Lehrstuhl für
Wirtschaftsinformatik
und integrierte
Informationssysteme

UNIVERSITÄT
DUISBURG
ESSEN
Offen im Denken

Bachelorarbeit

Identifizierung und Implementierung ausgewählter Design Patterns für die Versionierung von Wirkungszusammenhängen

Vorgelegt der Fakultät für Informatik der
Universität Duisburg-Essen

von

David Neuhauss
Janssenstraße 28a
45147 Essen
Matrikelnummer: 3115632
Studiengang: Wirtschaftsinformatik

Gutachter:	Prof. Dr. Reinhard Schütte
	Prof. Dr. Stefan Eicker
Studiensemester:	Sommersemester 2024
	7. Fachsemester

Inhaltsverzeichnis

Abkürzungsverzeichnis

API .....Application Programming Interface
ARGUS .....Achieving Respect for Grades by Using ordinal Scales only
CBR .....Case Based Reasoning
CPW.....Combined Preference with Weights
CQRS .....Command and Query Responsibility Segregation
CRUD .....Create Read Update Delete
DDD.....Domain Driven Design
DTO .....Data Transfer Object
EBITDA ....Earnings Before Interest, Taxes, Depreciation and Amortization
HTTP.....Hypertext Transfer Protocol
IKT .....Informations- und Kommunikationstechnologie
IR.....Information Retrieval
IT .....Informationstechnologie
MCDA.....Multi Criteria Decision Analysis
NOPLAT....Net Operating Profit Less Adjusted Taxes
ORM .....Object Relational Mapper
REST.....Representational State Transfer
UML .....Unified Modelling Language
VCC .....Value Contribution Controlling

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

1.1 Problemstellung

„You can see the computer age everywhere but in the productivity statistics." (Solow, 1987, S. 36). Mit dieser Formulierung beschrieb der Ökonom und Nobelpreisträger Robert Solow das sogenannte Produktivitätsparadoxon. Es besagt, dass Investitionen in Informations- und Kommunikationstechnologie (IKT) scheinbar in keinem positiven Zusammenhang mit der Produktivität auf unternehmerischer oder volkswirtschaftlicher Ebene stehen (Brynjolfsson, 1993, S. 72–76).

In den letzten Jahrzehnten hat die IT eine zentrale Rolle in der Transformation von Geschäftsprozessen und Organisationsstrukturen eingenommen. Unternehmen wie Amazon und Uber haben Geschäftsmodelle wie den Einzelhandel und das Personentransportgewerbe fundamental verändert. Diese Entwicklung begann bereits im letzten Jahrtausend. So verdoppelte sich in den Vereinigten Staaten von Amerika zwischen den Jahren 1980 und 2000 der Anteil von Investitionen in IKT an den nicht-wohnwirtschaftlichen Investitionen. Im Jahr 2000 betrug dieser dann ca. 30 % (Colecchia & Schreyer, 2001, S. 9–10).

Dabei blieb trotz der weitreichenden Investitionen in IT-Systeme der erzielte Produktivitätszuwachs anfangs oft hinter den Erwartungen zurück oder es konnte überhaupt keine Produktivitätsveränderung beobachtet werden (Cron & Sobol, 1983, S. 171–181; Roach, 1998, S. 49–56). So berichten Kemerer und Sosa (1991, S. 214–217) von zahlreichen Beispielen, bei denen Unternehmen IT-Projekte nach großen Investments wieder einstellen mussten. Darunter das Beispiel des Logistikunternehmens FedEx, das seinen als Alternative zum Fax positionierten Service Zapmail nach Verlusten von ca. 350 Millionen US-Dollar einstellen musste (Kemerer & Sosa, 1991, S. 215).

Seitdem war das Produktivitätsparadoxon Gegenstand ausführlicher wissenschaftlicher Auseinandersetzung (Brynjolfsson & Yang, 1996, S. 179–209). In dieser Diskussion erklärten einige Veröffentlichungen das Produktivitätsparadoxon mit überhöhten Erwartungen, Messschwierigkeiten und dem langfristigen Charakter von IKT-Investitionen (Brynjolfsson, 1993, S. 73–76; Schweikl & Obermaier, 2020, S. 493–494, 2020, S. 489–491). Solow selbst äußerte sich im Jahr 2000 wie folgend: „You can now see computers in the productivity statistics" (Uchitelle, 2000, S. 143). Dennoch bleibt es bis heute Gegenstand wissenschaftlicher Diskussion, inwiefern das Produktivitätsparadoxon existiert und zu erklären ist (Hajli et al., 2015, S. 470–471).

Eng verbunden mit der Diskussion um das Produktivitätsparadoxon ist die Frage danach, wie der Wertbeitrag von IT-Investitionen bestimmt werden kann. Auf diese Frage gibt

es bis heute keine eindeutige Antwort, stattdessen wurden verschiedene Herangehensweisen zur Bestimmung des Wertbeitrags vorgeschlagen (Wagner & Beckmann, 2022, S. 1634–1635). Ein Ansatz ist das von Schütte et al. (2022, S. 364–412) vorgeschlagene Value Contribution Controlling (VCC). Dabei handelt es sich um ein in Phasen unterteiltes Vorgehensmodell zum Wirkungsmanagement von IT-Systemen (Schütte et al., 2022, S. 365). Ein zentraler Aspekt des VCC sind Wirkungskataloge, in denen die durch die Einführung eines IT-Service erwarteten Wirkungen dokumentiert werden. Weitere Bestandteile des VCC sind IT-Servicekataloge und Prämissenkataloge, deren jeweilige Entitäten in einem Zusammenhang miteinander stehen (Schütte et al., 2022, S. 366). Eine ausführlichere Erläuterung des Vorgehensmodells wird in Kapitel 2.2.1 beschrieben.

Theoretisch könnten Unternehmen das VCC manuell durchführen und sämtliche Kataloge und Artefakte rein textuell pflegen. Es liegt aber nahe, dass dieses Vorgehen einen unverhältnismäßigen Aufwand erfordert und die tiefergehende Analyse der gesammelten Informationen ebenfalls unverhältnismäßig aufwändig ist. Eine softwaregestützte Implementierung des VCC erscheint daher die angemessenere Wahl zu sein.

Ein erster Ansatz der softwaregestützten Implementierung ist das im Rahmen eines Bachelorprojektes von Studierenden entwickelte VCC-Tool (Evers et al., 2023). Dabei handelt es sich um eine webbasierte Anwendung, die den Nutzer[1] durch die verschiedenen Phasen des VCC leitet und dabei die VCC-Kataloge in einer MySQL-Datenbank anlegt und verwaltet. Bei der Erarbeitung des VCC-Tools wurde deutlich, dass eine der wesentlichen Anforderungen an eine solche Implementierung die Versionierung der VCC-Kataloge darstellt (Evers et al., 2023, S. 16–17). Schütte et al. (2022, S. 370–371) beschreiben die Versionen eines Katalogs als fortlaufende Nummer, die nach Veränderungen am Katalog inkrementiert wird. Ergänzt wird diese Versionsnummer um eine vorangestellte Zahl, welche die Phase des Vorgehensmodells angibt, in welcher die Version erstellt wurde.

Die Versionierung soll es ermöglichen, die Entwicklung der Kataloge in der zeitlichen Dimension darstellen und analysieren zu können (Schütte et al., 2022, S. 371). Dadurch könnten Fragen beantwortet werden wie „Welche (negativen) Wirkungen wurde erst im produktiven Einsatz eines IT-Systems bekannt?" „Wie haben sich die erwarteten Wirkungen eines IT-Systems im Laufe des Projektfortschritts entwickelt?", oder „Wie häufig und wann mussten die Kataloge aufgrund neuer Erkenntnisse oder Fehler überarbeitet werden?". Diese Informationen könnten dabei helfen, die Projektdynamiken eines Unternehmens besser zu verstehen, und so einen Lernprozess für zukünftige Projekte zu

[1] Zur besseren Lesbarkeit wird in dieser Arbeit das generische Maskulinum verwendet. Die in dieser Arbeit verwendeten Personenbezeichnungen beziehen sich auf alle Geschlechter.

ermöglichen. Auch in ihren Ausführungen zum Produktivitätsparadoxon heben Schütte et al. (2022, S. 150) die zeitliche Betrachtung von IT-Projekten als bedeutend für das Verständnis der Wirkung von IT-Systemen auf die Produktivität von Unternehmen hervor. Diese Anforderungen an die Versionierung konnten im VCC-Tool allerdings nur unzureichend umgesetzt werden, so dass die zuvor beschriebene Darstellung und Analyse nicht möglich ist (Evers et al., 2023, S. 39).

1.2 Zielsetzung und Gang der Arbeit

Ziel dieser Arbeit ist es, eine softwareseitige Lösung zur Versionierung der VCC-Kataloge zu entwickeln, die den thematisierten Anforderungen entspricht. Nach der Erarbeitung der Lösung soll diese implementiert und in das VCC-Tool integriert werden. So sollen nutzbare und korrekt versionierte VCC-Kataloge bereitgestellt werden, die das von Schütte et al. (2022, S. 364–508) beschriebene Wirkungsmanagement ermöglichen.

Dieses Ziel wird durch die Verwendung von Design Patterns erreicht, die in der deutschsprachigen Literatur auch als Entwurfsmuster bekannt sind. Design Patterns bieten bewährte Lösungsmuster für wiederkehrende Probleme in der Softwareentwicklung (bin Uzayr, 2022, S. 1–2). Design Patterns erscheinen als geeignetes Mittel, um mit der Komplexität der zu implementierenden Versionierung umzugehen und auf bestehende Lösungen für den Umgang mit zeitlichen Dimensionen zurückzugreifen.

Die vorliegende Arbeit gliedert sich in sechs Kapitel. Kapitel 2 bietet eine Einführung in die Bedeutung eines adäquaten Wertbeitragscontrollings von IT-Systemen (Kapitel 2.1). Es wird erläutert, wie das VCC-Vorgehensmodell dieses Controlling unterstützt (Kapitel 2.2), und das VCC-Tool wird als erster Ansatz einer Implementierung des VCC beschrieben (Kapitel 2.2.2). Zudem wird der theoretische Rahmen für die Identifizierung und Selektion von Design Patterns gelegt (Kapitel 2.3).

In Kapitel 3 wird ein geeignetes Design Pattern identifiziert. Zunächst wird ein methodisches Vorgehen entwickelt (Kapitel 3.1), das die vorherigen Ausführungen zu Design Patterns berücksichtigt. Daraufhin werden die Auswahlkriterien für das Design Pattern festgelegt (Kapitel 3.3), und eine geeignete Entscheidungsmethode aus der Literatur wird ausgewählt (Kapitel 3.4). In einer umfassenden Recherche werden potenzielle Design Patterns identifiziert (Kapitel 3.5), und anschließend wird mittels der gewählten Entscheidungsmethode das am besten geeignete Design Pattern ausgewählt (Kapitel 3.6).

Kapitel 4 widmet sich der Implementierung des ausgewählten Design Patterns. Die Anforderungen an das zu implementierende System werden beschrieben (Kapitel 4.1), grundlegende Architekturentscheidungen erörtert (Kapitel 4.2) und der Aufbau der ent-

wickelten Architektur erläutert (Kapitel 4.3). Anhand der Funktionsweise der Architekturkomponenten wird gezeigt, wie das Design Pattern die gestellten Anforderungen erfüllt (Kapitel 4.4). Abschließend wird die Integration der entwickelten Anwendung mit dem VCC-Tool demonstriert (Kapitel 4.5).

Kapitel 5 betrachtet die erzielten Ergebnisse der Arbeit. Es wird diskutiert, inwiefern das Ziel der Arbeit erreicht wurde und welche Erkenntnisse bezüglich der Implementierung der Versionierung sowie der Selektion von Design Patterns gewonnen wurden (Kapitel 5.1). Eine Auseinandersetzung mit den Limitationen der Arbeit vertieft die rückblickende Betrachtung (Kapitel 5.2). Im letzten Abschnitt werden Ansätze für zukünftige Arbeiten im Bereich der Selektion von Design Patterns und der Speicherung von Informationen mit Zeitbezug aufgezeigt (Kapitel 5.3).

Die Arbeit schließt in Kapitel 6 mit einem Fazit ab. Darin werden die Ergebnisse und Limitationen der vorliegenden Arbeit zusammengefasst.

2 Bestimmung des Wertbeitrags der IT

In diesem Kapitel werden die für das Verständnis dieser Arbeit essenziellen Themen und Konzepte erläutert. Dazu wird zuerst auf die Relevanz des Wertbeitrag der IT eingegangen. Anschließend wird das VCC als Vorgehensmodell zum Wirkungsmanagement vorgestellt. Dabei wird auch die Rolle des VCC-Tools als Ansatz einer softwaregestützten Implementierung des VCC-Vorgehensmodells betrachtet. Zuletzt wird auf die Rolle von Design Patterns zur Implementierung der versionierten VCC-Kataloge eingegangen. Dabei wird besonders auf die Identifizierung und Selektion von Design Patterns thematisiert.

2.1 Relevanz des Wertbeitrags der IT

Der Begriff der Digitalisierung fällt in der öffentlichen Diskussion häufig im Zusammenhang mit Produktivitätssteigerungen und der Einsparung von Arbeitskräften. Dabei ist es nicht selbstverständlich, dass Investitionen in IT automatisch positive Renditen erzielen. So stoppte beispielsweise das Logistikunternehmen Deutsche Post DHL 2015 die Einführung eines neuen IT-Systems und schrieb bereits getätigte Investitionen in Höhe von 345 Millionen Euro ab, da das System nicht die erhofften positiven Effekte erzielen konnte (Deutsche Post AG, 2015).

Seit mehreren Jahrzehnten wird in der Wirtschaftsinformatik die Frage nach dem Wertbeitrag von IT-Projekten und der IT im Allgemeinen intensiv diskutiert. 1993 untersuchte Brynjolfsson (S. 67–77) mögliche Erklärungen für die beobachtete Diskrepanz zwischen erwarteten und gemessenen Produktivitätssteigerungen nach Investitionen in IT-Systeme – dem sogenannten Produktivitätsparadoxon der IT. Er kam zu dem Ergebnis, dass ein signifikanter Teil der Diskrepanz möglicherweise durch unzureichende Messmethoden der Produktivität erklärt werden könnte (Brynjolfsson, 1993, S. 73–76). Diese These wurde auch von folgenden Veröffentlichungen gestützt, wobei wie in Kapitel 1.1 ausgeführt bis heute über das Vorliegen, mögliche Ursachen und die Dynamiken des Paradoxons diskutiert wird (Hajli et al., 2015, S. 465–471; Hitt & Brynjolfsson, 1996, S. 136–139; Lehr & Lichtenberg, 1999, S. 354–357; Schütte et al., 2022, S. 181–186). Diese Diskussion unterstreicht die Bedeutung geeigneter Bewertungsmethoden, um den tatsächlichen Wertbeitrag von IT-Systemen bestimmen zu können.

Auch Schütte et al. (2022, S. 185) betonen, dass ohne eine Quantifizierung des Wertbeitrags von IT-Systemen und Projekten in Unternehmen kein wertbeitragsorientiertes IT-Controlling erfolgen kann. Folglich können so auch keine fundierten Investitionsentscheidungen in IT-Systeme getroffen werden. Dabei ist es besonders in Zeiten sinkender Investitionsbereitschaft erforderlich, zur Verfügung stehendes Investitionsbudget bestmöglich zu nutzen.

Die Bestimmung des Wertbeitrags der IT ist aus mehreren Gründen kein einfaches Unterfangen. Zum einen sind die Auswirkungen von IT-Investitionen häufig indirekt und treten verzögert über einen längeren Zeitraum ein (Brynjolfsson, 1993, S. 75; Schütte et al., 2022, S. 169–170). Zu berücksichtigen ist auch, dass der gesteigerte Wertbeitrag nicht nur unmittelbar finanzieller Natur sein kann. Er kann sich beispielsweise auch in Form einer gesteigerten Marketingleistung, einer gesteigerten Anpassungsfähigkeit des Unternehmens oder höherer Kundenzufriedenheit ausdrücken (Masli et al., 2011, S. 87–89).

Ein bedeutender Teil der mit der Einführung von IT-Systemen verbundenen Kosten ist zudem auf komplementäre Investitionen zurückzuführen machen (Schütte et al., 2022, S. 184–185). Eine IT-Investition kann beispielsweise eine Reorganisation der Geschäftsprozesse erfordern oder neue Kompetenzen und Schulungen der Mitarbeiter notwendig machen. Auch die dadurch entstehenden Kosten und möglicherweise entstehenden positiven Effekte müssen bei der Bestimmung des Wertbeitrags betrachtet werden.

2.2 Value Contribution Controlling

Nachdem im vorherigen Abschnitt gezeigt wurde, dass es adäquater Methoden bedarf, um den Wertbeitrag von IT-Projekten näher bestimmen zu können, wird in diesem Unterkapitel das VCC nach Schütte et al. (2022, S. 364–412) als Ansatz zur Bewältigung dieser Herausforderung beschrieben.

2.2.1 Value Contribution Controlling als Ansatz zur Bestimmung des Wertbeitrags der IT

Das VCC nach Schütte et al. (2022, S. 364–412) ist ein Referenz-Vorgehensmodell, das entwickelt wurde, um das Wirkungsmanagement von IT-Systemen in Unternehmen zu verbessern. Es zielt darauf ab, die rationale Handlungsbasis von Entscheidern zu stärken und den wirtschaftlichen Nutzen von IT-Projekten näher zu bestimmen. Es kann sowohl auf einzelne IT-Projekte als auch iterativ auf ein übergreifendes IT-Servicemanagement angewendet werden. In diesem Fall wird das Vorgehensmodell für jedes Teilprojekt separat initialisiert (Schütte et al., 2022, S. 371).

Das VCC folgt einem strukturierten Vorgehensmodell, das aus sechs Phasen besteht. Während die Phasen durchlaufen werden, werden Wirkungs- Service- sowie Prämissenkataloge erstellt und gepflegt. Wirkungen, IT-Services und Prämissen stellen die zentralen, miteinander verknüpften, Entitäten des VCC dar und sollen den Kontext des Wirkungsmanagements abbilden. Die Entitäten werden im Laufe dieses Kapitels näher beschrieben, zunächst soll jedoch auf die in Abbildung 1 dargestellten Phasen des VCC eingegangen werden.

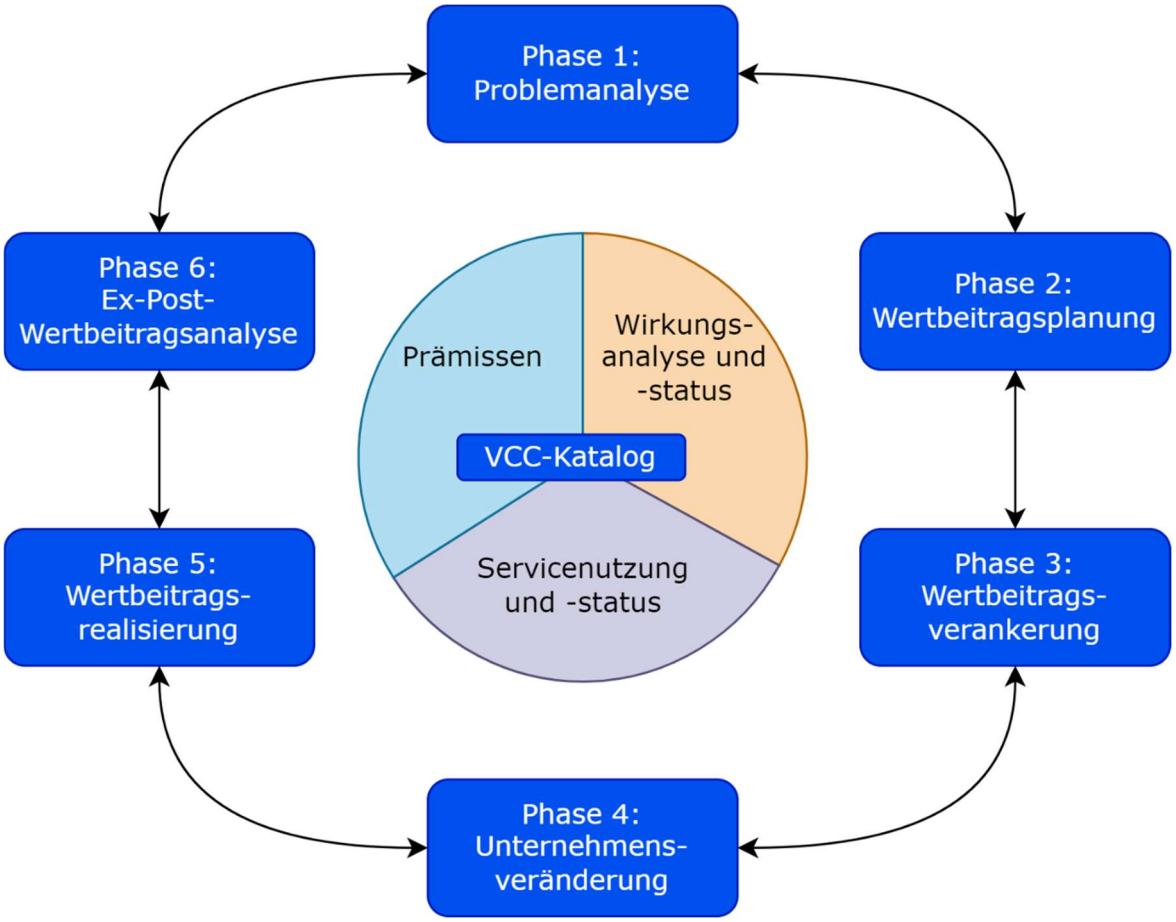

Abbildung 1: VCC-Vorgehensmodell nach Schütte et al. (2022, S. 366)

Jede Phase trägt spezifisch zur Identifizierung, Realisierung und Bewertung der im Rahmen des IT-Servicemanagements erwarteten Wertbeiträge bei. Die folgende Beschreibung der Phasen des VCC beruht auf Schütte et al. (2022, S. 373–412):

Phase 1: Problemanalyse

In der ersten Phase wird der Ist-Zustand der Organisation dokumentiert und der gewünschte Soll-Zustand skizziert. Dabei sollen durch Analyse die bestehenden Probleme und deren Auswirkungen auf die Organisation festgestellt werden. Durch eine Untersuchung der Lücke zwischen Ist- und Soll-Zustand soll ein tiefergehendes Verständnis der Probleme erzielt werden. Zudem werden erste Maßnahmen formuliert, durch die eine Überführung des Ist- in den Soll-Zustand erreicht werden kann und eine erste Schätzung möglicher Wirkungen der Maßnahmen ermittelt.

Phase 2: Wertbeitragsplanung

Die Wertbeitragsplanung ist eine zentrale, aus mehreren Schritten bestehende Phase des VCC. Zuerst werden mögliche Wirkungen einzuführender IT-Services sowie deren Wertbeiträge mittels einer Wirkungspotentialanalyse identifiziert. Hierfür können Referenzwerte aus früheren Projekten oder Benchmarks herangezogen werden. Anschlie-

ßend werden die Wertbeiträge in eine hierarchische Struktur gebracht und mit ökonomischen Zieldimensionen verknüpft, um die Messbarkeit zu gewährleisten. Zudem wird dokumentiert, welchen Prämissen die spezifizierten Wertbeiträge unterliegen. Abschließend werden erste Versionen der Kataloge angelegt, die die geplanten Wirkungen und die zugrunde liegenden Annahmen dokumentieren.

Phase 3: Wertbeitragsverankerung

In der Phase der Wertbeitragsverankerung werden die geplanten Wirkungen innerhalb der Organisation verortet. Das umfasst die Abstimmung mit Stakeholdern über die geplanten Wirkungen und die damit verbundenen Maßnahmen. Außerdem wird festgelegt, welche Personen oder Abteilungen für die Umsetzung und das Monitoring der Wirkungen verantwortlich sind.

Phase 4: Unternehmensveränderung

In dieser Phase werden die geplanten IT-Services eingeführt und die entsprechenden notwendigen organisatorischen Anpassungen vorgenommen. Dies umfasst das Change Management sowie das Monitoring und eventuell notwendige Anpassungen der umzusetzenden Maßnahmen.

Phase 5: Wertbeitragsrealisierung

Nachdem die Unternehmensveränderung stattgefunden hat, erfolgt in dieser Phase die Realisierung der Wertbeiträge. Durch den Einsatz geeigneter Methoden und Auswertung des betrieblichen Berichts- und Rechnungswesens werden die tatsächlich eingetretenen Ist-Werte erfasst und in den Katalogen festgehalten.

Phase 6: Ex-Post-Wertbeitragsanalyse

In der letzten Phase des Vorgehensmodells erfolgt eine nachträgliche Evaluation der realisierten Wertbeiträge. Es erfolgt ein Abgleich der Soll- mit den Ist-Werten, eventuelle Abweichungen werden diskutiert und dokumentiert. Auf dieser Basis kann bestimmt werden, ob und in welchem Umfang der erwartete Wertbeitrag durch den IT-Service / die IT-Services erzielt werden konnte. Zudem werden Maßnahmen geplant und durchgeführt, um eventuelle Lücken zwischen Soll- und Ist-Werten zu schließen. Aus den gewonnenen Erkenntnissen soll gelernt werden, um zukünftige Projekte besser planen und umsetzen zu können.

Im Folgenden werden die VCC-Kataloge und ihre Entitäten näher beschrieben. Die Beschreibung beruht auf den Ausführungen von Schütte et al. (2022, S. 497–507).

IT-Service bezeichnet ein spezifisches IT-System oder eine Anwendung, die im Rahmen des VCC eingeführt oder angepasst werden soll. IT-Services sind die zentralen Entitäten, deren Nutzen und Wertbeitrag analysiert und bewertet werden. Ein IT-Service

kann beispielsweise ein ERP-System, ein CRM-System oder eine spezielle Softwarelösung sein.

Service-Kataloge: Diese Kataloge dokumentieren die verschiedenen IT-Services, deren funktionalen und technischen Status, sowie deren Nutzung in der Organisation. Sie enthalten detaillierte Informationen über Eigenschaften, Servicestatus und Servicenutzung der IT-Services und dienen als Grundlage für die Bewertung der erwarteten und realisierten Wirkungen.

Prämissen sind Annahmen und Rahmenbedingungen, die den Kontext und die Voraussetzungen für die Realisierung von IT-Services und deren Wertbeiträgen definieren. Schütte et al. (2022, S. 503–505) unterscheiden zwischen Wirkungsvoraussetzungen, Servicevoraussetzungen und Strategische- und Umweltvoraussetzungen.

Prämissen-Kataloge: Diese Kataloge listen die unterschiedlichen auf. Dies umfasst beispielsweise technische Voraussetzungen wie die Kompatibilität von Systemen oder organisatorische Voraussetzungen wie die Verfügbarkeit von qualifiziertem Personal.

Wirkungen beschreiben die erwarteten und tatsächlichen Veränderungen und Ergebnisse, die durch den Einsatz eines IT-Services in der Organisation erzielt werden. Sie werden in vorökonomische Wirkungen, denen in der Regel keine konkreten Werte zugrunde liegen, und quantifizierbare Wertbeiträge unterteilt (Schütte et al., 2022, S. 200, 2022, S. 373).

Wirkungs-Kataloge: Diese Kataloge erfassen und strukturieren die verschiedenen Wirkungen der IT-Services. Sie beinhalten Informationen darüber, welche Veränderungen durch die IT-Services erwartet werden, wie diese gemessen werden können und welche wirtschaftlichen oder organisatorischen Ziele damit verbunden sind. Die Wirkungen werden in eine hierarchische Struktur gebracht, um ihre Bedeutung und ihre Beziehungen zueinander besser zu verstehen und bewerten zu können. So können vorökonomische Wirkungen in Wertbeiträgen aufgehen. Abbildung 2 zeigt einen beispielhaften Ausschnitt eines Wirkungskatalogs, der dieses Prinzip verdeutlicht.

VCC-Katalog: Der VCC-Katalog integriert die vorher genannten Wirkungs-, Service- und Prämissenkataloge in eine ganzheitliche Sicht.

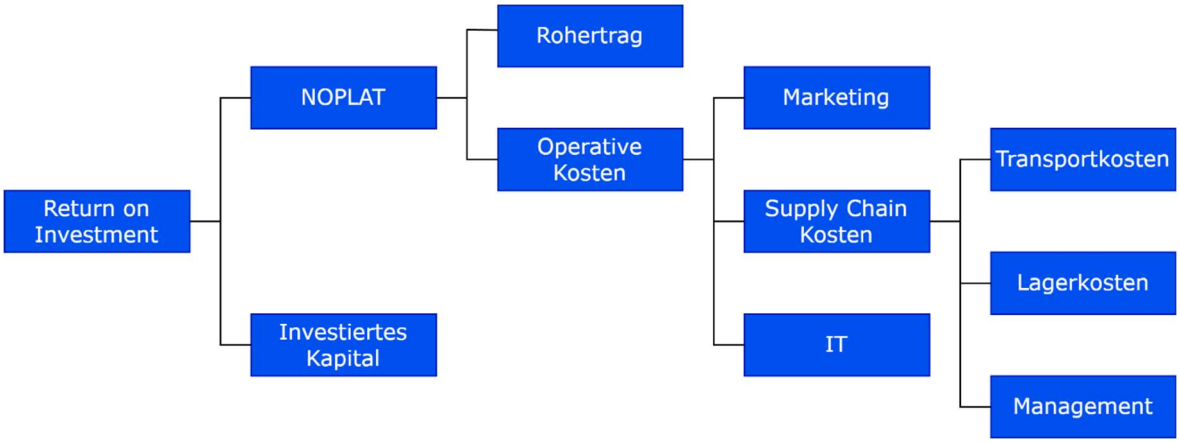

Abbildung 2: Auszug aus einem Wirkungskatalog nach Schütte et al. (2022, S. 433)

Für jede der sechs Phasen liegen jeweils eine oder mehrere Versionen der beschriebenen Kataloge vor. Dabei gilt die in Kapitel 1.1 beschriebene Versionierungslogik (Schütte et al., 2022, S. 370).

Schütte et al. (2022, S. 420–507) beschreiben Aufbau und Ablauf der einzelnen Phasen und Entitäten deutlich ausführlicher als es an dieser Stelle wiedergegeben werden kann. Dabei werden auch konkrete Methoden und Analysetechniken vorgeschlagen, die während der Phasen zum Einsatz kommen können.

2.2.2 Das VCC-Tool als Implementierung des Value Contribution Controlling

Das VCC-Tool wurde im Rahmen eines studentischen Projektes entwickelt und stellt den ersten Versuch einer informationstechnischen Implementierung des VCC-Vorgehensmodells dar. Das VCC-Tool ist eine webbasierte Anwendung, die sich an den von Schütte et al. beschriebenen Phasen und Katalogen orientiert. Dabei wurden allerdings einige Abstraktionen und Vereinfachungen an dem beschriebenen Vorgehensmodell vorgenommen (Evers et al., 2023, S. 22–25).

So wurde eine vierte Entität hinzugefügt, die Umrechnungsregel. Umrechnungsregeln werden ebenfalls während der sechs Phasen in Katalogen dokumentiert und dienen der Umrechnung von Wirkungen, die in nicht-monetären Einheiten gemessen werden, in eine finanzielle Zieldimension, in der sich letztlich alle eingetretenen Wirkungen äußern. Im VCC-Tool ist diese Zieldimension das EBITDA der betrachteten Organisation (Evers et al., 2023, S. 26–27). Die Kataloge und ihre Entitäten sind im VCC-Tool immer einem Projekt zugeordnet. Das Projekt nimmt dabei also die integrierende Rolle des VCC-Katalogs ein (Evers et al., 2023, S. 26). Folglich existiert im VCC-Tool kein gesonderter VCC-Katalog. Stattdessen ermöglicht die Projektansicht eine kombinierte Ansicht der einzelnen Kataloge, zwischen denen durch Klick auf eine verknüpfte Entität navigiert werden kann. Im weiteren Verlauf dieser Arbeit wird der Begriff VCC-Katalog für die Prämissen-, Service- und Wirkungskataloge verwendet.

Der Umfang der Attribute der Entitäten wurde im Vergleich zu den von Schütte et al. beschriebenen Attributen deutlich reduziert (Evers et al., 2023, S. 26). Abbildung 3 zeigt ein Klassendiagramm der für diese Arbeit wesentlichen Entitäten des VCC-Tools, aus dem die jeweiligen Attribute und Beziehungen hervorgehen.[2] In Anhang A - befindet sich ein Auszug aus der Dokumentation der im Rahmen dieser Arbeit erstellten Anwendung. In diesem werden einige der Attribute näher erläutert. Da der Quellcode des VCC-Tools vollständig in englischer Sprache verfasst ist, werden in dieser und folgenden Abbildungen die englischen Bezeichnungen der jeweiligen Entitäten verwendet.[3]

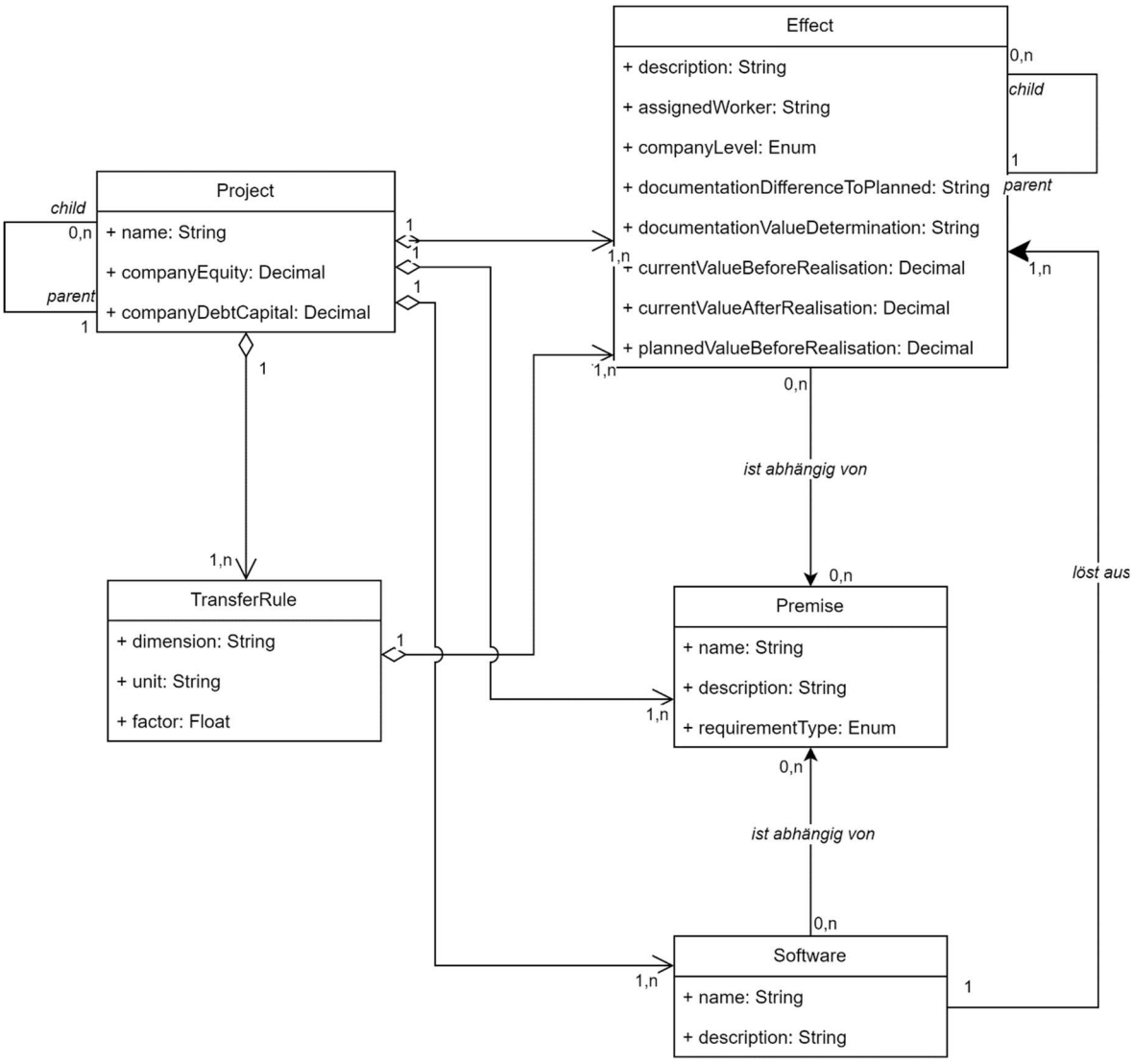

Abbildung 3: Klassendiagramm der Entitäten im VCC-Tool nach Evers et al. (2023, S. 26)

[2] Es wird an dieser Stelle festgelegt, dass alle Abbildungen, die Quellcode zeigen oder sich darauf beziehen, in englischer Sprache verfasst werden, da auch sämtlicher betrachteter Quellcode in englischer Sprache verfasst ist.

[3] Um Verwechselungen zu verhindern, wird aufgrund der geläufigen Verwendung des Begriffs Service als Bezeichnung für Architekturkomponenten einer Software, die Entität IT-Service im Programmiercode als Software bezeichnet.

Das VCC-Tool führt den Nutzer durch die Phasen des VCC-Vorgehensmodells. Auf Grundlage des dabei erstellen Wirkungskataloges, sowie der bei Projekterstellung angegebenen Finanzkennzahlen, kann in Phase 6, der Ex-Post-Wertbeitragsanalyse, der Wertbeitrag des Projektes zum EBITDA sowie der Return on Investment ermittelt und visualisiert werden. Während das VCC-Tool nur eine vereinfachte Abbildung des VCC bietet und gewissen Limitationen unterliegt, stellt sie dennoch eine erste nutzbare Implementierung des Vorgehensmodells dar (Evers et al., 2023, S. 35–40).

Während der Ausarbeitung des VCC-Tools wurde die Bedeutung einer umfassenden Versionierung der VCC-Kataloge deutlich. Zum einen ermöglicht sie die Nachvollziehbarkeit von Änderungen und Entscheidungen während des Projektverlaufs. Gleichzeitig stellt sie aber auch die Grundlage für tiefergehende Auswertungen wie Szenarioanalysen dar (Evers et al., 2023, S. 16). Details der Versionierung wurden bereits in Kapitel 1.1 beschrieben. Die im VCC-Tool implementierte Versionierung wird den beschriebenen Anforderungen nicht gerecht (Evers et al., 2023, S. 39). Die bisher implementierte Versionierung muss manuell durch den Nutzer angestoßen werden. Es werden also keine automatischen, granularen Versionen für jede Änderung angelegt. Wird eine neue Version angelegt, besteht diese aus einer vollständigen Kopie der VCC-Kataloge zum jeweiligen Zeitpunkt (Evers et al., 2023, S. 28–29). Dieser Ansatz ist unter mehreren Gesichtspunkten unzureichend. Die Speicherung der Informationen ist ineffizient und würde bei größeren VCC-Katalogen, die in der komplexen betrieblichen Realität zu erwarten sind, zu einem hohen Speicherbedarf führen. Als nachteilig zu bewerten ist auch die fehlende Granularität. Durch die Speicherung der gesamten VCC-Kataloge, wird nicht festgehalten, welche Änderungen konkret vorgenommen wurden. Diese Information könnte höchsten aufwendig durch einen Vergleich aller Attribute aller Entitäten rekonstruiert werden. Probleme könnten auch bei der parallelen Bearbeitung durch mehrere Benutzer entstehen. Ohne eine fein-granulare Versionierung besteht ein erhöhtes Risiko von Konflikten und Inkonsistenzen, wenn mehrere Benutzer gleichzeitig Änderungen vornehmen. Diese Konflikte könnten zu Datenverlust oder ungewollten Überschreibungen führen, was die Integrität der Daten gefährdet und die Nachvollziehbarkeit der Änderungen weiter erschwert.

2.3 Design Patterns

Die zuvor angesprochenen Limitationen der Versionierung sollen mit der in dieser Arbeit implementierten Versionierung der VCC-Kataloge überwunden werden. Dafür werden, wie in Kapitel 1.2 erläutert, Design Patterns verwendet. In den folgenden Unterkapiteln wird eine Einführung in das Thema Design Patterns gegeben. Besonders wird dabei auf den wissenschaftlichen Stand zur Identifizierung und Selektion geeigneter Design Patterns für ein vorliegendes Problem eingegangen.

2.3.1 Design Patterns als wiederverwendbare Entwurfsmuster

Design Patterns haben ihren Ursprung in der Architektur. In dieser Domäne bezeichneten Alexander et al. (1977, S. X) bereits 1977 Design Patterns als Beschreibung eines wiederkehrenden Problems in Verbindung mit einer Beschreibung einer wiederverwendbaren Lösung.

Dieses Konzept wurde dann durch Gamma et al. (1994) auf die Softwareentwicklung übertragen. Sie beschreiben in ihrem Werk „Design Patterns: Elements of Reusable Object-Oriented Software" 23 Entwurfsmuster für gängige Problemstellungen der Softwareentwicklung. Darunter sind Design Patterns die bis heute Verwendung finden, wie das Singleton Design Pattern oder das Facade Design Pattern (Gamma et al., 1994, S. 144–152, 1994, S. 208–217).

Design Patterns unterscheiden sich von anderen Ansätzen zur Lösung wiederkehrender Probleme in der Softwareentwicklung, wie Frameworks und Bibliotheken, durch ihre abstrakte Natur (Gamma et al., 1994, S. 41). Sie stellen keine implementierte, direkt nutzbare Lösung dar. Stattdessen beschreiben sie einen generischen Lösungsansatz für ein abstraktes Problem, welcher durch den Anwender unter Berücksichtigung des Technologie- und Problemkontextes zu implementieren ist (Buschmann et al., 1996, S. 7–8). Zdun (2007, S. 9) formuliert es folgendermaßen:

„A pattern does not describe a structural element of the system to which it is applied, but it can be seen as an event in a creative design process.".

Nach der Veröffentlichung von Gamma et al. (1994) wurde das Konzept der Design Patterns in der Softwareentwicklung in zahlreichen Veröffentlichungen weiterentwickelt und auf verschiedene Bereiche der Softwareentwicklung angewendet (Bafandeh Mayvan et al., 2017, S. 100; Buschmann et al., 1996, S. XII–XIII; Henninger & Corrêa, 2007, S. 3–4). Heutzutage gelten Design Patterns als etablierter Bestandteil der Softwareentwicklung. Sie erleichtern Softwareentwicklern die Kommunikation über Probleme und Lösungen und können dazu beitragen die Entwicklungseffizienz durch das Wiederverwenden existierender, bewährter Lösungen zu steigern (bin Uzayr, 2022, S. 8; McConnell, 2004, S. 103). Dadurch tragen sie dazu bei, Wissen über Lösungen verfügbar zu machen.

Mittlerweile wurde eine kaum überschaubare Vielzahl an Design Patterns beschrieben. Der 2000 von Rising (2000) veröffentliche Pattern Almanach listet über 1000 Design Patterns auf. 2007 identifizierten Henninger und Corrêa (S. 2) bereits über 2000 Patterns. Sie wiesen außerdem darauf hin, dass zwischen vielen Design Patterns Überschneidungen existieren, was die Komplexität der Betrachtung weiter erhöhe (Henninger & Corrêa, 2007, S. 6). Dies führe dazu, dass das bloße Studium von Fachbüchern nicht mehr ausreiche, um das geeignete Design Pattern für eine Problemstellung zu

kennen. Es bedürfe also geeigneter Werkzeuge um Patterns finden und auswählen zu können (Henninger & Corrêa, 2007, S. 13). Dazu komme, dass die Beschreibung und Charakterisierung von Design Patterns je nach Quelle variieren würde, so dass ein direkter Vergleich zwischen beschriebenen Patterns oft nicht unmittelbar möglich sei. Diese uneinheitliche Beschreibungsweise stelle eine Hürde für formale/werkzeugunterstützte Auswahlmethoden von Design Patterns dar (Henninger & Corrêa, 2007, S. 8).

Auf einer abstrakteren Ebene lassen sich Design Patterns als Tupel aus Kontext, Problem und Lösung beschreiben (Alexander, 1979, S. 247).

Eine bis heute verwendete Klassifizierung von Design Patterns stammt von Gamma et al. (1994, S. 21–22). Sie unterscheiden unter dem Überbegriff des Zwecks (Purpose) zwischen strukturbasierten Design Patterns (Structural Patterns), verhaltensbasierten Design Patterns (Behavioral Patterns) und entwurfsbasierten Design Patterns (Creational Patterns). Entwurfsbasierte Patterns betreffen die Erstellung von Objekten, während strukturbasierte Patterns sich auf die Zusammensetzung von Objekten und Klassen beziehen. Verhaltensbasierte Patterns dagegen beziehen sich auf die Art und Weise wie Klassen oder Objekte miteinander interagieren (Gamma et al., 1994, S. 21–22).

Sabatucci et al. (2015, S. 137–138) differenzieren zwischen natürlichsprachlichen, semi-formalen und formalen Ansätzen Design Patterns zu dokumentieren. Semi-formale Beschreibungen bestehen meist aus einer UML-Darstellung des Patterns. Sie kritisieren hier allerdings die Fokussierung auf die Struktur des Design Patterns anstelle seiner Konzepte und Ideen, die nicht in einem Diagramm dargestellt werden können. Formale Ansätze, wie die von Eden et al. (Eden et al., 1998) vorgeschlagene Sprache LePUS, versuchen die Spezifikation von Design Patterns in eine eindeutige, grammatikalischen Regeln folgende, Logik zu überführen. Dadurch soll eine automatisierte Verarbeitung einer Auswahl von Patterns möglich werden (Eden et al., 1998, S. 31).

Die geläufigste Beschreibungsweise besteht allerdings aus einer textuellen Beschreibung des Patterns, ergänzt um eine UML-basierte Darstellung der Funktionsweise (Sabatucci et al., 2015, S. 136). Gamma et al. (1994, S. 16–17) beschreiben Design Patterns anhand des Namen sowie einer Klassifizierung anhand des zuvor erläuterten Zwecks des Patterns. Des Weiteren nennen Gamma et al. (1994, S. 16–18) folgende Merkmale:

- **Absicht (Intent):** Was soll das Design Pattern tun?
- **Auch bekannt als:** Unter welchen Namen ist das Design Pattern noch bekannt?
- **Motivation:** Ein Szenario, das verdeutlicht, wie das Design Pattern ein Problem löst.
- **Anwendbarkeit:** In welchen Situationen kann das Design Pattern angewendet werden?

- **Struktur:** Eine graphische Darstellung des Design Patterns.
- **Zusammenarbeit (Collaborations):** Wie arbeiten die Elemente des Design Patterns zusammen, um die Absicht des Patterns zu erfüllen?
- **Konsequenzen:** Welchen Folgen hat die Verwendung des Design Patterns? Sind Nachteile zu berücksichtigen?
- **Implementierungen:** Was ist bei einer Implementierung des Design Patterns zu berücksichtigen?
- **Beispielcode:** Beispielhafter Quellcode, der die Funktionsweise des Design Patterns verdeutlicht.
- **Bekannte Anwendungen:** Beispiele für bekannte Anwendungen des Design Patterns in produktiven Systemen.
- **Verwandte Patterns:** Welche Patterns weisen eine Nähe zu diesem auf? Mit welchen Design Patterns wird das beschrieben Design Pattern zusammen genutzt?

Die von Gamma et al. (1994, S. 16–18) verwendeten Merkmale und Klassifizierungen finden sich immer noch in vielen Veröffentlichungen, um Design Patterns zu beschreiben. Je nach Quelle wird die Beschreibung dabei um weitere Merkmale ergänzt oder reduziert.

Während Design Patterns heutzutage etablierter Bestandteil der Softwareentwicklung sind, kann ihre Verwendung auch kritisch betrachtet werden. Bin Uzayr (2022, S. 14) kritisiert, dass ihre Verwendung zu ineffizientem Quellcode führe, da das bloße Anwenden eines Patterns zu einer Verkomplizierung des Codes führe, was letztlich den Wartungsaufwand erhöhe. Auch Gamma et al. (1994, S. 44) weisen darauf hin, dass Design Patterns nicht blindlings auf jedes Problem angewendet werden sollten. Es müsse eine Abwägung zwischen der zusätzlichen Komplexität und der gewonnenen Flexibilität stattfinden.

Es wird deutlich, dass die Nutzung von Design Patterns neben einigen Vorteilen auch einigen Risiken unterliegt. Daher erscheint es kritisch, neben der grundsätzlichen Entscheidung Design Patterns zu nutzen, das für die vorliegende Problemstellung optimale Design Pattern zu identifizieren. Wie bereits erläutert, ist die Auswahl eines Patterns aber aufgrund der Vielzahl an existierenden Patterns und Beschreibungsformen nicht trivial.

Birukou (2010, S. 2–3) unterteilt den Prozess zur Anwendung von Design Patterns auf ein Problem auf drei Schritte: die Identifizierung möglicherweise geeigneter Patterns, die Selektion eines Patterns und dessen Anwendung auf das vorliegende Problem. Kann das Problem dadurch nicht gelöst werden, beginnt die Identifizierung von Patterns erneut. Dieser iterative Prozess ist auf Abbildung 4 zu sehen.

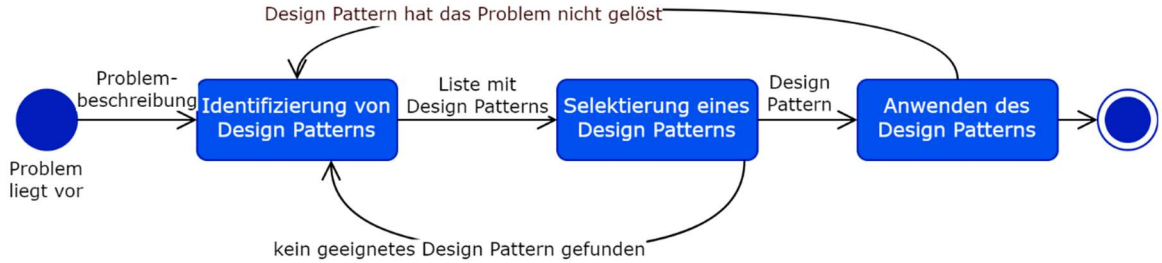

Abbildung 4: Prozess zur Anwendung von Design Patterns auf ein Problem nach Birukou (2010, S. 3)

Das in dieser Arbeit behandelte Design Problem wurde bereits in Kapitel 1.1 beschrieben. Die zwei folgenden Unterkapitel behandeln folglich die Identifizierung und Selektion von Design Patterns. Die Anwendung des ausgewählten Patterns wird im weiteren Verlauf dieser Arbeit in Kapitel 4 beschrieben.

2.3.2 Identifizierung von Design Patterns

Die Identifizierung in Frage kommender Design Patterns stellt einen kritischen Schritt im Prozess der Implementierung einer Softwarelösung dar. Dieses Unterkapitel zielt darauf ab, bestehende Ansätze zur Identifizierung von Design Patterns zu erörtern, die als Grundlage für das später zu wählende Vorgehen dienen können.

Das grundlegende Problem bei der Identifizierung von Design Patterns für eine Problemstellung der Softwareentwicklung ist, dass es keine einheitliche Quelle oder Datenbank gibt, auf die Suchende zurückgreifen könnten (Birukou, 2010, S. 3; Henninger & Corrêa, 2007, S. 4). Stattdessen existiert eine Vielzahl möglicher Medien, in denen Design Patterns beschrieben werden. Diese reichen von Büchern bis hin zu Websites (Henninger & Corrêa, 2007, S. 3). Um für die Lösung eines Problems möglichst viele existierende Design Patterns in Betracht zu ziehen, muss also eine medienübergreifende Recherche durchgeführt werden. Folgende Quellen können dabei herangezogen werden:

- **Bücher** können umfassende Sammlungen von Design Patterns enthalten und darüberhinausgehende Kontextinformationen bieten. Bekannte Beispiele sind „Design Patterns: Elements of Reusable Object-Oriented Software" von Gamma et al. (1994), „Pattern-Oriented Software Architecture – A System of Patterns" von Buschmann et al. (1996) oder „The Pattern Almanac 2000" von Rising (2000).
- **Wissenschaftliche Publikationen** bieten Einsichten in neue Forschungsergebnisse und innovative Design Patterns (Henninger & Corrêa, 2007, S. 7).
- **Online-Datenbanken / Design Pattern Suchmaschinen** Während es einige Versuche gab, spezialisierte Pattern Datenbanken/Suchmaschinen zu etablieren, scheinen diese keine Verbreitung gefunden zu haben und sind mittlerweile inak-

tiv. Birukou (2010, S. 6–8) nennt hier unter anderem „Patternforge", „Open Pattern Repository" oder „PatternShare". Mittlerweile ist keine der genannten Websites mehr aktiv.

- **Websites** bieten Zugang zu Patterns die nicht in Form einer Publikation veröffentlich wurden. Hierbei kann es sich um die Websites einzelner Entwickler oder Plattformen wie „StackOverflow" oder „GitHub" handeln.
- Aufzeichnungen und Publikationen von **Workshops oder Konferenzen** bieten Einblick in aktuelle Trends und Diskussionen innerhalb der Entwicklergemeinschaft (Henninger & Corrêa, 2007, S. 7).

Bei der Bearbeitung dieser Quellen muss sorgfältig bewertet werden, ob Design Patterns in die Liste möglicher Lösungen übernommen werden. Besonders Internetquellen wie Websites sind naturgemäß kritisch zu betrachten. Allerdings ist es wichtig zu erkennen, dass viele Design Patterns aus der Praxis und den Diskussionen innerhalb der Entwicklergemeinschaft entstehen. Daher sollten auch diesen Quellen nicht vernachlässigt werden. Dabei muss aber beachtet werden, dass diese Design Patterns nicht den gleichen Überprüfungsmechanismen unterliegen, wie sie in Fachbüchern üblich sind.

Anschließend sollten die identifizierten Design Patterns in eine einheitliche Darstellungsform überführt werden, die für die gewählte Selektionsmethode geeignet ist (Naghdipour et al., 2023, S. 1097–1098).

2.3.3 Selektion von Design Patterns

Nach der Identifizierung möglicher Design Patterns gilt es, das für das vorliegende Problem am besten geeignete Pattern auszuwählen. Die Frage, wie dieses Pattern auszuwählen ist, war in der Vergangenheit Thema zahlreicher Publikationen (Asghar et al., 2019; Naghdipour et al., 2023; Sahly & Sallabi, 2012; Zamani et al., 2009). Diese Aufmerksamkeit könnte teilweise durch die zunehmende Anzahl an verfügbaren Design Patterns, welche die Auswahl von Design Patterns komplexer gemacht hat, zu erklären sein. Gleichzeitig hat sich jedoch auch das Verständnis von Softwareentwicklung gewandelt. Eher als eine Art Kunsthandwerk betrachtet, zeichnete sich Softwareentwicklung durch den individuellen Stil und die persönlichen Techniken des Entwicklers aus. Code war geprägt von der persönlichen Note des Schöpfers. Dieser kunsthandwerkliche Ansatz in der Softwareentwicklung hat sich zunehmend zu einer wissenschaftlich fundierten Disziplin entwickelt (Austin & Devin, 2003, S. 93–95). Dieser Wandel wurde durch das Aufkommen strukturierter Programmiermethoden und die Akzeptanz von Konzepten wie Design Patterns vorangetrieben. In diesem neuen Paradigma spielen Design Patterns eine zentrale Rolle, da sie eine standardisierte Lösung für wiederkehrende Probleme bieten und somit die Kunst der Softwareentwicklung in eine systematische

Praxis überführen. Die Selektion des passenden Design Patterns wird damit zu einem entscheidenden Faktor für die Qualität und den Erfolg eines Softwareprojekts.

Gamma et al. (1994, S. 41–42) heben hervor, dass es schon bei 23 beschriebenen Design Patterns schwierig sei, das zum Problem passende Pattern auszuwählen. Sie schlagen mehrere Ansätze dazu vor. Diese Ansätze stellen weniger ein konkretes Vorgehen dar, als mehr Denkansätze, die bei der Auswahl eines Patterns berücksichtigt werden sollten. Der Kerngedanke dabei ist, jedes potenzielle Pattern eingehend zu studieren und im Hinblick auf seine Funktionsweise, seinen Zweck und den Zusammenhang mit anderen Design Patterns zu verstehen (Gamma et al., 1994, S. 42).

Mit der gestiegenen Anzahl an Design Patterns und Komplexität moderner Softwareprojekte, hat aber auch die Bedeutung formalisierter und standardisierter Verfahren zur Auswahl geeigneter Design Patterns zugenommen. Dabei haben sich unterschiedliche Herangehensweisen ausgebildet. Naghdipour et al. (2021, S. 5) identifizierten sieben, auf Abbildung 5 dargestellte, Ansätze zur Selektion von Design Patterns. Diese variieren hinsichtlich ihres Automatisierungs- und Formalisierungsgrades.

UML-basierte Ansätze umfassen die Modellierung des Problems in Form eines Diagramms. Ein geeignetes Design Pattern wird durch den Abgleich des Diagramms mit UML-Diagrammen der infrage kommenden Design Patterns identifiziert (Kim & El Khawand, 2007, S. 568–586; Kim & Shen, 2008, S. 329–347). Diese Verfahren gehen allerdings mit einem hohen Zeit- und Ressourcenaufwand einher und scheinen mittlerweile nicht mehr weiterentwickelt zu werden (Naghdipour et al., 2021, S. 4–5).

Bei den Verfahren des **Case-Based Reasoning** besteht ein Fall üblicherweise jeweils aus einem Problem, einer Lösung und einem Ergebnis. Eine Fallsammlung besteht aus mehreren Fällen, die jeweils ein vergangenes, erfolgreich gelöstes Problem beschreiben (Watson & Marir, 1994, S. 331). Die Fallsammlung kann genutzt werden, um ein neues Designproblem zu lösen, indem die Ähnlichkeit zwischen dem aktuellen Problem und den gespeicherten Fällen bewertet wird (Muangon & Intakosum, 2013, S. 20–21). Dadurch kann ein passendes Design Pattern vorgeschlagen werden (Naghdipour et al., 2021, S. 2). Naghdipour et al. (2021, S. 5) geben für diese Verfahren allerdings nur eine mittlere Güte der Vorhersagequalität bei hohem zeitlichen Aufwand an.

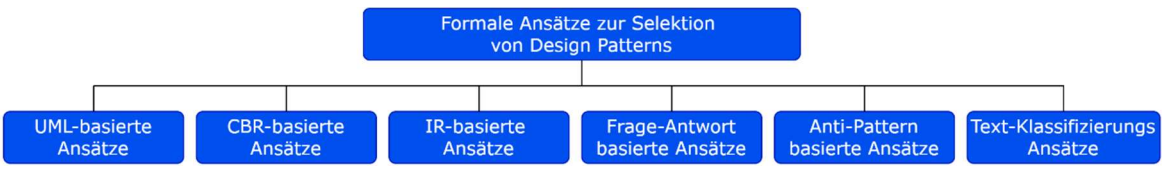

Abbildung 5: Formalisierte Ansätze zur Selektion von Design Patterns nach Naghdipour et al. (2021, S. 5)

Information Retrieval (IR) Verfahren extrahieren Merkmale aus den textuellen Beschreibungen von Design Patterns und Designproblemen, um die Ähnlichkeit zwischen ihnen zu berechnen und so ein geeignetes Design Pattern zu identifizieren. Diese Verfahren können vollautomatisch durchgeführt werden (Hamdy & Elsayed, 2018, S. 41–45; Rahmati et al., 2019, S. 247–249). Werden sie nicht mit anderen Verfahren kombiniert, weisen IR Verfahren allerdings eine geringe Vorhersagegenauigkeit auf (Naghdipour et al., 2021, S. 5).

Frage-Antwort-basierte Verfahren stellen dem Entscheider eine Reihe von Fragen zu dem zu lösenden Designproblem. Basierend auf den Antworten wird ein geeignetes Design Pattern vorgeschlagen. Dies setzt allerdings voraus, dass ein Fragen-Modell für die vorliegende Art von Design-Problem bereits erarbeitet wurde (Palma et al., 2012, S. 2–5).

Anti-Pattern-basierte Verfahren identifizieren Anti-Patterns, um zu bestimmen, welche Design Patterns für die Lösung eines bestimmten Problems ungeeignet sind. Dadurch können passende Design Patterns durch Ausschluss der Anti-Patterns vorgeschlagen werden (Nahar & Sakib, 2015, S. 2–6). Dieser Ansatz scheint allerdings noch wenig verbreitet zu sein (Naghdipour et al., 2021, S. 4).

Textklassifikationsverfahren nutzen Algorithmen des überwachten oder unüberwachten maschinellen Lernens, um Design Patterns basierend auf textuellen Beschreibungen automatisch zu klassifizieren und auszuwählen (Hussain et al., 2019, S. 5–7). Diese Verfahren zeigen vielversprechende Ergebnisse, setzen allerdings eine ausreichend große Menge an Trainingsdaten voraus (Hussain et al., 2017, S. 239–240).

Eine flexiblere Alternative zu diesen formalen, auf Design Patterns spezialisierten Verfahren, stellt die **Multi Criteria Decision Analysis (MCDA)** dar. Dabei handelt es sich um eine Klasse von Verfahren zur Lösung multikriterieller Entscheidungsprobleme (Nemery & Ishizaka, 2013, S. 1–5). Diese Verfahren sind nicht speziell für die Selektion von Design Patterns konzipiert, wurden jedoch bereits für die Lösung von Problemstellungen der Softwareentwicklung angewendet (Moaven & Habibi, 2020, S. 4578–4594; Nawaz et al., 2015, S. 129–137; Vijayalakshmi et al., 2010, S. 22–26). In der Literatur wird eine Vielzahl von der MCDA zuzuordnenden Verfahren beschrieben. Diese unterscheiden sich dabei unter anderem hinsichtlich der Komplexität der Verfahren, sowie der Art wie die Kriterien evaluiert und gegebenenfalls priorisiert werden.

Wątróbski et al. (2019, S. 112) identifizierten 56 unterschiedliche Verfahren. Angesichts dieser Menge an möglichen Methoden stellt bereits die Wahl der geeigneten Methode ein Entscheidungsproblem dar. Wątróbski et al. (2019, S. 109–119) entwickelten deshalb ein Vorgehensmodell, um Anwender bei der Auswahl einer geeigneten MCDA-Methode zu unterstützen. Das Modell strukturiert mittels eines Entscheidungsbaums die

Entscheidung anhand der Problemcharakteristika und der gewünschten Methodeneigenschaften. Dabei wird das vorliegende Entscheidungsproblem anhand von vier Fragen charakterisiert, gegebenenfalls erweitert um tiefergehende Nachfragen (Wątróbski et al., 2019, S. 111–112). Für jede dieser Fragen liegt als Ergebnis eine Teilmenge an möglichen Methoden vor. Die Schnittmenge aller Teilmengen repräsentiert die für das konkrete Entscheidungsproblem geeigneten MCDA-Methode (Wątróbski et al., 2019, S. 116).

Es wird deutlich, dass vielfältige methodische Ansätze existieren, um die Auswahl eines geeigneten Design Pattern vorzunehmen. Dabei fällt auf, dass einige der auf die Auswahl von Design Patterns spezialisierten Verfahren ein bereits für die spezifische Problemstellung vorbereitetes Modell (Frage-Antwort-basierte Verfahren) voraussetzen. Andere Verfahren basieren auf bereits dokumentierten Fällen (CBR) oder erfordern eine umfassende Kenntnis der Design Patterns (Anti-Pattern-basierte Verfahren). Dabei werden sie oft anhand von Entscheidungsproblemen demonstriert, bei denen aus heterogenen Pattern-Sammlungen wie der von Gamma et al. (1994) ausgewählt wird. Mit heterogen ist gemeint, dass die beschriebenen Design Patterns unterschiedlichen Zwecken dienen und sich stark in ihrer Struktur voneinander unterscheiden. Im Gegensatz dazu soll in dieser Arbeit jedoch eine Auswahl zwischen Design Patterns mit dem gleichen Zweck, der Versionierung von Wirkungszusammenhängen, getroffen werden. Es ist unklar, inwiefern die beschriebenen, auf Design Patterns spezialisierten Verfahren für die vorliegende Problemstellung geeignet sind.

Für den weiteren Verlauf dieser Arbeit wird sich daher dazu entschieden, das Entscheidungsmodell von Wątróbski et al. zu nutzen, um eine MCDA-Methode für die Auswahl des zu implementierenden Design Patterns auszuwählen. Auch wenn MCDA-Methoden im Gegensatz zu spezialisierten Methoden nicht konkret für die Selektion von Design Patterns entwickelt wurden, erscheinen sie hier die sinnvollere Wahl darzustellen. Dies liegt daran, dass sie die Flexibilität bieten, vom Anwender festgelegte Kriterien zu berücksichtigen und nicht auf vorbereitete Modelle oder dokumentierte Fälle angewiesen sind.

3 Identifizierung geeigneter Design Patterns zur Versionierung von Wirkungszusammenhängen

In diesem Kapitel wird zuerst das methodische Vorgehen beschrieben und angewendet, mit dem ein geeignetes Design Pattern für die Versionierung der VCC-Kataloge identifiziert und selektiert werden soll.

3.1 Methodisches Vorgehen

Unter Berücksichtigung der vorangegangenen Betrachtung zu Design Patterns und ihrer Identifizierung und Selektion im speziellen wird das folgende mehrschrittige Vorgehen verwendet, um das geeignete Design Pattern für die Versionierung der VCC-Kataloge auszuwählen.

1. **Festlegung einer einheitlichen Darstellungsform**
   Die Auswahl eines Design Patterns beginnt mit der Festlegung einer einheitlichen Darstellungsform für alle betrachteten Design Patterns. Dieser Schritt ist notwendig, um eine Vergleichbarkeit zu gewährleisten. Dabei soll auf die in Kapitel 2.3.1 erörterten Ansätze zur Darstellung von Design Patterns zurückgegriffen werden.
2. **Definition von Auswahlkriterien**
   Im nächsten Schritt werden Kriterien definiert, anhand derer Design Patterns auf ihre Eignung für die vorliegende Problemstellung der Versionierung hin bewertet werden können. Dabei werden sowohl in der Softwareentwicklung gängige Faktoren als auch Anforderungen, die aus dem Kontext des VCC-Vorgehensmodell hervorgehen, berücksichtigt werden.
3. **Auswahl einer geeigneten MCDA-Methode**
   Mithilfe des Entscheidungsmodells von Wątróbski et al. (2019, S. 116) wird eine MCDA-Methode ausgewählt, die dazu geeignet ist, ein Design Pattern für die behandelte Problemstellung auszuwählen.
4. **Identifizierung möglicher Design Patterns**
   Anschließend wird eine extensive Recherche in Frage kommender Design Patterns durchgeführt. Um eine möglichst umfassende Auswahl zu erhalten, werden wissenschaftliche Veröffentlichungen, Fachbücher, Konferenzbeiträge aber auch Websites oder Blogs von Entwicklern herangezogen werden. Die identifizierten Design Patterns müssen außerdem in die im ersten Schritt festgelegte Darstellungsform überführt werden.

5. **Selektion des Design Pattern**
   Im letzten Schritt wird die ausgewählte MCDA-Methode angewendet, um das geeignete Design Pattern auszuwählen.

3.2 Festlegung einer einheitlichen Darstellungsform

Je nach Quelle werden Design Patterns anhand verschiedener Merkmale beschrieben. Aus Gründen der Vergleichbarkeit ist es wichtig im Rahmen dieser Arbeit eine einheitliche Darstellungsform festzulegen. Dazu wird folgendes Schema verwendet:

- **Name**
- **Klassifikation** anhand der von Gamma et al. (1994, S. 21–22) verwendeten Klassifizierung in struktur-, verhaltens- und entwurfsbasierte Design Patterns.
- **Auch bekannt als**, andere Bezeichnungen unter denen das Design Pattern auch erwähnt wird.
- **Struktur** des Patterns als UML-Darstellung.
- **Funktionsweise**
- **Verwandte Design Patterns**, die inhaltliche Ähnlichkeiten aufweisen oder mit dem beschriebenen Design Pattern zusammen verwendet werden.

Dies sind im Wesentlichen die von Gamma et al. (1994, S. 16–18) verwendeten Merkmale. Dabei wurde allerdings bewusst auf Merkmale wie Motivation und Anwendbarkeit verzichtet. Da sich alle in der vorliegenden Arbeit betrachteten Design Patterns mit der Versionierung und Historisierung von Daten beschäftigen, ist die Motivation für alle eine ähnliche. Die Anwendbarkeit ist in dieser Arbeit nur für die für die vorliegende Problemstellung von Bedeutung. Diese wird durch die Anwendung der Entscheidungsmethode überprüft.

3.3 Auswahlkriterien für das Design Pattern

Die Auswahl eines optimalen Design Patterns erfordert eine sorgfältige Definition der Kriterien. Diese werden sowohl aus allgemeinen Prinzipien der Softwareentwicklung als auch aus spezifischen Anforderungen des VCC-Vorgehensmodells abgeleitet. Dadurch soll sichergestellt werden, dass die implementierte Lösung den Standards der Softwareentwicklung entspricht und das VCC-Vorgehensmodell optimal unterstützt. Die VCC-spezifischen Kriterien basieren auf den Ausführungen von Schütte et al. (2022, S. 370–371) sowie der an das VCC-Tool formulierten Anforderungen bezüglich der Versionierung und Bearbeitung der VCC-Kataloge (Evers et al., 2023, S. 16–17). Nicht-VCC-spezifische Kriterien (C4, C5, C6, C7) orientieren sich an den Ausführungen von McConnell (McConnell, 2004, S. 48, 80, 464) zu den Prinzipien qualitativ hochwertiger Software.

Dafür werden folgende Kriterien definiert:

- **C1: Wiederherstellbarkeit vergangener Zustände:** Das Pattern muss in der Lage sein, vergangene Zustände der VCC-Kataloge darzustellen. Wichtig ist dabei auch, dass Relationen zwischen einzelnen Katalogeinträgen zeitlich korrekt dargestellt werden.[4] Das bedeutet, dass sowohl die Relationen als auch alle verknüpften Einträge historisch korrekt wiederhergestellt werden können.
- **C2: Abbildung des Änderungsverlaufs:** Darüber hinaus sollte das Pattern es ermöglichen, die im Zeitverlauf vorgenommenen Änderungen möglichst detailliert abzubilden.
- **C3: Identifizierbarkeit einzelner Versionen:** Einzelne Versionen sollten konkret identifizierbar sein, beispielsweise anhand von Versionsnummern.
- **C4: Wartbarkeit:** Änderungen an der Versionierung und den betroffenen Katalogen, sowie das Implementieren neuer Anforderungen sollten ohne umfassende Überarbeitungen möglich sein.
- **C5: Skalierbarkeit:** Das Pattern soll in der Lage sein, mit wachsenden Kataloggrößen zu skalieren, ohne dass es zu Leistungseinbußen kommt.
- **C6: Leistungsfähigkeit:** Die Leistungsfähigkeit des Patterns in Bezug auf Antwortzeiten und Ressourcennutzung ist wichtig, um eine effiziente Systemnutzung sicherzustellen. Das bedeutet auch, dass das Speichern der Versionen möglichst wenig zusätzlichen Speicherbedarf erzeugen sollte.
- **C7: Komplexität:** Bei gleicher Leistung sind weniger komplexe Lösungen zu bevorzugen, da diese in der Regel einfacher zu warten und erweitern sind.
- **C8: Parallelität:** Mehrere Benutzer sollten gleichzeitig an den Katalogen arbeiten können, ohne dass es zu inkonsistenten Zuständen der Kataloge kommen kann.

Diese Kriterien werden in Kapitel 3.6 als Grundlage für die Entscheidung dienen, welches der identifizierten Design Patterns ausgewählt werden sollte.

3.4 Bestimmung einer Selektionsmethode

3.4.1 Anwendung des Wątróbski-Entscheidungsmodells

Zur Auswahl einer geeigneten MCDA-Methode wird das von Wątróbski et al. (2019, S. 109–121) entwickelte Vorgehensmodell verwendet. Den Kern des Modells ein aus vier Hauptfragen bestehender Entscheidungsbaum. Jede der vier Hauptfragen evaluiert ein Charakteristikum (C1 – C4) des Entscheidungsproblems, ergänzt durch eventuelle untergeordnete Charakteristika (z.B. C1.1). Im Folgenden werden die vier Charakteristika des in dieser Arbeit behandelten Entscheidungsproblems bestimmt, um eine geeignete MCDA-Methode zu identifizieren. Die folgenden Fragen leiten sich direkt aus dem von

[4] Diese Relationen liegen in Form von Fremdschlüsselbeziehungen vor.

Wątróbski et al. (2019, S. 116) aufgestellten Entscheidungsbaum ab. Der Entscheidungsbaum findet sich auch in Anhang B dieser Arbeit.

C1 – Sollen die Kriterien gewichtet werden? Grundsätzlich kann davon ausgegangen werden, dass bei der Auswahl eines Design Patterns manche Aspekte höher priorisiert werden sollten als andere. So erscheint beispielsweise das Kriterium der Wiederherstellbarkeit vergangener Zustände wichtiger als das der Skalierbarkeit, da es direkt aus den Anforderungen des VCC abgeleitet werden kann. Daher wird diese Frage mit *Ja* beantwortet. Dies führt zu Eigenschaft C1.1 – welche Art der Gewichtung vorliegt. Zur Auswahl stehen hier die Antworten *qualitativ*, *quantitativ* oder *relativ*. In dem vorliegenden Entscheidungsproblem liegt keine natürliche quantitative Gewichtung der Kriterien vor. Auch eine relative Gewichtung der Kriterien würde eine exakte Priorisierung erfordern, die mi. Daher wird sich für eine qualitative Gewichtung der Kriterien entschieden. Somit liegt als Ergebnis des Charakteristikums C1 die Teilmenge S1b vor. Diese und alle weiteren identifizierten Teilmengen werden in der Abbildung 6 gezeigt.

C2 – Auf welcher Art von Skala sollen die Varianten verglichen werden? Mögliche Antworten sind *qualitativ*, *quantitativ* oder *relativ*. Da eine quantitative Bewertung der Kriterien aller identifizierten Design Patterns den Umfang dieser Arbeit übersteigen würde und ein relativer Vergleich aufgrund der unterschiedlichen Quellen der Patterns nicht möglich ist, wird sich für eine qualitative Bewertung der Kriterien entschieden. Daraus ergibt sich die Teilmenge S2a.

C3 – Ist das Entscheidungsproblem von Unsicherheit geprägt? Mögliche Unsicherheiten können bezüglich der Eingabedaten oder der Präferenzen des Entscheiders bestehen. Für das vorliegende Entscheidungsproblem wird zur Vereinfachung angenommen, dass keine Unsicherheiten bestehen. Dies führt zu der Teilmenge S3a.

C4 – Welche Art von Entscheidung soll getroffen werden? Die Art des Entscheidungsproblem wird von der Art des gewünschten Ergebnisses bestimmt. Es kann sich in Form einer eindeutigen, optimalen Entscheidung (Auswahl), einer Sortierung (Ranking) oder einer Einordnung der Varianten in nach Präferenz geordneten Kategorien äußern. Da im vorliegenden Entscheidungsproblem eine eindeutige Entscheidung für ein zu implementierendes Pattern getroffen werden soll, wird hier die Option einer eindeutigen Auswahl gewählt. Daraus ergibt sich die Teilmenge S4a.

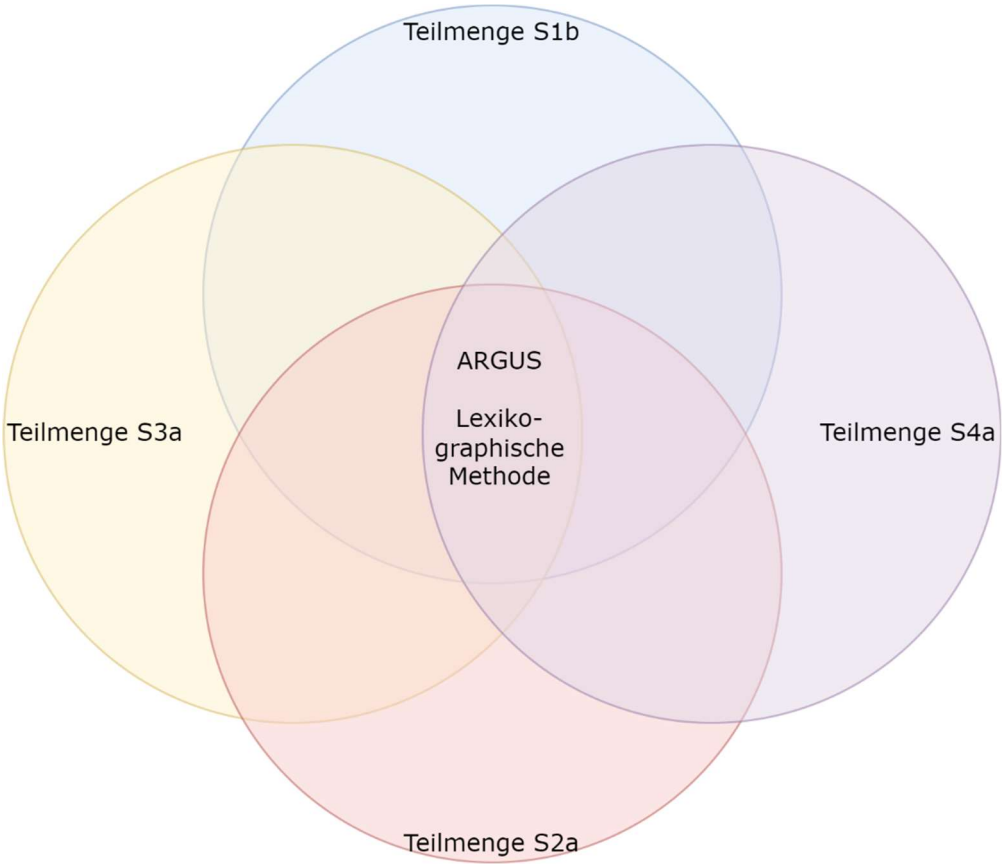

Abbildung 6: Schnittmenge der identifizierten Ergebnisteilmengen

Wie auf Abbildung 6 zu erkennen ist, entsprechen zwei MCDA-Methoden allen vier definierten Charakteristika des vorliegenden Entscheidungsproblems. Dabei handelt es sich um die Methoden ARGUS und eine Lexikographische Methode. ARGUS steht für „Achieving Respect for Grades by Using ordinal Scales only" und basiert auf der Idee, Entscheidungsprozesse durch ausschließliche Nutzung ordinaler Skalen zu vereinfachen, indem es sowohl die Präferenzintensitäten als auch die Wichtigkeit der Kriterien in einer mehrdimensionalen Entscheidungssituation ordinal modelliert (De Keyser & Peeters, 1994, S. 272).

Die Lexikographische Methode fokussiert sich dagegen darauf, Kriterien in einer streng hierarchischen Reihenfolge zu bewerten, wobei das wichtigste Kriterium zuerst betrachtet wird und nur bei Gleichstand das nächstwichtige Kriterium entscheidet (Fishburn, 1974, S. 1444–1447).

Während ARGUS komplexere Entscheidungsstrukturen durch die ausschließliche Nutzung ordinaler Skalen vereinfacht und eine gewisse Flexibilität in der Gewichtung der Kriterien ermöglicht, bietet die lexikographische Methode eine direktere, weniger flexible aber klar strukturierte Entscheidungsfindung, bei der die Reihenfolge und das Gewicht der Kriterien strikt festgelegt sind (De Keyser & Peeters, 1994, S. 272–277; Fishburn, 1974, S. 1444–1447).

Da bei der Auswahl des Design Patterns für die Versionierung ausdrücklich mehrere Kriterien berücksichtigt werden sollen, wird sich in dieser Arbeit für die ARGUS-Methode zur Auswahl eines Design Patterns entschieden. Im folgenden Abschnitt wird die Anwendung der Methode näher erläutert.

3.4.2 ARGUS-Methode zur Lösung multikriterieller Entscheidungsprobleme

ARGUS wurde von De Keyser und Peeters (1994, S. 263–274) entwickelt und basiert auf der Idee des paarweisen Vergleichs aller Alternativen miteinander. Dabei wird das Konzept des Outrankings angewendet. Dieses besagt, dass eine Alternative a einer Alternative b vorzuziehen ist (formalisiert als aSb), wenn a in Bezug auf die betrachteten Kriterien insgesamt vorzuziehen ist, auch wenn sie in einzelnen Kriterien unterlegen sein mag. Dabei ist zu berücksichtigen, dass nicht zwangsläufig für jedes Paar aus Alternativen eine eindeutige Präferenz ermittelt werden kann (Zimmermann & Gutsche, 1991, S. 204–206). Stattdessen können zwei Alternativen a und b auch nicht miteinander vergleichbar (aRb) oder indifferent (aIb) sein (Zimmermann & Gutsche, 1991, S. 205). Des Weiteren berücksichtigt ARGUS das Konzept der Diskordanz. Dieses besagt, dass Alternative a Alternative b trotz Überlegenheit nicht vorzuziehen ist, wenn für ein Kriterium von b gilt, dass es dem Kriterium von a in besonders starkem Maße überlegen ist (De Keyser & Peeters, 1994, S. 272). Wann dies der Fall ist, wird durch den Anwender der Methode festgelegt.

De Keyser und Peeters (1994, S. 272–274) beschreiben ein aus fünf Schritten bestehendes Verfahren, mit dem aus einer gegebenen Menge an Alternativen A eine oder mehrere optimale Alternativen bestimmt werden können. Die folgende Erläuterung der einzelnen Schritte beruht, sofern nicht anders angegeben, auf den Ausführungen von De Keyser und Peeters (1994, S. 263–277):

a. Datensammlung

Zuerst gilt es die Menge aller zu betrachtenden Alternativen A, sowie die Kriterien $f_1(x), \dots, f_k(x) \mid x \in A$ anhand derer diese verglichen werden, zu definieren. Dabei steht k für die Anzahl der zu betrachtenden Kriterien. Für jedes Kriterium gilt es nun eine Bewertung vorzunehmen, inwiefern die Alternative dem Kriterium entspricht. Diese Bewertung kann sowohl quantitativ als auch qualitativ erfolgen. Eine Überführung in eine ordinale Skala erfolgt im späteren Verlauf der Methode. Tabelle 1 zeigt die dabei entstehende Matrix anhand des von De Keyser und Peeters verwendeten Beispiels eines Vergleichs von potenziellen Standorten eines Atommüllendlagers. Dabei werden sieben alternative Standorte anhand von vier Kriterien bewertet.

Tabelle 1: Beispiel einer Bewertungsmatrix nach De Keyser und Peeters (1994, S. 274)

Kriterium / Alternative	Kosten	Freisetzung radioaktiver Abfall	Widerstand der Bevölkerung	Kapazität
Standort 1	9,7	ungenügend	keiner	0,3
Standort 2	17,5	keine	niedrig	0,37
Standort 3	13,6	ungenügend	hoch	0,33
Standort 4	12,0	vernachlässigbar	hoch	0,35
Standort 5	10,0	ungenügend	niedrig	0,31
Standort 6	14,2	keine	moderat	0,34

b. Präferenzmodellierung

Nun muss für jedes Kriterium eine Präferenzmatrix aufgestellt werden. Das Vorgehen für die Aufstellung ergibt sich aus dem vorliegenden Skalenniveau. Bei einer ordinalen Skala werden die möglichen Werte des Kriteriums auf der x- und y-Achse der Matrix, jeweils von unerwünschten zu erwünschten Werten sortiert, angeordnet. Das untere Dreieck der entstehenden Matrix sagt aus, ob und ggf. wie stark die Ausprägung eines Kriteriums $f_i(a)$ (auf der y-Achse) der Ausprägung $f_i(b)$ (auf der x-Achse) vorzuziehen ist ($f_i(a) > f_i(b)$). De Keyser und Peeters (1994, S. 264) verwenden dafür die Präferenzstufen indifferent, leicht Präferenz, moderate Präferenz, starke Präferenz und sehr starke Präferenz. Das obere Dreieck der Matrix sagt aus, dass für den entsprechenden Vergleich gilt, dass b a vorzuziehen ist.[5] Liegt für einen Vergleich Diskordanz vor, wird dies ebenfalls im oberen Dreieck der Matrix vermerkt.

Tabelle 2 zeigt eine beispielhafte Präferenzmatrix für eine Ordinalskala. In diesem Beispiel sind niedrige Werte vorzuziehen. Daraus ergibt sich, dass eine Alternative mit einem als niedrig bewerteten Kriterium einer Alternative mit einem als hoch bewerteten Kriterium stark (strong) vorzuziehen wäre.

Tabelle 2: Beispielhafte Präferenzmatrix für eine Ordinalskala nach De Keyser und Peeters (1994, S. 275)

Kriterium / Alternative	Kosten	Freisetzung radioaktiver Abfall	Widerstand der Bevölkerung	Kapazität
Standort 1	9,7	ungenügend	keiner	0,3
Standort 2	17,5	keine	niedrig	0,37
Standort 3	13,6	ungenügend	hoch	0,33
Standort 4	12,0	vernachlässigbar	hoch	0,35
Standort 5	10,0	ungenügend	niedrig	0,31
Standort 6	14,2	keine	moderat	0,34

[5] Wie stark diese Präferenz ausgeprägt ist, kann ermittelt werden, indem die Alternative b nun auf der y-Achse und die Alternative a auf der x-Achse betrachtet wird.

Das Vorgehen ist für Intervall- und Verhältnisskalen ähnlich, es wird aber zusätzlich betrachtet, wie groß der tatsächliche numerische Unterschied zwischen den Werten ist. Hier kann der Entscheidungsträger eine Funktion definieren, die den Grad der Präferenz basierend auf dem numerischen Unterschied der beiden Werte angibt. Für diesen Fall ist auf Tabelle 3 ein Beispiel abgebildet. Wie auf der Abbildung zu sehen ist, kann auch hier Diskordanz definiert werden.

Tabelle 3: Beispielhafte Präferenzmatrix für eine Verhältnisskala nach De Keyser und Peeters (1994, S. 275)

Präferenz (a über b)	$d = f_i(a) - f_i(b)$
indifferent	$0{,}0 \leq d < 1{,}6$
niedrig	$1{,}6 \leq d < 3{,}2$
moderat	$3{,}2 \leq d < 4{,}8$
stark	$4{,}8 \leq d < 6{,}5$
sehr stark	$6{,}5 \leq d < \infty$
diskordant	$d < -8{,}1$

c. Gewichtung der Kriterien

Im dritten Schritt der ARGUS-Methode gewichtet der Entscheidungsträger die Kriterien durch eine Einteilung in Prioritätsklassen. Wie auf Tabelle 5 gezeigt, verwenden De Keyser und Peeters (1994, S. 266) fünf Prioritätsklassen mit Ausprägungen von nicht wichtig bis hin zu extrem wichtig. Dabei können mehrere Kriterien der gleichen Prioritätsklasse zugeordnet werden.

Tabelle 4: Beispielhafte Gewichtung der Kriterien nach De Keyser und Peeters (1994, S. 276)

Gewicht	
nicht wichtig	
wenig wichtig	
moderat wichtig	C4
sehr wichtig	C1, C3
extrem wichtig	C2

d. Kombinierte Präferenzstruktur

In diesem Schritt werden die zuvor definierten Präferenzstufen zusammen mit den Prioritätsklassen verwendet, um eine Gesamtbewertung der Alternativen zu erstellen. Dabei wird jedes Paar aus Präferenzstufe und Prioritätsklasse eines Kriteriums in eine Hierarchie eingeordnet. Die Herleitung dieser Hierarchie wird methodisch über zwei Tabellen (hier Tabelle 5 und Tabelle 6) abgeleitet.

Zunächst werden die Präferenzstufen mit den Prioritätsklassen zu der in Tabelle 5 abgebildeten Matrix kombiniert. Diese enthält auf der x-Achse alle Prioritätsklassen in aufsteigender Reihenfolge. Auf der y-Achse befinden sich drei Abschnitte. Zunächst werden für den Fall $f_i(a) > f_i(b)$ die Präferenzstufen in absteigender Reihenfolge aufgelistet. Es

folgt der Fall der Gleichwertigkeit $f_i(a) = f_i(b)$. Der dritte Abschnitt listet für den Fall $f_i(a) < f_i(b)$ die Präferenzstufen in aufsteigender Reihenfolge. Die Zellen der Matrix werden entsprechend dem Schema $\sum_{i=1}^{9}\sum_{j=1}^{5} f_{ij} = k$ nummeriert. Dabei indexiert i die Reihen der Matrix während j die Spalten nummeriert. De Keyser und Peeters (1994, S. 271) bezeichnen die entstehende Matrix als Präferenz-Wichtigkeits-Tabelle.

Tabelle 5: Beispielhafte Darstellung der Präferenz-Wichtigkeits-Tabelle nach De Keyser und Peeters (1994, S. 271)

	Priorität Präferenz	nicht wichtig	wenig wichtig	moderat wichtig	sehr wichtig	extrem wichtig
$f_i(a) > f_i(b)$	sehr stark	f_1_1	f_1_2	f_1_3	f_1_4	f_1_5
	stark	f_2_1	f_2_2	f_2_3	f_2_4	f_2_5
	moderat	f_3_1	f_3_2	f_3_3	f_3_4	f_3_5
	leicht	f_4_1	f_4_2	f_4_3	f_4_4	f_4_5
$f_i(a) = f_i(b)$	indifferent	f_5_1	f_5_2	f_5_3	f_5_4	f_5_5
$f_i(a) < f_i(b)$	leicht	f_6_1	f_6_2	f_6_3	f_6_4	f_6_5
	moderat	f_7_1	f_7_2	f_7_3	f_7_4	f_7_5
	stark	f_8_1	f_8_2	f_8_3	f_8_4	f_8_5
	sehr stark	f_9_1	f_9_2	f_9_3	f_9_4	f_9_5

Um eine Gesamtbewertung zwischen zwei Alternativen vornehmen zu können, wird daraufhin eine eindimensionale Variable, die „combined preference with weights" Variable (CPW), erstellt (De Keyser & Peeters, 1994, S. 271). Die CPW-Variable wird jeweils für den Fall $f_i(a) > f_i(b)$ und $f_i(a) < f_i(b)$ erstellt. Sie ordnet alle Zellen des entsprechenden Abschnitts der Präferenz-Wichtigkeits-Tabelle in absteigender Reihenfolge gemäß ihrem Einfluss auf das folgende Outranking ein. Die Klassen repräsentieren gleichwertige Kombinationen aus Präferenzstufe und Prioritätsklasse. So wird im abgebildeten Beispiel davon ausgegangen, dass die wichtigste Kombination aus einer sehr starken Präferenz und einem extrem wichtigen Kriterium (f_{15} für $f_i(a) > f_i(b)$ und f_{95} für $f_i(a) < f_i(b)$) besteht. Die Kombination aus sehr starker Präferenz und sehr wichtigem Kriterium wird dagegen als gleichwertig mit der Kombination aus starker Präferenz und extrem wichtigem Kriterium betrachtet. Tabelle 6 zeigt diese Einordnung gemäß dem bereits zuvor abgebildeten Beispiel von De Keyser und Peeters (1994, S. 268–277). In diesem Beispiel ergeben sich acht Gewichtsstufen der CPW-Variable. Die Einstufung, welches Gewicht einem Paar aus Präferenz und Priorität zugeordnet wird, kann durch den Entscheidungsträger jedoch letztendlich basierend auf seiner Einschätzung angepasst werden (Martel & Matarazzo, 2016, S. 235)

Tabelle 6: Beispielhafte Darstellung zweier CPW-Variablen nach De Keyser und Peeters (1994, S. 271)

	$CPW - (f_i(a) > f_i(b))$	$CPW - (f_i(a) < f_i(b))$
1	g1 = f_1_5	h1 = f_9_5

2	g2 = f_1_4 + f_2_5	h2 = f_8_5 + f_9_4
3	g3 = f_1_3 + f_2_4 + f_3_5	h3 = f_7_5 + f_8_4 + f_9_3
4	g4 = f_1_2 + f_2_3 + f_3_4 + f_4_5	h4 = f_6_5 + f_7_4 + f_8_3 + f_9_2
5	g5 = f_1_1 + f_2_2 + f_3_3 f_4_4	h5 = f_6_4 + f_7_3 + f_8_2 + f_9_1
6	g6 = f_2_1 + f_3_2 + f_4_3	h6 = f_6_3 + f_7_2 + f_8_1
7	g7 = f_3_1 + f_4_2	h7 = f_6_2 + f_7_1
8	g8 = f_4_1	h8 = f_6_1

Tabelle 7 zeigt dem Beispiel folgend die entstehenden CPW-Variablen in ausgeschriebener Form. Diese Form in natürlicher Sprache kann von dem Entscheidungsträger im nächsten Schritt genutzt werden, um den paarweisen Vergleich zwischen den Alternativen durchzuführen.

Tabelle 7: Beispielhafte Darstellung der kombinierten Gesamtbewertung von Präferenz und Priorität absteigend sortiert nach Gewicht in Anlehnung an De Keyser und Peeters (1994, S. 270)

1	(sehr stark, extrem wichtig)
2	(sehr stark, sehr wichtig), (stark, extrem wichtig)
3	(sehr stark, moderat wichtig), (stark, sehr wichtig), (moderat, extrem wichtig)
4	(sehr stark, wenig wichtig), (stark, moderat wichtig), (moderat, sehr wichtig), (leicht, extrem wichtig)
5	(sehr stark, nicht wichtig), (stark, wenig wichtig), (moderat, moderat wichtig), (leicht, sehr wichtig)
6	(stark, nicht wichtig), (moderat, wenig wichtig), (leicht, moderat wichtig)
7	(moderat, nicht wichtig), (leicht, wenig wichtig)
8	(leicht, nicht wichtig)

e. Outranking der Alternativen

Im letzten Schritt wird nun durch Outranking ermittelt, welche Alternative(n) vorzuziehen ist (sind).

Zunächst können Alternativen, die von anderen Alternativen dominiert werden, aus dem Entscheidungsprozess ausgeschlossen werden. Eine Alternative wird als dominiert betrachtet, wenn es mindestens eine Alternative gibt, die in allen Kriterien gleich gut oder besser und in mindestens einem Kriterium besser ist.

Anschließend werden alle verbleibenden Alternativen paarweise miteinander verglichen. Dafür werden alle Gewichtsstufen der beiden CPW-Variablen (für den Fall $f_i(a) > f_i(b)$ und $f_i(a) < f_i(b)$) mit dem Wert null belegt. Folgend wird für jedes Kriterium mittels der in Schritt b erstellten Präferenzmatrix überprüft, ob $f_i(a) > f_i(b)$ oder $f_i(a) < f_i(b)$ vorliegt. Ist dies der Fall, wird der Wert in der entsprechenden Gewichtsstufe der entsprechenden CPW-Variable um eins inkrementiert. Tabelle 8 zeigt ein Beispiel für das Resultat eines solchen Vergleichs.

Tabelle 8: Beispielhafter paarweiser Vergleich zwischen zwei Alternativen mit dem Ergebnis der Indifferenz nach De Keyser und Peeters (1994, S. 276)

	$CPW - (f_i(Standort\ 1) > f_i(Standort\ 6))$	$CPW - (f_i(Standort\ 1) < f_i(Standort\ 6)$
1	0	0

2	0	1
3	1	0
4	1	0
5	0	1
6	0	0
7	0	0
8	0	0

Auf Grundlage der so befüllten CPW-Variablen, kann nun ermittelt werden, welche Beziehung zwischen den beiden Alternativen vorliegt. Mögliche Beziehungen sind:

- **S** (Überlegenheit): Alternative a ist Alternative b vorzuziehen (aSb)
- **I** (Indifferenz): Alternative a und b sind indifferent, also gleichwertig (aIb)
- **R** (Unvergleichbar): Die Alternativen sind nicht miteinander vergleichbar (aRb)

Die Beziehung zwischen zwei CPW-Variablen (hier g für die Variable von Alternative a und h für die Variable von Alternative b) wird mittels folgender Regeln ermittelt. Dabei gilt $l = Anzahl\ der\ Gewichtsklassen\ der\ CPW$.

$$\text{Wenn } \forall i: \sum_{j=1}^{i} g_j = \sum_{j=1}^{i} h_j \quad i = 1, \dots, l \quad \text{dann:} \quad a\,I\,b;$$

$$\text{Wenn } \forall i: \sum_{j=1}^{i} g_j \geq \sum_{j=1}^{i} h_j \quad i = 1, \dots, l \quad \text{dann:} \quad a\,S\,b;$$

$$\text{Wenn } \forall i: \sum_{j=1}^{i} g_j \leq \sum_{j=1}^{i} h_j \quad i = 1, \dots, l \quad \text{dann:} \quad b\,S\,a;$$

$$\text{In allen anderen Fällen: } a\,R\,b$$

Nachdem die Beziehungen zwischen allen Alternativen ermittelt wurden, können sie in tabellarischer Form festgehalten werden und in Form eines Graphen visualisiert werden. Tabelle 9 und Abbildung 7 zeigen dafür Beispiele. Gerichtete Kanten bedeuten die Dominanz einer Alternative über die andere, während ungerichtete Kanten die Indifferenz der Alternativen beschreiben.

Tabelle 9: Beispielhaftes Gesamtergebnis der paarweisen Vergleiche nach De Keyser und Peeters (1994, S. 277)

Unvergleichbar	Standort 1 R Standort 6; Standort 5 R Standort 6
Indifferenz	Standort 4 I Standort 5
Überlegenheit	Standort 2 S Standort 1; Standort 2 S Standort 4; Standort 2 S Standort 5; Standort 2 S Standort 6; Standort 4 S Standort 1; Standort 5 S Standort 1; Standort 6 S Standort 4

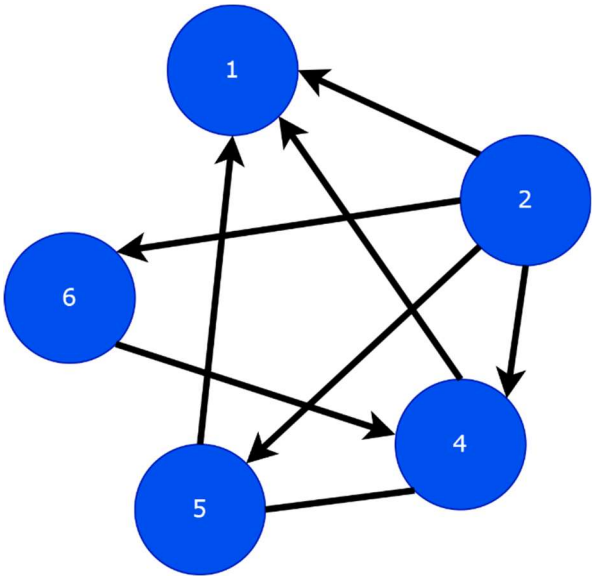

Abbildung 7: Beispielhafte Darstellung der Vergleichsergebnisse in Form eines Graphen nach De Keyser und Peeters (1994, S. 277)

Schließlich wird der Kern oder die Kerne des Graphen bestimmt. Diese(r) besteht / bestehen aus einer oder mehreren Alternativen, die von keiner Alternative dominiert werden und beschreibt die auszuwählende(n) Alternative(n). So besteht im gezeigten Beispiel auf Abbildung 7 der Kern des Graphen aus Standort 2. Auf diesem Beispiel ist die Alternative drei nicht abgebildet, da sie von anderen Alternativen dominiert wird und daher zu Beginn dieses Schrittes ausgeschlossen wurde.

3.5 Identifizierung möglicher Design Patterns

Um mögliche Design Patterns für die Versionierung der VCC-Kataloge zu identifizieren, wurden bewusst nicht nur Fachliteratur herangezogen. Neben Fachbüchern wurden auch wissenschaftliche Artikel, Konferenzbeiträge, Online-Foren wie StackOverflow sowie Blogs und Fachartikel von Branchenexperten für die Recherche herangezogen.

Jede der herangezogenen Quellen wurde auf ihre Relevanz und ihren Beitrag zur Lösung der in Kapitel 3.3 spezifizierten Kriterien hin bewertet. Besonders bei nicht-wissenschaftlichen Quellen wie Blogs einzelner Entwickler wurde dabei darauf geachtet, die Informationen mit anderen Quellen abzugleichen. Für die weitere Ausarbeitung wurde lediglich der Blog von Martin Fowler herangezogen, welcher bereits Fachliteratur zum Thema Design Patterns veröffentlicht hat (Fowler, 1996, 2003). Auf seinem Blog veröffentlicht Fowler speziell Beiträge zum Thema Design Patterns mit zeitlichem Bezug.

Die identifizierten Design Patterns wurden in das in Kapitel 3.2 festgelegte Schema überführt und werden im Folgenden beschrieben. Eine ausführlichere Diskussion ihrer Merkmale findet in Kapitel 3.6.1 statt, wenn die Design Patterns hinsichtlich der Auswahlkriterien bewertet werden.

3.5.1 Alternative 1: Audit Log

Klassifikation: Verhaltensbasiert

Auch bekannt als: Audit Trail, Transaction Log, Changelog, Change History Record, Tracking Log

Struktur:

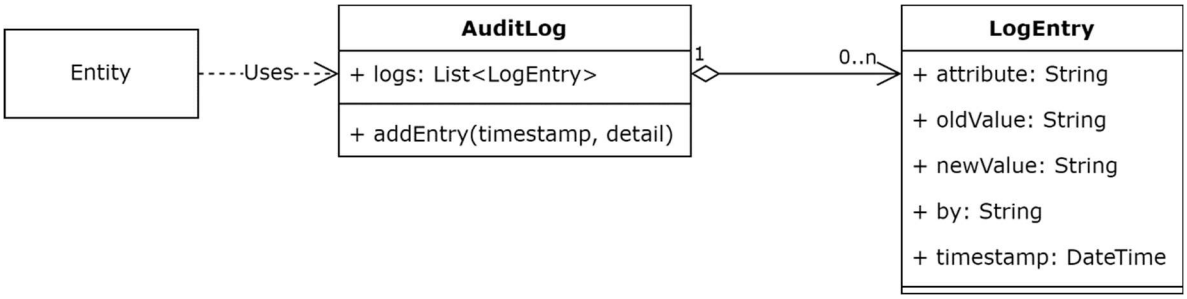

Abbildung 8: Struktur des Audit Log Design Patterns

Funktionsweise:

Wie auf Abbildung 8 zu sehen ist, besteht das Audit Log Pattern im Wesentlichen aus drei Komponenten. Einer Entität, deren Änderungen nachverfolgt werden sollen und einem Log, welcher wiederum aus Logeinträgen besteht. Der Logeintrag enthält Informationen zu der vorgenommenen Veränderung, wie beispielsweise durch wen, wann, welches Attribut verändert wurde (Rubis & Cardei, 2015, S. 7).

Wird an der Entität eine Änderung vorgenommen, z.B. bei einem Kunden der Name verändert, wird dem Log ein Eintrag hinzugefügt, aus dem hervorgeht, welche Änderung wann von wem durchgeführt wurde (Fowler, 2004a).

Verwandte Pattern: Change History, Business Data Object Versioning

Die Idee des Audit Logs stellt eine der einfachsten Formen der Historisierung dar. Es bietet eine chronologische Aufzeichnung von Änderungen und Aktivitäten, die zur späteren Analyse und Überprüfung verwendet werden können (Fowler, 2004a). Die Einfachheit des Patterns führt allerdings zu Problemen beim Lesen der Logs. Während einzelne Aktivitäten für menschliche Leser noch verständlich sind, bietet das Pattern keine konkrete Lösung zur Wiederherstellung von Zuständen zu bestimmten Zeitpunkten. Dies kann lediglich durch zusätzliche Software erreicht werden (Fowler, 2004a, 2004b).

3.5.2 Alternative 2: Temporal Property

Klassifikation: Verhaltensbasiert

Auch bekannt als: Historic Mapping, History on Association

Struktur:

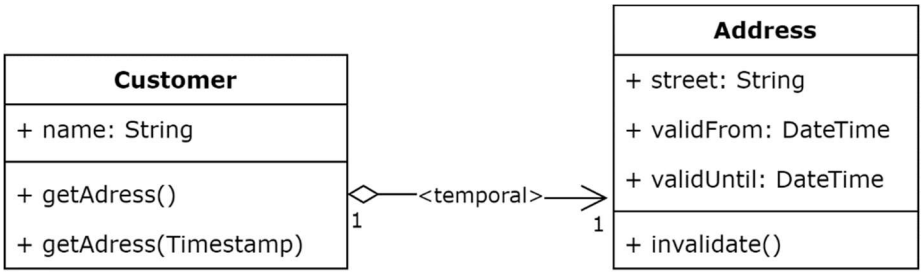

Abbildung 9: Beispiel für die Struktur eines Temporal Property in Anlehnung an Fowler (2004b)

Funktionsweise:

Das Temporal Property Design Pattern besteht im Kern aus einer Entität und einem Mechanismus zur Speicherung der zeitlichen Historie ausgewählter Attribute der Entität. Anstatt, dass Attribute direkt als Felder in der Entität gespeichert werden, werden sie in einer externen Struktur geführt, die es erlaubt, verschiedene Zustände über die Zeit abzurufen. Diese besteht neben dem eigentlichen Inhalt aus einem Zeitstempel, der darüber Auskunft gibt, ab wann sie gültig ist (Fowler, 1996, S. 303–305). Je nach Implementierung kann sie zusätzlich über einen „Gültig bis" Zeitstempel oder eine „Ist aktiv" Flagge verfügen.

Wird eine Änderung an der Entität vorgenommen wird eine neue Version des Attributs angelegt und das vorherige gegebenenfalls als inaktiv markiert. Da das Attribut nicht direkt als Feld der Entität vorliegt muss der Zugriff darauf, wie auf Abbildung 9 zu erkennen ist, über den Aufruf von Methoden erfolgen (Fowler, 2004b).

Verwandte Pattern: Temporal Object

Im Gegensatz zu simplen Audit-Logs bietet das Temporal Property Pattern eine strukturierte und abfragbare Historie. Es ist allerdings darauf ausgerichtet, lediglich bestimmte Attribute zu historisieren.

3.5.3 Alternative 3: Temporal Object

Klassifikation: Verhaltensbasiert, Objektbezogen

Auch bekannt als: Versioned Object, Business Data Object Versioning

Struktur:

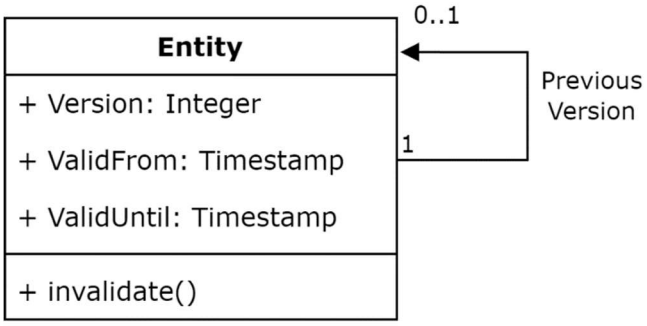

Abbildung 10: Struktur des Temporal Object Design Patterns

Funktionsweise:

Bei diesem Design Pattern wird die gesamte Entität versioniert. Das bedeutet, dass für jede Änderung an der Entität eine neue Version mit einem aktualisierten „Gültig ab" Attribut angelegt wird. Wenn auch für die eigentlich Funktionalität nicht zwingend erforderlich, kann die Entität, wie auf Abbildung 10 gezeigt, um einen Versionszähler und/oder einen Verweis auf die vorherige Version erweitert werden (Kimball & Ross, 2013, S. 54). Die auf Abbildung 10 angedeutete Verknüpfung mit der vorherigen Version ist dabei nicht unbedingt notwendig. Die Ermittlung einer vorherigen Version ist auch durch eine entsprechende Datenbankabfrage möglich (Snodgrass, 1999, S. 12–14).

Verwandte Pattern: Snapshot[6], Temporal Property

Beim Betrachten dieses Design Patterns fällt die Nähe zum vorherigen Design Pattern, dem Temporal Property, auf. Der Unterschied zwischen den beiden Pattern wird deutlicher bei der Betrachtung möglicher Implementierungen. Ein Temporal Property könnte als eine Liste mit Tupeln aus Zeitstempel und Änderung realisiert werden, wohingegen bei einem Temporal Object in der Praxis für jede Version ein eigenes Objekt benötigen wird.

3.5.4 Alternative 4: Event Sourcing

Klassifikation: Verhaltensbasiert, Objektbezogen

Struktur:

[6] Das Design Pattern Snapshot wird in diesem Abschnitt nicht gesondert behandelt, da es starke Ähnlichkeiten mit dem Design Pattern Temporal Object aufweist.

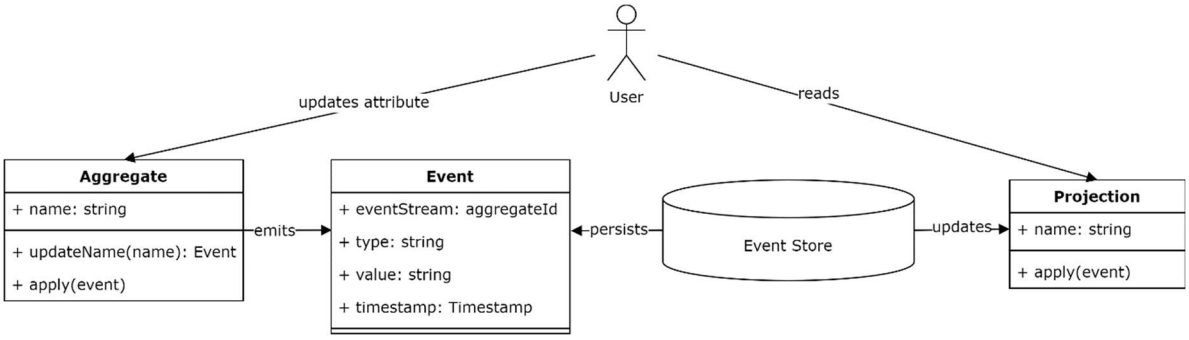

Abbildung 11: Logische Struktur des Event Sourcing Design Patterns

Funktionsweise:

Die Grundidee des Event-Sourcing Design Patterns ist es, den aktuellen Zustand einer Entität aus der Summe aller stattgefundenen Veränderungen (Events) zu konstruieren (Khononov, 2021, S. 101).

Wie auf Abbildung 11 zu sehen ist, besteht das Event Sourcing Design Pattern im Wesentlichen aus vier Elementen: Aggregaten (Aggregate), Projektionen (Projection), Events sowie einem Eventspeicher (Event Store).

Ein **Aggregat** repräsentiert eine im Event-Sourcing-System abgebildete Sammlung von einem oder mehreren Objekten der modellierten Domäne, die als Einheit behandelt werden (Khononov, 2021, S. 96). Im Kontext dieser Arbeit könnte ein Aggregat zum Beispiel eine Wirkung, ein Wirkungskatalog oder die Summe aller VCC-Kataloge sein. Wie der Umfang eines Aggregats definiert wird, liegt obliegt dem Entwickler.

Eine Änderung an einem Aggregat löst ein **Event** aus. Ein Event beschreibt ein Ereignis, das in der abgebildeten Domäne zu einem bestimmten Zeitpunkt stattgefunden hat, beispielsweise das Erstellen einer Wirkung (Betts et al., 2013, S. 113). Events sind unveränderlich, was bedeutet, dass einmal gespeicherte Events nicht mehr verändert werden können (Betts et al., 2013, S. 235).

Events werden in einem **Eventspeicher** persistiert (Khononov, 2021, S. 107). Um den Zustand eines Aggregats zu einem beliebigen Zeitpunkt wiederherzustellen, können nun alle Events, die sich vor dem Zeitpunkt ereignet haben, aus dem Eventspeicher ausgelesen werden. Werden diese Events in der korrekten Reihenfolge auf ein neu instanziiertes Aggregat angewendet, befindet sich dieses im gewünschten Zustand (Betts et al., 2013, S. 113).

Während Aggregate dafür zuständig sind, unter Berücksichtigung der Domänenlogik Events zu generieren, dienen Projektionen der Lese-Seite der Applikation.[7] Projektionen

[7] Es muss an dieser Stelle erwähnt werden, dass das Konzept des Event Sourcing grundsätzlich auch ohne Projektionen angewendet werden kann. Im produktiven Einsatz scheint die Verwendung von Projektionen aber Standard zu sein, so dass diese hier als Teil von Event Sourcing beschrieben werden.

konsumieren die Events eines oder mehrerer Aggregate und wenden diese, ähnlich wie Aggregate, an, um ihren Zustand verändern (Overeem et al., 2021, S. 7–8). Um nicht bei jeder Abfrage einer Projektion alle Events aus dem Eventspeicher lesen und anwenden zu müssen, wird in der Regel die aktuellste Version einer Projektion als Read-Model gespeichert (Khononov, 2021, S. 130–131). Durch die Möglichkeit Events verschiedener Aggregate zu konsumieren, können Projektionen erstellt werden, die für bestimmte Darstellungen in der Anwendung spezialisiert sind. Dadurch entfällt die Berechnung dieser (möglicherweise rechenintensiven) Darstellungen zum Zeitpunkt der Abfrage (Overeem et al., 2021, S. 7–8).

Verwandte Pattern: CQRS, DDD

3.6 Anwendung der ARGUS-Methode zur Auswahl eines geeigneten Design Patterns

In diesem Abschnitt wird mittels der zuvor ausgewählten ARGUS-Methode das für die Versionierung geeignete Design Pattern ermittelt.

3.6.1 Datensammlung

Die Alternativen und die Kriterien anhand derer sie bewertet werden, wurden bereits in Kapitel 3.3 und Kapitel 3.5 beschrieben. Alle Kriterien werden auf der auf Tabelle 10 abgebildeten Ordinalskala bewertet. Diese besteht aus fünf möglichen Werten, die von einer sehr niedrigen bis hin zu einer sehr hohen Bewertung reichen. Bis auf das Kriterium der Komplexität ist für alle Kriterien ein hoher Wert vorzuziehen.

Tabelle 10: Verwendete Ordinalskala für die Bewertung der Kriterien der Alternativen

sehr hoch
hoch
moderat
niedrig
sehr niedrig

Im Folgenden wird für jedes Kriterium jeder Alternative eine Einstufung auf dieser Skala vorgenommen. Diese erfolgt auf Grundlage der in Kapitel 3.5 erfolgten Ausführungen zu der Funktionsweise der Design Patterns.

Audit Log

Die Wiederherstellbarkeit vergangener Zustände wird beim Audit Log als niedrig bewertet, da es primär die Änderungen selbst protokolliert. Der Kontext der Änderung oder der Zustand des Objekts zum Zeitpunkt der Änderung wird nicht gespeichert. Eine Wiederherstellbarkeit kann nur über zusätzliche, komplexe Programmierung erreicht werden und ist in dem Pattern an sich nicht vorgesehen (Snodgrass, 1999, S. 218).

In Bezug auf die Abbildung des Änderungsverlaufs bietet das Audit Log eine hohe Leistungsfähigkeit, jede Änderung kann aufgezeichnet werden, was eine detaillierte Ansicht der Historie von Datenänderungen ermöglicht.

Die Identifizierbarkeit einzelner Versionen wird hingegen als niedrig eingestuft, da das Design Pattern nur Änderungen speichert und über keine Logik zur Versionierung verfügt.

Die Wartbarkeit dagegen wird als hoch bewertet, was auf die Einfachheit des Patterns zurückzuführen ist. Audit Logs können leicht zu einer bestehenden Anwendung hinzugefügt werden. Beispielsweise indem ein Datenbanktrigger genutzt wird, um einen Log Eintrag automatisch anzulegen (Snodgrass, 1999, S. 218).

Die Skalierbarkeit des Audit Log Design Pattern wird als moderat bewertet. Mit einer steigenden Menge an Änderungen wächst auch die die Menge der zu speichernden Informationen. Diese enthalten allerdings wenige redundante Informationen (vgl. Skalierbarkeit des Patterns Temporal Object).

Dadurch, dass Audit Logs in einer einzigen Tabelle gespeichert werden können, ist die Leistungsfähigkeit von Abfragen als hoch zu bewerten. Dies gilt auch für das Hinzufügen von Log Einträgen, da dies lediglich das Hinzufügen einer Reihe in der Tabelle bedeutet. Probleme bei der Leistungsfähigkeit sind allerdings zu befürchten, sollen die Einträge eines Audit Logs weiterverarbeitet werden, um vergangene Versionen wiederherzustellen. Da dies in dem Pattern aber nicht vorgesehen ist, fließt dieser Umstand nicht in die Bewertung ein.

Auf Grund seiner Einfachheit und hohen Wartbarkeit kann auch die Komplexität des Patterns als niedrig bewertet werden. Daraus resultieren allerdings auch die erwähnten eingeschränkten Möglichkeiten zur Weiterverarbeitung der gespeicherten Informationen.

Zuletzt wird die Eigenschaft der Parallelität als moderat bewertet. Durch die Speicherung in einer Tabelle werden grundsätzlich parallele Operationen unterstützt. Allerdings existieren keine Mechanismen zur Konfliktvermeidung, wenn mehrere Nutzer Änderungen bezüglich desselben Attributs erstellen. Des Weiteren können Inkonsistenzen entstehen, wenn zum Beispiel das Schreiben des Objektes fehlschlägt, das Schreiben des Logs aber erfolgt (Khononov, 2021, S. 114).

Temporal Property

Das Pattern Temporal Property bietet eine hohe Wiederherstellbarkeit vergangener Zustände. Es erlaubt es den Wert eines Attributes zu einem beliebigen Zeitpunkt zu re-

konstruieren. Die Wiederherstellung eines Objektes mit mehreren versionierten Attributen gestaltet sich jedoch aufwendiger, da jedes Attribut einzeln rekonstruiert werden muss.

Durch diese Versionierung auf Attributebene kann auch eine, als hoch zu bewertende Abbildung des Änderungsverlaufs erreicht werden. Um eine Änderung abzubilden, muss dafür lediglich der Wert des Attributes vor und nach Änderung abgerufen werden und kann dann verglichen werden.

Die Identifizierbarkeit einzelner Versionen wird als moderat eingestuft. Zwar ermöglicht das Design Pattern, Zustände zu bestimmten Zeitpunkten zu identifizieren, jedoch konzentriert sich Temporal Property auf einzelne Eigenschaften und nicht auf das Gesamtobjekt. Dies kann die Zusammenführung verschiedener Eigenschaftsstände zu einem vollständigen Versionsbild erschweren. Dieses könnte durch eine Art Versionszähler in dem Gesamtobjekt realisiert werden.

Sollen im Nachhinein Änderungen an den Attributen eines Objekts vorgenommen werden, ist dies ohne größeren zusätzlich Aufwand umsetzbar. Es muss allerdings berücksichtigt werden, dass der Zugriff auf neue versionierte Attribute über Methoden erfolgen muss und entsprechende Tabellen (bei Nutzung einer relationalen Datenbank) anzulegen sind. Daher wird auch die Wartbarkeit des Temporal Properties als moderat bewertet.

Das Gleiche gilt für die Komplexität. Im Gegensatz zum Audit Log muss der Zugriff auf versionierte Attribute eines Objektes angepasst werden, da dieser über eine Methode erfolgt. Darüber hinaus sind aber keine weiteren speziellen Verwaltungsmechanismen für die Objektstruktur erforderlich. Daher wird auch die Komplexität des Temporal Property Design Patterns als moderat eingestuft.

Im Vergleich zum Design Pattern Audit Log wird die Leistungsfähigkeit nur als moderat eingestuft. Das ist darin begründet, dass bei der Initialisierung eines Objektes jedes versionierte Attribut separat über einen Methodenaufruf geladen werden muss. Bei einer Speicherung in einer relationalen Datenbank würde dies zu einer erhöhten Anzahl von Datenbankabfragen führen, was sich nachteilig auf die Leistungsfähigkeit der Anwendung und die Auslastung des Systems auswirken kann.

Bei den Kriterien der Skalierbarkeit und Parallelität greifen ähnliche Überlegungen wie bei dem Audit Log Pattern, daher werden diese ebenfalls mit moderat bewertet.

Temporal Object

Das Temporal Object Pattern erlaubt eine als sehr hoch zu bewertende Wiederherstellbarkeit vergangener Zustände. Objekte können in jeden vorherigen Zustand zurückversetzt werden.

Die Abbildbarkeit des Änderungsverlaufes wird dagegen nur als moderat bewertet. Da bei jeder neuen Version das gesamte Objekt abgespeichert wird, sind Änderungen von einer Version zur nächsten nicht umgehend ersichtlich. Stattdessen müsste ein programmatischer Vergleich der zwei Versionen stattfinden, um die Unterschiede zu identifizieren. Besonders bei komplexeren Datentypen ist dies nicht trivial und benötigt zusätzlichen Programmiercode.

Durch die eindeutige Abspeicherung des gesamten Objektes als eine bestimmte Version ist dafür eine hohe Identifizierbarkeit einzelner Versionen gegeben.

Auch die Wartbarkeit des Temporal Object wird als hoch bewertet. Dadurch, dass eine Version eines Objektes als Ganzes gespeichert wird, kann die speichernde Tabelle (bei Nutzung einer relationalen Datenbank) mit den herkömmlichen Methoden angepasst werden, ohne dass spezifische Anforderungen der Versionierung zu berücksichtigen sind.

Als niedrig ist dagegen das Kriterium der Skalierbarkeit zu bewerten. Durch die Speicherung vollständiger Objektversionen bei jeder Zustandsveränderung resultiert eine erhöhte Speichernutzung. Obwohl dieser Aspekt in kleineren Systemen vernachlässigbar erscheint, stellt er bei der Skalierung des Systems einen Nachteil dar.

Die Speicherung vollständiger Objektversionen führt auch zu einer moderaten Bewertung für das Kriterium der Leistungsfähigkeit. Die Logik zum Speichern und Laden von Objekten funktioniert ähnlich wie in herkömmlichen Systemen. Lediglich beim Wiederherstellen eines Objektes zu einem bestimmten Zeitpunkt muss die Gültigkeit der vorhandenen Versionen berücksichtigt werden, was sich negativ auf die Leistungsfähigkeit des Systems auswirken kann.

Die Komplexität des Patterns ist als hoch einzustufen, insbesondere hinsichtlich des Zugriffs auf gespeicherte Objektversionen. In einem relationalen Datenbanksystem führt die Art der Objektspeicherung dazu, dass mehrere Instanzen desselben Objekts unter derselben ID gespeichert werden, jedoch mit unterschiedlichen Versionsnummern oder Gültigkeitszeitstempeln. Das erfordert, dass bei jeder Datenabfrage spezifische Selektionen anhand der Versionsnummer oder durch den Abgleich der Gültigkeitsangaben erfolgen müssen. Dieser Vorgang erhöht nicht nur die Komplexität der Datenabfragen, sondern auch die der Datenverwaltung insgesamt. Die Komplexität wird weiter gesteigert, wenn Objekte, die miteinander in Beziehung stehen, betroffen sind. Bei der Abfrage von Daten, die zu einem Objekt gehören, muss ebenfalls für jedes damit verknüpfte Objekt eine selektive Abfrage basierend auf der korrespondierenden Versionsnummer oder dem Gültigkeitszeitraum durchgeführt werden. Dies erweitert den Umfang der notwendigen Datenoperationen erheblich und kann die Leistungsfähigkeit des Systems beeinträchtigen. Das gilt besonders, wenn solche Abfragen häufig oder in einem

skalierenden System durchgeführt werden, in welchem die Datenmenge und die Anzahl der Beziehungen exponentiell ansteigen können.

Die Speicherung ganzer Objektversionen führt auch bei der Eignung zur Parallelität zu einer niedrigen Bewertung. Wird nur ein Attribut eines Objektes verändert, betrifft die resultierende Datenbanktransaktion dennoch das gesamte Objekt, was das Risiko von Konflikten bei paralleler Bearbeitung erhöht. Darüber hinaus muss sichergestellt werden, dass keine inkonsistenten Zustände entstehen, wenn gleichzeitige Transaktionen ältere oder neuere Zustände desselben Objektes betreffen.

Event Sourcing

Wie das Temporal Objekt bietet auch das Event Sourcing Design Pattern eine sehr hohe Wiederherstellbarkeit vergangener Objektzustände. Darüber hinaus erlaubt es aber auch eine als sehr hoch zu bewertende Abbildung des Änderungsverlaufs. Durch die Abbildung jeder Änderung als ein einzelnes Event, ergänzt um einen Zeitstempel, können an einem Objekt vorgenommene Änderungen granular abgebildet werden.

Da für jedes Aggregat eine chronologische Eventsequenz vorliegt, kann für jeden vergangenen Zustand des Aggregats eine eindeutige Versionsnummer identifiziert werden. Daher wird auch dieses Kriterium als hoch bewertet.

Eine Schwäche des Event Sourcing ist die als niedrig bewertete Wartbarkeit. Werden einem Aggregat Attribute hinzugefügt, müssen auch die entsprechenden Events zu dessen Änderung implementiert werden. Des Weiteren müssen die Attribute in den Aggregaten und Projektionen hinzugefügt werden. Da das System in der Lage sein muss, sowohl die alte als auch die neue Version eines Events zu verarbeiten, stellt auch das nachträgliche Verändern implementierter Events eine Herausforderung dar. Dem kann durch Ansätze wie dem Versionieren von Events begegnet werden (Fowler, 2005). Darüber hinaus ist auch die erstmalige Implementierung eines Event Sourcing basierten Systems mit erheblichem Aufwand verbunden, da die Logik für das Anlegen von Aggregaten und das Verarbeiten von Events von der üblichen Logik in relationalen oder dokumentenbasierten Datenbanken abweicht.

Eine Stärke von Event Sourcing hingegen ist die hohe Skalierbarkeit. Die Nutzung unveränderlicher Events und von Projektionen als spezialisierte Read-Models erlaubt eine individuelle Skalierung der für das Schreiben und Lesen zuständigen Komponenten (Betts et al., 2013, S. 244–245).

Die Leistungsfähigkeit wird insgesamt als moderat bewertet. Änderungen an Aggregaten erfordern ein Abrufen des Aggregats (was das Laden der Events des Aggregats bedingt), das Verarbeiten und Abspeichern des Events sowie die Speicherung dadurch veränderter

Projektionen. Auch das Abrufen älterer Zustände erfordert das erneute Abspielen vergangener Events. Diesen negativen Aspekten kann aber durch Techniken wie dem Anlegen von Snapshots entgegen gewirkt werden (Betts et al., 2013, S. 245–246).

Wie bereits thematisiert, erfordert die Implementierung des Event Sourcing Patterns spezielle Logiken zur Verwaltung und Verarbeitung von Events. Die Komplexität des Patterns übersteigt die der bisher diskutierten Patterns deutlich und wird daher als sehr hoch bewertet.

Zuletzt wird festgestellt, dass Event Sourcing in hohem Maße dem Kriterium der Parallelität entspricht. Durch die Nutzung voneinander unabhängiger Events, können diese parallel und unabhängig voneinander verarbeitet werden. Durch die Wiederherstellbarkeit aller vergangenen Zustände können inkonsistente Zustände zurückgesetzt werden. Sollte es zu konfliktären Events kommen, ist es auch denkbar den Nutzer entscheiden zu lassen, welches Event übernommen werden soll (Betts et al., 2013, S. 111–112). Dabei muss aber erwähnt werden, dass die Umsetzung dieser konsistenzsichernden Maßnahmen dem System zusätzliche Komplexität hinzufügen, so dass der Aspekt der Parallelität durchaus auch kritisch gesehen werden kann. Kritik wird hier besonders in Zusammenhang mit dem Command and Query Responsibility Segregation (CQRS) Design Pattern genannt, welches häufig in Verbindung mit Event Sourcing verwendet wird (Overeem et al., 2021, S. 9–10; Pacheco, 2018, S. 113).

Tabelle 11 fasst die Bewertung der Alternativen zusammen.

Tabelle 11: Zusammenfassung der bewerteten Kriterien der Alternativen

C		1 - Audit Log	2 - Temporal Property	3 - Temporal Object	4 - Event Sourcing
1	Wiederherstellbarkeit vergangener Zustände	niedrig	hoch	sehr hoch	sehr hoch
2	Abbildung des Änderungsverlaufs	hoch	hoch	moderat	sehr hoch
3	Identifizierbarkeit einzelner Versionen	niedrig	moderat	hoch	hoch
4	Wartbarkeit	hoch	moderat	hoch	niedrig
5	Skalierbarkeit	moderat	moderat	niedrig	hoch
6	Leistungsfähigkeit	hoch	moderat	hoch	moderat
7	Komplexität	niedrig	moderat	hoch	sehr hoch
8	Parallelität	moderat	moderat	niedrig	hoch

3.6.2 Präferenzmodellierung

Tabelle 12 zeigt die Skala der Präferenzstufen. Diese übernimmt die von De Keyser und Peeters (1994, S. 270) verwendeten fünf Stufen, die von Indifferenz bis hin zu sehr starker Präferenz reichen.

Tabelle 12: Festgelegte Präferenzskala

indifferent

leichte Präferenz
moderate Präferenz
starke Präferenz
sehr starke Präferenz

Für alle Kriterien außer dem der Komplexität ist ein hoher Wert zu präferieren, so dass sich die auf Tabelle 13 und

Tabelle 14 gezeigten Präferenzmatrizen ergeben. Sie beschreiben, wie stark eine Alternative a unter gegebenen bewerteten Kriterien vor Alternative b präferiert wird. Die leeren Zellen der Matrix bedeuten eine Präferenz von b vor a. Diskordanz wird nicht festgelegt.

Tabelle 13: Präferenzmatrix für Kriterien bei denen ein hoher Wert zu präferieren ist

$f_i(a)$ \ $f_i(b)$	sehr niedrig	niedrig	moderat	hoch	sehr hoch
sehr niedrig	indifferent				
niedrig	leicht	indifferent			
moderat	moderat	leicht	indifferent		
hoch	stark	moderat	leicht	indifferent	
sehr hoch	sehr stark	stark	moderat	leicht	indifferent

Tabelle 14: Präferenzmatrix für Kriterien bei denen ein niedriger Wert zu präferieren ist

$f_i(a)$ \ $f_i(b)$	sehr hoch	hoch	moderat	niedrig	sehr niedrig
sehr hoch	indifferent				
hoch	leicht	indifferent			
moderat	moderat	leicht	indifferent		
niedrig	stark	moderat	leicht	indifferent	
sehr niedrig	sehr stark	stark	moderat	leicht	indifferent

3.6.3 Gewichtung der Kriterien

Nun müssen die verschiedenen Kriterien gewichtet werden. Als Prioritätsklassen werden die von De Keyser und Peeters (1994, S. 266) verwendeten fünf Klassen von „nicht wichtig" bis hin zu „extrem wichtig" übernommen. Tabelle 15 zeigt die vorgenommene Einteilung der Kriterien in die Prioritätsklassen.

Tabelle 15: Einteilung der Auswahlkriterien in Prioritätsklassen

Gewichtung	Kriterium

nicht wichtig	
wenig wichtig	Skalierbarkeit, Leistungsfähigkeit
moderat wichtig	Wartbarkeit, Komplexität, Parallelität
sehr wichtig	Identifizierbarkeit einzelner Versionen
extrem wichtig	Wiederherstellbarkeit vergangener Zustände, Abbildung des Änderungsverlaufs

Als extrem wichtig werden die Wiederherstellbarkeit vergangener Zustände sowie die Abbildung des Änderungsverlaufs betrachtet. Dabei handelt es sich um die zu erfüllenden Anforderungen, die im Fokus dieser Arbeit stehen. Die Identifizierbarkeit einzelner Versionen wird für sehr wichtig gehalten, um die Versionierung in einer Benutzeroberfläche zu veranschaulichen. Dies erlaubt es dem Benutzer spezifische Versionen einer Entität auszuwählen und zu vergleichen.

Eine hohe Wartbarkeit, möglichst niedrige Komplexität und die Fähigkeit zur Parallelität werden als moderat wichtig festgelegt. Sie stellen keine Kernanforderungen an die Versionierung von Wirkungszusammenhängen dar, sind aber dennoch wichtig, um die Implementierung weiterentwickeln und nutzen zu können.

Da die Konzeptionierung und Implementierung der Versionierung in einem wissenschaftlichen Rahmen erfolgt und keine Nutzung durch eine große Menge an Nutzern in der näheren Zukunft abzusehen ist, liegt der Fokus stärker auf einer optimalen Abbildung der Versionierung als auf der Skalierbarkeit und Leistungsfähigkeit. Demzufolge werden diese Kriterien als weniger wichtig eingestuft.

3.6.4 Kombinierte Präferenzstruktur

Aus den aufgestellten Präferenzstufen und Prioritätsklassen wird, wie in Kapitel 3.4.2, erläutert eine kombinierte Präferenzstruktur erstellt. Da die verwendeten Stufen und Klassen identisch mit den von De Keyser und Peeters verwendeten Skalen sind, ist auch das entstehende Ranking von Prioritäts- und Präferenzpaaren in acht Klassen identisch (vgl. Tabelle 7 in Kapitel 3.4.2).

3.6.5 Outranking der Alternativen

Anhand der bewerteten Kriterien wird nun für jedes Paar aus Alternativen ein paarweiser Vergleich durchgeführt. Der in Kapitel 3.4.2 beschriebene Algorithmus wird angewendet, um für jedes Paar aus Alternativen festzustellen, ob ein Design Pattern dem Anderen überlegen ist, ob eine Indifferenz vorliegt, oder ob die Design Patterns nicht vergleichbar sind. Dafür wird ein Python-Skript verwendet, dieses findet sich in Anhang C. Tabelle 16 zeigt das Ergebnis der durchgeführten Vergleiche. Darauf ist zu erkennen, dass zwischen Design Patterns, die der Logik klassischer relationaler Datenbanken folgen, keine eindeutige Überlegenheit festgestellt werden kann. Das Event Sourcing Pattern hingegen überliegt allen anderen Patterns.

Tabelle 16: Ergebnis des paarweisen Vergleichs zwischen den betrachteten Design Patterns

Nicht vergleichbar	Audit Log I Temporal Property; Audit Log I Temporal Object; Temporal Property | Temporal Object
Indifferent	
Überlegenheit	Event Sourcing S Audit Log; Event Sourcing S Temporal Property; Event Sourcing S Temporal Object

Abbildung 12 zeigt diese Ergebnisse in Form eines Graphen. Darauf ist zu erkennen, dass das Event Sourcing Design Pattern mit drei ausgehenden Kanten den Kern des Graphen darstellt.

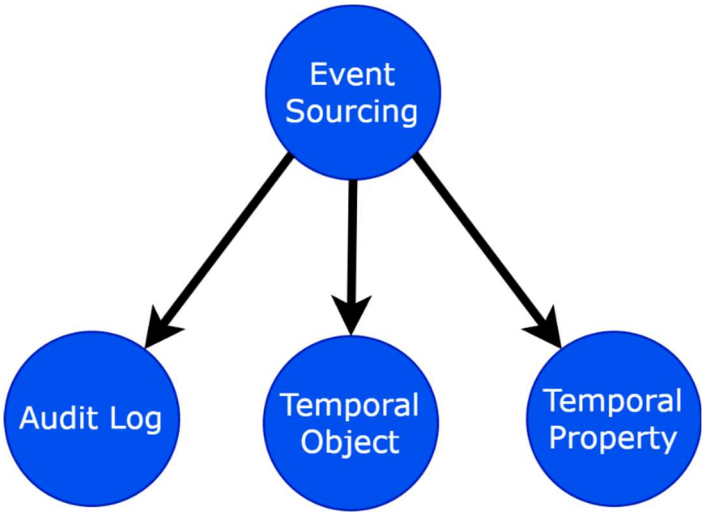

Abbildung 12: Darstellung der Vergleichsergebnisse in Form eines Graphen

Unter Betrachtung der definierten Kriterien und verfügbaren Alternativen zeigt sich, dass das nach der ARGUS-Methode das Design Pattern Event Sourcing am besten geeignet ist, um die Versionierung der VCC-Kataloge zu implementieren.

4 Implementierung versionierter Wirkungskataloge

Nachdem ein geeignetes Design Pattern für die Versionierung der VCC-Kataloge selektiert wurde, wird in diesem Kapitel die Implementierung des Patterns in einer Anwendung gezeigt. Wie erläutert, soll die Anwendung mit dem VCC-Tool integriert werden.

Dafür werden zunächst grundlegende Anforderungen an die Anwendung festgelegt, die neben der Implementierung des Event Sourcing Design Patterns zu berücksichtigen sind. Daraufhin werden Architekturentscheidungen, wie die Wahl der technischen Plattform, diskutiert. Anschließend wird die implementierte Architektur näher erläutert und anschließend in einem weiteren Abschnitt explizit die Einbindung des Event Sourcing Design Patterns in die Architektur aufgezeigt. Zuletzt wird die Integration mit dem VCC-Tool demonstriert.

4.1 Anforderungen an die zu implementierende Anwendung

Das ausgewählte Event Sourcing Design Pattern soll in einer Anwendung implementiert werden, die das Bearbeiten und Auswerten der VCC-Kataloge ermöglicht. In diesem Abschnitt werden die spezifischen Anforderungen an die Anwendung definiert. Die Anforderungen werden dabei, wie in der Softwareentwicklung üblich, in funktionale und nicht-funktionale Anforderungen unterteilt (Wiegers & Beatty, 2013, S. 6–10).

Funktionale Anforderungen:

- **Bearbeiten von Katalogen:** Die Anwendung soll es ermöglichen, in den VCC-Katalogen neue Entitäten anzulegen und bestehende Entitäten zu verändern.
- **Anzeigen der aktuellen Kataloge:** Die Anwendung soll den aktuellen Zustand der VCC-Kataloge ausgeben können.
- **Versionierung der Kataloge:** Wie zuvor ausgeführt, ist eine der Kernanforderungen an die Anwendung das Anlegen einer neuen Version für jede vorgenommene Änderung.
- **Anzeigen bestimmter Versionen:** Die Anwendung soll in der Lage sein, die VCC-Kataloge und ihre Entitäten in einer vom Nutzer gewählten Version anzuzeigen.
- **Nachverfolgbarkeit der Änderungen:** Die Anwendung soll in der Lage dazu sein, Änderungen die zu einer neuen Version geführt haben, granular abzubilden und darüber Auskunft zu geben, welcher Nutzer die Änderung durchgeführt hat.
- **Kompatibilität mit bestehender Software:** Die Anwendung soll mit der bestehenden Anwendung VCC-Tool kompatibel sein. Das bedeutet, dem VCC-Tool

muss es möglich sein, die VCC-Kataloge von der zu implementierenden Anwendung abzufragen, um sie entsprechend darzustellen.

Nicht-funktionale Anforderungen:

- **Leistungsfähigkeit:** Die Anwendung soll schnelle Antwortzeiten auch bei hohen Datenmengen und gleichzeitigen Nutzern gewährleisten.
- **Skalierbarkeit:** Es soll möglich sein, die Anwendung einfach zu skalieren, sowohl in Bezug auf die Datenmenge als auch die Anzahl der Benutzer.

4.2 Architekturentscheidungen

Die Wahl der Programmiersprache und des verwendeten Frameworks bildeten die technologische Basis der entwickelten Anwendung.

Ein möglicher Ansatz hätte darin bestanden, das Event Sourcing Pattern in das bestehende VCC-Tool zu integrieren. Gegen diese Möglichkeit wurde sich aber bewusst entschieden. Stattdessen wurde die Anwendung in dem .NET-Framework implementiert. Dabei handelt es sich um eine Plattform mit der sowohl Webanwendungen als auch rein auf HTTP basierende APIs erstellt werden können. Anwendungen in .NET werden in der Programmiersprache C# geschrieben, so dass diese Sprache für die Implementierung verwendet wurde. .NET bietet den Vorteil, dass es für den produktiven Einsatz in großen Systemen konzipiert ist, so dass dies Anforderung der Leistungsfähigkeit begünstigt. Die Entwicklung als API erlaubt außerdem eine Trennung zwischen der Logik der versionierten VCC-Kataloge und der Darstellungsebene in Gestalt des VCC-Tools. Dadurch ist die Implementierung der VCC-Kataloge nicht an eine spezifische Frontend-Technologie gebunden und kann in der Zukunft mit anderen Technologien zu unterschiedlichen Zwecken genutzt werden. Auch eine horizontale Skalierung der Anwendung wird dadurch ermöglicht, um der Anforderung der Skalierbarkeit zu entsprechen. Da die Anwendung in einem Docker-Container ausgeführt werden kann, können zur Skalierung dem ausführenden Cluster einfach weitere Anwendungscontainer hinzugefügt werden. Ein weiterer Vorteil des .NET-Frameworks ist die Möglichkeit, automatisiert eine dem OpenAPI Standard entsprechende Dokumentation der entwickelten API zu erzeugen. Dies erleichtert es zukünftigen Nutzern, die Anwendung zu nutzen.

Ein wesentlicher Aspekt des Architekturentwurfs war die Entscheidung zwischen der vollständigen Eigenentwicklung der Event-Sourcing-Logik und der Integration einer bestehenden Implementierung. Eine Eigenentwicklung erlaubt eine größere Kontrolle über die Merkmale des Systems und Anpassung an die vorliegende Domäne. Es muss aber auch berücksichtigt werden, dass Event Sourcing Systeme komplexe Logiken beinhalten, die Herausforderungen wie Parallelverarbeitung, Leistungsoptimierung und Daten-

integrität adressieren müssen. Um sich auf den Kern der Arbeit, die versionierte Abbildung der VCC-Kataloge, konzentrieren zu können, wurde sich daher für die Nutzung einer bestehenden Event Sourcing Implementierung entschieden. Dafür wurde die Marten-Bibliothek verwendet. Marten erlaubt die Verwendung einer PostgreSQL-Datenbank als dokumentenbasierte Datenbank und Event Store. Darüber hinaus stellt Marten eine Implementierung von Event Sourcing Konzepten wie Aggregaten und Projektionen bereit. Daraus ergab sich auch die zu verwendende Datenbank.

Event Sourcing als Design Pattern bildet den Kern der entwickelten Anwendung. Jedoch reicht es als Konzept nicht aus, um eine API vollständig zu entwickeln. Aus eingehenden Anfragen müssen die richtigen Events abgeleitet oder entsprechende Projektionen zurückgegeben werden. Des Weiteren müssen Anfragen auf ihre Validität und Konformität mit den Logiken des VCC-Vorgehensmodells hin überprüft werden. Daher wurden bei der Architekturkonzeptionierung zusätzlich die Konzepte Domain-driven Design (DDD) und CQRS angewendet.

CQRS ist ein Design Pattern, dessen Idee es ist, die Verantwortlichkeiten von Lese- und Schreibeoperationen innerhalb der Softwarearchitektur strikt voneinander zu trennen. Dies zielt darauf ab, Komplexität zu reduzieren und gleichzeitig die Skalierbarkeit der Anwendung zu verbessern (Oliveira Rocha, 2021, S. 160–162). CQRS wird häufig in Event Sourcing basierten Systemen verwendet, weil es die Grundprinzipien von Event Sourcing – die unveränderliche Aufzeichnung von Events und dedizierte Read Models – unterstützt und ergänzt (Oliveira Rocha, 2021, S. 160). Die Command-Seite nimmt Anweisungen (Commands) entgegen, validiert diese und erstellt die entsprechenden Events, während die Query-Seite Read-Models entsprechend einer Anfrage (Query) zurückgibt oder gegebenenfalls versionierte Zustände aus Events rekonstruiert (Betts et al., 2013, S. 244–245).

DDD ist ein Design Pattern, das darauf abzielt, die Komplexität von Software durch eine sorgfältige Modellierung der behandelten Domäne zu beherrschen (Millett & Tune, 2015, S. 6–9). Dieser Ansatz soll die Entwicklung einer gemeinsamen Sprache zwischen Entwicklern und Fachexperten fördern, was zu einer präziseren Verständigung führt (Khononov, 2021, S. 24–26). Auch im DDD werden Konzepte wie Aggregates und Events genutzt um die Domäne zu modellieren, was die gemeinsame Verwendung mit Event Sourcing begünstigt (Millett & Tune, 2015, S. 405–435).

Ein gängiges Architekturprinzip beim DDD ist die sogenannte Onion Architecture. Dabei werden die verschiedenen Architekturschichten wie in einer Zwiebel, von außen nach innen angeordnet. Die äußere Schicht interagiert mit der Umgebung wohingegen im Kern der Architektur die modellierte Domäne steht. Dazwischen liegen Schichten, die

die Anwendungslogik abbilden. Dieses Prinzip soll die klare Trennung von Zuständigkeiten (Separation of Concerns) unterstützen (Millett & Tune, 2015, S. 106). Kupplung findet in diesem Konzept immer von außen nach innen statt, das bedeutet zum Beispiel, dass die Schicht mit der Anwendungslogik das Domänenmodell kennt und darauf zugreift, das Domänenmodell aber unabhängig von der Anwendungslogik existiert (Millett & Tune, 2015, S. 107). Event Sourcing profitiert von der Zwiebelarchitektur durch die klare Abgrenzung zwischen der Domänenlogik, die die Events generiert, und den äußeren Schichten, die diese Events konsumieren und darauf reagieren. Die Trennung ermöglicht es, dass Änderungen in der Implementierung der Datenpersistenz oder in den Mechanismen zur Event-Verarbeitung die Logik des VCC nicht betreffen.

4.3 Architekturaufbau

Auf Grundlage der zuvor diskutierten Überlegungen wurde die Anwendungsarchitektur entworfen. Dabei wurde sich an dem Prinzip der Onion Architecture orientiert. In .NET besteht eine Anwendung (dort Lösung genannt) aus mehreren Projekten. Wie auf Abbildung 13 zu sehen ist, besteht die entworfene Anwendung aus vier Projekten, von denen drei als Schichten angeordnet sind. Es folgt eine Nennung der Rollen der jeweiligen Schichten. Eine ausführlichere Erläuterung der Komponenten der jeweiligen Schichten und ihrer Funktionsweise folgt im nächsten Kapitel. Es soll an dieser Stelle erwähnt werden, dass die folgenden Ausführungen sich auf die für das Verständnis der Anwendungslogik wesentlichen Komponenten beschränken. Details zu technischen Komponenten wie der Registrierung von Services oder Konfigurationen werden bewusst ausgespart. Der vollständige Quellcode der entwickelten Anwendung ist in Anhang D zu finden.

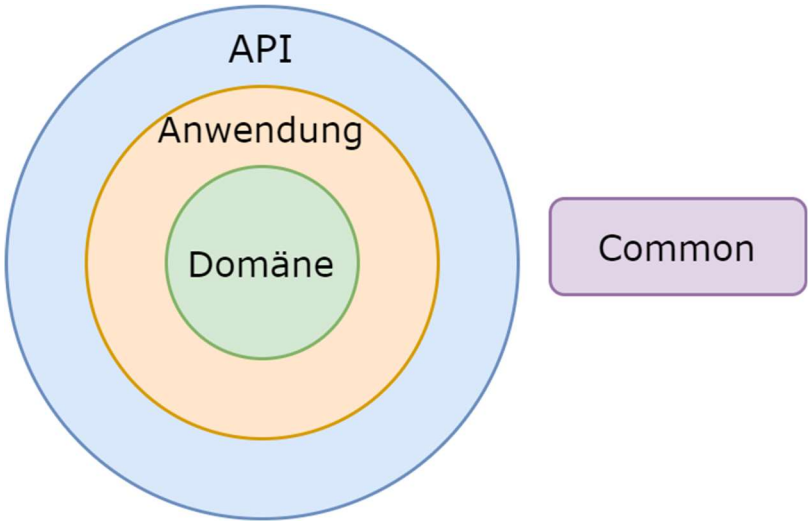

Abbildung 13: Schichten der entwickelten Architektur

API-Schicht

Die API-Schicht bildet die äußerste Schicht und dient als Schnittstelle zur Umwelt der Anwendung. Sie ist für die Kommunikationen mit externen Systemen, wie dem VCC-Tool verantwortlich und verarbeitet eingehende Anfragen. Es werden verschiedene Endpunkte bereitgestellt, die über gängige HTTP-Methoden wie *GET* und *POST* angesprochen werden können, um Änderungen an den Entitäten des VCC vorzunehmen oder Kataloge abzurufen.

Anwendungsschicht

Die Anwendungsschicht enthält die wesentliche Logik des VCC-Vorgehensmodells. Sie validiert eingehende Commands und führt diese gegebenenfalls aus. Auch Queries werden in der Anwendungsschicht bearbeitet. Je nachdem ob eine Query den Ist-Zustand der Read-Models abfragt oder eine historisierte Version anfordert, wird dafür auf gespeicherte Projektionen zurückgegriffen oder eine ältere Version über das Event Sourcing System rekonstruiert.

Domänenschicht

Die Domänenschicht stellt den Kern des Event-Sourcing Systems dar. Sie enthält die Aggregate der VCC-Entitäten und die jeweiligen Events. Die Aggregate repräsentieren die Entitäten des VCC und umfassen Methoden, die die entsprechenden Zustandsveränderungen aus einem Event ableiten.

Common-Projekt

Zuletzt existiert noch das Common-Projekt. Es kapselt unterstützende Funktionen, die von mehreren Schichten benötigt werden könnten. Dabei handelt es sich beispielsweise um Funktionen zur Validierung von Baumstrukturen. Dies dient der Wiederverwendbarkeit des Codes. Da alle Architekturschichten auf dieses Projekt zugreifen können, lässt sich argumentieren, dass dieses Projekt gemäß den Prinzipien der Onion Architecture auf Abbildung 13 innerhalb der Domänenschicht verortet sein müsste. Die gewählte Darstellungsform soll jedoch verdeutlichen, dass dieses Projekt keine Anwendungslogik im engeren Sinne beinhaltet, sondern nur zur Unterstützung anderer Schichten dient.

4.4 Architekturelemente

Nachdem im vorherigen Abschnitt ein über Überblick über die verschiedenen Anwendungsschichten gegeben wurde, wird in diesem Abschnitt erläutert aus welchen Komponenten die einzelnen Schichten bestehen, wie diese funktionieren und wie sie miteinander interagieren. Abbildung 14 zeigt die wesentlichen Schichten und ihre Komponenten. Um die Übersichtlichkeit zu erhöhen, werden einige Komponenten, wie beispielsweise Commands, in gestapelter Form dargestellt, anstatt jede einzelne Variante abzubilden

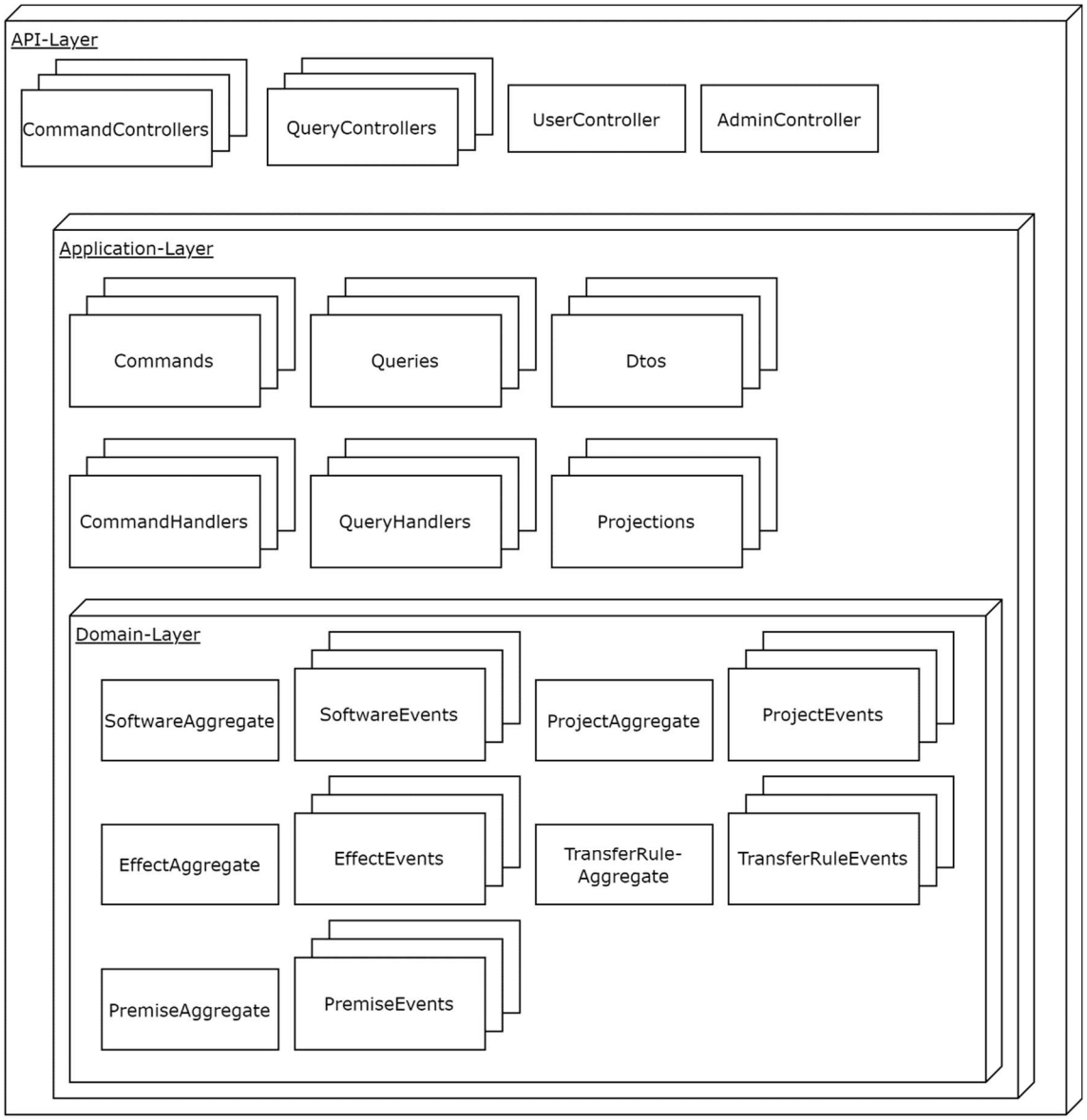

Abbildung 14: Struktureller Aufbau der Architekturschichten

Abbildung 15 zeigt die funktionalen Zusammenhänge der im Folgenden beschriebenen Komponenten. Auf die nummerierten Punkte der Abbildung wird an den entsprechenden Stellen im Text Bezug genommen, um das Zusammenspiel der verschiedenen Komponenten zu beschreiben.

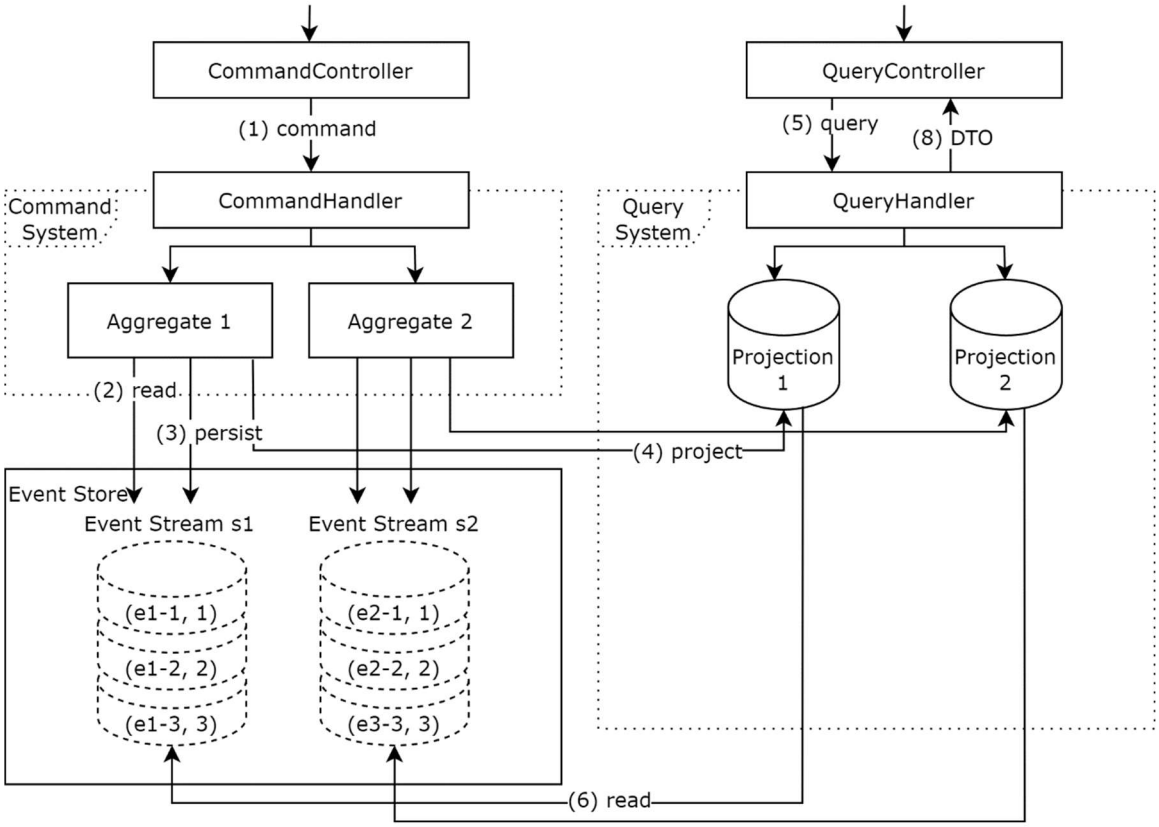

Abbildung 15: Funktionale Zusammenhänge der Architekturelemente in Anlehnung an Overeem et al. (2021, S. 10)

API-Schicht:

Die API-Schicht enthält hauptsächlich Controller. Controller existieren in den Ausprägungen CommandController (beispielsweise *EffectCommandController*, *PremiseCommandController* etc.) und QueryController (*EffectQueryController*, *PremiseQueryController* etc.). Sie enthalten die jeweiligen Endpunkte, die von externen Anwendungen über HTTP-Anfragen angesprochen werden können. Commandcontroller stellen für jedes Attribut der repräsentierten Entität einen Endpunkt bereit, der ein entsprechendes *Command* zur Zustandsänderung empfängt. Abbildung 16 zeigt beispielhaft das *UpdateEffectPlannedValue-Command*, mit dem der Soll-Wert einer Wirkung verändert werden kann. Um eine Rückverfolgbarkeit der Änderung zu gewährleisten, bezeichnet der „by"-Parameter die ID oder den Namen des Nutzers, der das Command ausführt.

```
public record UpdateEffectPlannedValue(
    Guid EffectId,
    decimal PlannedValue,
    string By
) : ICommand, IRequest<Result>;
```

Abbildung 16: Beispiel eines Command zur Zustandsänderung einer Entität

QueryController funktionieren nach einem ähnlichen Prinzip, sie stellen Endpunkte bereit, mit denen bestimmte Entitäten, ganze Kataloge oder Versionshistorien einer Entität abgerufen werden können. Analog zu den Commandcontrollern empfangen sie Queries, welche alle Informationen der Abfrage enthalten. Abbildung 17 zeigt beispielhaft die *GetEffect-Query*, mit der eine bestimmte Wirkung abgerufen werden kann. Auf der Abbildung ist zu erkennen, dass in der Query optional eine bestimmte Versionsnummer oder ein Zeitstempel angegeben werden kann, um den Zustand der Wirkung zu einem bestimmten Zeitpunkt abzurufen.

```
public record GetEffect(
    Guid EffectId,
    int? Version = null,
    DateTimeOffset? Timestamp = null) : IQuery<DTO.Effect>;
```

Abbildung 17: Beispiel einer Query zum Abruf einer versionierten Wirkung

Je nach Controllertyp wird der Anfrage ein Command oder eine Query im JSON-Format angehängt. Die Controller parsen das Command / die Query und übergeben diese/s an einen entsprechenden Command- oder Queryhandler in der Anwendungsschicht, welcher das Command oder die Query verarbeitet (Punkt 1 und 5 auf Abbildung 15).

Im Falle eines Commandhandler wird ein Statuscode zurückgegeben, der den Erfolg oder das Fehlschlagen des Command indiziert, gegebenenfalls ergänzt um eine Fehlermeldung. Queryhandler geben die angeforderte Entität / den angeforderten Katalog zurück (Punkt 8 auf Abbildung 15).

Anwendungsschicht

Wie auf Abbildung 14 zu erkennen ist, besteht die Anwendungsschicht aus Commands, Queries, den jeweiligen Handlern, Data Transfer Objects (DTOs) und Projektionen.

Wie zuvor erläutert, werden Commands und Queries in der API-Schicht erzeugt und an den jeweiligen Handler weitergegeben.

Um ein Command auszuführen, rekonstruiert der Commandhandler zuerst anhand der vorliegenden Events den Ist-Zustand des entsprechenden Aggregates. Dazu werden alle Events des entsprechenden Event Stream aus dem Eventspeicher ausgelesen (Punkt 2 auf Abbildung 15). Ein Event Stream besteht aus der chronologischen Reihenfolge der Events eines Aggregats. Nun können Prüfungen vorgenommen werden, um zu bestimmen, ob das Command zulässig ist. Gründe für eine Nichtzulässigkeit können zum Beispiel sein, dass die angegebene Entität überhaupt nicht existiert oder die angeforderte Zustandsänderung aufgrund der VCC-Logik nicht auf die Entität angewendet werden darf. Anschließend wird die entsprechende Aktion auf das Aggregat angewendet. Dieses

erzeugt daraus resultierende Events. So wird beispielsweise für ein *UpdatedEffectPlannedValue-Command* ein *EffectPlannedValueUpdated-Event* erzeugt. Der Commandhandler liest das Event dann aus dem Aggregat aus und persistiert es im Eventspeicher (Punkt 3 auf Abbildung 15). Unter Umständen greift ein Commandhandler auch auf mehrere Aggregate zu. Wird beispielsweise eine Wirkung aus dem System gelöscht, so muss die ParentId aller untergeordneten Wirkungen angepasst werden. Wird ein Event dem Eventspeicher hinzugefügt, wird es von Marten automatisch auf alle das Event konsumierende Projektionen angewendet und diese in der Read-Model-Datenbank aktualisiert (Punkt 4 auf Abbildung 15). Abbildung 18 zeigt ein Beispiel eines Commandhandler anhand des Handlers für das zuvor gezeigte *UpdateEffectPlannedValue-Command*.

```
public override async Task<Result> Handle(UpdateEffectPlannedValue command)
{
    var effect = Session.Events.AggregateStream<Domain.Aggregates.Effect.EffectAggregate>(command.EffectId);
    effect.UpdatePlannedValue(command.PlannedValue, command.By);
    Session.Events.Append(effect.Id, effect.GetUncommittedEvents());
    Session.SaveChanges();
    return new Result();
}
```

Abbildung 18: Methode zur Verarbeitung eines Commands am Beispiel des UpdateEffectPlannedValueHandler

Projektionen dienen als spezialisierte Read-Models. Genau wie Aggregate enthalten sie Methoden, die ein Event als Parameter nehmen und die entsprechenden Änderungen an der Projektion durchführen. Die Trennung von Aggregaten auf der Command-Seite und Projektionen auf der Query-Seite erlaubt es, Projektionen nicht nur als exaktes Abbild einer Entität zu gestalten. Stattdessen können Projektionen für bestimmte Darstellungszwecke erstellt werden, die Events verschiedener Entitäten oder nur bestimmte Events einer Entität konsumieren. Die Marten-Bibliothek stellt hierfür entsprechende Basisklassen zur Verfügung. Die vorliegende Implementierung enthält verschiedene Projektionen. Die simpelste Art von Projektionen sind Projektionen, die exakt eine Entität repräsentieren. Diese Projektionen sind inhaltlich identisch mit dem Aggregat der jeweiligen Entität. Eine komplexere Art von Projektion ist die *VersionList-Projektion*, die genutzt wird, um eine fortlaufende Versionsnummer auf Katalogebene zu ermöglichen. Auf diese Projektion wird am Ende dieses Kapitels ausführlicher eingegangen. Um nicht bei jedem Abruf eines Read-Models die Projektion anhand von Events rekonstruieren zu müssen, wird der aktuelle Zustand jeder Projektion in einer eigenen Read-Model-Datenbank gespeichert.[8]

[8] Technisch betrachtet handelt es sich um dieselbe PostgreSQL-Datenbank, die auch den Eventspeicher enthält. Die gewählte Formulierung soll verdeutlichen, dass die Speicherung von Projektionen unabhängig vom

Für jede Query existiert ein entsprechender Queryhandler. Einfachere Queries können beispielsweise darin bestehen, eine bestimmte Wirkung abzurufen. Ist in der Query keine bestimmte Versionsnummer oder ein Zeitstempel angegeben, kann dazu der aktuelle Stand der zugehörigen Projektion aus der Datenbank abgerufen werden. Bei Anforderung einer älteren Version oder eines spezifischen Zeitpunkts können alle entsprechenden Events mit geringerer Versionsnummer oder früherem Zeitstempel aus dem Eventspeicher geladen und zur Laufzeit auf eine neue Projektionsinstanz angewendet werden (Punkt 6 auf Abbildung 15). Auf diese Weise wird der gewünschte Zustand der Projektion wiederhergestellt. Die Marten-Bibliothek erleichtert dieses Vorgehen, so dass ein einfacher Queryhandler aus nur wenigen Zeilen Code besteht. Abbildung 19 zeigt solch einen Queryhandler am Beispiel des *GetEffect-Handlers*, der eine einzelne Wirkung abruft.

```
public override async Task<DTO.Effect> Handle(GetEffect query)
{
    Projection.Effect? effect = null;
    // If a specific version or timestamp is requested, we create a live projection
    if (query.Version.HasValue)
    {
        effect = Session.Events.AggregateStream<Projection.Effect>(query.EffectId, (long) query.Version);
    }
    else if (query.Timestamp.HasValue)
    {
        effect = Session.Events.AggregateStream<Projection.Effect>(query.EffectId, timestamp: query.Timestamp);
    }
    else
    {
        effect = Session.Load<Projection.Effect>(query.EffectId);
    }
    return effect?.ToDto() as DTO.Effect;
}
```

Abbildung 19: Methode zur Verarbeitung einer Query am Beispiel des GetEffect-Handler

Ein Queryhandler kann, analog zu einem Commandhandler, auch komplexere Anwendungslogik enthalten. Ein Beispiel hierfür ist der *GetPossibleParents-Handler*. Dies bestimmt für Entitäten, die in Baumstrukturen angeordnet sind (in der vorliegenden Implementierung des VCC sind dies Wirkungen und Prämissen), welche Entitäten zulässige „Eltern" sind. Dafür müssen alle Entitäten ausgeschlossen werden, die sich in der Baumstruktur unterhalb der entsprechenden Entität befinden, da eine Definition dieser als „Eltern" zu einem Zyklus innerhalb der Struktur führen würde.

Bevor die angeforderte Projektion an die API-Schicht zurückgegeben wird, wird sie durch den Queryhandler in ein DTO umgewandelt. DTOs sind spezielle Objekte, die dazu ver-

Eventspeicher ist. Theoretisch könnten Events und Projektionen auch in unabhängigen Datenbanken gespeichert werden.

wendet werden, Daten zwischen verschiedenen Schichten einer Anwendung oder zwischen Anwendungen und externen Systemen zu übertragen (Fowler, 2003, S. 401). DTOs kapseln die Datenhaltung der Anwendung von der Außenwelt ab, so dass anwendungsinterne Änderungen an der Datenstruktur sich nicht auf externe Systeme auswirken. Darüber gewährleistet ihre simple Struktur die Serialisierbarkeit (Fowler, 2003, S. 402–403).

Domänenschicht

Die Domänenschicht enthält den Kern des implementierten Event-Sourcing-Systems. Dieses besteht, wie in Abbildung 14 dargestellt, aus Aggregaten und den dazugehörigen Events. Für jede der im VCC-Tool implementierten Entitäten (IT-Service, Wirkung, Prämisse und Umrechnungsregel) wurde ein Aggregat angelegt. Diese Aggregate besitzen die im VCC-Tool definierten Attribute, die bereits in Kapitel 2.2.2 auf Abbildung 3 gezeigt wurden. Dabei handelt es sich um eine Abstraktion der umfangreicheren Attribute, welche von Schütte et al. (2022, S. 497–507) beschrieben wurden.

Zusätzlich wurde ein Projekt-Aggregat implementiert. Im VCC-Tool stellt dieses eine übergeordnete Einheit dar und bündelt dazugehörige Entitäten, indem jeder Entität ein Projekt zugeordnet wird. Während Schütte et al. (Schütte et al., 2022, S. 365–366) IT-Services projektübergreifend verorten, wurde im VCC-Tool die Abstraktion getroffen, IT-Services projektspezifisch zu behandeln (Evers et al., 2023, S. 26). Da der Fokus dieser Arbeit auf der Erarbeitung einer optimalen Versionierung liegt und das VCC-Tool eine geeignete Möglichkeit zur Visualisierung und Nutzung dieser bietet, wurde diese Abstraktion für diese Arbeit übernommen, um die Kompatibilität zwischen den Systemen zu gewährleisten.

Die Aggregate verfügen über Methoden, die von den Commandhandlern in der Anwendungsschicht genutzt werden, um ihre jeweiligen Attribute zu modifizieren. Diese Methoden verändern allerdings nicht direkt die Attribute, sondern erzeugen Events, welche die Zustandsveränderungen repräsentieren. Abbildung 20 zeigt ein Event am Beispiel des *EffectPlannedValueUpdated-Events*. Dieses wird erzeugt, wenn der Soll-Wert eine Wirkung verändert wurde.

```
public record EffectPlannedValueUpdated(
    Guid ProjectId,
    Guid Id,
    decimal PlannedValue,
    string By) : IEffectEvent;
```

Abbildung 20: Struktur eines Events am Beispiel des EffectPlannedValueUpdated-Events

Nachdem das Event erzeugt wurde, wird es auf das Aggregat angewendet und dort in einem internen Speicher abgelegt, so dass der ausführende Commandhandler es zur Speicherung im Eventspeicher auslesen kann. Abbildung 21 zeigt diesen Ablauf am Beispiel der *UpdatePlannedValue-Methode*, welche mit dem bereits erwähnten *UpdateEffectPlannedValue-Command* zusammenhängt.

```
public void UpdatePlannedValue(decimal commandPlannedValue, string commandBy)
{
    EffectPlannedValueUpdated effectPlannedValueUpdated =
      new EffectPlannedValueUpdated(ProjectId, Id, commandPlannedValue, commandBy);
    Apply(effectPlannedValueUpdated);
    AddUncommittedEvent(effectPlannedValueUpdated);
}

private void Apply(EffectPlannedValueUpdated @event)
{
    PlannedValueBeforeRealization = @event.PlannedValue;
}
```

Abbildung 21: Ablauf zur Erzeugung und Anwendung eines Events am Beispiel der UpdatePlannedValue-Methode im Effect-Aggregat

Für jedes Event existiert im Aggregat eine Methode, die das Event auf das Aggregat anwendet und das entsprechende Attribut tatsächlich verändert. Diese Apply-Methoden können später auch von einem Commandhandler genutzt werden, um den Zustand des Aggregates durch das erneute Anwenden von Events zu rekonstruieren.

Bei der Speicherung von Events im Eventspeicher reichert Marten diese automatisch um Metainformationen wie eine Versionsnummer und einen Zeitstempel an. Dadurch wird automatisch für jedes Aggregat eine Versionierung in Form einer fortlaufenden Versionsnummer angelegt. Schütte et al. (2022, S. 370) beschreiben allerdings eine Versionierung mit fortlaufender Nummerierung auf Katalogebene. Dies hätte beispielsweise erreicht werden können, indem anstatt eines einzelnen Wirkungsaggregat ein Wirkungskatalogaggregat implementiert worden wäre. Das hätte aber auch bedeutet, dass zum Rekonstruieren einer einzelnen Wirkung alle Events aller Wirkungen des Kataloges erneut angewendet werden hätten müssen. Bei größeren Katalogen hätte dies schnell zu einer Menge an Events geführt, die sich negativ auf die Leistungsfähigkeit des Systems auswirken könnte. Daher wurde sich für eine Implementierung von Aggregaten auf Entitätsebene entschieden.

Um dennoch eine Versionierung auf Katalogebene zu erreichen, wurde stattdessen eine *VersionList-Projektion* jeweils für IT-Services, Wirkungen, Prämissen und Umrechnungsregeln implementiert. Diese Projektion konsumiert alle Events der jeweiligen Entität und führt eine Liste aller Versionen, indexiert nach den jeweiligen Entitäten. Dabei wird ein

Versionszähler inkrementiert, sodass eine Versionsnummer auf Katalogebene entsteht. Diese Katalogversionsnummer kann auch in Queries genutzt werden, um eine bestimmte Version eines VCC-Katalogs abzurufen.

4.5 Integration in das VCC-Tool

Nach der Implementierung der versionierten VCC-Kataloge in Form einer API, wurde diese mit dem VCC-Tool integriert. Dazu mussten einige Änderungen an der Datenlogik des VCC-Tools vorgenommen werden. Zuvor speicherte dieses die VCC-Kataloge in einer relationalen MySQL-Datenbank. Diese Logik wurde modifiziert, so dass das VCC-Tool nun die Endpunkte der API nutzt, um Kataloge und Entitäten abzurufen oder zu verändern. Durch die Integration mit dem VCC-Tool können die in Kapitel 4.1 an die Anwendung gestellten funktionalen Anforderungen demonstriert werden.

Anhang E zeigt die Ansicht des Wirkungskatalogs im VCC-Tool nach der Integration mit der implementierten API. Nach dem Öffnen des auf Abbildung 22 zu sehenden Versionsdropdowns kann der Nutzer eine bestimmte Katalogversion auswählen. Wird eine bestimmte Katalogversion ausgewählt, wird der gesamte Katalog zum Zeitpunkt der gewählten Katalogversion dargestellt. Öffnet der Nutzer eine einzelne Wirkung, wird auch diese im Zustand entsprechend der Katalogversion angezeigt.

Wie auf der Abbildung 22 ebenfalls zu erkennen ist, wird für jede vorgenommene Änderung eine einzelne Version angezeigt. Bereits im Versionsdropdown ist zu erkennen, woraus die vorgenommene Änderung besteht.

Abbildung 23 zeigt die Detailansicht einer Entität am Beispiel einer Wirkung. Auch hier kann über ein Versionsdropdown eine beliebige Version der Wirkung ausgewählt werden. Des Weiteren können über die Detailansicht Änderungen an der Entität vorgenommen werden.

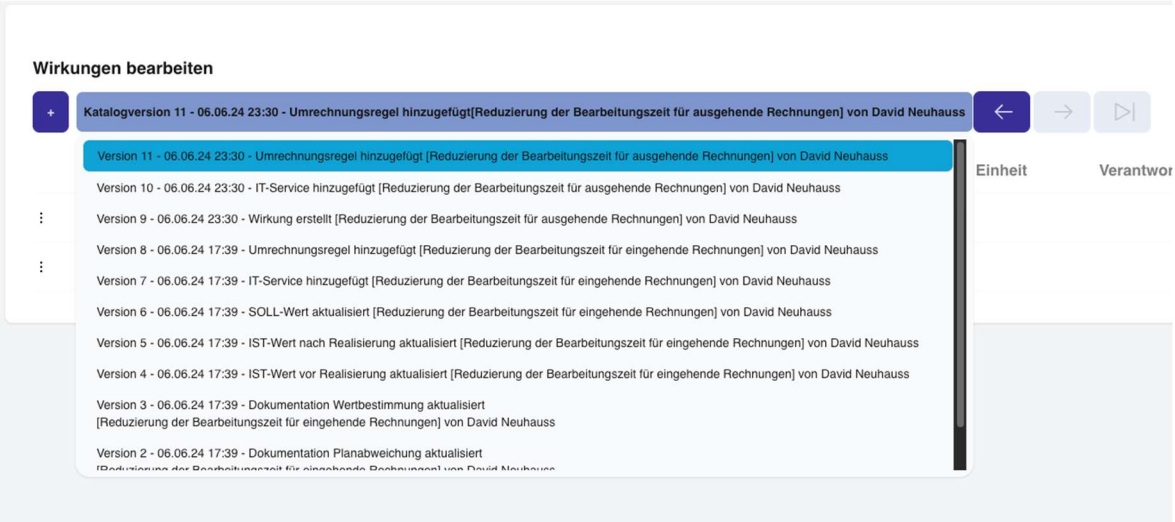

Abbildung 22: Versionsdropdown des Wirkungskatalogs im VCC-Tool

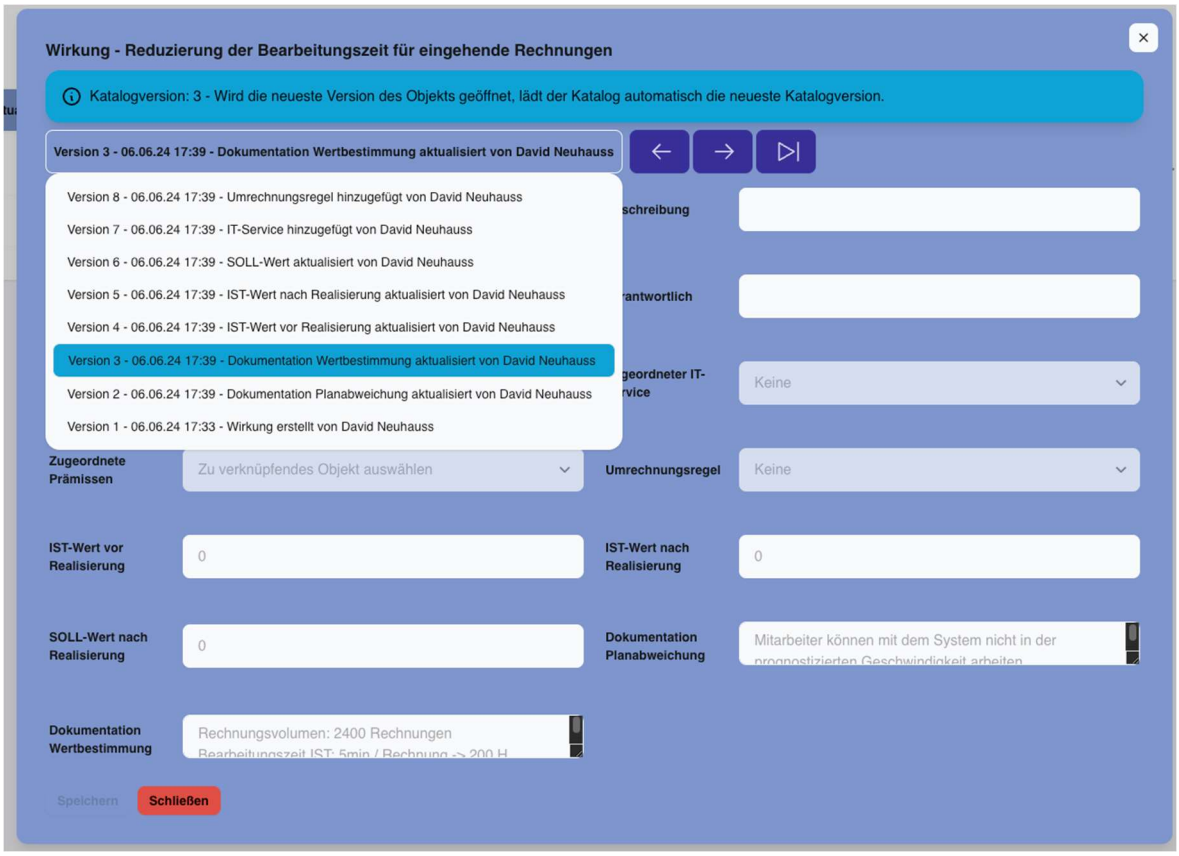

Abbildung 23: Versionsdropdown der Detailansicht einer Wirkung im VCC-Tool

5 Diskussion und Ausblick

5.1 Diskussion der Ergebnisse

Das Ziel der vorliegenden Arbeit bestand darin, ein geeignetes Design Pattern für die Versionierung von Wirkungskatalogen im VCC-Vorgehensmodell zu identifizieren und implementieren. Die Auswahl des Design Patterns sollte begründet erfolgen und sich an den in Kapitel 3.3 zuvor festgelegten Kriterien orientieren.

Dazu wurde zunächst in Kapitel 3.4.1 unter Anwendung des Entscheidungsmodells von Wątróbski et al. (2019, S. 109–119) die ARGUS-Methode für die Selektion eines Design Patterns unter gegebenen Alternativen ausgewählt. Um infrage kommende Design Patterns zu identifizieren, wurde daraufhin eine multimediale Recherche durchgeführt, bei der vier mögliche Design Patterns dokumentiert wurden. Die identifizierten Design Patterns wurden sodann hinsichtlich ihrer Entsprechung der Auswahlkriterien bewertet. Durch Anwendung der ARGUS-Methode wurde dann das Event Sourcing Pattern als am besten geeignetes Pattern identifiziert. Dabei erwies sich ARGUS als geeignete Methode, eine Auswahl unter den betrachteten Design Patterns vorzunehmen.

Anschließend wurde das Event Sourcing Design Pattern in Kapitel 4 in Form einer API implementiert. Anwendungen oder Nutzer können an die API entsprechende HTTP-Anfragen senden, um Änderungen an den VCC-Katalogen und ihren Entitäten vorzunehmen oder diese abzurufen. Optional kann dabei eine Versionsnummer oder ein Zeitstempel angegeben werden, um eine versionierte Version des Katalogs und seiner Entitäten abzurufen. Einzelne Änderungen können dabei durch den Nutzer nachverfolgt werden. Dabei ist einsehbar, welche Änderung, wann, durch wen vorgenommen wurde. Durch die in Kapitel 4.5 beschriebene Integration der API mit dem VCC-Tool können diese Funktionalitäten zudem visualisiert werden. Demnach erfüllt die Anwendung alle in Kapitel 4.1 festgelegten funktionalen Anforderungen. Durch die Implementierung als Docker-fähige API mit dem .NET-Framework wurden zudem die technischen Grundlagen gelegt, um auch die nicht-funktionalen Anforderungen der Skalierbarkeit und Zuverlässigkeit erfüllen zu können.

Konkludierend lässt sich sagen, dass das Ziel der Arbeit erreicht wurde. Es wurde ein Design Pattern identifiziert und implementiert, welches die Anforderungen des VCC-Vorgehensmodells an die Versionierung der VCC-Kataloge erfüllt. Somit haben sich auch das Entscheidungsmodell von Wątróbski et al. und die ARGUS-Methode als geeignet für die vorliegende Problemstellung erwiesen. Dennoch wurden im Rahmen der Implementierung einige Schwächen und Herausforderungen des gewählten Ansatzes deutlich.

Die Komplexität des Patterns stellte sich als klarer Nachteil dar. In relationalen Datenbanksystemen müsste für die grundlegende Verwaltung von Katalogen lediglich ein Datenbankschema definiert und gegebenenfalls ein Object Relational Mapper (ORM) konfiguriert werden. Für die vorliegende Event-Sourcing-basierte Implementierung hingegen mussten Aggregate, Projektionen und Events implementiert werden. Zusätzliche Komplexität entstand durch die Anwendung des CQRS-Patterns, dessen Verwendung im Kontext von Event Sourcing üblich ist, aber das Anlegen von Commands, Queries, entsprechenden Handlern und DTOs erforderte (Betts et al., 2013, S. 235; Khononov, 2021, S. 129; Pacheco, 2018, S. 113). Dies hat dazu geführt, dass Elemente wie die Struktur einer Entität an mehreren Stellen des Codes definiert werden mussten, was teilweise zu einer Redundanz des Codes geführt hat und sich negativ auf die Wartbarkeit auswirken dürfte.

Andererseits lässt sich feststellen, dass die Anwendung von CQRS- und DDD-Konzepten zu einer Trennung der Zuständigkeiten einzelner Codebestandteile beigetragen hat, was geholfen hat, mit der Komplexität der zu leistenden Implementierung umzugehen. Es stellt sich daher die Frage, ob die Verwendung von Event-Sourcing ohne CQRS zu einem simpleren Code mit höherer Wartbarkeit geführt hätte.

Ein anderer Ansatz, die Komplexität der Anwendung zu reduzieren, könnte die verstärkte Verwendung von Generics im Quellcode sein. Die implementierte Domäne der VCC-Kataloge weist einige Redundanzen hinsichtlich der Events auf. So könnten beispielsweise alle Commandhandler, die den Namen einer Entität aktualisieren, zu einem generischen *UpdateName-Handler* verallgemeinert werden. Dadurch könnte der Codeumfang reduziert und die Wartbarkeit erhöht werden. Dieser Ansatz sollte bei einer Weiterentwicklung der Anwendung berücksichtigt werden.

5.2 Limitationen der Arbeit

Trotz der erfüllten Anforderungen weist die vorliegende Implementierung einige Limitationen hinsichtlich der Abbildung des VCC-Vorgehensmodells auf. Diese Limitationen sollten bei einer Neu- oder Weiterentwicklung der versionierten VCC-Kataloge berücksichtigt und behoben werden.

Überwiegend beruhen die Limitationen auf der starken Orientierung dieser Arbeit an den Logiken des VCC-Tools. So wurde von diesem die den Katalogen übergeordnete Einheit des Projekts übernommen. Schütte et al. (2022, S. 371) hingegen betonen, dass beispielsweise der IT-Service projektübergreifend betrachtet werden sollte. Dieser Aspekt sollte bei einer Weiterentwicklung durch eine entsprechende Anpassung des Klassenschemas adressiert werden.

Des Weiteren weisen die implementierten Entitäten noch einen starken Abstraktionsgrade von den von Schütte et al. (2022, S. 497–507) beschriebenen Attributen auf. Diese Limitation kann behoben werden, indem die Implementierung um die entsprechenden Attribute erweitert wird. Die klare Trennung von Zuständigkeiten zwischen den einzelnen Codebestandteilen erleichtert dies.

Eine weitere Limitation hinsichtlich des VCC-Vorgehensmodells besteht darin, dass die implementierte Versionierung, die im VCC vorgesehenen Phasen nicht berücksichtigt. Die Nummerierung der Katalogversionen beginnt also nicht wie von Schütte et al. (2022, S. 370) beschrieben mit der Nummer der jeweiligen Phase, sondern inkrementiert die Nummer für jede vorgenommene Veränderungen. Dies hat den Vorteil, dass eine Nutzung des VCC-Ansatzes zum Wirkungsmanagement unabhängig von dem phasenbasierten Vorgehensmodell möglich ist. Um dennoch den Stand eines Kataloges zu einem bestimmten Zeitpunkt (beispielsweise dem Abschluss einer Phase) einsehen zu können, könnte eine Art Checkpoint System im VCC-Tool integriert werden. Dazu müsste lediglich der Zeitstempel des Phasenabschluss gespeichert werden. Dieser könnte dem Versionsdropdown in der Katalogansicht hinzugefügt werden und würde es so erlauben, den Katalog zu diesem bestimmten Zeitpunkt einzusehen. Um die Zugehörigkeit einer Version zu einer bestimmten Phase zu verdeutlichen, könnte die Phasennummer auf der Darstellungsebene der Katalogversion vorangestellt werden.

5.3 Forschungsausblick

Aus den ausgeführten Herausforderungen und Limitationen dieser Arbeit, eröffnen sich einige Ansätze für zukünftige Arbeiten im Bereich der Selektion von Design Patterns und Versionierung von Daten.

Es hat sich gezeigt, dass einige der spezifisch auf Design Patterns ausgelegten Selektionsmethoden eher theoretischer Natur sind und einer umfangreichen Datenvorbereitung bedürfen. Es ist außerdem schlüssig, dass die Ergebnisqualität der Selektionsmethoden maßgeblich von Umfang und Qualität der betrachteten Design Patterns abhängt. Dies wird aber durch die heterogene Quellenlage erschwert, in der Design Patterns beschrieben werden. Wie in Kapitel 2.3.2 erwähnt, gab es in der Vergangenheit bereits Ansätze für webbasierte Design-Pattern-Datenbanken oder Suchmaschinen. Diese Ansätze konnte sich allerdings alle nicht durchsetzen und werden heutzutage nicht mehr betrieben. Birukou (2010, S. 6–8) führt dies zumindest teilweise darauf zurück, dass keine Gemeinschaft gebildet werden konnte, die sich um die Pflege und Weiterentwicklung dieser Systeme kümmert. Ein interessanter Forschungsansatz könnte hier daraus bestehen, zu erarbeiten, wie eine Design-Pattern-Datenbank oder Suchmaschine aufgebaut sein müsste, um sich in der Entwicklergemeinschaft durchsetzen zu können.

Birukou (2010, S. 6–8) sowie Henninger und Corrêa (2007, S. 16–19) nennen hier einige (gescheiterte) Ansätze, die dazu analysiert werden sollten. Solch eine Datenbank könnte auch ein geeigneter Ansatz sein, um die beschriebenen, eher theoretischen Selektionsmethoden für Entwickler praktisch nutzbar zu machen.

Weitere Ansätze für zukünftige Arbeiten ergeben sich auf dem Gebiet der Versionierung, besonders im Zusammenhang mit dem Event Sourcing Design Pattern. Es wurde in dieser Arbeit gezeigt, dass dieses Design Pattern vielfältige Möglichkeiten im Bereich der Versionierung und Historisierung von Daten bietet. Die hohe Komplexität und Redundanz des entstehenden Codes behindern allerdings möglicherweise seine breitere Verwendung. Es erscheinen zwei mögliche Ansätze, um die Zugänglichkeit des Patterns zu verbessern.

Einerseits könnte erarbeitet werden, wie eine minimalistische Event Sourcing Implementierung aussehen könnte. Während die Verbindung von CQRS mit Event Sourcing gewisse Vorteile bietet, wäre ein minimalistischer Ansatz möglicherweise besser geeignet, um das Pattern auch für Entwickler mit begrenzten Ressourcen und Erfahrung im Umgang mit Event Sourcing nutzbar zu machen.

Der zweite Ansatz betrifft die grundsätzliche Art und Weise wie Event Sourcing implementiert wird. Event Sourcing wird häufig als Gegenentwurf zu relationalen Datenbanken betrachtet und in Systemen eingesetzt, die hohe Anforderungen an die Nachverfolgbarkeit von Zustandsänderungen stellen, wie beispielsweise Zahlungssystemen (Overeem et al., 2021, S. 7). In diesen Systemen sind die Daten oft bereits aufgrund ihrer wachsenden, eventbasierten Natur (beispielsweise die Summe aller Kontobewegungen in einem Finanzsystem) für Event Sourcing geeignet. Teilweise wird die Abbildung eines klassischen, Cread Read Update Delete (CRUD) System mit Event Sourcing auch als Antipattern bezeichnet (Millett & Tune, 2015, S. 127).

Diese Arbeit zeigt jedoch, dass Event-Sourcing auch für CRUD-orientierte Systeme vielversprechende Möglichkeiten zur Wiederherstellung vergangener Zustände und Nachverfolgung von Änderungen bietet. Ein vielversprechender Ansatz wäre die Entwicklung eines ORM / CRUD-Frameworks, das für den Nutzer wie ein gewöhnliches ORM funktioniert, jedoch auf Datenbankseite mit Event Sourcing arbeitet und dem Nutzer entsprechende Funktionalitäten zur Verfügung stellt. So könnten Nutzer in einem CRUD-System Änderungen nachverfolgen und Zustände wiederherstellen, ohne sich mit der Komplexität und den Besonderheiten von Event-Sourcing auseinandersetzen zu müssen.

Zudem sollten zukünftige Arbeiten untersuchen, wie die aus der Versionierung der VCC-Kataloge resultierenden Analysemöglichkeiten bestmöglich genutzt werden können, um das Wirkungsmanagement von IT-Systemen zu unterstützen.

6 Fazit

Diese Arbeit hat gezeigt, dass das Event Sourcing Design Pattern dazu geeignet ist, eine Versionierung von Wirkungszusammenhängen im Kontext des VCC-Vorgehensmodells zu ermöglichen. Dies wurde durch eine Implementierung des Event Sourcing Patterns in Form einer API demonstriert. Durch eine Integration mit dem VCC-Tool wurden die Ergebnisse dieser Arbeit für Anwender nutzbar gemacht, so dass diese das VCC-Vorgehensmodell für ein Wertbeitragscontrolling von IT-Projekten einsetzen können. Im Rahmen der Ausarbeitung konnte gezeigt werden, wie eine MCDA-Methode eingesetzt werden kann, um aus einer Auswahl von Design Patterns eine begründete Entscheidung zu treffen. Dabei hat sich die ARGUS-Methode als geeignet für die Selektion eines Design Patterns erwiesen.

Trotz des erreichten Ziels wurden einige Limitationen und Herausforderungen deutlich. Die hohe Komplexität des Event Sourcing Patterns führte zu einem umfangreichen und teilweise redundanten Code. Diese Komplexität könnte durch die Verwendung von Generics reduziert werden. Zudem zeigte sich, dass die Implementierung von Event Sourcing ohne CQRS möglicherweise zu einem simpleren und wartbareren Code führen könnte.

Darüber hinaus hat die vorliegende Arbeit gezeigt, welche Potentiale das Event Sourcing Pattern für die Nachverfolgbarkeit und Versionierung von Informationen bietet. Um die Zugänglichkeit des Patterns zu verbessern, wird die Entwicklung eines Event Sourcing basierten ORM in zukünftigen Arbeiten vorgeschlagen.

Schlussendlich lässt sich festhalten, dass die Arbeit einen Beitrag dazu geleistet hat, die praktische Anwendung des VCC-Vorgehensmodells zu ermöglichen. Es wurde die Grundlage dafür gelegt, die Möglichkeiten versionierter Wirkungszusammenhänge für zeitbezogene Analysen von IT-Projekten zu nutzen.

Literaturverzeichnis

Alexander, C. (1979). *The timeless way of building*. Oxford University Press.

Alexander, C., Ishikawa, S., & Silverstein, M. (1977). *A pattern language: Towns, buildings, construction*. Oxford University Press.

Asghar, M. Z., Alam, K. A., & Javed, S. (2019). Software Design Patterns Recommendation: A Systematic Literature Review. *2019 International Conference on Frontiers of Information Technology (FIT)*, 167–172. https://doi.org/10.1109/FIT47737.2019.00040

Austin, R., & Devin, L. (2003). Beyond Requirements: Software Making as Art. *IEEE Software*, *20*(01), 93–95. https://doi.org/10.1109/MS.2003.1159037

Bafandeh Mayvan, B., Rasoolzadegan, A., & Ghavidel Yazdi, Z. (2017). The state of the art on design patterns: A systematic mapping of the literature. *Journal of Systems and Software*, *125*, 93–118. https://doi.org/10.1016/j.jss.2016.11.030

Betts, D., Domínguez, J., Melnik, G., Simonazzi, F., & Subramanian, M. (2013). *Exploring CQRS and Event Sourcing: A journey into high scalability, availability, and maintainability with Windows Azure*. Microsoft.

bin Uzayr, S. (2022). *Software Design Patterns: The Ultimate Guide*. CRC Press. https://doi.org/10.1201/9781003308461

Birukou, A. (2010). A survey of existing approaches for pattern search and selection. *Proceedings of the 15th European Conference on Pattern Languages of Programs*, 1–13. https://doi.org/10.1145/2328909.2328912

Brynjolfsson, E. (1993). The Productivity Paradox of Information Technology. *Commun. ACM*, *36*(12), 66–77. https://doi.org/10.1145/163298.163309

Brynjolfsson, E., & Yang, S. (1996). Information Technology and Productivity: A Review of the Literature. In M. V. Zelkowitz (Hrsg.), *Advances in Computers* (Bd. 43, S. 179–214). Elsevier. https://doi.org/10.1016/S0065-2458(08)60644-0

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., & Stal, M. (1996). *Pattern oriented software architecture: Volume 1: A system of patterns* (2001. Aufl.). Wiley.

Colecchia, A., & Schreyer, P. (2001). *ICT Investment and Economic Growth in the 1990s: Is the United States a Unique Case? A Comparative Study of Nine OECD Countries* (OECD Science, Hrsg.; Nummer 7). https://doi.org/10.1787/078325004705

Cron, W. L., & Sobol, M. G. (1983). The relationship between computerization and performance: A strategy for maximizing the economic benefits of computerization. *Information & Management*, *6*(3), 171–181. https://doi.org/10.1016/0378-7206(83)90034-4

De Keyser, W., & Peeters, P. (1994). Argus—A New Multiple Criteria Method Based on the General Idea of Outranking. In M. Paruccini (Hrsg.), *Applying Multiple Criteria Aid for Decision to Environmental Management* (S. 263–278). https://doi.org/10.1007/978-94-017-0767-1_17

Deutsche Post AG. (2015, Oktober 28). *Ad hoc: Deutsche Post DHL Group entscheidet über die IT-Neuausrichtung von DHL Global Forwarding*. DPDHL. https://group.dhl.com/de/investoren/mitteilungen/ad-hoc-mitteilungen/ad-hoc-dpdhl-20151028.html

Eden, A., Hirshfeld, Y., & Yehudai, A. (1998). *LePUS - A Declarative Pattern Specification Language*. *326*.

Evers, C., Féaux de Lacroix, S., Neuhauss, D., & Tenkamp, M. (2023). *Entwicklung eines Tools zur Unterstützung des Value Contribution Controllings*.

Fishburn, P. C. (1974). Exceptional Paper—Lexicographic Orders, Utilities and Decision Rules: A Survey. *Management Science*, *20*(11), 1442–1471. https://doi.org/10.1287/mnsc.20.11.1442

Fowler, M. (1996). *Analysis Patterns: Reusable Object Models*. Pearson Education.

Fowler, M. (2003). *Patterns of enterprise application architecture* (5. Aufl.). Addison-Wesley.

Fowler, M. (2004a, März 7). *Audit Log*. martinfowler.com. https://martinfowler.com/eaaDev/AuditLog.html

Fowler, M. (2004b, März 7). *Temporal Property*. martinfowler.com. https://martinfowler.com/eaaDev/TemporalProperty.html

Fowler, M. (2005, Dezember 12). *Event Sourcing*. martinfowler.com. https://martinfowler.com/eaaDev/EventSourcing.html

Gamma, E., Helm, R., Johnson, R. E., & Vlissides, J. (1994). *Design patterns: Elements of reusable object-oriented software* (1. Aufl.). Pearson Education.

Hajli, M., Sims, J. M., & Ibragimov, V. (2015). Information technology (IT) productivity paradox in the 21st century. *International Journal of Productivity and Performance Management*, *64*(4), 457–478. https://doi.org/10.1108/IJPPM-12-2012-0129

Hamdy, A., & Elsayed, M. (2018). Topic modelling for automatic selection of software design patterns. *Proceedings of the International Conference on Geoinformatics and Data Analysis*, 41–46. https://doi.org/10.1145/3220228.3220263

Henninger, S., & Corrêa, V. (2007). Software pattern communities: Current practices and challenges. *Proceedings of the 14th Conference on Pattern Languages of Programs*, 1–19. https://doi.org/10.1145/1772070.1772087

Hitt, L. M., & Brynjolfsson, E. (1996). Productivity, Business Profitability, and Consumer Surplus: Three Different Measures of Information Technology Value. *MIS Quarterly*, *20*(2), 121–142. https://doi.org/10.2307/249475

Hussain, S., Keung, J., & Khan, A. A. (2017). Software design patterns classification and selection using text categorization approach. *Applied Soft Computing*, *58*, 225–244. https://doi.org/10.1016/j.asoc.2017.04.043

Hussain, S., Keung, J., Sohail, M. K., Khan, A. A., & Ilahi, M. (2019). Automated framework for classification and selection of software design patterns. *Applied Soft Computing*, *75*, 1–20. https://doi.org/10.1016/j.asoc.2018.10.049

Kemerer, C. F., & Sosa, G. L. (1991). Systems development risks in strategic information systems. *Information and Software Technology*, *33*(3), 212–223. https://doi.org/10.1016/0950-5849(91)90136-Y

Khononov, V. (2021). *Learning Domain-Driven Design*. O'Reilly Media, Inc.

Kim, D.-K., & El Khawand, C. (2007). An approach to precisely specifying the problem domain of design patterns. *Journal of Visual Languages and Computing*, *18*(6), 560–591.

Kim, D.-K., & Shen, W. (2008). Evaluating pattern conformance of UML models: A divide-and-conquer approach and case studies. *Software Quality Journal*, *16*(3), 329–359. https://doi.org/10.1007/s11219-008-9048-5

Kimball, R., & Ross, M. (2013). *The data warehouse toolkit: The definitive guide to dimensional modeling* (3. Aufl.). Wiley.

Lehr, B., & Lichtenberg, F. (1999). Information Technology and Its Impact on Productivity: Firm-Level Evidence from Government and Private Data Sources, 1977-1993. *The Canadian Journal of Economics*, *32*(2), 335–362. https://doi.org/10.2307/136426

Martel, J.-M., & Matarazzo, B. (2016). Other Outranking Approaches. In S. Greco, M. Ehrgott, & J. R. Figueira (Hrsg.), *Multiple Criteria Decision Analysis: State of the Art Surveys* (2. Aufl., S. 221–282). Springer. https://doi.org/10.1007/978-1-4939-3094-4_7

Masli, A., Richardson, V., Sánchez, J., & Smith, R. (2011). The Business Value of IT: A Synthesis and Framework of Archival Research. *Journal of Information Systems*, *25*. https://doi.org/10.2308/isys-10117

McConnell, Steve. (2004). *Code complete* (2. Aufl.). Microsoft Press.

Millett, S., & Tune, N. (2015). *Patterns, principles, and practices of domain-driven design* (1st edition). Wrox.

Moaven, S., & Habibi, J. (2020). A fuzzy-AHP-based approach to select software architecture based on quality attributes (FASSA). *Knowledge and Information Systems*, *62*(12), 4569–4597. https://doi.org/10.1007/s10115-020-01496-7

Muangon, W., & Intakosum, S. (2013). Case-based Reasoning for Design Patterns Searching System. *International Journal of Computer Applications*, *70*, 16–24. https://doi.org/10.5120/12231-8433

Naghdipour, A., Hasheminejad, S. M. H., & Barmaki, R. L. (2023). Software design pattern selection approaches: A systematic literature review. *Software: Practice and Experience*, *53*(4), 1091–1122. https://doi.org/10.1002/spe.3176

Naghdipour, A., Hossien Hasheminejad, S. M., & Reza Keyvanpour, M. (2021). DPSA: A Brief Review for Design Pattern Selection Approaches. *2021 26th International Computer Conference, Computer Society of Iran (CSICC)*, 1–6. https://doi.org/10.1109/CSICC52343.2021.9420629

Nahar, N., & Sakib, K. (2015, Dezember 1). *Automatic Recommendation of Software Design Patterns Using Anti-patterns in the Design Phase: A Case Study on Abstract Factory*.

Nawaz, F., Mohsin, A., Fatima, S., & Janjua, N. K. (2015). Rule-Based Multi-criteria Framework for SaaS Application Architecture Selection. In T. Dillon (Hrsg.), *Artificial Intelligence in Theory and Practice IV* (S. 129–138). Springer International Publishing. https://doi.org/10.1007/978-3-319-25261-2_12

Nemery, P., & Ishizaka, A. (2013). *Multi-criteria decision analysis: Methods and software*. Wiley.

Oliveira Rocha, H. F. (2021). *Practical Event-Driven Microservices Architecture: Building Sustainable and Highly Scalable Event-Driven Microservices*. Apress. https://doi.org/10.1007/978-1-4842-7468-2

Overeem, M., Spoor, M., Jansen, S., & Brinkkemper, S. (2021). An empirical characterization of event sourced systems and their schema evolution—Lessons from industry. *Journal of Systems and Software*, *178*, 110970. https://doi.org/10.1016/j.jss.2021.110970

Pacheco, V. F. (2018). *Microservice patterns and best practices: Explore patterns like CQRS and event sourcing to create scalable, maintainable, and testable microservices* (1st ed.). Packt Publishing.

Palma, F., Farzin, H., Guéhéneuc, Y.-G., & Moha, N. (2012). *Recommendation System for Design Patterns in Software Development: An DPR Overview*. 1–5. https://doi.org/10.1109/RSSE.2012.6233399

Rahmati, R., Rasoolzadegan, A., & Dehkordy, D. T. (2019). An Automated Method for Selecting GoF Design Patterns. *2019 9th International Conference on Computer and Knowledge Engineering (ICCKE)*, 345–350. https://doi.org/10.1109/ICCKE48569.2019.8965221

Rising, L. (2000). *The Pattern Almanac 2000*. Addison Wesley.

Roach, S. S. (1998). No Productivity Boom for Workers. *Issues in Science and Technology*, *14*(4), 49–56.

Rubis, R., & Cardei, I. (2015). The business data object versioning and change history patterns. *Proceedings of the 22nd Conference on Pattern Languages of Programs*, 1–9.

Sabatucci, L., Cossentino, M., & Susi, A. (2015). A goal-oriented approach for representing and using design patterns. *Journal of Systems and Software*, *110*, 136–154. https://doi.org/10.1016/j.jss.2015.07.040

Sahly, E. M., & Sallabi, O. M. (2012). *Design pattern selection: A solution strategy method*. 2012 International Conference on Computer Systems and Industrial Informatics, ICCSII 2012. Scopus. https://doi.org/10.1109/ICCSII.2012.6454337

Schütte, R., Seufert, S., & Wulfert, T. (2022). *IT-Systeme wirtschaftlich verstehen und gestalten: Methoden - Paradoxien - Grundsatze* (1st Aufl 2022nd edition). Springer.

Schweikl, S., & Obermaier, R. (2020). Lessons from three decades of IT productivity research: Towards a better understanding of IT-induced productivity effects. *Management Review Quarterly*, *70*(4), 461–507. https://doi.org/10.1007/s11301-019-00173-6

Snodgrass, R. T. (1999). *Developing Time-Oriented Database Applications in SQL*. Morgan Kaufmann.

Solow, R. (1987, Juli 12). We'd better watch out. *New York Times Book Review*, 36.

Uchitelle, L. (2000, März 12). ECONOMIC VIEW; Productivity Finally Shows The Impact Of Computers. *The New York Times*, 143.

Vijayalakshmi, S., Zayaraz, G., & Vijayalakshmi, V. (2010). Multicriteria Decision Analysis Method for Evaluation of Software Architectures. *International Journal of Computer Applications*, *1*(25). https://doi.org/10.5120/463-767

Wagner, G.-L., & Beckmann, H. (2022). *Literatur-Review: State of the Art des Wertbeitrags der IT zum Unternehmenserfolg*. https://doi.org/10.18420/INF2022_140

Wątróbski, J., Jankowski, J., Ziemba, P., Karczmarczyk, A., & Zioło, M. (2019). Generalised framework for multi-criteria method selection. *Omega*, *86*, 107–124. https://doi.org/10.1016/j.omega.2018.07.004

Watson, I., & Marir, F. (1994). Case-based reasoning: A review. *The Knowledge Engineering Review*, *9*(4), 327–354. https://doi.org/10.1017/S0269888900007098

Wiegers, K. E., & Beatty, J. (2013). *Software Requirements* (Third edition.). Microsoft Press.

Zamani, B., Butler, G., & Kayhani, S. (2009). Tool Support for Pattern Selection and Use. *Electronic Notes in Theoretical Computer Science*, *233*, 127–142. https://doi.org/10.1016/j.entcs.2009.02.065

Zdun, U. (2007). Systematic pattern selection using pattern language grammars and design space analysis. *Software: Practice and Experience*, *37*(9), 983–1016. https://doi.org/10.1002/spe.799

Zimmermann, H.-J., & Gutsche, L. (1991). *Multi-Criteria Analyse*. Springer. https://doi.org/10.1007/978-3-642-58198-4

Anhang

Anhang A - Dokumentation der Attribute der VCC-Entitäten im Quellcode

Wirkung (Effect)

Wirkungen entstehen durch die Einführung von IT-Services. Sie sind das Kernelement der Wertbeitragsanalysen.

Attribute

- Name
- Description
- AssignedEmployee -> der für die Umsetzung verantwortliche Mitarbeiter
- CompanyLevel -> Unternehmensebene, auf der die Wirkung eintritt
- DocuDifferenceToPlanned -> Ein Freitextfeld um die Abweichung zwischen geplantem und endgültigem Wert zu dokumentieren
- DocuValueDetermination -> Ein Freitextfeld um die Bestimmung der Werte zu dokumentieren
- CurrentValueBeforeRealization -> der IST-Wert vor der Projektrealisierung
- CurrentValueAfterRealization -> der IST-Wert nach der Projektrealisierung
- PlannedValueBeforeRealization -> der vor der Projektrealisierung geplante IST-Wert nach der Projektrealisierung
- SoftwareId -> die ID des der Wirkung zugeordneten IT-Services
- PremiseIds -> die IDs der Prämissen denen die Wirkung unterliegt
- TransferRuleId -> die der Wirkung zugeordnete Umrechnungsregel

IT-Service (Software)

IT-Services (im Code zur Eindeutigkeit Software genannt) stellen im Rahmen eines Projektes einzuführende oder zu verändernde IT-Services dar.

Attribute

- Name
- Description
- ProjectId
- PremiseIds -> die IDs der dem Service zugeordneten Prämissen
- EffectIds -> wie PremiseIds

Prämisse (Premise)

Prämissen sind entweder Wirkungs- oder Serviceprämissen. Logisch bedingen sie die Einführung eines Services / das Eintreten einer Wirkung. Diese Tatsache

findet sich noch nicht in den Wertbeitragsanalysen wieder.

Prämissen können einer Prämisse untergeordnet werden, so dass ein Projekt einen oder mehrere Prämissenbäume enthalten kann.

Attribute

- Name
- Description
- PremiseType -> "Service" oder "Effect"
- ParentId
- SoftwareIds
- EffectIds

Umrechnungsregel (TransferRule)

Umrechnungsregeln überführen eine nicht-monetäre Größe (wie Mitarbeiterstunden) in eine Auswirkung auf den EBITDA. Sie werden von einer oder mehreren Wirkungen referenziert.

Attribute

- Name
- Dimension -> Die quantitative und messbare Größe die in EBITDA überführt wird
- Unit -> Einheit der Dimension, z.B. "€" oder "H"
- Factor -> Der Faktor um den eine Einheit der Dimension den EBITDA verändert

Beispiele

Einfacher Monetärer Gewinn

Dimension: Gewinn

Unit: €

Factor: 1

Bedeutet: 1€ Gewinn steigert den EBITDA um 1€

Einfache Kosten

Dimension: Kosten

Unit: €

Factor: -1

Bedeutet: 1€ Kosten mindern den EBITDA um 1€

Anhang B - Entscheidungsbaum von Wątróbski et al.

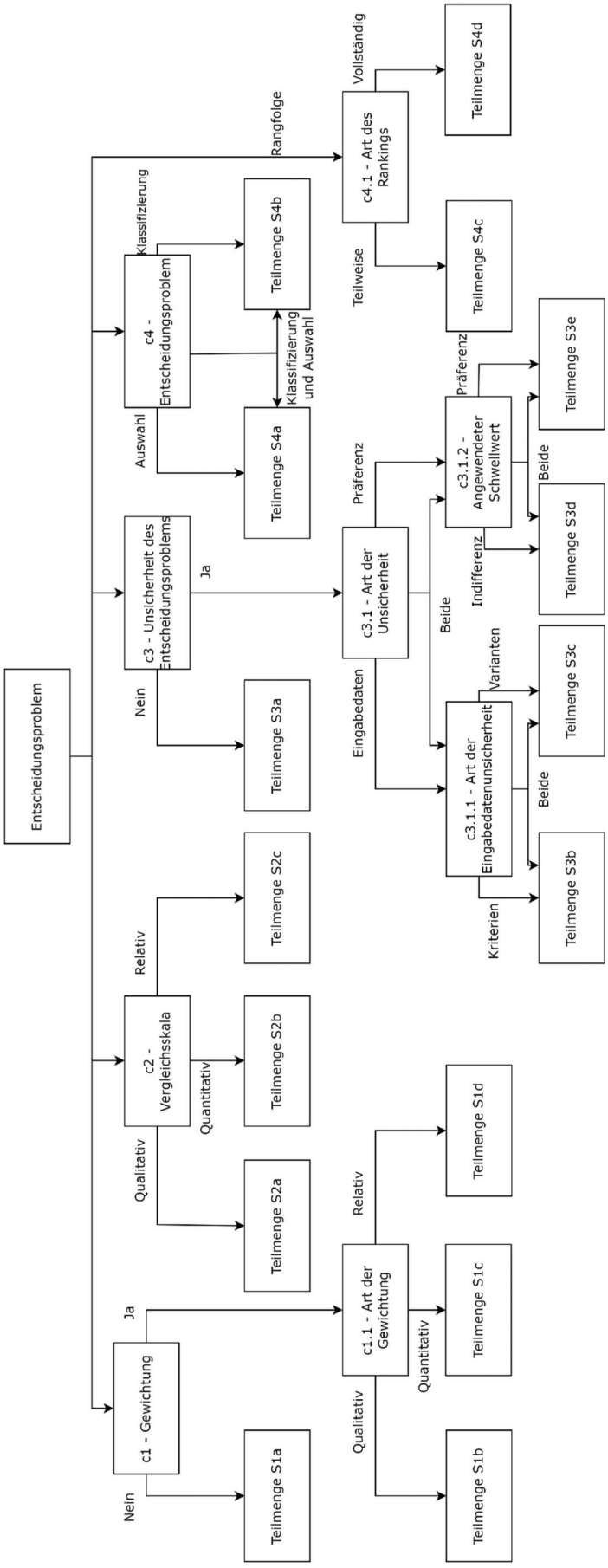

Abbildung 24: Entscheidungsbaum zur Auswahl einer MCDA-Methode nach Wątróbski et al. (2019, S. 116)

Anhang C – Skript zur Durchführung der paarweisen Vergleiche zwischen den Alternativen

Das Python-Skript, mit dem das Entscheidungsmodell von Wątróbski et al. angewendet wurde, findet sich auf dem dieser Arbeit beigelegten Datenträger unter der Bezeichnung „*Outranking.py*".

Anhang D – Quellcode der implementierten API

Der Quellcode der in dieser Arbeit entwickelten AI befindet sich auf dem dieser Arbeit beigelegten Datenträger unter der Bezeichnung *„API.zip*".

Anhang E – Ansicht des Wirkungskatalogs im VCC-Tool

Abbildung 25: Ansicht des Wirkungskatalogs nach Integration mit der implementierten API

Eidesstattliche Erklärung

Ich versichere an Eides statt durch meine Unterschrift, dass ich die vorstehende Arbeit selbständig und ohne fremde Hilfe angefertigt und alle Stellen, die ich wörtlich oder annähernd wörtlich aus Veröffentlichungen entnommen habe, als solche kenntlich gemacht habe, mich auch keiner anderen als der angegebenen Literatur oder sonstiger Hilfsmittel bedient habe. Die Arbeit hat in dieser oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen.

Krummesse, 20.06.2024,

Ort, Datum, Unterschrift