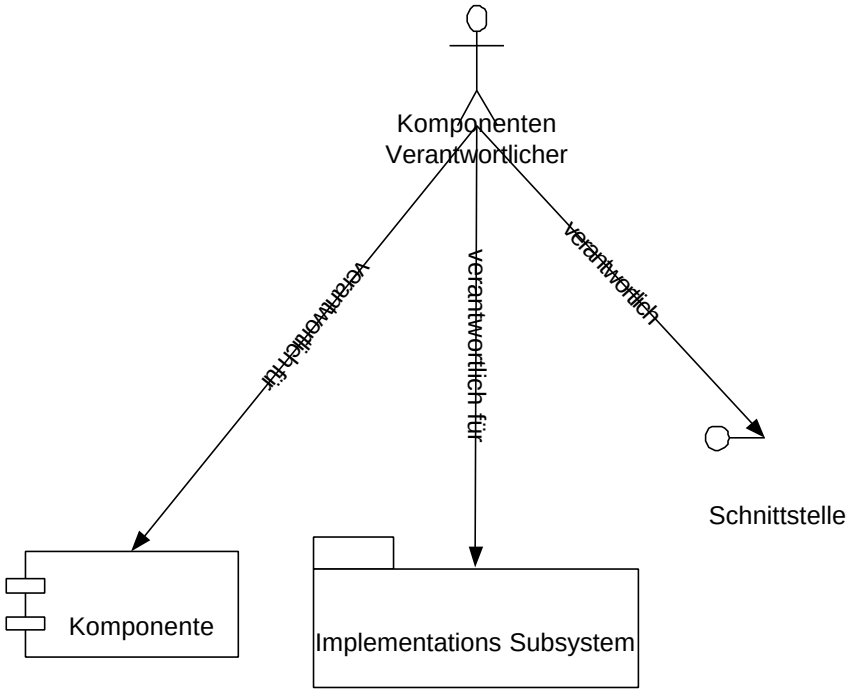
Der Architekt muss auch die Verteilung der Applikation auf die verschiedenen Knoten bestimmen.



Aber der Architekt ist nicht für die Weiterführung der Modelle verantwortlich.

16.4.2. 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.



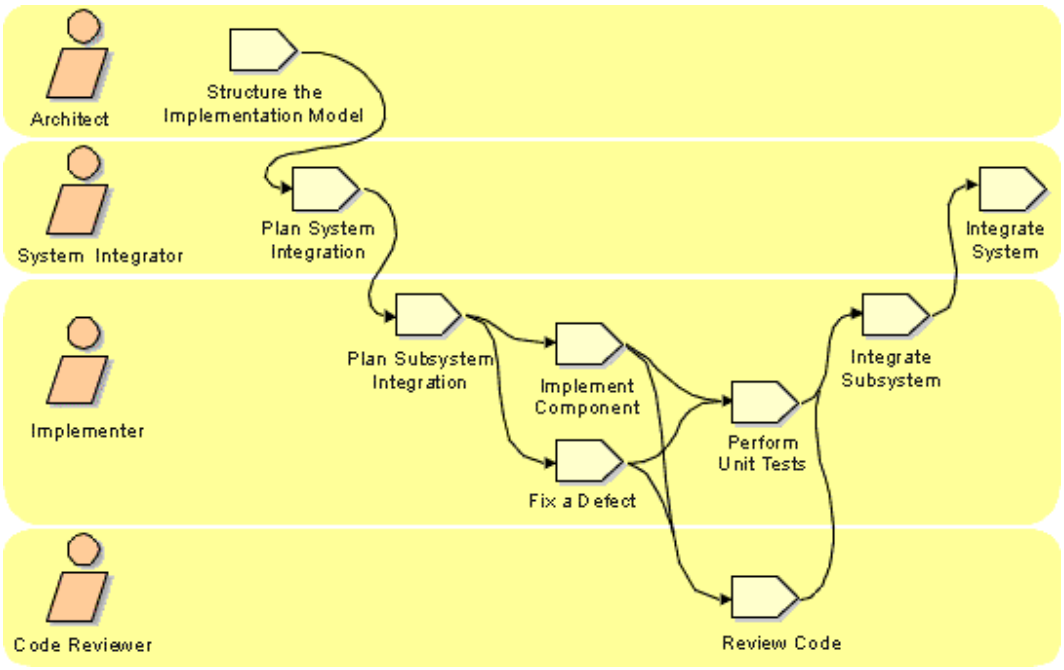
Der Verantwortliche für die Komponenten ist oft auch für einzelne Subsysteme verantwortlich. Da die Implementationssysteme eins-zu-eins mit dem Design korrespondieren müssen, müssen auch allfällige Änderungen an einem der Systeme im andern nachgeführt werden.

16.4.3. Mitarbeiter : System Integrator

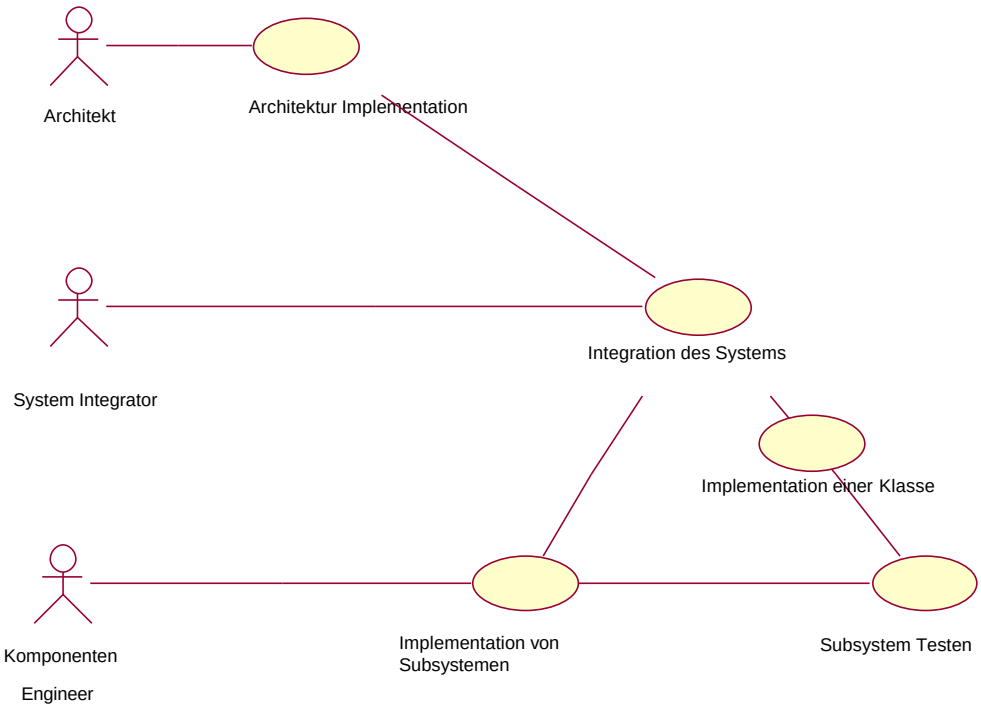
Die Person, die die Verantwortung für das Gesamtsystem trägt, muss die einzelnen Releases oder Builds planen und den Releaseplan nachführen, in Abstimmung mit Marketing und Verkauf, falls es sich um ein verkäufliches Produkt handelt.

16.5. Workflow

Bisher haben wir die Implementation Aktivitäten einfach als Ansammlung von Implementation Ergebnissen beschrieben. Jetzt wollen wir Implementation 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.



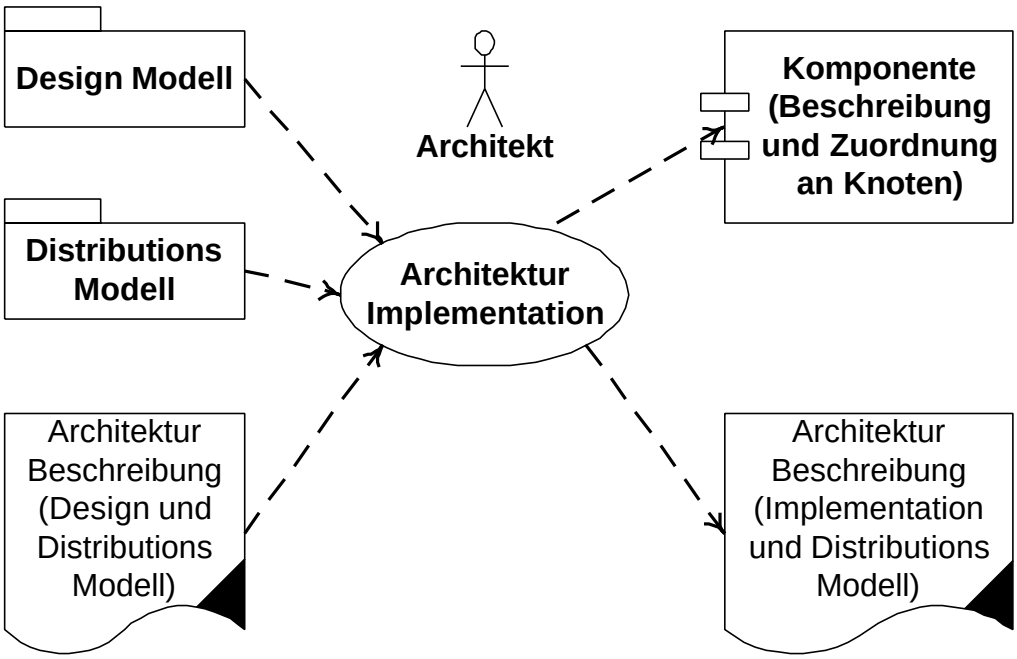
16.5.1. Aktivität : Architektur Implementation

Das Ziel der Architektur Implementation ist es zu garantieren, dass die im Design festgelegte Architektur auch korrekt implementiert wird:

- identifizieren der architektonisch relevanten Komponenten, zum Beispiel ausführbare Komponenten
- Verteilung der Komponenten auf Knoten der relevanten Netzwerk Konfiguration

Im Architektur Design wurden Schnittstellen, Klassen (generische dh. applikationsübergreifende und applikationsspezifische Klassen) , Subsysteme identifiziert. Daher ist die Arbeit des Architekten in der Implementation eher einfach. Es geht höchstens darum, Verfeinerungen vorzunehmen.

Zusammenfassend : Tätigkeiten und Ergebnisse des Architekten zur Implementation



16.5.1.1. Identifikation von architektonisch relevanten Komponenten

In der Regel müssen architektonisch relevante Komponenten möglichst früh identifiziert werden. Allerdings muss man darauf achten, dass man nicht jedes Detail bereits festlegt.

Dateien und Klassen festzulegen ist in der Regel nicht extrem schwierig. Primär muss aber die Architektur auf der Ebene „Architektur Pattern“ und „Design Pattern“ festgelegt werden: Pattern sind Lösungsmuster und zeigen somit WIE die Lösung aussehen kann.

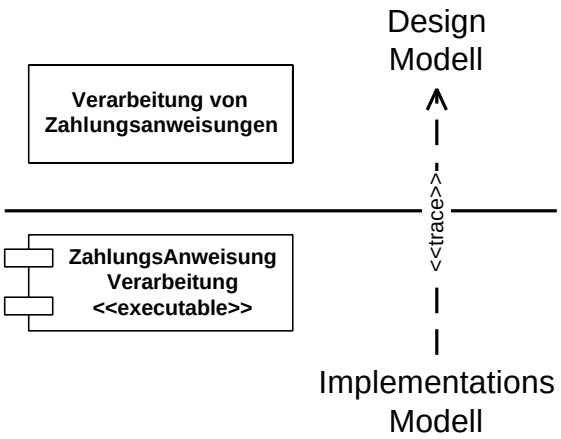
Falls die Architektur zu detailliert festgelegt wird, hat der Implementator keinen Raum mehr für die Ausnutzung von Programmsprachen spezifischen Eigenschaften.

16.5.1.2. Identifikation der Knoten und Netzwerk-Konfiguration

Das Vorgehen ist recht einfach: wir nehmen die als ausführbar erkannten Klassen und ordnen diesen ausführbare Komponenten zu. Zudem identifizieren wir allfällig benötigte Dateien und falls nötig Hilfsprogramme.

Beispiel **Identifikation von ausführbaren Komponenten**

Im Design Modell haben wir eine aktive Klasse „Verarbeitung von Zahlungsanweisungen“. In der Implementation nennen wir die dazu gehörende ausführende Komponente „ZahlungsAnweisungVerarbeitung“. Schematisch sieht diese wie folgt aus:

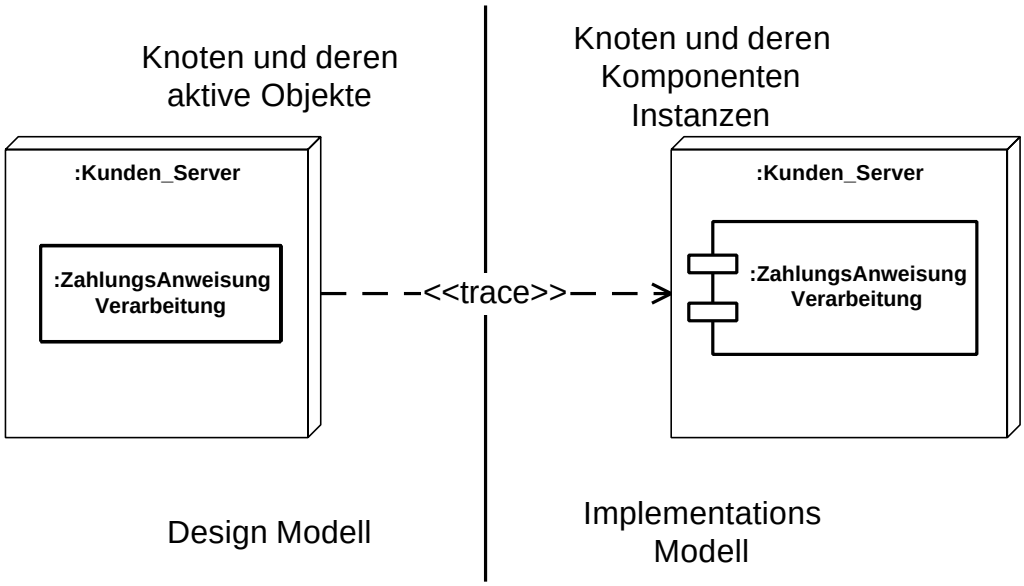


Der aktiven Klasse wird eine ausführbare Komponente zugeordnet.

In einem weiteren Schritt müssen wir nun die Komponenten auf die Knoten verteilen. Betrachten wir ein Beispiel dazu:

Beispiel **Verteilung der Komponenten auf Knoten**

Die ausführbare Komponente „ZahlungsAnweisungVerarbeitung“ wird sinnvollerweise auf dem Knoten „Käufer Server“ installiert, da wir beim Design die aktive Klasse, aus der die Komponente hergeleitet wurde, auch auf diesen Knoten gelegt hatten.



Der aktiven Klasse wird eine ausführbare Komponente zugeordnet.

Die Verteilung der ausführbaren Komponenten ist offensichtlich eine zentrale Aufgabe des Architekten!

16.5.2. Aktivität : Systemintegration

Das Ziel der Systemintegration aus Architektensicht dürfte klar sein:

- einen Plan für die Freigabe von Teilsystemen zu erstellen
- Integration des jeweiligen Programmstandes vor den Systemtests

16.5.2.1. Planung einer Freigabe

Bei komplexen Systemen kann man das System nicht gleich als Ganzes freigeben. In der Regel wird man Teilapplikationen identifizieren, Prioritäten setzen und dann ein Teilsystem nach dem andern implementieren, testen und integrieren.

Diese Zerlegung in Freigabesysteme wird in der Regel auf Grund der Anwendungsfälle, der Use Cases bestimmt. In einzelnen Fällen wird man auch Szenarios verwenden.

Kriterien für die Definition von Freigabesystemen:

- die auszuliefernden Systeme („Builds“) sollten funktional umfangreicher sein als ihre Vorgänger. Es sollte erkennbar sein, welche Anwendungsfälle mit dem neuen „Build“ zusätzlich abgedeckt werden.
- das neue „Build“ sollte nicht zu umfangreich, zu verschieden vom Vorgänger sein

Die Gründe für die Kriterien sind recht einfach:

- der Benutzer möchte wissen, was er zusätzlich erhält
- der Benutzer kann nicht viele komplexe neue Funktionen auf einmal einführen

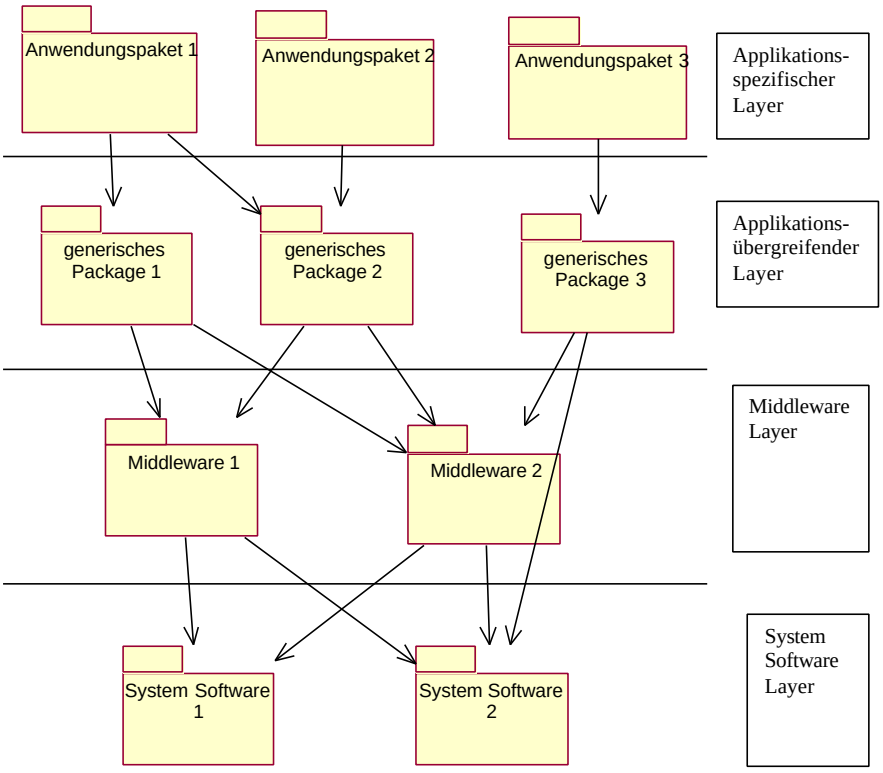
Pro Anwendungsfall müssen folgende Punkte geklärt werden:

- wie sieht die ursprüngliche Beschreibung des Anwendungsfalles aus, wie sieht die Umsetzung in Analyse und Design aus?
- welche Subsysteme und Klassen sind an diesem Anwendungsfall beteiligt und aus welchen Klassen und Design Subsystemen stammen diese?
- welche Seiteneffekte haben die neu zu schaffenden Komponenten auf das bestehende System?

Die Ergebnisse der Abklärung werden in einem Implementationsplan festgelegt.

16.5.2.2. Integration einer auslieferbaren Version

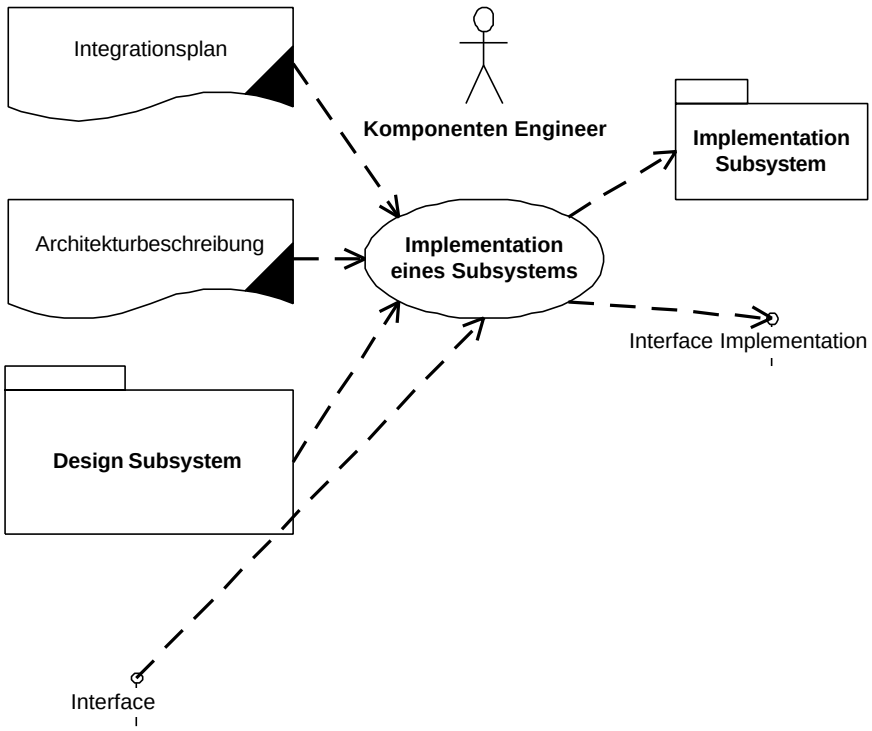
Sofern die einzelnen Releases / Builds klar geplant wurden, ist die Integration uns zeitgerechte Auslieferung sicher das Ziel des Implementations-Teams. Die Praxis, speziell bei grossen Softwarefirmen (Microsoft, SAP, Oracle), zeigt, dass eine saubere Planung im Software Bereich eher schwierig ist.
Die Auslieferung eines Systems kann je nach verwendetem Architektur Pattern auch schichtweise , bottom up geschehen.
Betrachten wir ein System, welches mit Hilfe des Layer Patterns implementiert wird: zuerst muss die Middleware, dann die applikationsübergreifenden Programme und schliesslich die eigentlichen Anwendungsprogramme ausgeliefert werden.



16.5.3. Aktivität : Implementation eines Subsystems

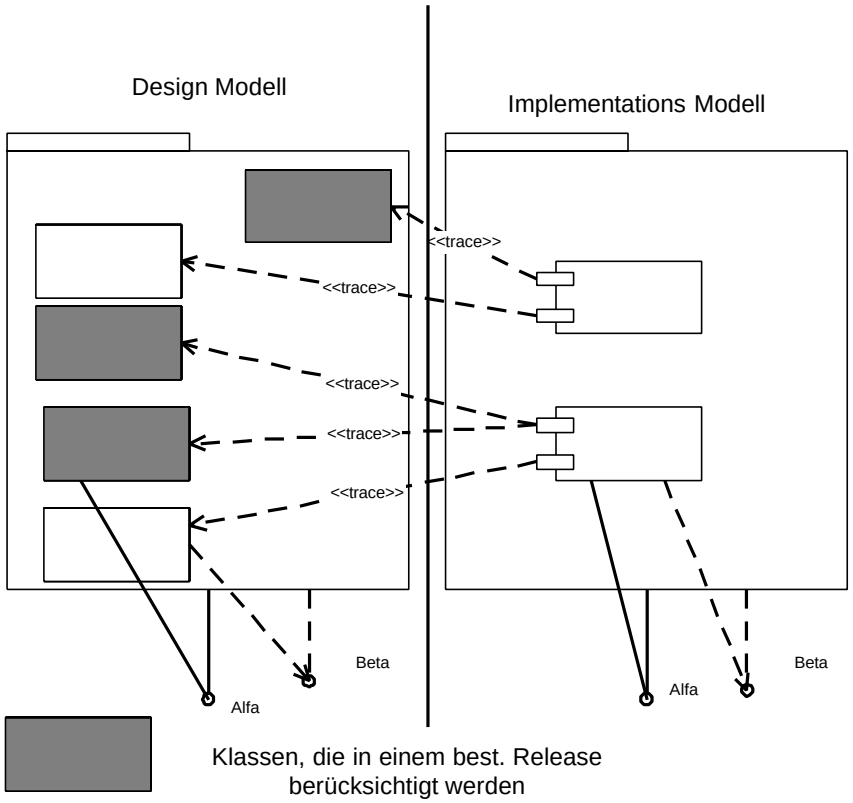
Subsysteme sind im obigen Falle einer Layer / Ebenen Architektur typischerweise mit den applikationsübergreifenden Systemen verbunden. Diese Subsysteme müssen so stabil definiert und implementiert werden, dass nicht bei jeder neuen Freigabe die „Middleware“ ausgetauscht werden muss.

16.5.3.1. Wartung des Subsystem-Inhaltes



Wie die Design Klassen und die Implementations Subsysteme zusammen hängen haben wir bereits bei den Artifakten besprochen..

Bei der Festlegung eines Releases muss festgehalten werden, welche Klassen und Subsysteme berücksichtigt werden. Dies ermöglicht eine saubere Planung und eine bessere Strukturierung des Gesamtsystems.

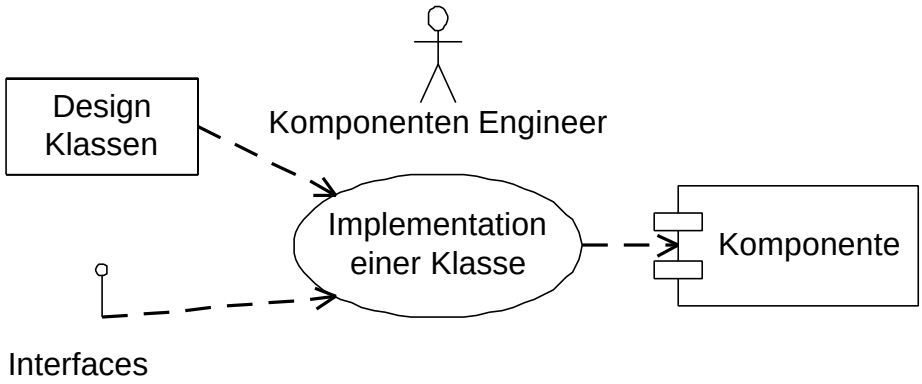


16.5.4. Aktivität : Implementation einer Klasse

Diese Aktivität ergibt sich einfach aus dem Design. Die Umsetzung in eine Implementationsklasse wurde bereits beschrieben: es gilt spezielle Gegebenheiten der Programmiersprache zu berücksichtigen.

In modernen Sprachen wie Java werden die gängigen Konzepte (Class, Package, Interface) und Varianten davon (abstract, protected, public) direkt unterstützt.

In andern Programmiersprachen (Cobol, Visual Basic) werden diese Konzepte zum Teil implementiert, zum Teil mit andern Namen.



16.5.4.1. Skizzieren einer Dateikomponente

Das konkrete Ausgestalten einer Dateikomponente hängt von der Programmiersprache und dem zu Grunde liegenden Dateisystem ab.

In Java hat man versucht, mit Hilfe von Packages und den „file“ Objekten eine Dateisystemunabhängigkeit zu erreichen. Aber früher oder später muss der Zusammenhang mit dem Dateisystem festgelegt werden.

16.5.4.2. Kodegenerierung für die Designklassen : Datenfelder und Methoden

Die Kodegenerierung für die Implementationsklassen wird zum Teil durch die Design Tools wie Rose unterstützt. Aber es wird uns nicht abgenommen!

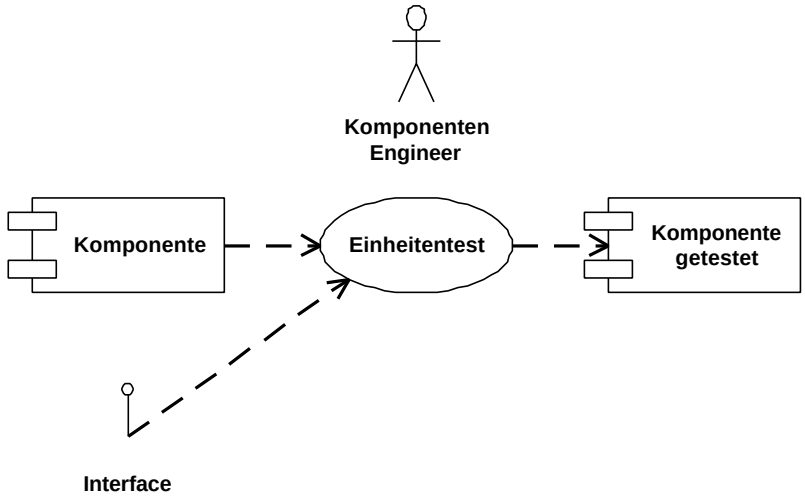
Im Falle von C++ und Java geht es eigentlich nur um die Generierung der Rahmen, Klassen Definitionen, Package Definitionen, MethodenSkeleten.

Algorithmen werden in der Regel nicht automatisch umgesetzt. Das wäre auch zu umständlich, da in diesem Falle die Spezifikation eigentlich bereits so detailliert sein müsste, wie sie eben erst in der Implementation nötig ist. Aber immer hin!

16.5.5. Aktivität : Einheiten testen

Beim Testen kennt man zwei grundsätzliche Vorgehensweisen:

- *Spezifikations-Testen,* oder „black-box testing“, bei der das externe verhalten einer Komponente getestet wird
- *Struktur-Testen,* oder „white-box testing“, bei der die interne Struktur der Komponente getestet wird.



16.5.5.1. Spezifikationstests

Diese Art des Testens beschränkt sich auf die Beantwortung der Frage:

- funktioniert die Komponente so, wie sie spezifiziert wurde?

Als erstes muss man sogenannte *Äquivalenzklassen* definieren, da sich ein System hier eine Komponente, in der Regel auf Eingaben nicht eindeutig verhält:

- mehrere Eingaben können ein und das selbe Verhalten zeigen

Beispiel **Äquivalenzklasse**

Wenn Sie ein Bankkonto haben und Geld abheben können, solange der resultierende Kontostand >= o bleibt, dann führt beim Kontostand von 20.—die Eingabe : Abheben von 30.—oder 50.—zum selben Ergebnis : es wird nichts abgehoben!

16.5.5.2. Struktur Tests

Diese Art des Testens kann nie vollständig sein, weil reale Programme zu komplex sind. Der Tester beschränkt sich also zum Beispiel auf einen „Walkthrough“, bei dem der Programmcode vollständig besprochen und erläutert wird, vor Experten.

Diese Art der Prüfung geschieht typischerweise bei Hochsicherheitsprogrammen.

16.6. Zusammenfassung : Implementation

Das Hauptergebnis der Implementation sind folgende Elemente:

- Subsysteme, mit Interfaces und Inhalt (implementierten Funktionen / Methoden)
- Komponenten : Dateien, ausführbare Komponenten und deren Abhängigkeiten voneinander
- das System als Ganzes, basierend auf einer sauberen Architektur