

Kooperation zwischen RE und SA: Eine Untersuchung von Problemen und Lösungsansätzen

Franz-Dominik Dahmann

Master Informatik

Hochschule Bonn-Rhein-Sieg

<https://www.h-brs.de/de>

Grantham-Allee 20, 53757 Sankt Augustin

Email: franz.dahmann@smail.inf.h-brs.de

Jan Eric Müller

Master Informatik

Hochschule Bonn-Rhein-Sieg

<https://www.h-brs.de/de>

Grantham-Allee 20, 53757 Sankt Augustin

Email: jan-eric.mueller@smail.inf.h-brs.de

Zusammenfassung—In der Softwareentwicklung sind das Requirements Engineering (RE) und der Entwurf einer Software-Architektur (SA) äußerst wichtige Bestandteile. Ausgehend von einer Menge von Anforderungen wird eine Software-Architektur konzipiert. Die Softwarelösung wird dann entsprechend der Software-Architektur implementiert. Im Idealfall entspricht die fertiggestellte Softwarelösung dann der System-Vision des Kunden. In der Realität gibt es jedoch an vielen Stellen Probleme, die dazu führen, dass schon die Software-Architektur sich deutlich von den Anforderungen oder der System-Vision unterscheidet. Besonders in der Phase, in der aus den Anforderungen die Software-Architektur generiert wird, können starke Unterschiede zu der System-Vision auftreten. In dieser Arbeit werden Methoden des Requirements Engineering und der Software-Architektur Konzeption vorgestellt und verglichen, die das Potenzial haben, diese Probleme zu lösen.

I. EINFÜHRUNG

Bei der Realisierung eines Softwareentwicklungsprojektes kann eine Kluft zwischen den Software-Architekten auf der einen Seite und den Requirements Engineers auf der anderen Seite bestehen. Software-Architekten können mit dem Problem konfrontiert sein, dass Anforderungsdokumente nicht ausreichend sind, um weitreichende Architekturentscheidungen in Bezug auf den Softwareentwurf zu treffen. Daher entsteht in diesem Fall für den Software-Architekten häufig der Mehraufwand, sodass dieser über weitere Interviews ein präziseres Bild von der gewünschten Software-Architektur gewinnen muss [10]. Dies resultiert in Verzögerungen, die sich dann darin widerspiegeln, dass Termine nicht eingehalten werden können. Betreibt der Software-Architekt diesen Mehraufwand nicht und trifft eigene Annahmen bezüglich der Software-Architektur [10] kann dies wiederum zu einer geringeren Akzeptanz des Kunden und im schlimmsten Fall zum Auftragsverlust führen. Die Requirements Engineers wissen auf der anderen Seite nicht, welche Anforderungen konkret wichtig für den Software-Architekten sind, da ihnen die entsprechende Fachkenntnis fehlt [10]. Somit können diese nicht zielführend die notwendigen Informationen mit dem Kunden erarbeiten.

Ohne diese Informationen ist eine ausreichende Grundlage der Anforderungen für den Architekturentwurf nicht gegeben. Um diese Kluft zu überbrücken ist es notwendig Verfahren zu ermitteln, die es Requirements Engineers ermöglicht, die richtigen Informationen einzuholen und die Zusammenarbeit mit den Software-Architekten zu optimieren.

Im Folgenden wird zunächst in der Untersuchung möglicher Probleme klar hervorgehoben, welche Probleme es in der Zusammenarbeit von Software-Architekten und Requirements Engineers gibt. Nachfolgend werden vier Verfahren untersucht die das Potenzial haben die Zusammenarbeit zwischen Requirements-Engineer und Software-Architekt zu verbessern. Diese vier Verfahren sind:

- Probing wird beschrieben in III-A
- CBSP wird beschrieben in III-B
- COSMOD-RE wird beschrieben in III-C
- ADD 3.0 wird beschrieben in III-D

Diese vier Verfahren werden nach einem festgelegten Schema untersucht. Hierbei wird sowohl untersucht, was die Methoden bezeichnen wollen, als auch wie die Methoden arbeiten. Auf dieses Schema wird in III näher eingegangen.

Nach der Beschreibung der einzelnen Methoden werden diese vergleichend in IV ausgewertet. Hierfür werden die Methoden zunächst unter den eben genannten Punkten verglichen. Danach werden sie in den Kontext der in II genannten Probleme gebracht und hinsichtlich ihrer Fähigkeit, diese zu lösen, betrachtet. Es wird überprüft, wo diese Verfahren ihre Stärken und Schwächen offenbaren.

Abschließend wird überprüft, wie diese Verfahren gewinnbringend kombiniert werden oder sich gegenseitig ergänzen können.

II. UNTERSUCHUNG MÖGLICHER PROBLEME

Während der Zusammenarbeit zwischen Requirements Engineer und Software-Architekt können vielfältige Probleme auftreten. Bei der Untersuchung möglicher Probleme lässt sich eine Kategorisierung dieser vornehmen. So sind einige Probleme bedingt durch den Menschen selbst, während andere Probleme sich aus den Anforderungen ergeben. Im Folgenden werden die kategorisierten Probleme genauer ausgeführt.

A. Durch den Menschen bedingt

Bei der Betrachtung der direkt durch den Menschen bedingten Probleme fallen folgende auf:

P1: *Schlechte Kommunikation*

P2: *Konkurrierende Interessen*

P3: *Fehlendes Know-How*

1) Schlechte Kommunikation: Eine schlechte Kommunikation ist in diesem Zusammenhang als eine solche zu sehen, bei der wesentliche Intentionen und Informationen nicht übermittelt werden. Ein wichtiger Aspekt schlechter Kommunikation ist die Distanz zwischen den Akteuren innerhalb eines Projektes.

Nach [1] kann vor allem Distanz ein Grund für schlechte Kommunikation sein. Distanzen in der Kommunikation zwischen Requirements Engineer und Software-Architekt können die Koordination reduzieren, was wiederum Projektverzögerungen hervorrufen kann, sowie das Nichterfüllen von Kundenwünschen. Distanzen werden nach [1] vor allem als räumliche Distanzen bei global agierenden Projektteams gesehen. Bei räumlichen Distanzen können nach [2] beispielsweise Begleitprobleme wie das Beherrschen einer Sprache auftreten. So kann das schlechte Beherrschen einer Sprache zu Missverständnissen führen und Fehler in den Anforderungen hervorrufen. Zusätzlich können nach [1] soziale, kulturelle oder zeitliche Distanzen hinzukommen.

Bei nicht ausreichendem Kommunikationsfluss zwischen Requirements Engineer und Software-Architekt kann das Problem auftreten, dass der Requirements Engineer während der Anforderungsgewinnung keine Rücksprache mit dem Software-Architekten hält [2]. Ohne ausreichende Rücksprache kann beispielsweise eine Beeinträchtigung der Qualität der architekturelevanten Anforderungen auftreten [2]. Dies kann von fehlenden bis fehlerhaften Anforderungen reichen. Ein weiteres mögliches Problem ist, dass der Software-Architekt dem Requirements Engineer nicht ausreichend vermittelt, welche Informationen er für einen gültigen Architekturentwurf benötigt [2]. Aus diesen Kommunikationsproblemen können Missverständnisse in den Verantwortlichkeiten folgen [2]. Auch hier ist eine mögliche Folge die negative Beeinflussung der Qualität der

Architekturanforderungen.

2) Konkurrierende Interessen: Unter konkurrierenden Interessen ist zu verstehen, dass die Stakeholder des Entwicklungsprozesses der Software-Architektur in der Projektarbeit verschiedene Interessen verfolgen, die miteinander in Konflikt stehen. Die Stakeholder haben zwar eine gemeinsame Vision von dem fertigen Produkt, verfolgen aufgrund variierender Interessen jedoch eine unterschiedliche Art der Zielerreichung [9]. Weiter kann es sein, dass Stakeholder durch ihre unterschiedlichen Sichten auf das Projekt auf verschiedene Arten mit diesem interagieren. Dies kann wiederum mit den Projektzielen konkurrieren. Der Requirements Engineer vertritt hierbei eher die Sicht des Kunden, während der Software-Architekt, den Standpunkt des Entwicklers vertritt [8]. Ein Problem kann ebenfalls auftreten, wenn Stakeholder verschiedene Tools, Prozesse und Arbeitsweisen nutzen, um das Projektziel zu erreichen. Wenn hier die Kompromissbereitschaft fehlt, kann dies die Arbeit erschweren [2].

3) Fehlendes Know-How: Fehlendes Know-How kann zu diversen Problemen führen. Dabei kann hier zwischen zwei Ebenen unterschieden werden. Einerseits können Probleme aus fehlendem Know-How über die Methodik der jeweils anderen Rolle (Software-Architekt oder Requirements Engineer) entstehen [9][2]. Hierbei können unter anderem Probleme bei der Planung und Umsetzung eines Projektes resultieren, da hierdurch potenziell relevante Vorgehensweisen oder Methoden nicht berücksichtigt werden können. Andererseits können Probleme aus inhaltlich fehlendem Know-How entstehen [9]. So kann es dem RequirementsEngineer zum Beispiel schwer fallen die Schwerpunkte so zu setzen, dass der Software-Architekt die Informationen erhält, die er benötigt [12]. Dies wird dann zum Problem wenn der Requirements Engineer nicht weiß welche Informationen der Software-Architekt für die Umsetzung benötigt. Ferner können Missverständnisse auftreten, wenn der Software-Architekt bei fachspezifischen Begriffen ein anderes Verständnis hat, als der Requirements Engineer [2].

B. Qualität der Anforderungen

Bei der Untersuchung der Probleme, die sich auf die Qualität der Anforderungen beziehen, fallen folgende auf:

P4: *Zu restriktive/detaillierte architekturelevante Anforderungen*

P5: *Ungenaue/fehlende architekturelevante Anforderungen*

P6: *Nicht klar hervorgehobene architekturelevante Anforderungen*

P7: *Wechselwirkungen zwischen architekturelevanten Anforderungen*

1) Zu restriktive/detaillierte architekturelevante Anforderungen: Vor allem bei sehr großen Projekten

kann es passieren, dass es tausende von Anforderungen, die für die Architektur relevant sind, gibt [9]. Es besteht eine hohe Wahrscheinlichkeit, dass in einem solchen Fall viele Anforderungen Widersprüche erzeugen und den Lösungsraum zu sehr einschränken. Hier können restriktive oder detaillierte Anforderungen die Arbeit von Requirements Engineer und Software-Architekt unnötig einschränken und dadurch erschweren.

2) Ungenaue/Fehlende architekturelle relevante Anforderungen: In der Anforderungsgewinnung kann es passieren, dass wichtige architekturelle relevante Anforderungen nicht erhoben werden oder wichtige Details fehlen [10][11]. Dies kann jedoch weitreichende Auswirkungen auf die Software-Architektur haben, da dem Software-Architekten notwendige Informationen bei der Konzeption fehlen. Dieser sieht sich dann gezwungen entweder eine Software-Architektur frei zu entwerfen oder zusätzliche klärende Gespräche mit dem Kunden zu führen. Auch hier kann es passieren, dass bei besonders großen Projekten mit mehreren tausenden Anforderungen, wichtige nicht erhoben werden [9]. Vor allem im Kontext der agilen Softwareentwicklung bedingt ein iteratives Vorgehen, dass manche architekturellen Anforderungen erst während der Modellierung oder sogar Implementierung der Architektur gewonnen werden [9].

3) Nicht klar hervorgehobene architekturelle relevante Anforderungen: Eines der Hauptprobleme architektureller Anforderungen liegt darin, dass es üblicherweise schwer ist, diese zu identifizieren und zu spezifizieren [9]. Werden architekturelle Anforderungen für den Software-Architekten nicht hervorgehoben, kann es passieren, dass dieser wichtige Anforderungen erst spät, wenn überhaupt, wahrnimmt [12]. Dies kann zur Folge haben, dass wichtige architekturelle Entscheidungen nicht rechtzeitig getroffen werden können und zu einem Mehraufwand zu einem späteren Zeitpunkt führen [10]. Ein weiteres Problem liegt in der fehlenden Ausführung der Auswirkungen von architekturellen Anforderungen auf die spätere Software-Architektur [12]. So kann hieraus folgen, dass wichtige Projektziele nicht erfüllt werden, da der Zeitrahmen des Projektes überschritten wird.

4) Wechselwirkung zwischen Anforderungen: Unter Wechselwirkung zwischen Anforderung ist zu verstehen, dass Anforderungen teilweise voneinander abhängig sein können. Diese Abhängigkeit kann auf verschiedene Arten und Weisen erfolgen. Ist beispielsweise ein Ansatz gegeben, bei dem mehrere Abstraktionsstufen vorhanden sind, kann es passieren, dass wichtige Abhängigkeiten, die über mehrere Abstraktionsstufen hinweg reichen, nicht berücksichtigt werden oder weitere Probleme verursachen [6]. Änderungen von Anforderungen auf verschiedenen Hierarchieebenen treffen kontinuierlich ein und können somit Einfluss auf die Architektur haben und erheblichen Änderungsbedarf verursachen [7]. Auch ist es schwierig, die Konsistenz

und Nachverfolgbarkeit der Projektion von Anforderungen auf die Software-Architektur zu gewährleisten, da eine Anforderung mehrere architekturelle relevante Artefakte betreffen kann. Zusätzlich kann ein architekturell relevanter Artefakt mehrere Anforderungen betreffen [9].

Neben den aufgeführten Problemen gibt es weitere, die hier nicht näher behandelt werden. Darunter fällt zum Beispiel ein phasenbezogenes Requirements Engineering.

Die hier aufgeführten Probleme ermöglichen es, nach vergleichbaren Kriterien Methoden zu untersuchen. Hierdurch ist es möglich, zu überprüfen, inwiefern diese geeignet sind, eine Verbesserung der Kooperation zwischen Requirements Engineer und Software-Architekt zu erzielen.

III. UNTERSUCHUNG GEGEBENER METHODEN

Da die beiden Bereiche des Requirements Engineering und der Software-Architektur ein enormes Maß an Wissen und Fähigkeiten erfordern, werden diese in der Regel von verschiedenen Teams betreut [11]. Wie bereits genannt, entstehen hier häufig Probleme durch fehlendes technisches Know-How über Software-Architektur spezifische Aspekte bei den Requirements Engineers. Auch können diese oft nicht zwischen funktionalen Anforderungen (FR) und architektspezifischen funktionalen Anforderungen (ASFR) unterscheiden [12]. ASFRs sind FRs, welche kritisch, sehr risikobehaftet, volatil und bei Änderungen ein aufwendiges oder teures Refactoring mit sich bringen würden oder einen anderweitig großen *signifikanten* Impact auf die zu konzipierende Software-Architektur hätten [11]. Durch das fehlende Know-How entstehen unvollständige Anforderungs-Artefakte, in welchen wesentliche ASFRs für den Software-Architekten fehlen.

Sei ein vereinfachter Entwicklungsprozess wie in Abbildung 1 gegeben. Zu sehen ist ein Kunde, der eine System-Vision hat. Mit dieser Vision wendet er sich an einen Requirements Engineer, der einerseits über Gespräche und andererseits über seine Fähigkeiten versucht, die System-Vision in Form von Anforderungen festzuhalten. Diese Anforderungen werden an den Software-Architekten weitergegeben, der wiederum versucht aus den Anforderungen einen Architekturentwurf zu generieren. Diesen Architekturentwurf wird er an die Entwickler weitergeben. Diese entwickeln dann das konzipierte System und liefern es als fertiggestelltes System an den Kunden. In der Betrachtung der Zusammenarbeit zwischen Software-Architekt und Requirements Engineer sind vor allem die Punkte von Interesse, die vor der fertiggestellten Software-Architektur stehen. Somit sind als Stakeholder vor allem Kunde, Requirements Engineer und Software-Architekt von Bedeutung. Einfach formuliert soll es das Ziel sein, die System-Vision in einer korrekten Software-Architektur festzuhalten, die die Anforderungen erfüllt.

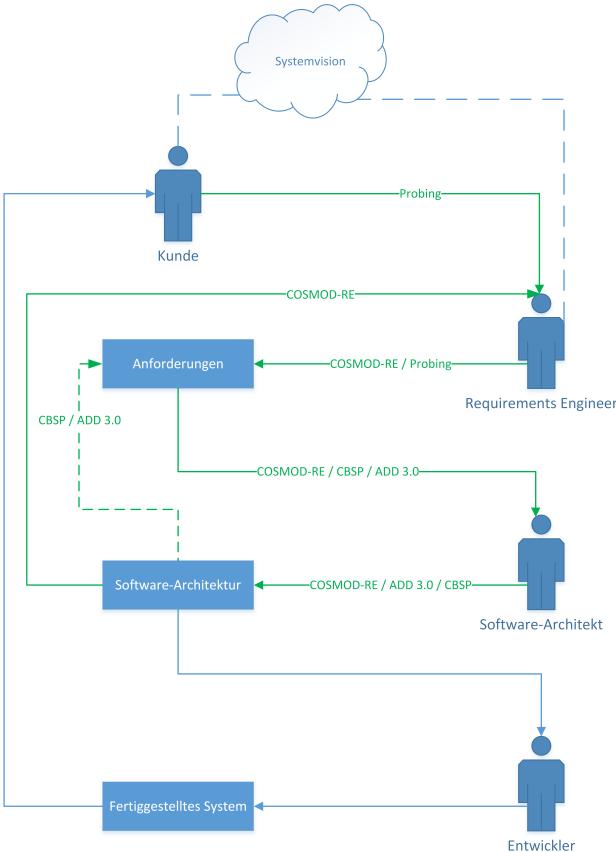


Abbildung 1. Überblick über die untersuchten Methoden

Die Pfeile in der Abbildung sind als Informationsflüsse zu sehen, die methodisch unterstützt werden. So stellt der Pfeil von Kunde zu Requirements Engineer dar, wie Anforderungen oder System-Vision des Kunden beispielsweise über ein Interview an den Requirements Engineer weitergegeben und in Form von Anforderungen formalisiert festgehalten wird. Im Fall von COSMOD-RE wird die System-Vision formalisiert in den Anforderungen festgehalten und an den Software-Architekten weitergegeben. Dieser entwirft eine entsprechende Software-Architektur, die dann zu einem weiteren Abgleich wieder an den Requirements Engineer überreicht wird. Der gestrichelte Pfeil von der Software-Architektur zu den Anforderungen soll das iterative Vorgehen der beiden annotierten Methoden verdeutlichen.

Im Folgenden sollen vier verschiedene Methoden untersucht werden, welche den Anspruch haben, die Kluft in der Zusammenarbeit zwischen Software-Architekt und Requirements Engineer zu überbrücken. Dabei werden zunächst die Ziele der Methoden herausgearbeitet und anschließend die Funktionsweise näher erläutert, bevor die vier Methoden im nachfolgenden Kapitel ausgewertet werden. Bei der Funktionsweise wird auf folgende Aspekte genauer eingegangen:

- **Randbedingungen:** In den Randbedingungen wird erklärt, welche Einschränkungen die Methode hat und welche Voraussetzungen geklärt sein müssen um die Methodik anzuwenden.
- **Eingabe:** Die Eingabe beschreibt, welche Artefakte für die Ausführung der Methode benötigt werden.
- **Vorgehensmodell:** Mit dem Vorgehensmodell wird beschrieben, wie die Methode anzuwenden ist.
- **Ausgabe:** Die Ausgabe beschreibt, was am Ende das Resultat der Methode ist und welchen Mehrwert dieses haben könnte.

A. Probing (JEM)

Das Ziel des Probing ist es, Requirements Engineers mit Fragen auszustatten, die eine Erhebung architektureller Anforderungen, den ASFRs, ermöglichen. Diese Fragen werden Probing Questions (PQ) genannt.

1) Ziele der Methode: Die Haupttreiber architektureller Entscheidungen sind nicht-funktionale (NFR) bzw. qualitative Anforderungen [12]. Diese haben eine explizite Auswirkung auf die zu erstellende Architektur. Bei funktionalen Anforderungen bzw. ASFRs sind diese Auswirkungen meist implizit und müssen zunächst durch weitere Interviews mit dem Kunden herausgearbeitet werden. Auch sind hier die für einen Software-Architekten erforderlichen Informationen nicht immer klar in der Anforderung aufgeführt [12]. Dies kann zu falschen architektonischen Entscheidungen führen. Um dieses Problem der nicht klar hervorgehobenen architekturellen Anforderungen anzugehen, beschäftigen sich die Probing Questions mit den ASFRs. Requirements Engineers sollen mit Probing Questions ausgestattet werden um zusätzliche relevante Fragen stellen zu können. Dadurch soll während der Anforderungserhebung eine vollständigere Anforderungsspezifikation erstellt werden, in welcher die ASFRs aussagekräftiger sind [12]. Auch helfen diese den Requirements Engineers, ein genaueres Verständnis für den Software-Architekten aufzubauen, welche Informationen er für die Konzeption und Implementierung der Software-Architektur benötigt.

2) Funktionsweise der Methode: Da das Probing eine Art Interview-Leitfaden bestehend aus PQs für die ausführlichere Extraktion von ASFRs und keinen methodischen Leitfaden für die Anwendung dieser zur Verfügung stellt, wird in den folgenden Paragraphen auf die PQs und deren Kategorisierung und nicht auf die Durchführung eines Interviews mit diesen eingegangen.

a) Randbedingungen: Das Probing benötigt, außer den mitgebrachten PQs, keine weiteren Artefakte. Allerdings müssen zwei offensichtliche Bedingungen erfüllt sein, damit das Probing den Nutzen bringt den es verspricht. Zum einen müssen die PQs oder sogar die PQ-Flows dem Requirements

Engineer zur Verfügung gestellt werden. Zum anderen muss dieser auch bereit sein, diese bei der Anforderungsgewinnung zu nutzen.

Damit die PQs und PQ-Flows genauer auf ein Projekt zugeschnitten sind und besser auf einen konkreten Anwendungsfall angewandt werden können, müssen diese von einem erfahrenen Software-Architekten ausgewählt, angepasst oder ergänzt werden. Dafür benötigt dieser allerdings ausreichend Wissen und Erfahrung in der Domäne und über den Kunden [11]. Mehr dazu unter III-A2c.

b) *Eingabe:* Für die Anwendung stellt das Probing sowohl einen Katalog aus PQs als auch zugehörige PQ-Flows bereit. Damit die PQs möglichst alle Bereiche der ASFRs abdecken wurden letztere, wie in Tabelle I zu sehen, in 15 Kategorien unterteilt. Die Kategorien wurden durch die Analyse von 450 FRs aus 30 Anforderungsspezifikationen und durch mehrere Interviews mit erfahrenen Software-Architekten aus verschiedenen Domänen und Ländern aufgestellt [11] [12]. Sie bilden einen ersten Ansatz für die Strukturierung von PQs und sollen in Zukunft verfeinert werden [12].

Tabelle I
ASFR KATEGORIEN NACH [11]

ASFR Kategorie	Beschreibung
Audit Trail	Auditionierung der Systemausführung ermöglichen
Batch-Verarbeitung (Batch Processing)	Batch-Verarbeitung ermöglichen
Lokalisierung	Unterstützung für mehrere Sprachen anbieten
Geschäftsprozess "Status" Meldungen (Kommunikation)	Unterstützung für Kommunikation anbieten
Datenbezogene Dialoge	Unterstützung verschiedener Dateneingabemechanismen
Bezahlung (Payment)	Finanzielle Transaktionen ermöglichen
Druck	Unterstützung zum Drucken von Dokumenten anbieten
Berichte (Report)	Die Generierung von Berichten ermöglichen
Suche	Anbieten von Suchfunktionen
Drittanbieter-Interaktion	Interaktion mit Drittanbieter-Komponenten ermöglichen
Arbeitsfluss (Workflow)	Bereitstellung von Funktionen um Arbeitselemente zu verschieben, Gutachten und Genehmigungen ermöglichen
Online-Hilfe	Online-Hilfe ermöglichen
Lizenziierung	Dienste für die Beschaffung, Installation und Beobachtung der Lizenzbenutzung ermöglichen
Analyse von Benutzerverhalten	Sammeln, Analysieren und Aggregieren von Benutzerverhaltensdaten ermöglichen
Speichermechanismen	Bereitstellung von automatischen Mechanismen zur Speicherung und Konvertierung von Dokumenten

Zu jeder Kategorie von ASFRs sind sowohl mehrere PQs als auch beispielhafte ASFRs für ein besseres Verständnis der Kategorie notiert. Für die vierte Kategorie (Kommunikation) lautet eine Beispiel-ASFR *Ein Arbeitsfluss ist erforderlich, um*

Benachrichtigungen an die Versicherer zu senden. Mögliche PQs hier sind Soll die Kommunikation unidirektional oder bidirektional verlaufen? oder Soll die Kommunikation vom Kunden konfigurierbar sein? [12]. Für die Kategorie Suche wurden PQs wie Soll die Suche schlüsselwort- oder filterbasiert sein? oder Ist eine semantische Suche notwendig? aufgenommen [12].

Des Weiteren sind die aufgestellten PQs in sechs Typen unterteilt [11]:

1. *Geschäftsregeln im Unternehmen des Kunden:* Einige PQs beziehen sich auf Einschränkungen, welche durch Unternehmensregeln gegeben sind, so beispielsweise *Wie lange muss ein Audit-Trail gespeichert werden?* oder *Welche Informationen möchte das Unternehmen im Audit-Trail sehen?* [11].
2. *Strategische Technologieentscheidungen:* PQs von diesem Typ beziehen sich auf die architektonischen Auswirkungen, welche durch Investitionsentscheidungen des Unternehmens des Kunden begründet sind [11].
3. *Nicht-funktionale Aspekte bezogen auf die ASFRs:* Viele PQs behandeln relevante nicht-funktionale Aspekte, die bei einer architekturelevanten Entscheidung berücksichtigt werden müssen. Zum Beispiel beziehen sich folgende PQs auf Skalierbarkeit und Sicherheit: *Wie groß ist die Menge an Daten?, Wie viele Haupttabellen werden für die Auditierung benötigt? oder Welches Sicherheitslevel wird bei sensitiven Daten benötigt?* [11].
4. *Einhaltung gesetzlicher Vorschriften:* PQs, welche von gesetzlichen Vorschriften abhängen haben einen großen Einfluss auf architektonische Entscheidungen. Wenn in einem Land beispielsweise bestimmte Inhalte von Gesetzeswegen verboten sind, dann hat diese Einschränkung einen großen Einfluss auf die zu konzipierende Software-Architektur [11].
5. *Projektkontext:* Hierzu gehören PQs, die kontextuelle Parameter des Projekts betreffen und Auswirkungen auf die Software-Architektur haben. Diese beinhalten PQs, welche die funktionale Passform, einen Geschäftsfall, Kosten, den Zeitrahmen, die zur Verfügung stehenden Technologien, Fähigkeiten auf Kunden- und Verkäuferseite, Unternehmenskultur, Kundenmentalität oder sogar strategische Anforderungen betreffen [11].
6. *Zusammengesetzte Wirkung auf zwei oder mehr ASFRs:* Eine PQ, welche Antworten zu mehr als einer ASFR Kategorie liefert, gehört zu diesem Typ. Eine Beispiel-PQ für die Kategorien Geschäftsprozess "Status" Meldungen und Drittanbieter-Interaktion lautet: *Was für eine Bestätigung möchten sie vom Drittanbieter-Gateway bei erfolgreicher Zahlung empfangen?* [11]

Diese Typisierung zeigt die Abhängigkeiten von ASFRs zu anderen ASFRs, NFRs oder anderen Strukturen im Projekt oder Unternehmen auf [11]. Dies ermöglicht, auf diese zu

reagieren und daraus resultierenden Problemen rechtzeitig vorzubeugen.

Zudem wurden für fünf der 15 ASFR-Kategorien ganze PQ-Flows erstellt. Die fünf Kategorien sind *Audit Trail*, *Batch-Verarbeitung*, *Geschäftsprozess "Status"* *Meldungen (Kommunikation)*, *Berichte* und *Arbeitsfluss*. Die Idee hier ist, das architektonische Wissen, welches den PQs innewohnt [11], in einer Wissensdatenbank aus PQ-Flows zu sammeln, ähnlich wie die Design Patterns nach Gamma et al. [12] um eine verbesserte und vollständigere Spezifikation zu erhalten.

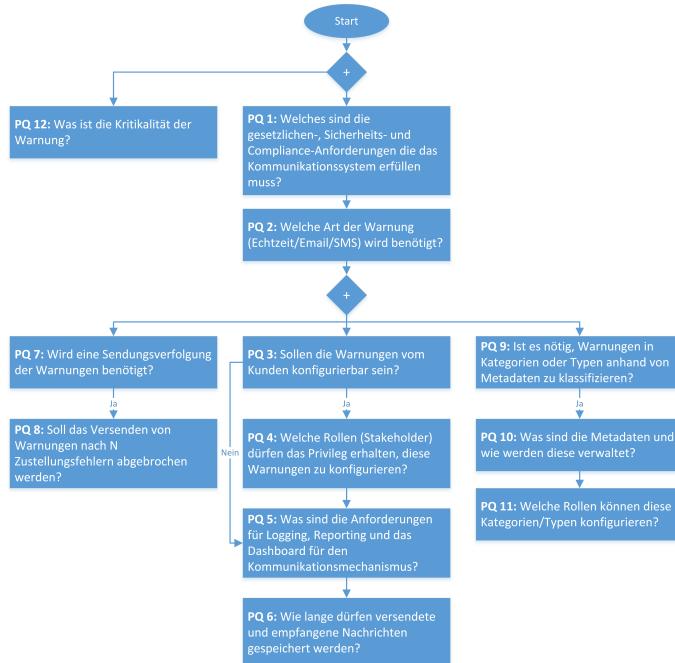


Abbildung 2. PQ-Flow für Geschäftsprozess "Status" Meldungen (Kommunikation) [10]

Ein PQ-Flow ist eine Art Fluss- oder Ablaufdiagramm aus PQs. Hier sind mehrere PQs so angeordnet, dass die Reihenfolge, in welcher sie im Interview benutzt werden eine Rolle spielt. Dadurch soll dem Requirements Engineer eine bessere Hilfestellung geben werden, da hier einige Fragen abhängig von den Antworten vorhergegangener Fragen irrelevant werden können. So ist es logisch sinnvoll die PQ *Welche Rollen (Stakeholder) dürfen das Privileg erhalten, diese Warnungen zu konfigurieren?* (PQ 4) aus Abbildung 2 nur zu stellen, wenn die vorherige PQ (PQ 3) mit einem Ja beantwortet wurde.

c) *Vorgehensmodell:* Die bereits existierenden PQs und PQ-Flows können ohne weitere Änderungen dem Requirements Engineer an die Hand gegeben werden, da diese bereits in der Praxis erfolgreich getestet wurden und laut [10], [11] und [12] einen Mehrwert bieten. Allerdings bieten diese einen relativ allgemeinen Ansatz und sind nicht auf ein konkretes Projekt einer bestimmten Domäne zugeschnitten.

Alternativ können die PQs mit zusätzlichem Wissen des Software-Architekten eines konkreten Projektes angepasst oder ergänzt werden, wenn ihm bestimmte Aspekte für sein Projekt in den PQs nicht ausreichend abgedeckt erscheinen [11]. Hierzu benötigt dieser jedoch Erfahrung und Wissen in folgenden Bereichen [11]:

1. Detailliertes Domänen-Wissen
2. Kenntnisse über das Geschäftsfeld des Kunden sowie über den Projektzyklus im Unternehmen des Software-Architekten
3. Wissen über das Unternehmen des Kunden

Für diese Anpassungen gilt allerdings: "Erfahrung spielt eine signifikante Rolle" [11].

d) *Ausgabe:* Nachdem die PQs vom Requirements Engineer bei der Anforderungserhebung benutzt wurden erhält dieser daraufhin eine vollständige Spezifikation mit allen vom Software-Architekten, benötigten ASFRs.

B. Component-Bus-System-Property (CBSP) (JEM)

Component-Bus-System-Property, kurz CBSP, ist ein leichtgewichtiger Ansatz, um Anforderungen und Architektur mit Hilfe von Zwischenmodellen in Übereinstimmung zu bringen [9]. Dazu wird iterativ aus den Anforderungen ein CBSP-Modell gebildet, welches Anforderungs- und Architektur-Modellelemente miteinander verknüpft.

1) *Ziele der Methode:* Der CBSP-Ansatz hilft Software-Architekten dabei, Architekturelemente wie Komponenten und Konnektoren, Architektur-Eigenschaften, Abhängigkeiten zwischen diesen Elementen und passende Architekturstile zu finden. Dabei unterstützt das CBSP den Software-Architekten dabei folgende Herausforderungen, um Anforderungen und Architektur in Übereinstimmung zu bringen, zu behandeln:

- *Überbrückung unterschiedlicher Formalitätsebenen:* Das Zwischenmodell von CBSP reduziert die semantische Lücke zwischen meist informell in natürlicher Sprache gehaltenen Anforderungen auf einer höheren Abstraktionsebene und eher formell gehaltenen Architekturbeschreibungen [9].
- *Modellierung nicht-funktionaler Anforderungen:* CBSP ermöglicht, die sonst schwere Modellierung von nicht-funktionalen Anforderungen als Architekturmodell sowohl auf der System- wie auch der Architekturbene [9].
- *Aufrechterhaltung evolutionärer Konsistenz:* Aufrechterhaltung von Konsistenz und Nachverfolgbarkeit ist ein schwieriges Unterfangen, da eine einzelne Anforderung mehrere architektonische Anliegen betreffen kann und ein einzelnes Architekturelement typischerweise mehrere Beziehungen zu verschiedenen Anforderungen hat [9]. Diese Schwierigkeiten werden vom CBSP-Zwischenmodell behoben.

- *Unvollständige Modelle und iterative Entwicklung:* Das CBSP-Zwischenmodell benötigt auf Grund seines iterativen Vorgehensmodells keine von Beginn an vollständige Anforderungsspezifikation. Des Weiteren können bestimmte Anforderungen erst vollständig verstanden werden, wenn die Software-Architektur modelliert oder sogar partiell implementiert wurde [9].
- *Größe und Komplexität behandeln:* Großsysteme müssen meist hunderte bis tausende von Anforderungen erfüllen. Da CBSP sich in jeder Iteration nur mit einem Teil der architektonisch relevanten Anforderungen beschäftigt, kann es diese Komplexität beherrschen und den Fokus erhöhen. Jede Aktivität von CBSP befasst sich mit der Filterung von Anforderungen oder dem Zusammenfassen mehrerer Anforderungen in eine [9].
- *Verschiedene Stakeholder mit unterschiedlichen Interessen:* Anforderungen und Architektur in Einklang zu bringen ist auch ein Prozess, in welchem heterogene Stakeholder mit widersprüchlichen Zielen, Erwartungen und Terminologien involviert sind. CBSP versucht hier die richtige Balance zwischen diesen abweichenden Interessen zu finden, indem es wichtige Stakeholder involviert [9].

Um bei der Bewältigung dieser Herausforderungen zu unterstützen bietet das CBSP:

- einen leichtgewichtigen Ansatz Anforderungen zu verfeinern durch die Bereitstellung eines kleinen erweiterbaren Sets von Architektur-Schlüsselkomponenten,
- einen Mechanismus um die Anzahl relevanter Anforderungen zu reduzieren und auf die relevantesten ASFRs zu fokussieren,
- Beteiligung von wichtigen Stakeholdern,
- einen regulierbaren Voting-Mechanismus um Konflikte und unterschiedliche Auffassungen zwischen Architekten zu beheben und
- Toolunterstützung für bestimmte Schritte des Ansatzes [9].

2) *Funktionsweise der Methode:* Das CBSP erweitert, wie in Abbildung 3 zu sehen ist, den Gedanken des Twin-Peaks Modells. Das Zwischenmodell des CBSP verknüpft die Anforderungen mit der Architektur. Das iterative Vorgehen kann dabei sogar auf verschiedenen Detail- bzw. Abstraktionsstufen angewandt werden [9]. Um das Vorgehen genauer verstehen zu können wird zunächst die Taxonomie des CBSP betrachtet.

CBSP Taxonomie: Das Metamodell von CBSP beinhaltet die Basiskonstrukte einer Architektur und ist in sechs Dimensionen aufgeteilt:

1. C: sind Elemente, welche individuelle Komponenten der Software-Architektur beschreiben oder involvieren. Diese

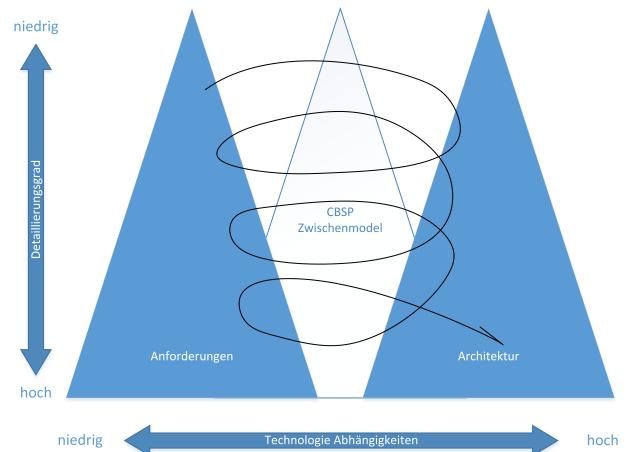


Abbildung 3. CBSP Modell Kontext [9]

sind in Datenkomponenten (C_D) und Verarbeitungskomponenten (C_P) unterteilt. So könnte beispielsweise die Anforderung:

R: Benutzern das direkte manipulieren von Kalkulations-tabellen erlauben.

in folgende CBSP Modellelemente verfeinert werden:

C_P: UI Komponente für Kalkulationstabellen Manipula-tion.

C_D: Daten für Kalkulationstabellen. [9]

2. B: sind Elemente, welche Konnektoren beschreiben. Zum Beispiel:

R: Manipulierte Daten in Kalkulationstabellen müssen im Dateisystem gespeichert werden.

kann in folgendes CBSP Modellelement verfeinert wer-den:

B: Konnektor, der die Interaktion zwischen UI und Persistenz-Komponenten ermöglicht. [9]

3. S: sind Elemente, welche systemweite Features beschreiben oder solche, die zu einer größeren Untermenge von Komponenten und Konnektoren passen. Zum Beispiel:

R: Der Benutzer soll angemessene Filter und Visualisierungen auswählen können.

kann in folgendes CBSP Modellelement verfeinert werden:

S: Das System soll eine strikte Trennung von Datenhaltungs-, -Bearbeitungs- und -Visualisierungs-Komponenten vornehmen. [9]

4. CP: sind Elemente, welche Eigenschaften, wie Zuverlässigkeit, Portabilität, Skalierbarkeit, Anpassbarkeit und Erweiterbarkeit, von Daten- oder Verarbeitungskomponenten beschreiben. Zum Beispiel:

R: Der Benutzer soll die Daten entfernt mit minimale wahrgenommener Latenz visualisieren können.

kann in folgendes CBSP Modellelement verfeinert wer-den:

C_P: Die Datenvisualisierungs-Komponente soll effizient sein und inkrementelle Updates unterstützen. [9]

5. *BP*: sind Elemente, welche Eigenschaften von Konnektoren beschreiben. Zum Beispiel:

R: Updates für Systemfunktionen sollen mit minimaler Ausfallzeit möglich sein.

kann in folgendes CBSP Modellelement verfeinert werden:

BP: Robuste Konnektoren sollen zur Verfügung gestellt werden, um das Hinzufügen und Entfernen von Laufzeitkomponenten zu erleichtern. [9]

6. *SP*: sind Elemente, welche systemweite Eigenschaften bzw. Eigenschaften von Subsystemen beschreiben. Zum Beispiel:

R: Die Daten der Kalkulationstabellen müssen verschlüsselt werden, bevor sie über das Netzwerk versandt werden.

kann in folgendes CBSP Modellelement verfeinert werden:

SP: Das System soll sicher sein. [9]

Aus diesen Elementen wird das CBSP Zwischenmodell aufgebaut. Das Metamodell zu diesen Elementen ist in Abbildung 4 dargestellt. Jedes CBSP Artefakt beschreibt ein architekturerelevantes Anliegen und beschreibt eine frühe Architekturentscheidung des zu erstellenden Systems [9].

Von einer Anforderung zu einem CBSP Element kann eine Verfeinerung, siehe Beispiel (5), oder Generalisierung, siehe Beispiel (6), erfolgen. Diese beiden möglichen Anpassungen basieren auf Notwendigkeiten, welche durch bestimmte Eigenschaften des zu erstellenden Systems, Charakteristika der Anwendungsdomäne oder Hintergrund und Erfahrung des Software-Architekten, erwachsen [9]. Da es für Verfeinerung und Generalisierung keine allgemeinen formalen Regeln gibt, ist es häufig notwendig, hierfür den Kunden bzw. weitere Stakeholder wie den Requirements Engineer zu konsultieren [9].

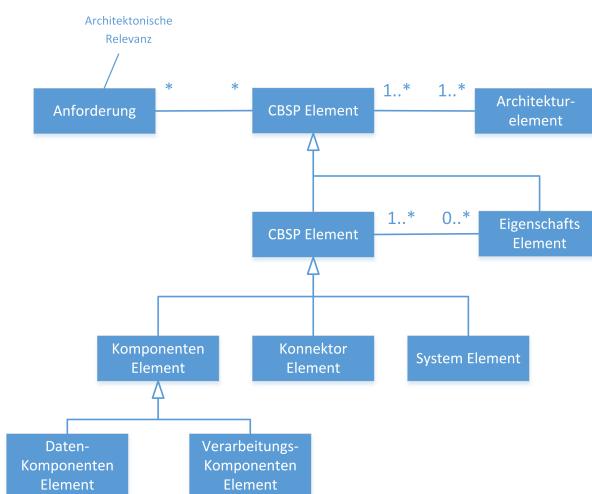


Abbildung 4. CBSP Metamodell [9]

a) *Randbedingungen*: Damit CBSP erfolgreich angewandt werden kann, muss die Bereitschaft zur Mitarbeit aller involvierten Stakeholder gegeben sein. In fast allen Schritten können diese bei Unklarheiten oder für Feedback herangezogen werden. In einigen Schritten ist deren Mitarbeit verpflichtend.

Des Weiteren müssen die Software-Architekten über ein gewisses Mindestmaß an Erfahrung und Wissen verfügen um die einzelnen Schritte kompetent durchführen zu können. Dafür gibt es allerdings keine Liste mit Mindestvoraussetzungen oder ein Minimum an Jahren an Berufserfahrung. Hier muss klar sein, ob der Architekt die nötigen Kompetenzen besitzt um wichtige Entscheidungen selbstständig treffen zu können.

b) *Eingabe*: Als Eingabe benötigt das CBSP eine erste Anforderungsspezifikation. Diese muss noch nicht vollständig sein, denn sie kann in jeder Iteration weiter ergänzt werden. Da CBSP nicht auf FRs oder NFRs zugeschnitten ist, werden alle Anforderungen genutzt, denn beide Anforderungstypen haben sowohl explizit als auch implizit Informationen, welche für die Software-Architektur relevant sind [9].

c) *Vorgehensmodell*: Jede Iteration von CBSP besteht aus fünf Schritten, welche im folgenden vorgestellt werden.

Schritt 1: Auswahl der Anforderungen für die nächste Iteration - Um die Komplexität zu reduzieren werden hier, wie bereits angesprochen, durch ein Team von Architekten nur Anforderungen für die Iteration ausgewählt, welche mittels zweier Kriterien durch den Requirements Engineer als wichtig oder durchführbar bewertet wurden: (1) die *Bedeutung* zeigt die Relevanz und den Wert für den Projekterfolg, (2) während die *Durchführbarkeit* technische, ökonomische und politische Bedingungen berücksichtigt [9]. Diese Anforderungen können formell, informell oder semi-formell notiert sein.

Schritt 2: Architektonische Klassifizierung der Anforderungen - In Schritt zwei klassifizieren die Software-Architekten die selektierten Anforderungen mit Hilfe der CBSP Taxonomie. Des Weiteren werden alle so klassifizierten Anforderungen in einer Tabelle anhand ihrer Relevanz zu den sechs CBSP Dimensionen bewertet. Die Bewertung erfolgt über eine Ordinalskala (nicht=0; partiell=1; weitgehend=2; voll=3) und beschreibt die Auswirkung, welche die Anforderung auf eine oder mehrere der jeweiligen CBSP Elemente hat [9]. Eine beispielhafte Bewertung der Anforderungen R01: *Unterschiedliche Ladungstypen unterstützen*, R02: *Verschiedene Fahrzeuge unterstützen* und R09: *Schätzung der Frachtankunft und Fahrzeugverfügbarkeit unterstützen* ist Tabelle II zu entnehmen [9].

Schritt 3: Identifizierung und Auflösung von falschen Zuordnungen bei der Klassifizierung - Da mehrere Software-Architekten diese Klassifizierung unabhängig voneinander vornehmen, müssen im dritten Schritt mögliche Abweichungen aufgedeckt, diskutiert und behoben werden.

Tabelle II
BEISPIEL RELEVANZPROFILE NACH [9]

Anforderungen	C	B	S	CP	BP	SP
R01	1,33	0,33	1,67	1,00	0,33	0,33
R02	2,00	0,00	1,00	1,33	0,00	0,00
R09	2,67	1,33	2,00	0,33	0,00	0,00

Dies ist wichtig um Missverständnisse, mehrdeutige Anforderungen, implizites Wissen, und widersprüchliche Auffassungen zu identifizieren und Risiken bei der Verfeinerung der Anforderungen zu reduzieren [9]. Bei der Diskussion können auch der Kunde und Requirements Engineer involviert sein. Der erarbeitete Konsens sorgt für ein einheitliches Verständnis der Anforderungen und die Relevanz dieser für die Software-Architektur. Wenn keine Unstimmigkeiten mehr vorliegen, werden die Anforderungen weiter ausgedünnt indem alle Anforderungen, welche eine geringere Relevanz haben als ein vorher definierter Wert der Ordinalskala, wie beispielsweise "weitgehend", vorerst verworfen werden. Dabei wird auf Abhängigkeiten zwischen den Anforderungen geachtet um kritische nicht zu verwerfen. Alle akzeptierten Anforderungen werden inklusive gesammelter Probleme und Unklarheiten in den nächsten Schritt übernommen.

Schritt 4: Architektonische Verfeinerung der Anforderungen
- Im vorletzten Schritt werden die CBSP Elemente umformuliert oder aufgeteilt, welche überlappende CBSP Dimensionen und architektonische Anliegen aufweisen. Wenn ein CBSP Element beispielsweise weitgehend Komponenten-relevant, voll Konnektor-relevant und weitgehend Konnektor-Eigenschafts-relevant ist, erhöht es das Verständnis und die Präzision, wenn es in mehrere Architekturentscheidungen, verkörpert als CBSP Element, aufgesplittet wird. Ein einzelnes CBSP Artefakt kann während dieses Prozesses mehrfach als Nebenprodukt bei verschiedenen Anforderungen auftauchen, während diese Redundanzen identifiziert und eliminiert werden [9].

In diesem Schritt gibt es zwei Varianten, welche entweder einzeln oder sogar kombiniert ausgeführt werden können. Variante eins betrachtet zunächst nur die großen strukturellen Aspekte des Systems (C, B und S Elemente). Dies hat den Vorteil, dass der Architekt leichter einen Überblick behalten kann, da die beiden Aspekte (Struktur und Funktionalität vs. nicht-funktionale Eigenschaften) separat betrachtet werden. Die Eigenschaften der drei Elemente werden in einem gesonderten (Sub-)Prozess identifiziert. Die zweite Variante identifiziert in einem Schritt sowohl die strukturellen Elemente als auch ihre nicht-funktionalen Eigenschaften. Der Vorteil hierbei ist, dass der Software-Architekt sich auf einen einzelnen System-Aspekt (oder eine kleine Menge von Aspekten) konzentrieren kann.

Schritt 5: Abgleich der Entscheidungen zwischen

Architekturelementen und -stilen mit CBSP - Wenn alle ausgewählten Anforderungen im CBSP Modell modelliert wurden, sodass keine Konflikte zwischen Software-Architekt und Requirements Engineer bestehen und alle CBSP Modellelemente weitgehend relevant zu einer der sechs CBSP Dimensionen sind, kann in Schritt fünf ein erster Architektur-Prototyp abgeleitet werden.

Die Bestimmung eines passenden Architekturstiles mit den zu den CBSP Modellelementen passenden Architekturmustern ist nicht immer einfach. Zum einen existiert kein formaler Ansatz für diesen Prozess, zum anderen können mehrere Architekturstile zum generierten CBSP Modell passen. Für diese Aufgabe gibt es verschiedene Verfahren. Hier wird die Selektion eines passenden Architekturstiles mit Hilfe einer Matrix aufgestellt, siehe Tabelle III. Diese Matrix stellt die CBSP Dimensionen mit geeigneten Eigenschaften potentiellen Architekturmustern gegenüber. Allerdings ist diese Tabelle nicht vollständig, denn weitere Eigenschaften für die CBSP Elemente sind denkbar und es werden auch nicht alle existierenden Architekturstile angeboten. Zum einen ist sie dafür nicht gedacht, sie soll lediglich eine Hilfestellung bei der Wahl eines passenden Architekturstils bieten, sobald mehr als einer passend erscheint. Dafür muss sie nur so umfassend wie eben nötig sein. Zum anderen würde eine vollständige Tabelle jedes Softwaresystem darstellen und nicht auf eine konkrete Domäne zurecht geschnitten sein. Außerdem wäre eine solche unmöglich zu erstellen [9]. Die endgültige Wahl liegt allerdings beim Software-Architekten.

Anschließend muss der Software-Architekt die CBSP Modellelemente in entsprechende Architekturelemente wie Komponenten, Konnektoren, Konfigurationen, Pakete und Daten mit den gewünschten Eigenschaften umwandeln [9].

d) Ausgabe: Nach der Ausführung der fünf Schritte und mehrfachen Iterationen sollte eine Software-Architektur gegeben sein, welche die selektierten Anforderungen zu einem möglichst hohen Grad erfüllt.

C. scenario and goal based System development method (COSMOD) (FDD)

COSMOD-Requirements Engineering (COSMOD-RE) beschreibt ein iteratives Vorgehen zum gleichzeitigen Design von Anforderungen und Software-Architektur. Kerngedanke bei COSMOD-RE ist eine Aufteilung in vier Hierarchiestufen, wo sowohl Architektur als auch Anforderungen definiert werden. In diesen vier Hierarchiestufen wird einerseits aus der Anforderungssicht und andererseits aus der Architektsicht betrachtet, welche Anforderungen und Komponenten dem System zuzuordnen sind.

1) Ziele der Methode: Kernziel von COSMOD-RE ist es, die Entwicklung von Anforderungs- und Architekturartefakten für softwareintensive eingebettete Systeme zu unterstützen. Ein ziel- und szenario-basierter Ansatz wie COSMOD-RE, der das Co-Design von Architekturartefakten und Anforderungen

Tabelle III
CBSP ZU ARCHITEKTURSTIL MAPPING NACH [9]

CBSP Dimensionen	Eigenschaften	Client-Server	C2	Event-basiert	Schichten-Architektur	Pipes und Filter
Datenkomponente	aggregiert	++	++	++	+	-
	persistent	++	o	o	o	o
	gestreamt	-	-	-	-	+
	gecached	++	+	-	-	-
Verarbeitungskomponente	Dienste anbieten / Nur konsumieren	++	o	o	o	o
	hat N Schnittstellen	++	+	++	-	-
	stateful	+	++	++	+	-
	lose gekoppelt	+	+	++	-	++
	kann migriert werden	+	++	++	-	-
Konnektor	synchron	++	-	+	++	-
	asynchron	-	++	++	-	++
	lokal	-	++	o	++	+
	verteilt	++	++	++	-	+
	sicher	+	o	o	+	o
(Sub-)System	effizient	o	+	+	o	-
	skalierbar	+	o	-	-	+
	weiterentwickelbar	++	++	++	-	++
	portierbar	o	+	o	++	o
	zuverlässig	o	o	-	o	o
	dynamisch	+	++	++	-	++

ermöglicht, muss jedoch einige Anforderungen erfüllen um einen Nutzen zu bringen. Die Anforderungen an eine solche Methodik sind: [4]

- *Entwicklung von Anforderungen und Architekturartefakten.*

Da Anforderungen und die Software-Architektur die Kerntreiber hinter innovativen Projekten sind ist es wichtig, beide gleichermaßen zu nutzen, sodass keines von beiden in den Vordergrund tritt [4].

- *Unterstützung der Anordnung von Anforderungen und Architekturartefakten.*

Werden Anforderungen und Architekturartefakte parallel entwickelt kann es passieren, dass Inkonsistenzen auftreten. Um diese zu vermeiden ist es notwendig, dass COSMOD-RE diese erkennt und behebt [4].

- *Definition detaillierter Anforderungen auf der Basis von Architekturartefakten.*

Ohne technisches Wissen fällt es Stakeholdern schwer, notwendige Details für die Erhebung architektureller Anforderungen zu liefern. Daher sollte es COSMOD-RE ermöglichen, die detaillierten Anforderungen erst nach der initialen Definition von Architekturartefakten zu liefern [4].

- *Nutzung einer Abstraktionshierarchie.*

Da bei einem Co-Design Vorgehen eine große Komplexität bestehen kann, ist es notwendig eine Abstraktionshierarchie einzuführen um diese entsprechend zu behandeln [4].

Ferner basieren die in der Methode genutzten Prozesse auf folgenden Ideen: [3]

- Initiale Unterteilung in die Architektursicht und die

Systemnutzungs-Sicht

- Ziel- und Szenario-basierte Integration der Sichtweisen
- Definition von Systemanforderungen basierend auf einer festgelegten Systemnutzungs- und Architektur-Sicht

2) *Funktionsweise der Methode:* Das wichtigste Element von COSMOD-RE ist die Abstraktionshierarchie. Über diese werden alle Aktivitäten, die im Rahmen der Methodik stattfinden, eingeordnet und miteinander in einen Kontext gesetzt. So wird auf jeder Ebene der Abstraktionshierarchie eine Architektursicht und eine Anforderungssicht erzeugt. Hierbei ist jedoch zu beachten, dass bei COSMOD-RE nicht unbedingt ein Top-Down-Ansatz zu wählen ist, da Anforderungsartefakte und Architekturartefakte auf allen Ebenen gleichzeitig bearbeitet werden können.

In der Abbildung 5 ist schematisch dargestellt, wie COSMOD-RE auszuführen ist. Es gibt vier Hierarchieebenen, auf denen drei Co-Design Prozesse arbeiten, die wiederum aus fünf Subprozessen bestehen.

a) *Randbedingungen:* Bei der Anwendung von COSMOD-RE ist zu beachten, dass einige Bedingungen erfüllt sein müssen um die Methodik richtig anzuwenden:

- *Definition einer Abstraktionshierarchie.*

Es gibt keine Richtlinien bezüglich der Definition der Abstraktionshierarchie. Dies bedeutet, dass die verwendeten Hierarchiestufen in jedem Projekt neu festgelegt werden müssen. Ferner ist hierbei von Relevanz, zu definieren, wo die Trennlinien zwischen verschiedenen Abstraktionsstufen sind [6].

- *Verknüpfung von Anforderungs- und Architekturmodellen.*

Da im Rahmen von COSMOD-RE Anforderungsartefakte und Architekturartefakte parallel entwickelt werden, ist

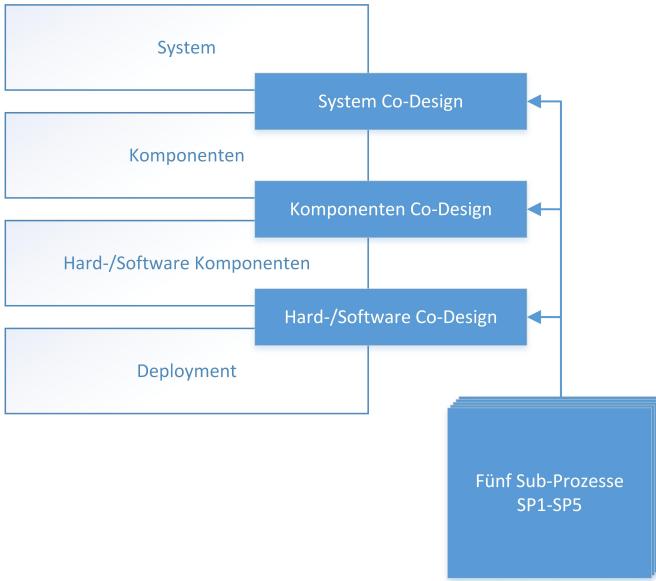


Abbildung 5. Schematische Darstellung der COSMOD-RE Methode

es notwendig diese miteinander zu verknüpfen, um sie in den Kontext des Ziels der Software zu bringen. Für diese Verknüpfung ist die Anwendung von Methodenfragmenten notwendig [6].

- **Definition von Konsistenzbedingungen.**

Für die Definition von Konsistenzbedingungen gibt es keinen allgemeingültigen Ansatz. Daher ist es notwendig, in jedem Projekt neu zu definieren, ob Konsistenz gegeben ist oder nicht [6].

- **Definition der System-Vision.**

Es muss eine System-Vision vorhanden sein um COSMOD-RE einsetzen zu können, da die Methode diese zu Anforderungen verfeinert [3].

Sind diese Randbedingungen geklärt, ist es möglich COSMOD-RE anzuwenden.

b) Eingabe: Als Eingabe benötigt COSMOD-RE eine System-Vision. Dies bedeutet, in initialen Gesprächen mit Stakeholdern muss bereits festgehalten sein, welche Vision von dem zu konzipierenden System gegeben ist. Die System-Vision ist vor allem in der Systemebene von Relevanz. Auf den niedrigeren Ebenen sind die Ausgaben der oberen Ebenen bedeutsam.

c) Vorgehensmodell: Um das Vorgehensmodell zu beschreiben muss zunächst geklärt werden, was unter den vier Hierarchieebenen zu verstehen ist. Danach werden die drei Co-Design Prozesse erläutert. Darauf folgt dann die Erläuterung der fünf Sub-Prozesse.

Die vier in Abbildung 5 dargestellten Hierarchieebenen sind:

1. *System:* Die Systemebene beschreibt die oberste Ebene, bei der in der Anforderungssicht das System als Ganzes betrachtet wird. Im Fokus stehen dabei die Interaktionen mit dem System. Weiter werden funktionale Anforderungen und Qualitätsanforderungen erstellt, die sich auf das Gesamtsystem beziehen. Die Architektsicht konzentriert sich hier auf die Definition von externen Systemschnittstellen. Hier definierte Artefakte sollen primär die Kommunikation mit Stakeholdern unterstützen.
2. *Komponenten:* Die Komponentenebene bezeichnet die Aufteilung des Systems in einzelne Komponenten aus denen sich dieses zusammensetzen soll. Für jede Komponente werden funktionale Anforderungen und Qualitätsanforderungen formuliert. Da in dieser Ebene die Basis für die Systemarchitektur gelegt wird, ist die Kommunikation zwischen dem Software-Architekten und dem Requirements Engineer von besonderer Bedeutung.
3. *Hard-/Software Komponenten:* Auf dieser Ebene werden die zuvor erstellten Komponenten in Hard- und Software Komponenten aufgeteilt und weiter verfeinert. Anforderungen auf dieser Ebene sind somit speziell auf die Komponentenart bezogen.
4. *Deployment:* Auf dieser Ebene werden Softwarekomponenten programmierbaren Hardwarekomponenten zugeordnet. Anforderungen auf dieser Ebene beziehen sich auf das Deployment der Softwarekomponenten und ihrem Einfluss auf vorher definierte Anforderungen.

Da die Zusammenarbeit zwischen Software-Architekt und Requirements Engineer vor allem in den obersten beiden Ebenen von Relevanz ist, sind die unteren beiden Ebenen in diesem Kontext zu vernachlässigen.

Ein Vorteil von COSMOD-RE ist unter anderem, dass Beziehungen die sich über mehrere Hierarchieebenen erstrecken, nicht vernachlässigt werden. Dies ist durch die CO-Design Prozesse möglich. Diese geben an, dass bei der Ausführung der Subprozesse mehrere Hierarchieebenen zu berücksichtigen sind. So wird beim System-Co-Design sowohl die Systemebene als auch die Komponentenebene betrachtet. Bei dem Komponenten Co-Design wird die Komponentenebene und die Hard-/Software Komponentenebene betrachtet. Im Hard-/Software Co-Design wird die Hard-/Software Komponentenebene und die Deploymentebene berücksichtigt [4]. Jeder dieser drei CO-Design Prozesse führt dann die fünf Subprozesse aus.

Das grundsätzliche Vorgehen von COSMOD-RE ist, dass eine Systemvision unter der Verwendung von Szenarien und Zielen zu einer Menge von Systemartefakten verfeinert wird [3]. Hierfür nutzt die Methode fünf Subprozesse.

In Abbildung 6 nach [3] sind diese zu sehen.

SP1: Entwicklung der Systemnutzungs-Sicht - Ziel dieses Subprozesses ist die Verfeinerung der System-Vision.

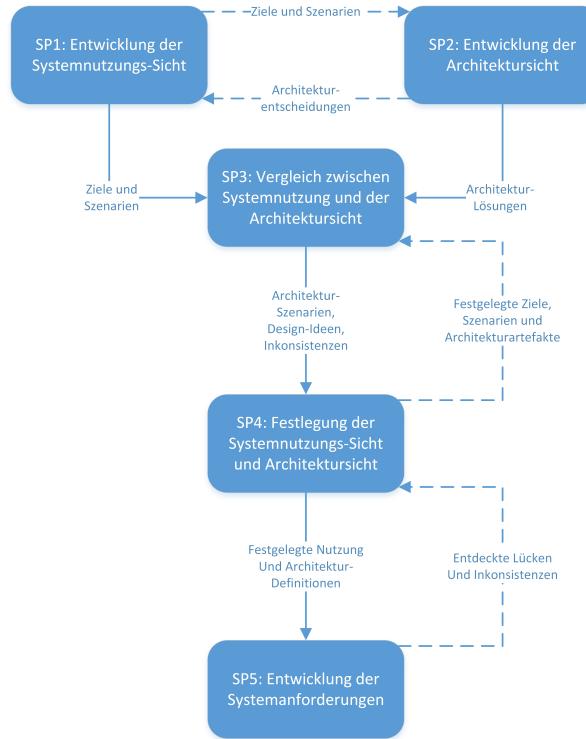


Abbildung 6. Die fünf Subprozesse der COSMOD-RE Methode

Zunächst sollen hier potenzielle Akteure identifiziert werden. Daraufhin wird die System-Vision in nutzerbezogene Unterziele aufgeteilt. Danach soll für jedes Ziel ein Nutzungs-Szenario gebildet werden, welches die Bedingungen für die Zielerreichung dokumentiert. In diesem Schritt ist als Eingabe die Eingabe der Methode, die System-Vision, gegeben [3]. Bei diesem Prozess sind vor allem das Know-How des Requirements Engineer von Bedeutung. Dieser muss in Kooperation mit dem Kunden die Ziele dieses Subprozesses erreichen.

SP2: Entwicklung der Architektsicht - Hauptziel dieses Subprozesses ist die Erzeugung grober Architekturlösungen für das geplante System. Eingabe in diesem Subprozess ist einerseits wieder die System-Vision, andererseits jedoch die in SP1 gebildeten Systemnutzungs-Ziele und -Szenarien. Ausgabe dieses Prozesses ist ein grober Entwurf der Systemarchitektur [3].

In diesem Subprozess werden vier Kernaktivitäten ausgeführt:

- **Analyse der Systemziele und Nutzungs-Szenarien:** Ziel dieser Aktivität ist die Identifikation architekturrelevanten Aussagen in den Systemnutzungs-Zielen und Systemnutzungs-Szenarien. Diese können zum Beispiel hinweise auf spezielle Komponenten sein.
- **Kreative Entwicklung neuer Architekturlösungen:** Ziel dieser Aktivität ist es, innovative Lösungen für das Sys-

tem zu erarbeiten. Auch wenn die in SP1 erzeugten Ausgaben von Bedeutung sind, ist hier vor allem das Know-How der Software-Architekten von Relevanz. Das technische Hintergrundwissen und der Einfallsreichtum sind von zentraler Bedeutung.

- **Bewertung der entwickelten Architekturlösungen:** Ziel dieser Aktivität ist die Auswertung der entwickelten Lösungsansätze um so die besten auszuwählen.
- **Definition einer vorläufigen groben Architektur:** Ziel dieser Aktivität ist die Erzeugung einer (partiellen) Lösung basierend auf der zuvor ausgeführten Bewertung [3].

SP3: Vergleich zwischen Systemnutzungs-Sicht und Architektsicht - Hauptziel dieses Subprozesses ist einerseits die Überprüfung, ob die Architektur die identifizierten Systemnutzungs-Ziele und Szenarien unterstützt und andererseits die Identifikation neuer Verwendungszwecke basierend auf der aktuellen groben System-Architektur [3].

Die Überprüfung lässt sich auch betrachten als Vergleich der Ergebnisse von SP1 und SP2. Um die Systemnutzungs-Szenarien mit der Architektur zu vergleichen, ist es notwendig, die Systemnutzungs-Szenarien zu Architektur-Szenarien zu verfeinern. Diese Verfeinerung, die auch mithilfe von Message-Sequence-Charts durchgeführt werden kann, wird üblicherweise über drei Schritte durchgeführt [3].

- Das System wird in eine Untermenge funktionaler Anforderungen verfeinert, welche in der groben Architektur definiert werden.
- Jeder Systemnutzung wird eine funktionale Komponente zugeordnet, die verantwortlich für die Realisierung der Interaktion mit externen Akteuren ist.
- Systeminterne Interaktionen, die externe Interaktionen ermöglichen, werden definiert [3].

SP4: Festlegung der Systemnutzungs-Sicht und Architektsicht - Wie auch in SP3 gibt es hier zwei zentrale Ziele. Einerseits sollen die in SP1 erzeugten Systemnutzungs-Ziele und -Szenarien mithilfe der Erkenntnisse aus SP3 verbessert und angepasst werden und andererseits soll die in SP2 gebildete grobe Architektur mithilfe der Erkenntnisse aus SP3 verbessert und angepasst werden [3].

Hierfür muss zunächst bei den Ausgaben von SP3 überprüft werden, ob sie zur Verbesserung der Ergebnisse von SP1 und SP2 geeignet sind. Hierfür ist es angemessen ein System zur Kategorisierung einzuführen, über das entschieden werden kann, ob eine Verbesserung notwendig oder nicht notwendig ist. Danach werden die Ergebnisse von SP3 priorisiert. Hier wird überprüft, welche Verbesserungen zuerst umgesetzt werden sollten und welche nicht direkt notwendig sind. Zuletzt werden die Verbesserungen umgesetzt. Hier ist jedoch zu beachten, dass Verbesserungen neue Ideen

hervorrufen oder Inkonsistenzen erzeugen könnten. Daher kann es notwendig sein, SP3 zu wiederholen. Diese Iteration ist solange durchzuführen, bis die Ergebnisse angemessen angeordnet und stabil sind [3].

SP5: Entwicklung der Systemanforderungen - Hauptziel dieses Subprozesses ist die Spezifikation detaillierter Systemanforderungen. Grundlage für die Spezifikation sind die Systemnutzungs-Sicht und die Architektsicht. Mithilfe von Zielen, Szenarien und dem groben Architekturentwurf ist es möglich, detaillierte Anforderungen zu formulieren und in der Spezifikation aufzuführen.

d) Ausgabe: Ausgabe der Methode ist eine kompakte Menge von System-Nutzungs-Zielen, System-Interaktions-Szenarien und grobe Architekturartefakte. Ferner wird eine detaillierte System-Anforderungsspezifikation generiert.

Vor allem im Rahmen der fünf Subprozesse werden folgende Artefakte generiert:

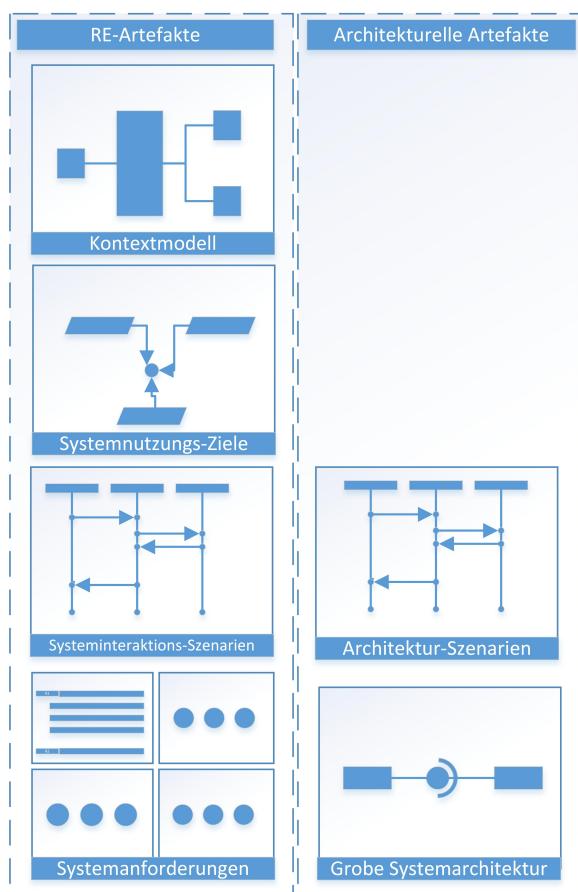


Abbildung 7. Die Artefakte der COSMOD-RE Methode

- Kontextmodell:** Das Kontextmodell dokumentiert die beabsichtigte Einbettung des Systems in seine Systemumgebung. Es werden externe Akteure definiert, die mit dem

System interagieren. Ein Akteur kann hierbei ein Mensch oder ein System sein. Ferner wird modelliert, wie Akteure mit dem System interagieren [3].

- Systemnutzungs-Ziele:** Systemnutzungs-Ziele verfeinern die System-Vision. Ein Systemnutzungs-Ziel dokumentiert eine Eigenschaft des Systems in Bezug zu externen Akteuren, die das System nutzen. Ein Ziel kann hierbei in Unterziele unterteilt werden, wodurch eine Hierarchie von Zielen möglich ist [3].
- Systeminteraktions-Szenarien:** Mit diesen Szenarien werden Interaktionen von Akteuren mit dem System dokumentiert. Für die Dokumentation solcher Szenarien bieten sich modellbasierte Ansätze an [3].
- Architektur-Szenarien:** Mithilfe dieser Szenarien werden Interaktionen innerhalb des Systems dokumentiert. Hiermit sind vor allem Interaktionen zwischen Systemkomponenten gemeint [3].
- Große System-Architektur:** Hier wird eine Zerlegung der System-Architektur in eine Menge funktionaler Komponenten vollzogen, die über Schnittstellen verknüpft sind [3].
- Systemanforderungen:** Als Systemanforderungen werden hier funktionale, strukturelle oder verhaltensbezogene Anforderungen gemeint. Weiter werden Qualitätsanforderungen, die sich beispielsweise auf Performance oder Sicherheit beziehen, spezifiziert [3].

Durch die iterativen Verbesserungen von Anforderungen und Systemarchitektur ist es abhängig von der Anzahl der Iterationen möglich ein Abbild der System-Vision zu erzeugen, sofern diese zu Beginn korrekt definiert wurde.

D. ADD 3.0 (FDD)

Attribute-driven-Design (ADD) bezeichnet ein Vorgehensmodell, bei dem iterativ ein Architekturdesign ausgearbeitet wird. ADD wird in Form von sogenannten Design Rounds durchgeführt. Eine Design-Round kann hierbei beispielsweise einem Sprint in SCRUM zugeordnet werden. Dies bedeutet, in einem Projekt kann es mehrere Design-Rounds geben, mit denen die Software-Architektur verfeinert wird.

1) Ziele der Methode: ADD 3.0 soll einen konkreten und genauen Ansatz zum Entwurf einer Software-Architektur bieten. Der Fokus soll bei ADD 3.0 auf dem Design der Software-Architektur liegen, die nach [5] Grundvoraussetzung für agile Projekte ist.

Eine Eigenschaft, die bei ADD besonders hervorsticht ist, dass es innerhalb der Design-Rounds eine klare Folge von Anweisungen gibt, die auszuführen sind um die Software-Architektur zu entwickeln. Hierbei ist relevant zu erwähnen, dass in ADD die Dokumentation und Analyse als wichtigste Elemente zur Entwicklung der Software-Architektur betrachtet

werden.

2) Funktionsweise der Methode: Insgesamt umfasst ADD sieben Schritte, die innerhalb einer Design-Round auszuführen sind. In Abbildung 8 ist schematisch dargestellt, wie eine Design-Round aufgebaut ist und welche Schritte auszuführen sind.

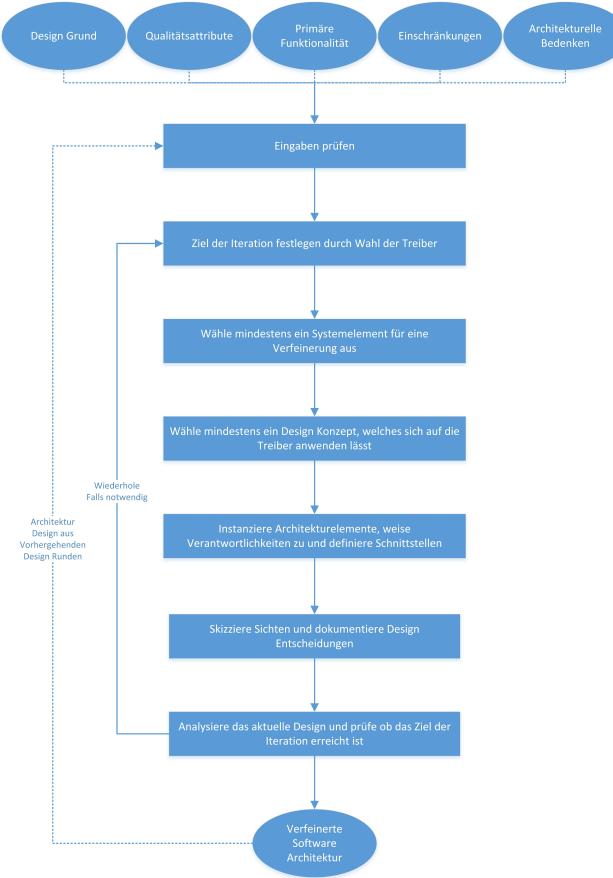


Abbildung 8. Attribute Drive Design 3.0

a) Randbedingungen: Die Voraussetzung für die Nutzung von ADD 3.0 ist, dass bereits primäre funktionale Anforderungen und Szenarien erhoben sind. Dies bedeutet, ADD 3.0 findet nicht direkt in der Anforderungsgewinnung Anwendung, sondern erst danach.

b) Eingabe: ADD 3.0 wird von sogenannten Projektträgern gesteuert. Projektträger sind die Eingaben, die vor einer Design-Round vorbereitet werden müssen. Die vorzubereitenden Eingaben sind:

- **Design Grund**

Es kann vielfältige Gründe für den Entwurf einer Software-Architektur geben. So kann es beispielsweise im Rahmen eines Projektes sein oder als Vorbereitung auf kommende Projekte. Vor dem Beginn muss jedoch

geklärt werden, aus welchem Grund das Design benötigt wird [5].

- **Qualitätsattribute**

Unter Qualitätsattributen sind in dem Kontext von ADD 3.0 Eigenschaften eines Systems zu sehen, die entweder testbar oder messbar sind. Aus diesem Grund haben Qualitätsattribute den größten Einfluss auf die zu konzipierende Software-Architektur. Da die Qualitätsattribute besonders wichtig sind, hat ihre Erhebung eine besondere Bedeutung [5].

- **Primäre Funktionalität**

Unter der primären Funktionalität ist zu verstehen, dass ein System die Aufgaben erfüllt, die es erfüllen soll. Während Qualitätsattribute vor allem das "wie" prüfen, prüft die primäre Funktionalität das "was". Relevant ist hierbei, dass es in den Anforderungen vorkommen kann, dass die primäre Funktionalität direkt mit Qualitätsattributen verknüpft ist [5].

- **Einschränkungen**

Bei dem Entwurf einer Software-Architektur gibt es immer Einschränkungen, die berücksichtigt werden müssen. Obwohl es möglich sein kann diese zu lockern, sind sie dennoch konstante Faktoren, die bei dem Design zu berücksichtigen sind [5].

- **Architektonische Bedenken**

Es gibt einige architektonische Bedenken, die eine Rolle spielen können:

- Unter **grundlegenden Bedenken** sind Bedenken zu verstehen, die mit dem gesamten Design-Prozess in Bezug stehen. Diese können beispielsweise die Zuteilung von Arbeitspaketen an Projektteams, Organisation des Programmcodes, Deployment oder Updates beinhalten [5].
- **Spezielle Bedenken** formulieren Bedenken, die sich auf detailliertere systeminterne Aspekte beziehen. Diese können beispielsweise Aspekte wie Exception Management, die Verwaltung von Abhängigkeiten, Logging oder Authentifikation beinhalten [5].
- Unter **internen Anforderungen** sind Anforderungen zu verstehen, die häufig nicht in den Anforderungs-dokumenten genannt werden. Diese Anforderungen betreffen beispielsweise Entwicklung, Einsatz oder Wartung des Systems [5].
- **Probleme** sind in der Regel nicht bei der initialen Design-Round von Bedeutung. Nach der Ausführung einer Design-Round kann es jedoch sein, dass mit dem aktuellen Software Design Risiken verbunden sind, die behoben werden müssen [5].

c) Vorgehensmodell: Grundsätzlich entspricht ein Durchlauf von ADD 3.0 in einem iterativen Entwicklungsverfahren einer Iteration. Wird das Wasserfall-Modell genutzt entspricht ein Durchlauf einer Menge von Architektur-Entwurf-Aktivitäten [5].

Vor der erstmaligen Ausführung von ADD 3.0 sollte überprüft werden, ob alle notwendigen Eingaben vorhanden sind.

Erhebung von Qualitätsattributen - Da von den Eingaben die Qualitätsattribute besonders wichtig sind, ist es wichtig, dass sie angemessen erhoben wurden [5]. Im Rahmen von ADD 3.0 sind Qualitätsattribute als Szenarien zu formulieren. Szenarien bieten hier den Vorteil, dass mithilfe von diesen präziser formuliert werden kann, was genau ein Attribut aussagen soll und wie genau es messbar oder testbar sein kann. Die Erhebung solcher Szenarien kann beispielsweise mithilfe eines Qualitäts-Attribut-Workshops (QAW) realisiert werden, der den folgenden Aufbau hat [5]:

1. QAW Präsentation und Vorstellung
2. Geschäftsziel Vorstellung
3. Architekturplan Vorstellung
4. Identifikation der architektonischen Treiber
5. Szenario Brainstorming
6. Szenario Konsolidierung
7. Szenario Priorisierung
8. Szenario Verfeinerung

Nachdem alle Eingaben gegeben sind, sind die sieben Schritte von ADD 3.0 auszuführen.

Schritt 1: Eingaben prüfen - Zunächst muss sichergestellt werden, dass die übergeordnete Zielstellung für die darauffolgenden Design-Aktivitäten festgelegt ist. Dies bedeutet, zunächst muss der Design Grund validiert werden. Dieser kann beispielsweise die erstmalige Erstellung eines Design-Entwurfes oder die Verbesserung eines vorhandenen Architektur-Designs sein. Danach wird überprüft, ob die für die Design-Round relevanten Anforderungen und Szenarien korrekt sind. Hier ist unter anderem zu prüfen, ob alle relevanten Stakeholder berücksichtigt werden und ob die erhobenen Anforderungen richtig priorisiert sind. Zuletzt muss noch geprüft werden, ob es Einschränkungen bezüglich der Software-Architektur gibt, die in der Design-Round zu berücksichtigen sind.

Schritt 2: Ziel der Iteration festlegen durch Wahl der Treiber - ADD 3.0 wird in der Regel in mehreren Iterationen ausgeführt, wobei jede Design-Round ein Ziel verfolgt. Durch die Auswahl von Design Gründen, Qualitätsattributen und primären Funktionalitäten wird festgelegt, welche Ziele in einer Design-Round verfolgt werden [5].

Schritt 3: Wähle mindestens ein Systemelement für eine Verfeinerung aus - Um die Treiber zu bedienen ist es notwendig, mindestens eine architektonische Struktur zu konzipieren. Ansonsten ist es beispielsweise unmöglich, die Qualitätsattribute vollständig zu erfüllen. In der ersten Iteration könnte eine solche Struktur beispielsweise das System als Ganzes sein, welches in mehreren Iterationen

in mehrere Teilsysteme aufgeteilt wird, die miteinander in Beziehung stehen oder voneinander abhängig sein können [5].

Schritt 4: Wähle mindestens ein Design Konzept, welches sich auf die Treiber anwenden lässt - Abhängig von den ausgewählten Treibern gibt es eine Vielzahl von Design Konzepten, die sich anwenden lassen. In diesem Schritt sind die Alternativen abzuwägen und die Kosten der Implementierung einzuschätzen um zu entscheiden, welche Design Konzepte anzuwenden sind. So gibt es beispielsweise viele gut dokumentierte Design Konzepte, die eine hohe Verfügbarkeit erzielen und gleichzeitig den Single-Point-of-Failure vermeiden [5].

Schritt 5: Instanziere Architekturelemente, weise Verantwortlichkeiten zu und definiere Schnittstellen - Nach der Auswahl der Design Konzepte kann es abhängig von der Auswahl notwendig sein, diese zu konfigurieren. Wird beispielsweise ein Schichten Pattern verwendet, muss entschieden werden, wie viele Schichten es geben soll. Neben der Instanzierung der Architekturelemente ist es notwendig, diese zuzuordnen. Weiter ist es notwendig, die instanzierten Architekturelemente miteinander zu verbinden um Kollaborationen zu ermöglichen [5].

Schritt 6: Skizziere Sichten und dokumentiere Design Entscheidungen - Nachdem nun die Architekturentscheidungen der Design Round getroffen sind, ist es notwendig, diese formalisiert festzuhalten und getroffene Design-Entscheidungen zu dokumentieren. Bei der Dokumentation müssen die Architekturentscheidungen weiter verfeinert und überprüft werden [5].

Schritt 7: Analyisiere das aktuelle Design und prüfe, ob das Ziel der Iteration erreicht ist - In diesem Schritt gilt es das aktuelle Design zu analysieren und zu überprüfen, ob es Fehler oder Inkonsistenzen gibt. Ferner ist zu prüfen, ob das in den Treibern vorgegebene Ziel erreicht ist. Ist dies nicht der Fall, ist es notwendig zu iterieren und die Schritte ab Schritt zwei zu wiederholen. Im Gesamtprojekt kann das aktuelle Design als weitere Eingabe für eine weitere Design-Round angesehen werden.

d) Ausgabe: Nach der Ausführung der sieben Schritte sollte ein verfeinertes Architekturentwurf erzeugt sein. Dieser kann in einer frühen Iterationen noch über Inkonsistenzen verfügen, daher gilt: Je mehr Iterationen durchgeführt werden desto solider ist der Architekturentwurf.

IV. AUSWERTUNG DER METHODEN

In den vorhergehenden Kapiteln sind verschiedene Methoden vorgestellt worden, die das Potenzial haben, das Zusammenwirken von Requirements Engineer und Software-Architekt zu verbessern und dadurch eine verbesserte Software-Architektur zu generieren. Nun gilt es jedoch zu

Tabelle IV
ÜBERSICHT ÜBER DIE KERNATTRIBUTE DER METHODEN

	Probing	CBSP	COSMOD-RE	ADD 3.0
Eingabe	PQ-Katalog / PQ-Flow	Erste Anforderungsspezifikation	System-Vision	Design Grund / Qualitätsattribute / Primäre Funktionalität / Einschränkungen / Architekturelle Bedenken
Ausgabe	Vollständigere Spezifikation (mit benötigten ASFRs)	Architektur-Prototyp	System-Nutzung-Ziele / System-Interaktions-Szenarien / grobe Architekturartefakte	Verfeinerte Software-Architektur
Randbedingungen	Der REler muss gewillt sein, die PQs zu benutzen	Alle Stakeholder müssen bei Unklarheiten zur Verfügung stehen / Software-Architekten brauchen ein Mindestmaß an Erfahrung und Wissen	Definition einer Abstraktionshierarchie / Verknüpfung von Anforderungen und Architekturmödellen / Definition von Konsistenzbedingungen / Definition einer System-Vision	Eingaben müssen gegeben sein / Anforderungserhebung muss abgeschlossen sein / Qualitätsattribute müssen erhoben sein
Ziele	Den REler mit PQs ausstatten / Erhebung der ASFRs erleichtern / verbessern	Leichtgewichtigen Ansatz zur Generierung von Architekturartefakten, welche die Anforderungen erfüllen, bereitstellen / Komplexität Behandeln / Konsistenz gewährleisten / Abhängigkeiten zwischen Anforderungen auflösen / wichtige Stakeholder involvieren / Mechanismen um Konflikte zu lösen	Entwicklung von Anforderungen und Architekturartefakten / Unterstützung der Anordnung von Anforderungen und Architekturartefakten / Definition detaillierter Anforderungen auf der basis von Architekturartefakten	Konkreter Ansatz zum Entwurf einer Software-Architektur / Design einer Software-Architektur

prüfen in wie weit dieses Potenzial zutrifft. Eignet sich eine Methode besser als die anderen? Wie sieht es aus, wenn die Methoden miteinander verglichen werden? Ergänzen sich die Methoden womöglich? Diese Fragen gilt es in der Auswertung zu beantworten.

Hierfür werden zunächst in tabellarischer Form die Kernattribute der besprochenen Methoden dargestellt. Danach werden die Methoden in Bezug zu den Problemen P1-P7 gebracht und es wird mittels eines Ratings überprüft, wie geeignet die Methode ist, um das Problem zu bewältigen. Danach werden die Methoden gegenübergestellt, wodurch überprüft werden soll, wo sich die Methoden potenziell ergänzen, bzw. an welchen Stellen die eine Methode besser geeignet ist als die andere.

A. Vergleich

In der Tabelle IV sind die Kernattribute der Methoden dargestellt.

1) *Eignung der Methoden zur Lösung der Probleme:* In der Tabelle V sind die Methoden und ihr Rating in Bezug zu den Problemen dargestellt.

Eine 1 steht hierbei dafür, dass eine Methode nicht geeignet ist, das Problem zu lösen. Die 2 steht dafür, dass eine Methode teilweise dafür geeignet sein kann, das Problem zu lösen und eine 3 steht dafür, dass eine Methode gut dafür geeignet ist, das Problem zu lösen.

Tabelle V
RATING DER METHODEN

	Probing	CBSP	COSMOD-RE	ADD 3.0
P1	2	3	3	1
P2	2	3	3	1
P3	2	3	2	1
P4	1	3	1	3
P5	3	2	3	2
P6	3	3	1	2
P7	1	3	3	1
Rating	14	20	16	11

a) *Probing (JEM):* Da das Probing, wie in Figur 1 zu sehen ist, während bzw. vor der Anforderungserhebung angewandt wird, behandelt es die durch den Menschen bedingten Probleme P1, P2, P3 nur teilweise. Es setzt, außer den Austausch der Probing Questions (PQs), keine Kommunikation zwischen Software-Architekt und Requirements Engineer voraus oder unterstützt diese. Allerdings erzwingt es mindestens diese eine, den Austausch der PQs. Dadurch verringert das Probing die Distanz zwischen Software-Architekt und Requirements Engineer und sorgt dafür, dass überhaupt eine Kommunikation stattfindet. P2 behandelt das Probing nur teilweise, da hier zumindest ein gemeinsames Arbeitsmittel, die PQs, von Software-Architekt und Requirements Engineer benutzt wird. Das fehlende Know-How wird hier nur teilweise beim Requirements Engineer durch die PQs aufgebessert, aber nicht beim Software-Architekt.

Die Probleme, die mit der Qualität der Anforderungen einhergehen, können zur Hälfte sehr gut und zur Hälfte gar

nicht durch das Probing gelöst werden. Probleme P5 und P6 werden ganz explizit durch das Probing behandelt, da es ausschließlich nach architekturelevanten Anforderungen fragt. Allerdings beschäftigt es sich nach der Anforderungserhebung nicht mehr mit den Anforderungen um Probleme mit zu restriktiven oder detaillierten architekturelevanten Anforderungen behandeln zu können. Wechselwirkungen zwischen den aufgestellten Anforderungen könnten teilweise aufgedeckt oder berücksichtigt werden, wenn die Typisierung der PQs betrachtet wird. Jedoch würde dieses Vorgehen nicht vollständig alle Wechselwirkungen aufdecken und ist zudem auch nicht Teil des Modells, sondern wäre nur eine kleine Hilfestellung.

b) *CBSP (JEM)*: CBSP liefert mit Abstand das beste Ergebnis. Da es in den Schritten eins bis vier Kommunikation zwischen Software-Architekt und Requirements Engineer voraussetzt um Konflikte zu lösen, behandelt es P1 sehr gut. Die konkurrierenden Interessen aus P2 werden ebenfalls durch die erzwungene Konfliktlösung und durch ein festes Vorgehensmodell angegangen. Fehlendes Know-How aus einer der beiden Sparten, Software-Architekt oder Requirements Engineer, wird hierdurch zwar nicht ersetzt, aber die daraus resultierenden Probleme wie sie in II-A3 aufgeführt sind werden angegangen. So ist sowohl inhaltlich fehlendes als auch fehlendes Know-How über die Methodik durch das Vorgehensmodell von CBSP irrelevant. Inhaltlich fehlendes Know-How und Missverständnisse bei Fachbegriffen können ebenfalls durch die verbesserte Kommunikation angegangen werden.

Die Probleme, die mit der Qualität der Anforderungen in Verbindung gebracht werden, werden durch das schrittweise Vorgehen von CBSP bis auf eine Ausnahme vollständig behandelt. Dadurch, dass immer nur eine kleine Menge an Anforderungen pro Iteration identifiziert, verfeinert, auf Abhängigkeiten und Redundanzen geprüft werden und alle Anforderungen explizit oder implizit Auswirkungen auf die Architektur haben, wird den Problemen P4, P6 und P7 ausreichend vorgebeugt. Allein P5 wird nur teilweise behandelt, denn durch das iterative Vorgehen und die Beschränkung auf eine Untermenge von Anforderungen pro Iteration können zwar ungenaue aber nicht fehlende Anforderungen berücksichtigt werden. Außerdem muss die Spezifikation zu Beginn nicht vollständig sein, was diese Möglichkeit auch noch unterstützt.

c) *COSMOD-RE (FDD)*: COSMOD-RE eignet sich gut um vier der Probleme zu lösen und teilweise um eines der Probleme zu lösen. Zwei der Probleme werden durch COSMOD-RE nicht gelöst. Bei COSMOD-RE werden sowohl die Architektur als auch die Anforderungen parallel entwickelt. Es findet gegenseitiger Austausch statt, was Kommunikation voraussetzt. Auch bei größeren Distanzen muss trotzdem ein Austausch stattfinden, da ansonsten die parallele Entwicklung von Anforderungen und Software-Architektur

nicht möglich ist. Daher bedingt die korrekte Ausführung von COSMOD-RE die Lösung des Problems P1. P2, also konkurrierende Interessen, wird durch den gegenseitigen Austausch ebenfalls gelöst. Bei paralleler Entwicklung und gegenseitigem Austausch müssen Kompromisse gebildet werden, welche die konkurrierenden Interessen auflösen. Fehlendes Know-How wird durch COSMOD-RE nicht konkret behandelt. Jedoch ist die Auswirkung von fehlendem Know-How bei paralleler Entwicklung und gegenseitigen Austausch eher gering, weswegen das Problem P3 teilweise gelöst wird. COSMOD-RE bietet keine Lösungsansätze gegen einen zu detaillierten und restriktiven Anforderungskatalog mit zu vielen Anforderungen, weswegen dieses Problem P4 offen bleibt. Durch die vier Ebenen, in denen bei COSMOD-RE gearbeitet wird und die Untersuchung von ebenenübergreifenden Beziehungen, werden ungenaue Anforderungen verfeinert und fehlende nachgearbeitet, weswegen P5 gelöst wird. COSMOD-RE sieht es nicht vor architekturelle Anforderungen konkret hervorzuheben weswegen P6 offen bleibt. Durch die Untersuchung auf verschiedenen Ebenen wird P7 gelöst.

d) *ADD 3.0 (FDD)*: ADD 3.0 eignet sich gut um eines der Probleme zu lösen und eignet sich nicht dafür vier der Probleme zu lösen. Zwei Probleme werden teilweise gelöst. Da ADD 3.0 eine Methode ist, die die Konzeption der Software-Architektur fördert, werden keine der durch den Menschen bedingten Probleme P1, P2 oder P3 behandelt. Dadurch, dass bei jeder Design-Round ein Ziel festgelegt wird, was mit dieser erreicht werden soll werden in jeder Design Round Anforderungen ausgewählt. Dies bedeutet, von der großen Menge an restriktiven Anforderungen werden nur einige zielführende ausgewählt um die aktuelle Design Round durchzuführen. Dadurch wird P4 verhindert. P5 bildet einen Widerspruch zu den Voraussetzungen um ADD 3.0 anzuwenden. Um ADD 3.0 auszuführen müssen die Projekttreiber vorhanden sein, was bedeutet, dass die Anforderungen mithilfe von Szenarien spezifiziert sein sollten. Gleicher gilt für P6. Sind die Projekttreiber nicht wie vorgeschrieben vorbereitet, lässt sich ADD 3.0 nicht anwenden. Es gilt: *Garbage in Garbage out* [5]. Die in P7 beschriebenen Wechselwirkungen werden nicht weiter beachtet.

2) Gegenüberstellung der Methoden:

a) Vergleich zwischen *CBSP* und *Probing (JEM)*:

Wenn man Eingaben und Ausgaben der beiden Methoden miteinander vergleicht, haben diese keine Schnittmenge. Jedoch fällt auf, dass die Ausgabe des Probing der Eingabe von CBSP entspricht. Es ist also möglich, diese beiden Methoden zusammen, also hintereinander auszuführen. Dadurch könnte sogar das Problem P5, welches durch CBSP nur teilweise behandelt wurde vollständig behandelt werden. Auch bei dem Vergleich der Ziele sind keine Redundanzen festzustellen. Beide Methoden setzen an vollkommen verschiedenen Punkten im Entwicklungsprozess,

wie in Abbildung 1 skizziert wurde, an. Somit sind Probing und CBSP theoretisch kompatibel zueinander. Dies wurde in der Praxis noch nicht überprüft. Hierbei ist allerdings zu beachten, dass Probing sich ausschließlich mit ASFRs beschäftigt, wohingegen CBSP auch NFRs mit berücksichtigt.

Eine Gemeinsamkeit beider Methoden ist, dass sie Vorwissen und Erfahrung beim Software-Architekten voraussetzen, damit die Methoden überhaupt angewandt werden können.

b) Vergleich zwischen CBSP und COSMOD-RE (JEM):

Im Vergleich von CBSP und COSMOD-RE fällt auf, dass beide ein ähnliches iteratives Vorgehen mit mehreren Schritten oder (Sub-)Prozessen haben. Sonst unterscheiden sich beide Methoden allerdings in vielen Punkten. Der größte Punkt besteht darin, dass COSMOD-RE sowohl Anforderungen als auch die Software-Architektur gleichzeitig in einem Design Prozess generiert, während bei CBSP das CBSP Zwischenmodell sowie die Software-Architektur erstellt werden. Die Anforderungen müssen bei CBSP, wenn auch unvollständig, bereits vorliegen und werden nicht verändert. Ein weiterer großer Unterschied ist die Sichtweise auf die Struktur des gesamten Prozesses. COSMOD-RE führt vier Abstraktionsstufen und drei Co-Design Prozesse sowie verschiedene Sichten auf alle Artefakte ein. Wohingegen CBSP sich nach dem Twin-Peaks Modell orientiert und nur eine Zwischenebene bzw. das Zwischenmodell eingeführt hat, welches für die Verknüpfung von Anforderungs- und Software-Architektur-Artefakten zuständig ist. Für diese Verknüpfung sorgen allerdings beide Ansätze gleichermaßen. Beim Vergleich der Attribute Eingabe, Ausgabe, Randbedingungen und Ziele, wie in Tabelle IV zu sehen ist, unterscheiden sich die beiden Methoden ebenfalls. Überschneidungen sind lediglich, dass Architekturartefakte erzeugt werden. Somit kann gesagt werden, dass diese beiden Methoden nicht kompatibel zueinander sind. Hier muss genauer überlegt werden, welche Methode besser für die Projektdurchführung geeignet ist.

c) Vergleich zwischen CBSP und ADD 3.0 (JEM):

CBSP und ADD 3.0 sind sich sehr ähnlich, da Eingaben und Ausgaben beider Methoden nahezu identisch sind. Beide erwarten Anforderungen und produzieren iterativ mit fest definierten Schritten eine Software-Architektur. ADD 3.0 benötigt für das Vorgehen allerdings noch weitere Informationen, wie einen Design Grund, Einschränkungen und architektonische Bedenken. Auch erwartet es die Qualitätsattribute im Gegensatz zu CBSP nicht als Anforderungen, sondern diese müssen als Szenarien abgebildet werden.

Der größte Unterschied zwischen den beiden Methoden im Hinblick auf die herausgearbeiteten Probleme ist, dass ADD 3.0 keine durch den Menschen bedingten Probleme behandelt. Es verlangt keine weitere Kommunikation

zwischen den beiden Rollen Requirements Engineer und Software-Architekt, wohingegen dies bei CBSP ein fester Bestandteil ist, um bei Problemen oder Missverständnissen den aktuellen Schritt abschließen zu können.

Außerdem funktioniert CBSP auf Grund der Selektion von Anforderungen pro Iteration auch mit größeren Projekten, wohingegen ADD 3.0 besser mit kleineren zurecht kommt. Dafür ist CBSP, vermutlich auch durch die bessere Kommunikation, wesentlich zeitaufwändiger.

Somit sind diese beiden Methoden nicht kompatibel zueinander. Es muss sich auch hier entschieden werden, welche der beiden Methoden in einem bestimmten Projekt den höheren Nutzen erbringt.

d) Vergleich zwischen ADD 3.0 und Probing (FDD):

Vergleicht man ADD 3.0 und Probing fällt vor allem auf, dass die Methoden an sehr unterschiedlichen Stellen eingesetzt werden. Probing wird eingesetzt um ASFR zu gewinnen und ADD 3.0 wird eingesetzt um auf der Basis von Anforderungen eine Software-Architektur zu erzeugen. Vergleicht man die Ziele fällt dies besonders auf. Bei der Betrachtung der Randbedingungen fällt auf, dass hier ebenfalls keine vergleichbare Schnittmenge gegeben ist. Jedoch fällt auf, dass eine der Randbedingungen von ADD 3.0 durch Probing gut zu erfüllen wäre. Das Festhalten von architekturelevanten Anforderungen ist eine gute Grundlage für die Qualitätsattribute, die ADD 3.0 benötigt. Probing ließe sich als eine Alternative zu einem Qualitäts-Attribut-Workshop ansehen, die es ermöglicht mit womöglich geringerem Personalaufwand alle ASFR zu gewinnen.

e) Vergleich zwischen COSMOD-RE und ADD 3.0 (FDD): Im Vergleich von COSMOD-RE und ADD 3.0 gibt es eine Vielzahl von Auffälligkeiten. So haben sowohl ADD 3.0 als auch COSMOD-RE als ein Ziel den Entwurf einer Software-Architektur. Zusätzlich hat COSMOD-RE jedoch noch weitere Ziele, wie die Gewinnung von Anforderungen. Um COSMOD-RE anzuwenden, müssen einige Randbedingungen geklärt werden, wie beispielsweise die Definition der Grenzen zwischen verschiedenen Ebenen der Abstraktionshierarchie oder eine Möglichkeit der Verknüpfung von Architektur- und Anforderungsmodellen. Außerdem muss eine System-Vision vorhanden sein. Ähnlich umfangreich sind die Randbedingungen von ADD 3.0. Hier wird vorausgesetzt, dass die Anforderungserhebung abgeschlossen ist und die Projekttreiber vorhanden sind. Besonders die Qualitätsattribute sollten vernünftig erhoben sein. Die ersten großen Unterschiede ergeben sich bei der Betrachtung der Eingaben. COSMOD-RE benötigt als Eingabe lediglich die System-Vision, während ADD 3.0 eine Menge von Projekttreibern benötigt. Die große Menge an ausführlichen Projekttreibern sind ein Nachteil an ADD 3.0, den COSMOD-RE nicht hat. Wenn es darum geht, einen schnellen Einstieg in die Methode zu erlangen,

hat COSMOD-RE hier einen Vorteil. Ein Nachteil von COSMOD-RE offenbart sich jedoch, wenn die Ausgaben betrachtet werden. Während COSMOD-RE hier lediglich grobe Architekturartefakte liefert, kann ADD 3.0 eine verfeinerte Software-Architektur generieren. Dies bedeutet, wenn der Mehraufwand zu Beginn betrieben wird, ist es möglich eine bessere Ausgabe zu produzieren.

Besonders auffällig sind die Geschwindigkeiten, in denen die Methoden arbeiten. COSMOD-RE hat vier Hierarchieebenen auf denen Anforderungen erhoben werden und die Software-Architektur erstellt wird. Dies wird über die drei CO-Design Prozesse geregelt, in denen jeweils zwei Hierarchieebenen untersucht werden. Jeder dieser drei CO-Design Prozesse führt die fünf Subprozesse aus, die wiederum vergleichbar mit einem Sprint bei SCRUM sind. Somit ist eine Mindestlaufzeit von drei SCRUM-Sprints gegeben, wobei hier nicht berücksichtigt ist, dass jeder der CO-Design Prozesse aufgrund der fünf Subprozesse länger als ein Sprint dauern kann. Bei ADD 3.0 ist ein Durchlauf vergleichbar mit einem Sprint bei SCRUM. Dies bedeutet, die verfeinerte Software-Architektur kann nach einem Sprint bereits fertig sein.

ADD 3.0 kann schnell Ergebnisse liefern. Dies ist jedoch hauptsächlich bei kleineren Projekten ein Vorteil. Bei größeren Projekten kann es passieren, dass man sehr unpräzise Ergebnisse erhält, wenn man bei ADD 3.0 nicht genügend Iterationen vollzieht. Hier hat COSMOD-RE den Vorteil, dass hier das Vorgehen durch die verschiedenen Hierarchieebenen und CO-Design Prozesse sehr strukturiert ist.

COSMOD-RE beginnt bei der Anforderungsgewinnung und arbeitet bis sowohl die Anforderungen als auch die Software-Architektur erstellt sind. In SP2 wird die Architektsicht erstellt. Es wird jedoch nicht konkret vorgegeben, wie die Architektsicht zu erstellen ist. Dies bedeutet, hier wäre die Freiheit gegeben, ADD 3.0 einzusetzen. Dadurch würde COSMOD-RE um ADD 3.0 erweitert werden. Dies würde zudem Lösungsansätze für die zwei Probleme liefern, die COSMOD-RE nicht gelöst bekommt.

f) Vergleich zwischen Probing und COSMOD-RE (FDD): COSMOD-RE hat als Ziele sowohl Anforderungen zu gewinnen als auch eine Software-Architektur zu erzeugen. Probing hat hingegen die Gewinnung von ASFR als Ziel. Probing soll es ermöglichen, schnell alle ASFR zu finden um so eine korrekte Software-Architektur zu ermöglichen. Währenddessen werden bei COSMOD-RE ausgehend von der System-Vision sowohl Anforderungen als auch Architekturartefakte abgeleitet. Betrachtet man die fünf Subprozesse bei COSMOD-RE ließe sich Probing der Entwicklung der Systemnutzungs-Sicht (SP1) zuordnen. Dies bedeutet bei einer Kombination der Ansätze würde durch Probing die initiale Systemnutzung-Sicht gewonnen werden,

die dann in weiteren Iterationen verfeinert würde. Eine Kombination der beiden Ansätze würde eine Lösung des Problems P6 für COSMOD-RE darstellen.

B. Bewertung

Unter der Berücksichtigung der genannten Probleme haben einige Methoden ein höheres Rating erzielt als andere. Insgesamt betrachtet hat CBSP mit 20 Punkten das höchste Rating, wenn als Grundlage die Menge der behandelten Probleme gegeben ist. COSMOD-RE hat mit 16 Punkten das zweithöchste Rating, Probing mit 14 das dritthöchste und ADD 3.0 mit 11 Punkten das niedrigste Rating. Dies bedeutet, CBSP und COSMOD-RE liefern bessere Lösungsansätze für die meisten Probleme. Hierbei ist allerdings zu beachten, dass CBSP und COSMOD-RE einen sehr großen Bereich des in Abbildung 1 skizzierten Prozesses abdecken, während Probing und ADD 3.0 nur einen Teil davon behandeln. Zudem bieten sowohl Probing als auch ADD 3.0 hauptsächlich nur für einen Teil der unter II aufgestellten Probleme einen Lösungsansatz an.

Bei der Betrachtung der Kompatibilität der Methoden fällt auf, dass sowohl CBSP als auch COSMOD-RE um Probing erweiterbar sind. In Kombination mit Probing würde CBSP ein Rating von 21 erzielen. Bei COSMOD-RE würde sich das Rating bei der Kombination mit Probing und ADD 3.0 auf 20 erhöhen, womit alle Probleme außer dem fehlenden Know-How gut zu lösen wären. Eine Kombination aus Probing und ADD 3.0 würde lediglich ein Rating von 16 erzeugen, was bei der Kombination von Methoden das schwächste Ergebnis liefert. Andere Kombinationen wie beispielsweise CBSP und COSMOD-RE oder CBSP und ADD 3.0 sind aufgrund von Konflikten in der methodischen Ausführung nicht möglich.

V. FAZIT

In der Untersuchung hat sich gezeigt, dass es eine Vielzahl von Problemen gibt, die Einfluss auf die Kooperation zwischen Requirements Engineer und Software-Architekt haben können. Ferner gibt es eine Vielzahl an Lösungsansätzen, die das Potenzial haben, die genannten Probleme zu lösen. In der Untersuchung der genannten Methoden hat sich jedoch gezeigt, dass diese geeignet sind um eine Teilmenge der Probleme zu lösen. Allerdings haben die Methoden unterschiedliche Schwächen, Stärken sowie Einsatzgebiete in welchen sie sinnvoll Anwendung finden. Zusätzlich hat sich gezeigt, dass die verschiedenen Methoden verschiedene Randbedingungen und Voraussetzungen haben, die zu erfüllen sind um die Anwendbarkeit zu gewährleisten. Um alle Probleme zu behandeln ist es jedoch notwendig verschiedene Methoden zu kombinieren.

VI. AUSBLICK

Im Vergleich der verschiedenen Methoden haben sich theoretische Erkenntnisse ziehen lassen, die in der Praxis

noch evaluiert werden müssen. Um dies zu realisieren sind jedoch verschiedene Schritte notwendig. Zunächst muss in der Praxis untersucht werden, ob die hier aufgezeigten Probleme tatsächlich auftreten und wie groß der Einfluss dieser auf die Kooperation von Requirements Engineer und Software-Architekt ist. Danach ist zu testen ob die Methoden in der praktischen Anwendung tatsächlich die Probleme in der ausgeführten Art lösen und den genannten Mehrwert bringen. Des Weiteren sind weitere Faktoren wie beispielsweise Projektgröße, Zeit oder weitere Ressourcen zu evaluieren, im Hinblick darauf, wie groß der Einfluss dieser auf Durchführbarkeit und den Erfolg der Methoden ist. Ferner ist zu validieren ob die Methoden sich in der Praxis tatsächlich zusammenführen lassen. Zuletzt ist zu untersuchen ob eine Zusammenführung der Methoden tatsächlich eine Verbesserung herbeiführt.

Neben den hier aufgeführten Methoden gibt es weitere wie die in [6] ausgeführte Erweiterung von COSMOD-RE, die das Potenzial haben die gegebenen Probleme zu lösen. Durch einen Vergleich mit weiteren Methoden könnten die Ergebnisse weiter verfeinert und validiert werden. Zusätzlich ist es möglich die einzelnen Methoden um weitere Aspekte zu erweitern und durch die Anwendung in der Praxis zu verbessern. So sind beispielsweise beim Probing nur zu fünf der 15 ASFR Kategorien PQ-Flows ausgearbeitet worden.

LITERATUR

- [1] E. Bjarnason, *Distances between Requirements Engineering and Later Software Development Activities: A Systematic Map*, Springer Berlin Heidelberg, 2013
- [2] J. D. Herbsleb, *Global Software Engineering: The Future of Socio-technical Coordination*, IEEE Computer Society Washington, 2007
- [3] K. Pohl, E. Sikora, *Structuring the Co-design of Requirements and Architecture*, Springer Berlin Heidelberg, 2007
- [4] K. Pohl, E. Sikora, *COSMOD-RE: Supporting the Co-Design of Requirements and Architectural Artifacts*, IEEE, 2007
- [5] H. Cervantes, R. Kazman *Designing Software Architectures: A Practical Approach*, Addison-Wesley Professional, 2016
- [6] E. Sikora, *Ein modellbasierter Ansatz zur verzahnten Entwicklung von Anforderungen und Architektur über mehrere Abstraktionsstufen hinweg*, Logos Verlag Berlin, 2011
- [7] G. Zorn-Pauli, B. Paech, T. Beck, H. Karey, G. Ruhe, *Analyzing an Industrial Strategic Release Planning Process A Case Study at Roche Diagnostics*, Springer Berlin Heidelberg, 2013
- [8] R. C. de Boer, H. van Vliet, *On the similarity between requirements and architecture*, Elsevier, 2009
- [9] P. Grnbacher, A. Egyed, N. Medvidovic, *Reconciling software requirements and architectures with intermediate models*, Springer Berlin Heidelberg, 2004
- [10] P. Rose Anish, B. Balasubramaniam, A. Sainani, J. Cleland-Huang, M. Daneva, R. J. Wieringa, S. Ghaisas, *Probing for requirements knowledge to stimulate architectural thinking*, ACM New York , 2016
- [11] P. Rose Anish, M. Daneva, J. Cleland-Huang, R. J. Wieringa, S. Ghaisas, *What you ask is what you get: Understanding architecturally significant functional requirements*, IEEE Computer Society Washington, 2015
- [12] P. Rose Anish, B. Balasubramaniam, J. Cleland-Huang, R. Wieringa, M. Daneva, S. Ghaisas, *Identifying architecturally significant functional requirements*, ACM New York, 2015