Konzeption und prototypische Umsetzung eines Frameworks für Case Management Applikationen

Masterarbeit

vorgelegt von André Zensen

Angefertigt im Studiengang Master of Science (M.Sc.) in Wirtschaftsinformatik an der Fachhochschule Bielefeld,

Fachbereich Wirtschaft und Gesundheit

Sommersemester 2019

Erstprüfer und Betreuer:	Prof. Dr. Jochen M. Küster
Zweitprüfer:	Prof. Dr. Hans Brandt-Pook

Inhaltsverzeichnis

Inhaltsverzeichnis ..... **I**
Abbildungsverzeichnis ..... **IV**
Tabellenverzeichnis ..... **VI**
Abkürzungsverzeichnis ..... **VII**
Kurzfassung **1**
1. Einleitung **2**
1.1 Motivation und Problembeschreibung 2
1.2 Abgrenzung 3
1.3 Methode und Aufbau der Arbeit 4
2. Theoretischer Hintergrund **6**
2.1 Case Management (CM) 6
2.1.1 Charakteristika von Case Management 6
2.1.2 Ausprägungen von CM 8
2.1.3 Anwendungsbeispiele für CM 8
2.1.4 CM im Vergleich zu traditionellen Ansätzen 10
2.2 Case Management Model and Notation (CMMN) 12
2.2.1 Metamodell und Strukturen 13
2.2.2 Übersicht über die grafischen Elemente 17
2.2.3 Ausführungssemantik 19
2.2.3.1 Zustände und Lebenszyklen 19
2.2.3.2 Verhaltensregeln durch *decorators* 22
2.2.3.3 Sentry-Konzept 23
2.2.4 Modellierungs- und Ausführungsunterstützung 27
2.3 Verwandte Arbeiten 29
2.3.1 Flexible und artefaktzentrierte Prozesse 29
2.3.2 Modellierung von CM Anwendungen mit CMMN 31
2.3.3 Case Management Implementierungen in der Forschung 32
3. Konzept eines Case Management Frameworks **34**
3.1 Übergeordnetes Ziel des Frameworks 34
3.2 Anforderungen für Case Management Applikationen 34
3.2.1 Basisanforderungen 34
3.2.2 Anforderungen an einen CM-Kern 35
3.2.3 Aus CMMN abgeleitete Anforderungen 35
3.2.4 Anforderungen an eine Serviceschicht 36
3.2.5 Anforderungen an eine Persistenzschicht 36
3.2.6 Anforderungen abgeleitet aus Problembereichen des traditionellen Prozessmanagement 36
3.3 Architektur und Komponenten einer CM Anwendung 37

3.3.1 Präsentationsschicht (*Web Frontend/Custom User Interface*) 38
3.3.2 Serviceschicht (*Case Management Services*) 39
3.3.3 Fall-Blaupausen und Instanzen (*Case Blueprints and Instances*) 40
3.3.4 Anwendungsspezifische Implementierungen (*Case Specific Implementations*) 40
3.3.5 Case Management Kern (*Case Management Core*) 41
3.3.6 REST-Schnittstellen (*REST-Interfaces*) 41
3.3.7 Persistenzschicht (*Persistence*) 42
4. Prototypische Umsetzung des Frameworks 43
4.1 Technologische Basis der Implementierung 43
4.1.1 Kurzeinführung in die Java Enterprise Edition 43
4.1.2 Überblick über verwendete Standards 44
4.1.3 Vaadin-CDI für grafische Benutzerschnittstellen von Tasks 45
4.2 Vereinfachte CMMN-Basis als Grundlage 46
4.3 Paketstruktur der Framework-Implementierung 47
4.4 Bausteine der Referenzarchitektur 49
4.4.1 Case Management Basiselemente 50
4.4.2 Rollensystem 51
4.4.3 Datenstrukturen 52
4.4.4 Sentry-Strukturen 53
4.5 Case Management Services 54
4.6 Adaptierte Entwurfsmuster für CM Anwendungen 55
4.6.1 Erzeugungsmuster Fabrik-Methode (*Factory Pattern*) in verschiedenen Anwendungsbereichen 55
4.6.1.1 Erzeugung von Fall-Instanzen aus Blaupausen 56
4.6.1.2 Erzeugung von IfPart-Implementierungen 57
4.6.1.3 Erzeugung von ProcessTask-Implementierungen 58
4.6.1.4 Erzeugung von *decorator* Rules 58
4.6.1.5 Erzeugung von Zustandsausprägungen 59
4.6.1.6 Erzeugung von CaseTask-Implementierungen 60
4.6.2 Entwurfsmuster Zustand (*State Pattern*) zur Abbildung der Zustände und Lebenszyklen 61
4.6.3 Entwurfsmuster Beobachter (*Observer Pattern*) zur Abbildung des Sentry-Konzepts 65
4.6.4 Entwurfsmuster Befehl (*Command Pattern*) für die Befehlsverarbeitung der Services 67
5. Praktische Anwendung des Frameworks 68
5.1 Fallbeispiel Urlaubsantrag 68
5.1.1 Modellierung in CMMN 68
5.1.2 Analyse des Modells 69
5.2 Übertragung des Modells in die Strukturen des Frameworks 70
5.2.1 Wurzelelement „Urlaubsantrag“ 70
5.2.2 Hinzufügen von Human- und ProcessTasks 71

5.2.3 Verknüpfung durch Sentry-Strukturen....72
5.2.4 Datenstruktur des CaseFileItems „Urlaubsantrag“ 73
5.2.5 Initialisierung des CaseModels und enthaltener Elemente 73
5.3 Individuelle Implementierungen....74
5.3.1 IfPart, IfPartImplementation und IfPartImplementationFactory....74
5.3.2 ProcessTask, ProcessTaskImplementation und ProcessTaskImplementationFactory 75
5.3.3 Maske für `HumanTask` „Antrag prüfen“ 76
5.3.4 Übersicht über Paketstruktur und erstellte Klassen des Fallbeispiels....77
5.4 Ausführung und Bearbeitung einer Fall-Instanz....78
5.4.1 Login, Fall-Instanziierung und Task-Beanspruchung....79
5.4.2 Erfassung der Antragsdaten 81
5.4.3 Antrag prüfen als Benutzer „John Admin“....82
5.4.4 Ausführung von ProcessTask-Implementierungen....83
5.4.5 Blick auf die Elemente der Instanz mit Hilfe einer REST-Schnittstelle....84
6. Kritische Betrachtung 85
6.1 Aufwandsschätzung für Entwicklungen mit dem Framework....85
6.2 Vorteile und Nachteile des erstellten Frameworks 86
6.3 Camunda im Kurzvergleich....88
6.4 Ausgewählte Forschungsimplementierungen im Vergleich 89
7. Zusammenfassung und Ausblick....90
Literaturverzeichnis.... XCIII

Abbildungsverzeichnis

Abbildung 1: Verlauf eines Falles....6
Abbildung 2: Spektrum des Prozessmanagements....10
Abbildung 3: Prozess- und Datenorientierung....11
Abbildung 4: Klassendiagramm CMMN Case....14
Abbildung 5: Klassendiagramm Informationsmodell....15
Abbildung 6: Klassendiagramm PlanItemDefinition....16
Abbildung 7: Grafische Elemente (eigene Modellierung)....17
Abbildung 8: Lebenszyklus eines CaseFileItems....19
Abbildung 9: Lebenszyklus der Klassen Stage und Task....20
Abbildung 10: Lebenszyklus von Milestone und EventListener....21
Abbildung 11: Zulässige Kombinationen von Elementen und *decorators*....22
Abbildung 12: Klassendiagramm decorators....23
Abbildung 13: Klassendiagramm Sentry-Konzept....24
Abbildung 14: Beispiel zur Veranschaulichung des Sentry-Konzepts (eigene Modellierung)....25
Abbildung 15: Einfache Verknüpfung durch Sentry (eigene Modellierung)....25
Abbildung 16: AND-Verknüpfung durch Sentrys (eigene Modellierung)....26
Abbildung 17: AND-Zusammenführung durch ein Sentry (eigene Modellierung)....26
Abbildung 18: OR-Struktur durch Sentrys (eigene Modellierung)....27
Abbildung 19: Grundlegende Architektur des Frameworks....37
Abbildung 20: Custom User Interfaces....38
Abbildung 21: Case Management Services....39
Abbildung 22: Case Blueprints....40
Abbildung 23: Case Specific Implementations....40
Abbildung 24: REST-Interfaces....41
Abbildung 25: Anwendung des Vaadin CDI-Navigator-Mechanismus....46
Abbildung 26: Paketstruktur der Framework-Implementierung....48
Abbildung 27: Framework Basiselemente....50
Abbildung 28: Framework Rollensystem....51
Abbildung 29: Framework Datenstrukturen....52
Abbildung 30: Framework Sentry-Strukturen....53
Abbildung 31: Service Implementierungen....54
Abbildung 32: Sequenzdiagramm Zugriff auf CaseFactory....56
Abbildung 33: Sequenzdiagramm IfPart und IfPartImplementationFactory....57
Abbildung 34: Sequenzdiagramm ProcessTask und ProcessTaskImplementationFactory....58
Abbildung 35: Sequenzdiagramm Rule und RuleExpressionFactory....59
Abbildung 36: Fabrik-Methode loadContextState der Klasse CaseFileItem....59
Abbildung 37: Sequenzdiagramm CaseTask und CaseTaskImplementationFactory....60

Abbildung 38: Klassendiagramm der Zustände für Stage und Task 61
Abbildung 39: Sequenzdiagramm State Pattern 63
Abbildung 40: Klassendiagramm Beobachter-Muster im Element-Kontext 64
Abbildung 41: Sequenzdiagramm Beobachter-Muster im Element-Kontext 66
Abbildung 42: Befehlsmuster im TaskService 67
Abbildung 43: CMMN-Modell "Urlaubsantrag" 68
Abbildung 44: CaseModel der Fall-Blaupause 70
Abbildung 45: Task-Elemente der Fall-Blaupause 71
Abbildung 46: Sentry-Strukturen der Fall-Blaupause 72
Abbildung 47: CaseFileItem-Struktur der Fall-Blaupause 73
Abbildung 48: Initialisierung des CaseModels der Fall-Blaupause 73
Abbildung 49: Individuelle IfPart-Implementierung 74
Abbildung 50: Konfiguration der IfPartImplementationFactory 75
Abbildung 51: Individuelle ProcessTask-Implementation 76
Abbildung 52: Vaadin-CDI-View und injizierte Services 76
Abbildung 53: Paketstruktur und Klassen der individuellen Implementierungen 77
Abbildung 54: Login als Jane Worker 78
Abbildung 55: Case-List mit neuer Fall-Instanz 79
Abbildung 56: Task-List für "Jane Worker" 80
Abbildung 57: Erfassung der Antragsdaten 81
Abbildung 58: Task-List für "John Admin" als Prüfer 82
Abbildung 59: Maske für HumanTask "Antrag prüfen" 83
Abbildung 60: Log-Einträge der ProcessTask-Implementierungen 83
Abbildung 61: Blick auf das aktualisierte Urlaubskonto 84
Abbildung 62: Elemente einer Fall-Instanz im JSON-Format 84

Tabellenverzeichnis

Tabelle 1: Ausgewählte forschungsorientierte CM-Implementierungen 32
Tabelle 2: Datenstruktur des CaseFileItems Urlaubsantrag 70
Tabelle 3: Lines of Code für das Fallbeispiel (ohne Masken) 85
Tabelle 4: Schätzwerte für LoC der Framework-Elemente.....86
Tabelle 5: Vor- und Nachteile der camunda Plattform 88

Abkürzungsverzeichnis

ACM	Adaptive Case Management
BPM	Business Process Management
BPMN	Business Process Model and Notation
CDI	Context and Dependency Injection
CM	Case Management
CMIS	Content Management Interoperability Services
CMMN	Case Management Model and Notation
CRUD	Create, Retrieve, Update and Delete
DCM	Dynamic Case Management
DMN	Decision Model and Notation
ECA	Event Condition Action
EJB	Enterprise Java Bean(s)
GPM	Geschäftsprozessmanagement
GSM	Guard Stage Milestone
HTTP	Hypertext Transfer Protocol
JAX-RS	Java API for RESTful Web Services
JEE	Java Enterprise Edition
JPA	Java Persistence API
JSON	JavaScript Object Notation
JSON-P	Java API for JSON Processing
JTA	Java Transaction API
LoC	Lines of Code
LTL	Linear Temporal Logic
OCL	Object Constraint Language
OMG	Object Management Group
ORM	Object Relational Mapping
PCM	Production Case Management
POJO	Plain Old Java Object
REQ	Requirement

REST	Representational State Transfer
SOAP	Simple Object Access Protocol
STAN	Structure Analysis for Java
TomEE	"Tomcat + Java EE = TomEE"
WFM	Workflow Management
WfMC	Workflow Management Coalition
WFMS	Workflow Management System
XML	eXtended Markup Language

Kurzfassung

„Traditionelles Geschäftsprozessmanagement“ mit bekannten und vorhersehbaren Abläufen lässt sich durch IT-Systeme unterstützen und automatisieren. Wird jedoch ein erhöhter Grad an Flexibilität gefordert, um komplexe Prozesse flexibel im jeweiligen Kontext zu bearbeiten, stoßen etablierte Ansätze schnell an ihre Grenzen. Ein Ansatz, um diese Grenzen zu überwinden, ist das Case Management. Es soll Wissensarbeitern ermöglichen, flexible Prozessstrukturen zu erstellen und unter Einbezug ihres Expertenwissens dynamisch bearbeiten zu können. Mit der „Case Management Model and Notation“ (CMMN) gibt es eine erste Modellierungssprache für Case Management Anwendungen und flexible Prozesse. Sie greift Case Management charakterisierende Elemente auf und ermöglicht eine grafische Modellierung. Ihr deklarativer Charakter und die ihr zugrundeliegende Ausführungssemantik ermöglichen es, flexible Prozesse, beziehungsweise Fälle, zu modellieren. Zwar existieren kommerzielle Lösungen, um Case Management Anwendungen zu realisieren, doch basieren diese nicht auf einem einheitlichen Modellierungsstandard. Hinzu kommen Lizenzkosten für herstellerabhängige Systeme zum Betreiben der Anwendungen. Alternativ kann ein Open Source Angebot eingesetzt werden. Die verfügbaren Plattformen sind aber sehr komplex und im Kern für stark strukturierte Prozesse ausgelegt. Eine weitere Möglichkeit ist es, von Grund auf Case Management Anwendungen neu zu entwickeln. Hier setzt die Arbeit an und entwickelt ein Konzept für die leichtgewichtige Implementierung von Case Management Anwendungen. Das Framework zeigt hierzu einerseits eine grundlegende Struktur für Case Management Anwendungen auf, andererseits bietet es als Software-Framework Strukturen zur Implementierung dieser an. Das Konzept des Frameworks baut hierzu auf vereinfachten CMMN Strukturen auf, setzt aber die Ausführungssemantik aus CMMN um. Mit Hilfe des Frameworks können Entwickler systematisch leichtgewichtige Applikationen erstellen. Für eine Anwendung werden CMMN Modelle zunächst in wiederverwendbare Bausteine des Frameworks transformiert. Die so entstandenen Blaupausen werden um individuelle Geschäftslogik und grafische Benutzerschnittstellen erweitert. Sie können anschließend durch das Framework instanziiert und ausgeführt werden. Der entwickelte Prototyp des Frameworks unterstützt bis auf einzelne Elemente und den Planungsmechanismus die CMMN Spezifikation. Ein mit dem Framework erstelltes Fallbeispiel zeigt, dass sich Entwickler durch das Framework auf die Umsetzung von Geschäftslogik und die Erstellung von grafischen Benutzerschnittstellen konzentrieren können. Die korrekte Ausführung und Steuerung von Fall-Instanzen übernimmt das Framework entsprechend der CMMN Ausführungssemantik. Das Framework ermöglicht durch seine starke Orientierung an CMMN die Entwicklung flexibler Prozesse und unterstützt Wissensarbeiter durch eine dynamische Ausführung dieser.

1. Einleitung

Dieses Kapitel erläutert die Motivation und Zielsetzung der Arbeit. Es wird der Problembereich umrissen und eine Abgrenzung vorgenommen. Abschließend wird die Forschungsmethode beschrieben und eine Übersicht über den Aufbau der Arbeit gegeben.

1.1 Motivation und Problembeschreibung

Sogenannte Wissensarbeit und wissensintensive Prozesse gewinnen vor allem in industrialisierten Informationsgesellschaften immer mehr an Bedeutung (s. [1], insb. S. 21ff.). Viele Routinearbeiten und bekannte Abläufe können durch IT- und Workflow-Systeme unterstützt und automatisiert werden. Flexible und unstrukturierte Arbeiten hingegen sind aber verstärkt notwendig, um komplexe Probleme zu lösen und tragen vermehrt zur Wertschöpfung bei. Auch Ausnahmen bekannter Prozesse, die nicht vor der Ausführung in Modellen erfasst sind oder erfasst werden können, müssen durch sogenannte Wissensarbeiter auf Basis ihres ausgeprägten Expertenwissens behandelt und gelöst werden.

„Klassisches“ Geschäftsprozessmanagement stößt hier durch seinen Fokus auf stark strukturierte und vordefinierte Prozessabläufe mit der einzelnen, meist isoliert betrachteten aktuellen Aufgabe im Zentrum an seine Grenzen [2]. Ein Ansatz, um diese Grenzen zu überwinden, ist das sogenannte Case Management.

Case Management ist ein neues Paradigma in der Wirtschaftsinformatik und zielt darauf ab, wissensintensive und flexible Prozesse transparenter und nachvollziehbarer zu machen. Es soll Wissensarbeiter bei der Ausführung und Lösung flexibler Prozesse unterstützen. Der am Ende sichtbare vollzogene Lösungsweg ist im Voraus nicht oder nur teilweise bekannt und entwickelt sich kontextbasiert durch die Wissensarbeiter. Er wird dabei während der Bearbeitung durch neu gewonnene Informationen und interne und externe Ereignisse beeinflusst.

Mit der „Case Management Model and Notation“ wurde im Mai des Jahres 2014 eine Modellierungssprache für Case Management veröffentlicht. 2016 erfuhr sie einige Überarbeitungen und ist seitdem in der Version 1.1 verfügbar. Sie ist der Versuch, einen offenen Standard für die Modellierung von Case Management Anwendungen zu etablieren. Neben einer einheitlichen grafischen Notationssprache enthält die zugrundeliegende Spezifikation verschiedene Modelle. Diese Modelle können in einem einheitlich definierten Austauschformat gespeichert und so mit verschiedenen Modellierungswerkzeugen und Ausführungsumgebungen verarbeitet werden.

Zwar existieren verschiedene proprietäre Case Management Systeme, wie etwa von den Anbietern IBM oder ISIS Papyrus, um Lösungen zu implementieren und durch IT zu unterstützen. Jedoch nutzen diese im Kern zur Modellierung und Ausführung von Case Management Applikationen keinen einheitlichen Standard. Entscheidet man sich für eine dieser Lösungen, bindet man sich zwangsläufig an die jeweilige Modellierungssprache, die nicht standardisiert und mit anderen Systemen austauschbar ist. Hinzu kommt oft die Notwendigkeit, weitere proprietäre Systeme zu erwerben, beziehungsweise Lizenzen für diese zu kaufen, wie etwa für spezifische Applikationsserver oder auch Datenbanksysteme, um das System zu betreiben. Zusammen resultiert dies in einer kostspieligen Herstellerabhängigkeit.

Eine Alternative ist es, eine Case Management Applikation auf Basis komplexer Geschäftsprozessmanagementplattformen zu erstellen, welche auch mit der „Case Management Model and Notation“ erstellte Modelle verarbeiten können. Zwar bestehen auch erste kostenlos verfügbare Lösungen für die Verarbeitung der neuen Notationssprache. Diese unterstützen aber nicht alle Bereiche der Spezifikation, sind sehr komplex und beinhalten daneben zusätzlich eine sehr komplexe Ausführungsumgebung für Modelle, die mit der Notationssprache „Business Process Model and Notation“ erstellt wurden. Durch die hohe Komplexität greift man früher oder später vermutlich auf kommerzielle Unterstützungsangebote der Anbieter zurück, um auf Basis dieser Systeme erstellte Applikationen zu verstehen, ausführen und schließlich warten zu können. Zusammen mit anbieterspezifischen Erweiterungen führt dies wiederum zu einer Herstellerabhängigkeit.

Eine weitere Möglichkeit ist es, Case Management Lösungen von Grund auf neu zu erstellen und beispielsweise in der Programmiersprache Java zu implementieren. Auf Grund der hohen Komplexität der neuen Notationssprache und der zugrundeliegenden Ausführungssemantik ist dieser Ansatz aber ebenso kostspielig und zeitaufwändig.

Hier setzt die vorliegende Arbeit an: Die „Case Management Model and Notation“ wird als zukünftiger Standard für die Modellierung von Case Management Anwendungen angesehen, der zentrale Charakteristika von Case Management ausdrückt. Auf Basis der Spezifikation der Notationssprache wird ein Framework erstellt. Dieses soll es Entwicklern ermöglichen, systematisch und strukturiert leichtgewichtige Case Management Applikationen erstellen zu können, die anschließend von Wissensarbeitern zur Bearbeitung und Lösung von Fall-Instanzen eingesetzt werden können.

Die zentrale Forschungsfrage ist somit, wie ein Framework für die Entwicklung leichtgewichtiger Case Management Applikationen konzipiert werden kann, das auf der „Case Management Model and Notation“ basiert. Hierzu wird sich der Frage aus verschiedenen Blickwinkeln genähert. Es wird geklärt, welche Anforderungen an solch ein Framework gestellt werden können, wie eine Architektur des Frameworks aufgebaut sein kann und wie die Ausführungssemantik mit Hilfe von etablierten Entwurfsmustern des Software Engineerings implementiert werden können. Außerdem wird geklärt, wie solch ein Framework praktisch eingesetzt werden kann, um Case Management Applikationen zu erstellen.

1.2 Abgrenzung

Ein besonderer Fokus der Arbeit liegt auf der Spezifikation der „Case Management Model and Notation“. Es können aber auf Grund der hohen Komplexität und des Umfangs der Spezifikation nicht alle Aspekte erläutert werden. Somit beschränkt sich die Darstellung auf die für das Framework wichtigsten Aspekte der Spezifikation und schafft ein Grundverständnis für die Notationssprache.

Ziel der Arbeit ist es nicht, eine ausgereifte oder vollumfängliche Plattform für Case Management Anwendungen zu erstellen, sondern ein Konzept und eine prototypische Implementierung für ein Framework zur Erstellung leichtgewichtiger Case Management Applikationen zu entwickeln, die auf der neuen Notationssprache basieren. Das Framework soll dabei eigenständig genutzt, oder in bestehende Applikationen eingebettet genutzt werden können.

1.3 Methode und Aufbau der Arbeit

Die Arbeit ist im Rahmen des Forschungsprojektes „Case Management Framework – CaMa-Frame“ an der Fachhochschule Bielefeld entstanden[1]. Die Arbeit setzt die in der (deutschen) Wirtschaftsinformatik übliche konstruktivistische Methode der Prototyperstellung ein. Ausgehend von einer Anforderungsanalyse für das Framework wird ein Konzept für dieses erstellt. Auf Basis des Konzeptes wird ein Prototyp erstellt. Das entstandene Framework-Artefakt wird schließlich eingesetzt, um auf Basis eines Fallbeispiels eine Applikation zu erstellen und mit ihr zu arbeiten. Durch die Anwendung wird das Framework evaluiert. Der Aufbau der Arbeit wird im Folgenden beschrieben.

In Kapitel 2 wird zunächst der theoretische Hintergrund geschaffen. Zunächst werden Charakteristika und Ausprägungen von Case Management aufgearbeitet und durch Anwendungsbeispiele verdeutlicht, bevor ein Vergleich zu einem „traditionellem“ Geschäftsprozessmanagement gezogen wird. Außerdem wird ein Grundverständnis für die „Case Management Model and Notation“ geschaffen, welche die Basis für das Framework darstellt. Neben beleuchteten Metamodellen und Strukturen der Notationssprache wird eine Übersicht über grafische Elemente gegeben. Daneben liegt ein besonderer Fokus auf der Erläuterung der Ausführungssemantik. Ferner werden bestehende Modellierungswerkzeuge und Ausführungsumgebungen für die Notationssprache kurz beleuchtet.

Eine Auswahl verwandter Arbeiten rundet das zweite Kapitel ab. Es werden der Stand der Technik und Forschungsarbeiten rund um flexible Prozesse, Modellierungen auf Basis der neuen Notationssprache und Case Management Implementierungen aus der Forschung aufgezeigt.

Kapitel 3 breitet das grundlegende Konzept für das Framework aus. Ausgehend von Anforderungen für Case Management Applikationen wird eine Architektur des Frameworks erstellt, um eine grundlegende Struktur zu schaffen. Die aus den Anforderungen entstandenen Architekturkomponenten werden detailliert beschrieben.

In Kapitel 4 wird die prototypische Umsetzung des Frameworks dargestellt. Es wird die technologische Basis für die Implementierung erläutert, bevor Bausteine zur Umsetzung der Anforderungen den verschiedenen Architekturaspekten zugeordnet werden. Ein besonderer Fokus liegt hierbei wie im zweiten Kapitel auf der Umsetzung der Ausführungssemantik der Notationssprache. Verschiedene Entwurfsmuster wurden identifiziert und für das Framework angepasst übernommen. Beispielhaft wird die Anwendung der verschiedenen Muster in den verschiedenen Bereichen des Frameworks gezeigt.

Kapitel 5 zeigt, wie ein Fallbeispiel mit Hilfe des Frameworks implementiert wird. Ausgehend von einem grafischen Modell wird mit Hilfe der Framework-Komponenten eine Case Management Applikation erstellt. Es wird aufgezeigt, welche individuellen Implementierungen nötig sind, um das Modell umsetzen. Anschließend wird ausschnittsweise gezeigt, wie mit dem Framework die Anwendung ausgeführt und Fall-Instanzen bearbeiten werden können.

Kapitel 6 widmet sich einer kritischen Betrachtung des erstellten Frameworks. Der Einsatz des Frameworks wird vor dem Hintergrund der erstellten Beispielapplikation betrachtet. Abgeleitet aus dem

[1] Siehe https://www.fh-bielefeld.de/wug/forschung/ag-pm/cama-frame.

Fallbeispiel wird aufgezeigt, welcher Aufwand zu erwarten ist, um Case Management Anwendungen mit dem Framework zu erstellen. Daneben werden Vorteile und Nachteile des Frameworks aufgezeigt. Zudem wird dargestellt, welcher Aufwand zur Erstellung einer Case Management Applikation auf Basis des Frameworks zu erwarten ist. Abgerundet wird die kritische Betrachtung durch einen Kurzvergleich mit der BPM-Plattform „camunda“ und ausgewählten Implementierungen aus der Forschung.

Kapitel 7 fasst abschließend die Ergebnisse der Arbeit zusammen und gibt einen Ausblick auf weitere Arbeiten, um den erstellten Prototyp des Frameworks weiter auszubauen.

Der Arbeit liegt eine CD bei, die das Framework und das implementierte Fallbeispiel enthält.

2. Theoretischer Hintergrund

Dieses Kapitel stellt den theoretischen Rahmen und Hintergrund der Arbeit dar. Hierzu werden zunächst in Abschnitt 2.1 Definitionen und Strömungen des Begriffs Case Management sowie Anwendungsbereiche von Case Management beleuchtet. Das Themengebiet Case Management wird aus verschiedenen Blickwinkeln betrachtet und mit traditionellen Ansätzen des Geschäftsprozessmanagements verglichen. In Abschnitt 2.2 wird die „Case Management Modeling Notation“ als Standard für die Modellierung von CM Anwendungen vorgestellt. Neben den grundlegenden Strukturen und Elementen der grafischen Palette ist ein besonderer Fokus hierbei auf die komplexe Ausführungssemantik gelegt. Abschnitt 2.3 gibt einen Überblick über verwandte Arbeiten.

2.1 Case Management (CM)

In diesem Abschnitt wird in das Thema Case Management eingeführt. In [3] werden verschiedene Aspekte des Case Managements betrachtet, die in den folgenden Abschnitten näher beleuchtet werden.

2.1.1 Charakteristika von Case Management

In diesem Abschnitt soll sich dem Begriff Case Management (seltener auch als Case Handling bezeichnet [4]) genähert werden. Hierzu werden verschiedene Charakteristika herausgestellt.

Ein Case Management kann im Kern als Management oft langlebiger, kollaborativ bearbeiteter Prozesse verstanden werden, die Wissen, Inhalte und Ressourcen koordinieren, um einen Fall zu einem spezifischen Ziel oder Abschluss zu führen. Der Ausführungspfad wird hierbei durch menschliche Entscheidungen während der Bearbeitung beeinflusst und ist nicht immer vorbestimmt [5]. Im Zentrum steht der Fall (beispielsweise ein Schadensfall, oder eine Kundenanfrage), dessen Status von Datenobjekten, ihren Werten und ihren Abhängigkeiten bestimmt und durch Wissensarbeiter vorangetrieben wird (vgl. [6]).

Abbildung 1: Verlauf eines Falles

Abbildung 1 (angelehnt an [7], S. 79) veranschaulicht diese Arbeitsweise: Am Anfang werden für einen zu lösenden Fall Meilensteine und ein übergreifendes Ziel definiert. Die geöffneten Ordner symbolisieren Aktivitäten, wie etwa das Hinzufügen von neuen Daten. Der Fall wird von Daten und der Expertise der Fallbearbeiter maßgeblich beeinflusst, bis schließlich das Ziel erreicht wird. Dabei verläuft die Bearbeitung nicht linear (siehe Pfeile zwischen den Ordner-Symbolen): Rücksprünge zu vorherigen Aufgaben können vollzogen werden, oder Aufgaben auch übersprungen werden.

Case Management als fester Begriff findet seinen Ursprung in den Pflegewissenschaften[2]. Der Begriff wird hier als ganzheitliche Betrachtung eines Patientenfalls und der Abstimmung der (weiterführenden) Behandlungsmaßnahmen unter Absprache mit und Berücksichtigung der individuellen Bedürfnisse des Patienten verstanden, um eine maßgeschneiderte Versorgung zu organisieren. Eine Fallbearbeitung kann sich hierbei in vier Schritte gliedern. Eine strukturierte Analyse des Ist-Zustands des Patienten und der verfügbaren Ressourcen führt zur Definition von Meilensteinen und Zielen. Geeignete Maßnahmen zur Erreichung dieser werden ausgewählt, vertraglich festgehalten und realisiert. Eine mit den Fallbeteiligten koordinierte Umsetzung der Maßnahmen wird schließlich hinsichtlich der angestrebten und erreichten Ziele geprüft (vgl. [8], S. 24).

Swenson definiert Case Management als Methode oder Praxis, die Arbeit um eine organisierte Sammlung relevanter Informationen koordiniert. Eine Sammlung entspricht hierbei dem Begriff eines Falles. Ein Fall steht im Zentrum für alle anfallenden Arbeiten und Prozesse, die auf Basis der relevanten Informationen initiiert und ausgeführt werden. Darüber hinaus dient der Fall der Aufzeichnung des bisherigen Verlaufs (vgl. [9], S. 4).

Der Begriff Case (Fall) lehnt sich an eine Aktenmetapher an: Im Rahmen von Aktivitäten werden in einer (digitalen) Akte strukturierte und unstrukturierte Dokumente und Informationen gesammelt, bearbeitet und verwaltet, die zur Zielerfüllung in bestimmten Zuständen benötigt werden. Diese Akte steht im Zentrum eines Falls. Die Beschaffung der Daten und die Bearbeitung dieser kann wiederum Unterprozesse oder weitere Fälle anstoßen, die hierarchisch dem ursprünglichen Fall und einem übergeordneten Ziel untergeordnet sind. Weitere Definitionen finden sich in [10].

Falltypen

Fälle können in die drei Typen „mass“, „regular“ und „special“ unterteilt werden, wie eine Studie über die IT-Unterstützung von Fällen in Justizorganisationen aus dem Jahre 2006 zeigt (vgl. [11], S. 28). Die Unterscheidung erfolgt unter den Gesichtspunkten der automatischen Ausführung und Bekanntheit der möglichen anfallenden Aufgaben: Tritt der Fall sehr häufig auf und ist sehr strukturiert (zum Beispiel basierend auf einem genormten rechtlichen Verfahren), entspricht er einem „mass case“, wie etwa bei standardisierten Mahnverfahren. Aufgaben sind bekannt und können in Teilen automatisch durch IT-Systeme erledigt werden. Gewöhnliche Fälle, die aber nicht (vollständig) automatisierbar sind und (verstärkt) menschliche Entscheidungen benötigen, können als „regular cases“ angesehen werden. Diese machen Gebrauch von Meilensteinen, um den Fall zu planen, zu steuern und durch die Erreichung der Meilensteine das Gesamtziel zu erreichen. Aufgaben können entsprechend der zu erreichenden und bereits erreichten Meilensteine gewählt und erledigt werden. Außergewöhnliche Fälle, sogenannte „special cases“, sind unbekannte oder mit der geringsten Routine behaftete Fälle.

Nachdem grundlegende Charakteristika von CM herausgestellt wurden, werden im nächsten Abschnitt Ausprägungen von CM beschrieben, die sich nicht auf eine Patienten- oder Rechtsdomäne beschränken und weiter gefasst sind.

[2] Siehe beispielsweise auch die Definition der Case Management Society of America (CMSA) unter http://www.cmsa.org/who-we-are/what-is-a-case-manager/.

2.1.2 Ausprägungen von CM

In diesem Abschnitt werden zentrale Ausprägungen von CM beschrieben. In den letzten 10 Jahren haben sich zwei grundlegende Strömungen des Case Managements herauskristallisiert: Adaptive Case Management (ACM) und Production Case Management (PCM). Diese unterscheiden sich primär in der Flexibilität der Prozessgestaltung während der Ausführung. Daneben gibt es noch den in der Forschung weniger verwendeten Begriff des Dynamic Case Management (DCM).

Adaptive- und Dynamic-Case Management

DCM-Systeme sollen Wissensarbeitern zur Laufzeit eine kontextabhängige Anpassung bestehender Geschäftsprozessspezifikationen ermöglichen (vgl. [2; 12]). Der Begriff entspricht im Kern dem Begriff des ACM, welches als flexibles System zur Gestaltung wissensintensiver Prozesse beschrieben wird [12]. Der Unterschied zwischen DCM und ACM liegt darin, dass ACM hinsichtlich der Flexibilität noch weiter geht. ACM erlaubt es Wissensarbeitern, einen Fall nicht nur anzupassen, sondern ad-hoc selbst zu gestalten. Aktivitäten, Ziele und hierfür benötigte Daten und Regeln können zur Laufzeit erstellt und angepasst werden. ACM bietet dem Wissensarbeiter somit eine „Do It Yourself"-Arbeitsweise und größtmögliche Flexibilität an (siehe auch [9; 12–15]).

Production Case Management

Im Gegensatz zu ACM macht PCM Gebrauch von festen Strukturen und bewahrt dabei Flexibilität hinsichtlich konkreter Ausführungspfade. PCM Anwendungen stellen den Wissensarbeitern verschiedene vordefinierte Aktivitäten zur Verfügung, die je nach Fall kombiniert eingesetzt werden können. Bekannte Muster benötigter Aktivitäten können im Vorfeld festgelegt und modelliert werden. Bei Bedarf werden diese zur Ausführungszeit dynamisch von Wissensarbeitern zusammengestellt, um ein Ziel zu erreichen (vgl. [9], S. 11).

PCM kann somit als "Best Practice aus ACM" angesehen werden, oder als Arbeitsmuster und (Teil-)Prozessschablonen verstanden werden, die aus erprobten und im Groben bekannten, wissensintensiven Prozessen entstanden sind. Wiederkehrende Aktivitäten, benötigte Daten und Ziele zur Lösung eines Falls werden zur Bearbeitung angeboten. Aus diesem Angebot, das zur Design-Zeit in Zusammenarbeit mit IT-Mitarbeitern erstellt wird, wählt die Wissensarbeiterin zur Ausführungszeit dann eine individuelle Konfiguration zur Bearbeitung und Lösung des Falls. Der Einsatzbereich ist hierbei auf domänenspezifische Lösungen beschränkt, dafür aber zum Beispiel auch grafisch im Vorfeld darstellbar und bietet konkrete Lösungsansätze für einen Fall an (vgl. [14; 16]).

2.1.3 Anwendungsbeispiele für CM

Neben Anwendungen im Gesundheitswesen und der Patientenversorgung (s. [8; 17]), bietet sich das Konzept des Case Management für die verschiedensten Anwendungsbereiche und Probleme an, da prinzipiell jeder Vorgang als ein Fall betrachtet und bearbeitet werden kann. Insbesondere wissensintensive Arbeit, die wenig strukturiert oder schlecht vorhersehbar abläuft, kann von Case Management-Lösungen profitieren und diese Wissensarbeiter unterstützen. Das „Korsett" der strikten Ablaufreihenfolge im „klassischen" Prozessmanagement wird so gelockert, aber zeitgleich können

definierte Regeln für die Zielerreichung eingehalten werden. Die durchgeführten Abläufe – die sich oft verschiedener Hilfsmittel und unterschiedlicher, nicht integrierter Systeme bedienen und in den Köpfen der Wissensarbeiter abspielen – können so nachvollziehbar gestaltet und auditierbar werden.

In der Literatur finden sich entsprechend der vielfältigen Einsatzmöglichkeiten die unterschiedlichsten Anwendungsbeispiele aus verschiedenen Branchen, von denen eine Auswahl in diesem Abschnitt kurz vorgestellt werden soll. Der Nutzen der jeweiligen Lösungen liegt unter anderem in den Bereichen der Kostenreduzierung, Zeiteinsparungen, einer Steigerung der Kundenzufriedenheit, Regeleinhaltung und auch darin, eine Basis für die Analyse durchlaufener Prozesse zu schaffen.

Vision Service Plan, eine im englischsprachigen Raum operierende Krankenversicherung mit über 70 Millionen Kunden, implementierte eine ACM-Lösung im Bereich des Kundenmanagements. Diese unterstütze die Mitarbeiter bei der Abwicklung von Versicherungsansprüchen, Beschwerden, Anfragen von Ärzten und bei Betrugsfällen. Eine weitere ACM-Lösung für das Kundenmanagement, das funktionell über ein Customer-Relationship-Management-System hinausgehe, implementierte die australische Rentenfondgesellschaft QSuper in ihrem Kundenservice, um eine integrierte „360-Grad-Sicht“ auf die Kunden zu erhalten. Durch die Lösung wurden verschiedene Legacy-Systeme abgeschafft und die IT-Infrastruktur modernisiert (s. [10], S. 17ff. und S. 23ff.).

Auch in der Softwareentwicklung finden Case Management-Lösungen ihre Anwendung. Der vorgesehene Prozess der Softwareentwicklung in einer deutschen Bank wird mit Hilfe einer Case Management-Lösung begleitet. Meilensteine und Qualitätsziele werden definiert und im System hinterlegt. Arbeitszuweisung und ihre Überwachung können flexibler gestaltet werden, als mit gängigen Lösungen. Der Entwicklungsprozess ist auditierbar und orientiert sich an den zentral organisierten Vorgaben und Artefakten [18].

Eine auf CMMN-Modellen (vgl. Abschnitt 2.2) basierende Lösung findet ihre Anwendung im Bereich des Enterprise Architecture Management, um diese zu planen und zu realisieren. In einer webbasierten Laufzeitumgebung können laufende Prozessinstanzen mit einem eingebetteten grafischen Modellierungswerkzeug manipuliert werden [19].

Um komplexe Not- und Krisenfälle mit unvorhersehbaren Ereignissen besser bewältigen zu können, wurde ein domänenspezifischer ACM-Ansatz im Rahmen eines sogenannten Emergency Response Systems als Erweiterung einer kommerziellen CM-Lösung von IBM erprobt [20].

Auch in der Lehre sind CM-Lösungen denkbar. In [21] wurde eine ACM-Lösung als e-Learning-Plattform zum kollaborativen Lernen prototypisch erprobt.

Eine weitere Implementierung stellt ein System aus zwei Komponenten zur Unterstützung der Arbeit der norwegischen Behörde für Lebensmittelsicherheit dar. Das System bietet die Möglichkeit zwischen ACM- und PCM-Ansatz mit konfigurierbaren Prozessschablonen zu wählen. Wissensarbeiter sind unter anderem Tierärzte, Biologen und Ingenieure. Eingesetzt wird das System für Routineprüfungen, aber auch für Ausnahmefälle mit gefährlichen biologischen Risiken [22].

Weitere Fallstudien und Anwendungsbeispiele von CM sind im Rahmen des jährlichen „Workflow Management Coalition (WfMC) Awards for Excellence in Case Management“, der innovative ACM Implementierungen nominiert und kürt, zu finden. [23] listet verschiedene Fallstudien als Gewinner des Jahres 2012 auf. Die aufgeführten Lösungen[3] unterstützen das Kundenmanagement unter anderem in der Finanzbranche, die Flugzeug- und Passagierabfertigung am Flughafen Heathrow in Großbritannien, unterstützen die Arbeit im Gerichtswesen (siehe hierzu auch [24] für eine Implementierung zur Unterstützung der Strafgerichte in Los Angeles, USA), das Patienten- und Pflegemanagement in einem Krankenhaus, oder auch das Management von DNA-Proben von Schwerverbrechern zur Unterstützung der Polizeiarbeit in Los Angeles.

2.1.4 CM im Vergleich zu traditionellen Ansätzen

Im Fokus „traditioneller Ansätze“ des Geschäftsprozessmanagements (GPM) oder Workflow-Managements (WFM) stehen im Vergleich zu CM weniger flexible, oft vereinfachte, sehr strukturierte und sich häufig wiederholende und flussbasierte Prozesse [25]. Einmal modelliert, können diese als Instanzen gestartet und entlang des vorgesehenen Ausführungspfads bearbeitet werden. Die Bearbeitung kann teilweise oder vollständig automatisiert werden, wie etwa bei einem Zahlungsvorgang.

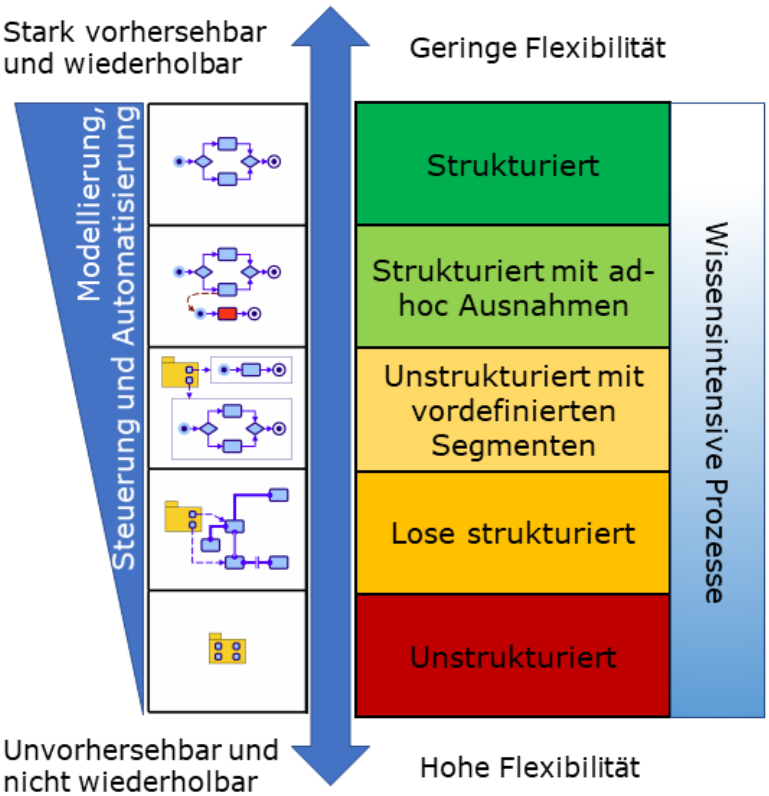

Abbildung 2: Spektrum des Prozessmanagements

Abbildung 2 (angepasst übernommen aus [6], S. 6) verdeutlicht das Spektrum des Prozessmanagements. Vorhersehbare und wiederholbare Prozesse können vollständig modelliert werden, in bekannten Bahnen gesteuert und (in Teilen) automatisiert werden. Sie bieten allerdings die geringste Flexibilität und

[3] Siehe auch http://www.adaptivecasemanagement.org/awards_2012_winners.html für weitere Gewinner des Jahres, insbesondere auch die „Honorary Category“ des PCM-Ansatzes.

Unterstützung für wissensintensive Prozesse, wie sie im CM Ausdruck finden. Dieses Ende des Spektrums kann als Ausdruck „traditioneller Ansätze“ gesehen werden. Das andere Ende des Spektrums beschreibt ACM: Lose strukturierte oder unstrukturierte Modelle (sofern es noch möglich ist, Modelle zu erstellen) bieten die höchste Flexibilität und Unterstützung für wissensintensive Prozesse an.

Im „traditionellen“ Prozessmanagement übernimmt ein WFMS die korrekte Ablaufverarbeitung der definierten Aktivitäten entlang des vorgegebenen Ausführungspfads. Hierzu bedient es sich beispielsweise eines *Token*-Systems (wie etwa im Modellierungsstandard für Geschäftsprozesse, „Business Process Model and Notation“ (BPMN) [26]), um gerade auszuführende Aktivitäten zu markieren. Der *Token* wandert von einer Startmarkierung zur nächsten Aktivität. Nach Abschluss einer Aktivität verfolgt er sukzessive den Ausführungspfad weiter, über andere Aktivitäten, bis zu einer Endmarkierung. Er kann für parallele Pfade geteilt und wieder zusammengeführt werden.

Auch wenn durch Abzweigungen Einfluss auf den Ausführungspfad genommen werden kann, stehen alle Pfade bereits fest; auf Aktivität A folgt entweder Aktivität B, oder Aktivität C. Rücksprünge oder das Überspringen von Aktivitäten ist außerhalb modellierter Pfade und auf Entscheidungen (oft durch die Abfrage von Datenwerten) basierender Abzweigungen nicht vorgesehen. Die Bearbeitung von Aufgaben ist somit auch zeitlich eingeschränkt: Ist ein Pfad von Aktivität A zu Aktivität X zu einem bestimmten Zeitpunkt nicht vorgegeben, kann Aktivität X erst später oder gar nicht erreicht werden.

Die Bearbeitung von Aufgaben findet meist durch einen zur Ausführung der Aktivität berechtigten Rollenträger statt. Aktivitäten werden dann häufig vom Workflowsystem automatisch in die Aufgabenliste der Rollenträgerin übertragen, welche ihre Übersicht und Entscheidungsmöglichkeiten einschränkt.

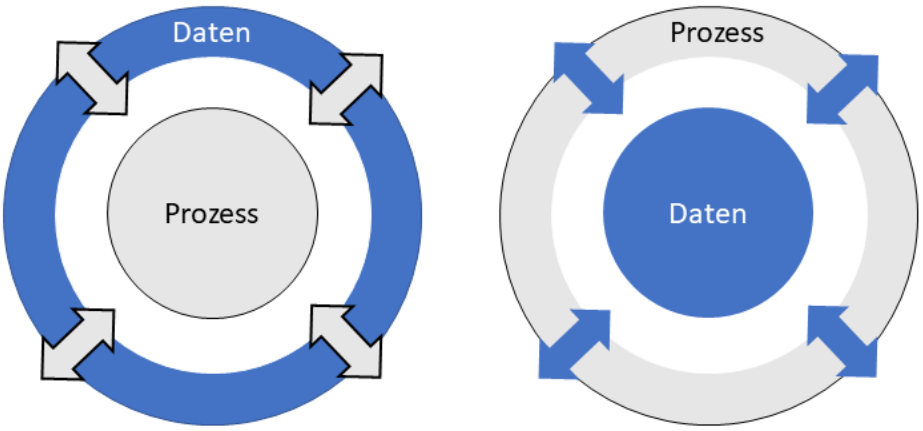

Abbildung 3: Prozess- und Datenorientierung

Bei der Bearbeitung einer Aufgabe sind typischerweise nur die Daten sichtbar, die zur Erfüllung der Aufgabe benötigt werden. Eine Übersicht über den Prozessverlauf oder über die verfügbaren Daten – also ein über die Aktivität hinausgehender größerer Kontext – ist oft nicht vorgesehen. Daten spielen „klassisch“ eine untergeordnete Rolle im Gegensatz zur stark datenbasierten Ablaufgestaltung in CM.

Abbildung 3 (angepasst übernommen aus [27], S. 214) hebt diesen Unterschied hervor: „Klassische“ Ansätze stellen den Prozess ins Zentrum, wohingegen CM Daten als Grundlage ins Zentrum stellt. Eine tiefergehende Betrachtung der Unterschiede zwischen imperativen und deklarativen Ansätzen findet sich beispielsweise in [28] und [29].

Unzulänglichkeiten „klassischer“ Ansätze

Verglichen mit „klassischen“ Workflowmanagementsystemen (WFMS) und deren Arbeitsweise soll Case Management vier zentrale Probleme der Realisierung von CM Lösungen unter Verwendung gängiger WFMS adressieren (vgl. [25], S. 131):

1) „Context tunneling“ wird in CM vermieden, das heißt alle Informationen des zu bearbeitenden Falls sind für (berechtige) Fallmitarbeiter einsehbar und der gesamte Kontext sichtbar. In der „klassischen Arbeitsweise“ sind Informationen typischerweise auf einzelne zu bearbeitende Aktivitäten beschränkt. Ein größerer Kontext fehlt und mögliche Alternativen zur Erreichung von Meilensteinen und Zielen werden nicht gesehen.
2) CM macht notwendige Aktivitäten in Abhängigkeit von Informationen verfügbar. Aktivitäten sind nicht durch einen festen Ausführungspfad und vorherige Aktivitäten bestimmt. „Klassische“ Ansätze verfolgen deterministische Abläufe mit geringem Freiheitsgrad in der Ausführung und stellen meist nur für die einzelne, gerade bearbeitete Aktivität benötigte Daten zur Verfügung.
3) Die Arbeitsteilung beschränkt sich in CM nicht auf eine zur Ausführung autorisierte Rolle, sondern sieht verschiedene Rollen zur gemeinsamen Bearbeitung eines Falles vor. Dies unterstreicht den kollaborativen Charakter von Wissensarbeitern als Ausführende.
4) Daten und Informationen können vor oder nach der Ausführung einzelner Aktivitäten hinzugefügt und bearbeitet werden, also unabhängig von bestimmten Aktivitäten. Dies stellt die (digitale) Akte ins Zentrum eines Falls. „Klassische“ Workflows machen die Bearbeitung und Einsicht von Daten abhängig von einer bestimmten, gerade zu bearbeitenden Aktivität.

2.2 Case Management Model and Notation (CMMN)

Nachdem CM charakterisiert wurde, soll die „Case Management Model and Notation“ (CMMN) [30] der Object Management Group (OMG) vorgestellt werden. Alle folgenden Ausführungen beziehen sich auf die aktuelle Version 1.1, die im Jahre 2016 veröffentlicht wurde.

CMMN strebt an, ein Standard für die Modellierung von Case Management zu werden. CMMN greift hierzu die im vorherigen Kapitel 2.1 beschriebenen Charakteristika von CM auf. Unter der Schirmherrschaft der OMG wurde CMMN in Zusammenarbeit mit CM-Praktikern und -Theoretikern entwickelt, um den Unzulänglichkeiten „traditioneller“ Ansätze Rechnung zu tragen und einen Standard für flexiblere Prozesse und Case Management zu schaffen. CMMN kann dem PCM-Ansatz zugeordnet werden (siehe Abschnitt 2.1.2), da vor der Ausführung modellierte Strukturen verwendet werden.

In der Spezifikation der CMMN wird CM einleitend als Vorgänge bezüglich eines Geschäftsobjekts in einer bestimmten Situation beschrieben, um einen gewünschten Ausgang zu erreichen. Dabei kann ein

Fall durch ad-hoc Entscheidungen gelöst werden. Erfahrung im Lösen ähnlicher Fälle führt mit der Zeit zu wiederholbaren Mustern, um einen Fall zu lösen. Dies führe zur Praxis des Case Managements. Es kann sich hierbei – breiter gefasst als die Definition aus den Pflegewissenschaften – um ein Geschäftsobjekt handeln, um das sich Handlungen, Vorgänge und Maßnahmen orientieren, um ein bestimmtes Ziel zu erreichen. Aus Erfahrungswerten in der Bearbeitung ähnlicher vergangener Fälle soll mit CMMN ein flexibel ausführbares Modell aus bekannten Strukturen erstellt werden können (vgl. [30], S.5f.).

Als Zielgruppe beschreibt CMMN Business Analysten, die Modelle zur Design-Zeit zusammen mit Fallexperten und Endanwendern entwickeln. Geschäftsregeln fließen ein, um erprobte Vorgänge abzubilden und Fälle standardisieren zu können. Die Flexibilität während der Ausführung (die der Design-Zeit also nachgeordnet ist) soll dabei nicht wie in einem „klassischen“ WFMS eingeschränkt werden: Der Ablauf ist stärker durch Fallbearbeiter und den einsehbaren Kontext gesteuert: Das Expertenwissen von Mitarbeitern, Daten und Entscheidungen auf Basis dieser Daten rücken in den Vordergrund, um einen Fall in einer bestimmten Situation flexibel zu lösen (vgl. [30], S. 7).

In den folgenden Abschnitten werden zunächst die grundlegenden Strukturen von CMMN-Modellen betrachtet, bevor eine Übersicht über die grafischen Elemente gegeben wird. Anschließend wird ein besonderer Fokus auf die Ausführungssemantik gelegt. Schließlich werden bestehende Möglichkeiten zur Modellierung mit CMMN und Ausführung von CMMN Modellen kurz betrachtet.

2.2.1 Metamodell und Strukturen

Die CMMN Spezifikation vereint Metamodelle, eine Notation zur grafischen Darstellung von Fällen, sowie ein XML Modell für den Austausch der erstellten Modelle zwischen verschiedenen Werkzeugen und Ausführungssystemen. Die für diese Arbeit wichtigsten Metamodelle und grundlegenden Strukturen werden folgend dargestellt. Die CMMN Spezifikation ist durch mehrere Abstraktionsebenen sehr komplex und wird nicht vollständig behandelt. In den sich anschließenden Abschnitten wird auf spezifische Bereiche und weitere Modelle eingegangen, die für die weitere Arbeit wichtig sind. Auf Definitionen für die Import- Erweiterungsmechanismen von CMMN wird nicht weiter eingegangen[4].

Bezieht sich der Text auf Klassen, Objekte (beziehungsweise grafische CMMN Elemente), Methoden oder Attribute, werden diese im Text durch den Font `Consolas` hervorgehoben.

[4] Siehe Kapitel 5.1 der Spezifikation.

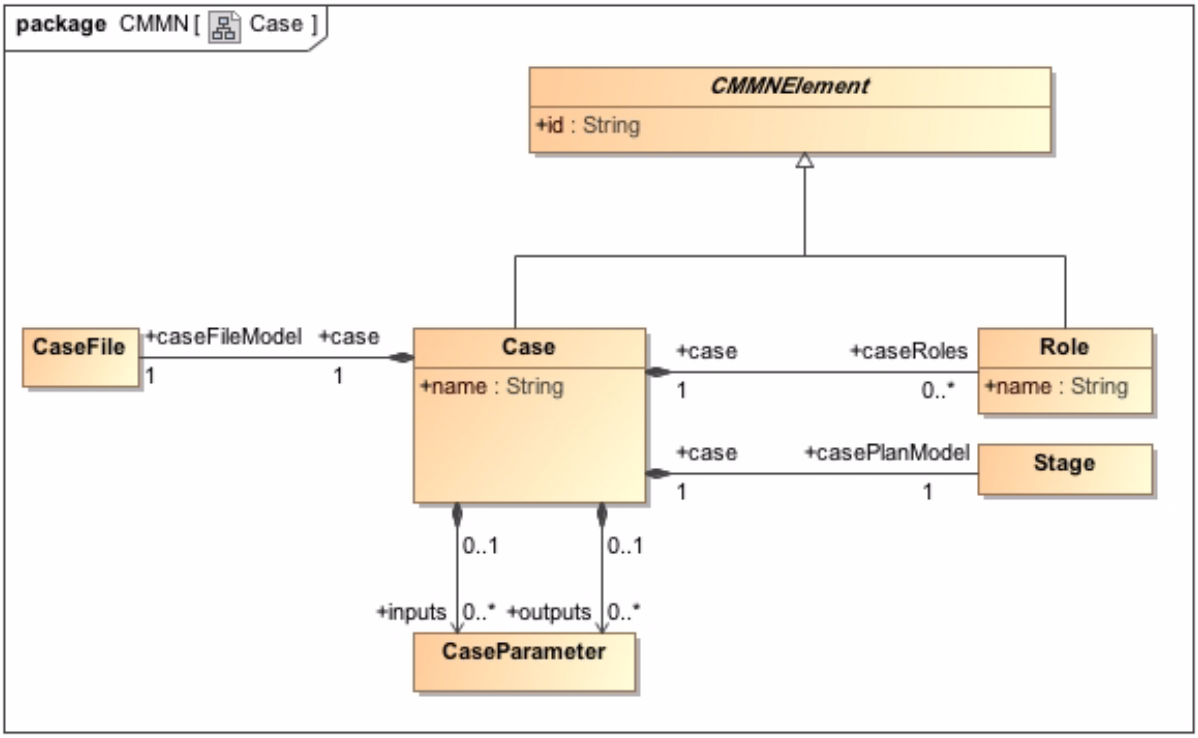

Abbildung 4: Klassendiagramm CMMN Case

Abbildung 4 (s. [30], S. 19ff.) zeigt das Klassendiagramm der Spezifikation für zentrale Elemente um die Klasse `Case`. Wie viele andere Klassen, erbt `Case` von der abstrakten Klasse `CMMNElement` und hierdurch einen im Modell eindeutigen Identifikator `id`. Einem jedem `Case` ist ein `CaseFile` zugeordnet, welches die (digitale) Akte eines Falls wie in Kapitel 2.1 beschrieben darstellt. In ihr werden Daten und Dokumente gespeichert, beziehungsweise referenziert. Ein- und Ausgangsparameter können über die Klasse `CaseParameter` definiert werden.

Daneben kann ein `Case` (optionale) Rollen in Form der Klasse `Role` besitzen, die frei über ihren Namen definiert werden können. Rollen können genutzt werden, um den Zugriff auf bestimmte Aktivitäten und Aktionen in einem Fall zu beschränken. Die Spezifikation lässt offen, wie Rollen Benutzern oder Gruppen zugewiesen werden können.

Der eigentliche `Case` und enthaltene Elemente, wie Aktivitäten, sind in der einem `Case` zugeordneten `Stage` enthalten. Diese ist hier nicht zu verwechseln mit der eigentlichen Klasse `Stage`, die der Gruppierung von Elementen dient, sondern als Spezialausprägung dieser, ausgewiesen durch die Bezeichnung des Attributs `casePlanModel`. Die Klasse `Stage` erlaubt hierarchische Strukturen (`Stages` eingebettet in `Stages`). Die „äußerste“ `Stage` entspricht dem grafischen Containerelement für alle Elemente (siehe Abschnitt 2.2.2). Bevor weiter auf Elemente eingegangen wird, die im `casePlanModel` enthalten sein können, soll das Informationsmodell der (digitalen) Akte erläutert werden.

Abbildung 5 (s. [30], S. 21ff.) zeigt das Informationsmodell der Spezifikation und die mit der Klasse `CaseFile` assoziierte Klasse `CaseFileItem`. Sie repräsentiert Dokumente und Daten der (digitalen) Akte, die über die Klasse `CaseFileItemDefinition` spezifiziert und mit Hilfe der Informationen der Klasse `Import` beispielsweise aus einem Dateisystem importiert werden können. Durch reflexive Assoziationen können (gerichtete) hierarchische Strukturen aufgebaut werden. Das Attribut `multiplicity` gibt an, wie oft ein `CaseFileItem` potenziell vorhanden sein kann.

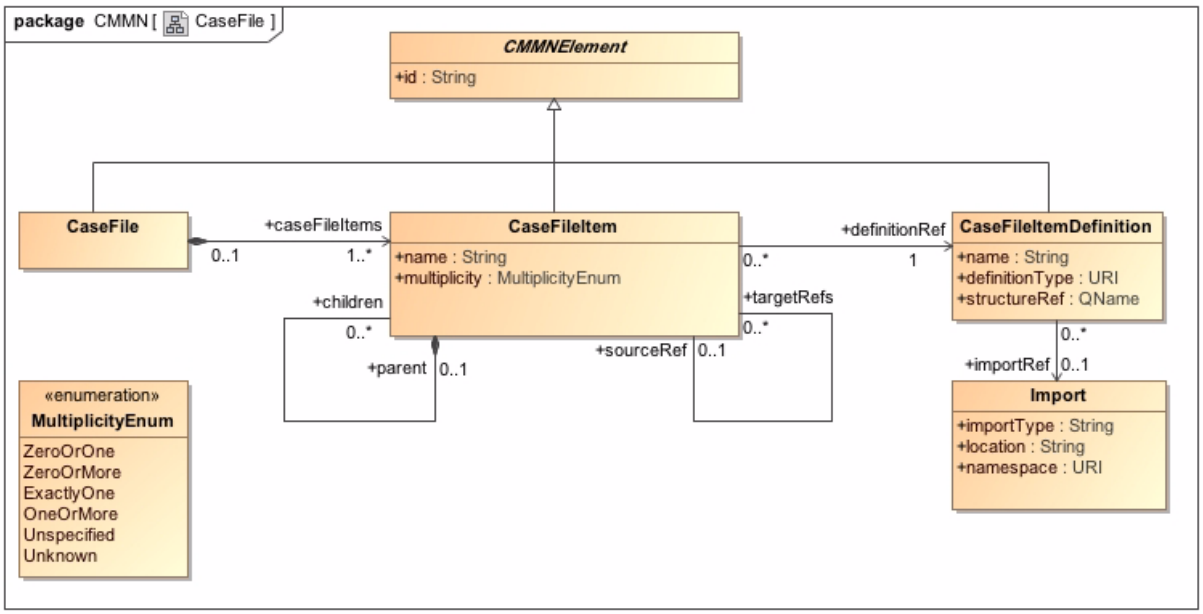

Abbildung 5: Klassendiagramm Informationsmodell

Abbildung 6 (s. [30], S. 23ff.) zeigt die grundlegenden CMMN Bausteine, aus der das eigentliche – zu großen Teilen auch grafisch darstellbare – Modell aufgebaut wird. Alle Elemente sind Spezialisierungen der abstrakten Klasse `PlanItemDefinition`, welche von der abstrakten Klasse `CMMNElement` einen im Modell eindeutigen Identifikator `id` erbt. Auf die assoziierte Klasse `PlanItemControl` wird in Abschnitt 2.2.3.2 näher eingegangen. Sie kann eingesetzt werden, um das Verhalten eines Elements zur Ausführungszeit zu bestimmen.

Basiselemente eines CMMN-Modells sind durch die vier Klassen `Stage`, `Task`, `EventListener` und `Milestone` repräsentiert. Die Klasse `Stage` ist in diesem Fall ein gruppierendes Element, welches Objekte seines eigenen Typs sowie der anderen von Klasse `PlanItemDefinition` erbenden Typen enthalten kann. Alle vier Klassen, beziehungsweise Objekte dieser Typen, können direkt der äußersten `Stage` oder einer in dieser befindlichen `Stage` hinzugefügt werden. Die Klasse `PlanFragment`, von der die Klasse `Stage` erbt, dient dem Planungsmechanismus von CMMN: Durch sie können gruppierte Elemente beliebig komplex miteinander verbunden werden. Diese `Stage` wird aber im Gegensatz zu einer normalen `Stage` zunächst von einer Fallbearbeiterin geplant, beziehungsweise zur Laufzeit aktiviert. Diese Strukturen sind variabel und optional, aber zur Design-Zeit definierte Teile eines Falls und sind daher bereits in Grundzügen bekannt.

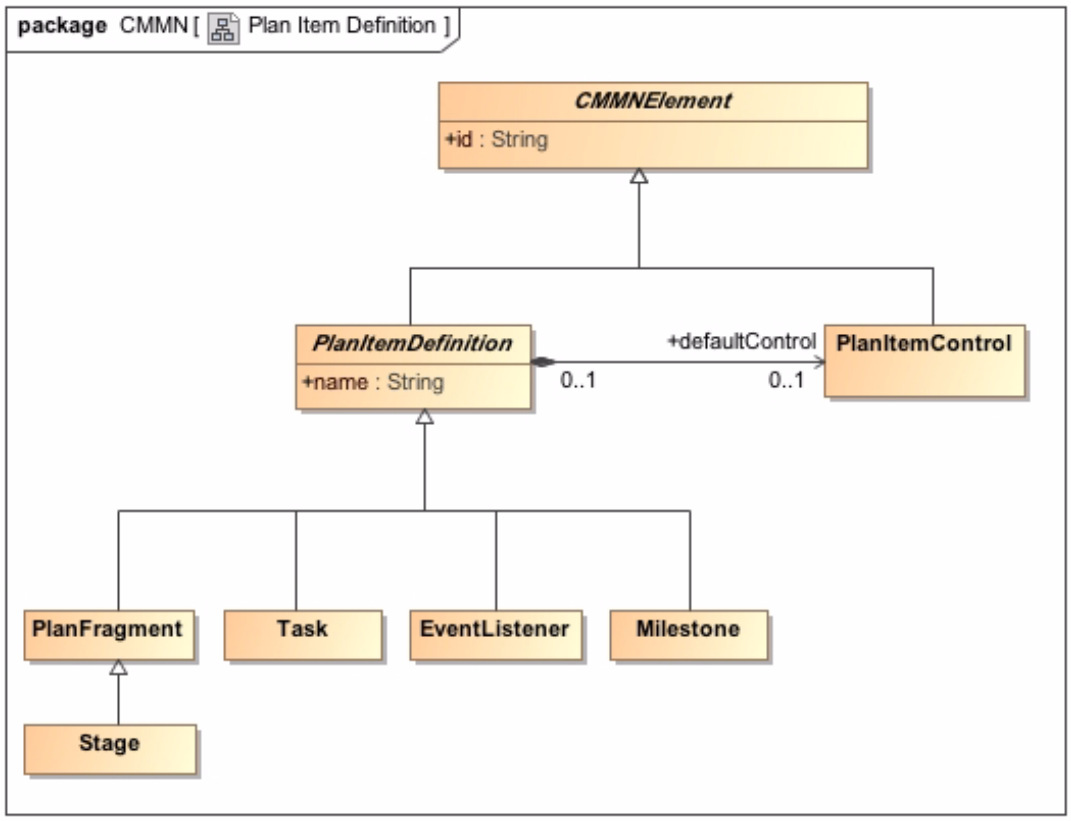

Abbildung 6: Klassendiagramm PlanItemDefinition

Die Klasse `Task` stellt eine unspezifische Aktivität dar. Von ihr erben vier Spezialisierungen: `HumanTask` zur Abbildung von durch Fallbearbeitern durchgeführte Aktivitäten, `ProcessTask` als Bindeglied zu (BPMN-)Prozessen oder automatisierten Vorgängen, `DecisionTask` zur Verknüpfung mit Regeln (etwa mit der „Decision Model and Notation“ (DMN) modelliert, die auch von der OMG verwaltet wird[5]) und `CaseTask`, welcher einen Fall im Fall auslösen kann und als Bindeglied fungiert.

Spezialisierungen finden sich auch für Klasse `EventListener`. `EventListener` sind in CMMN unspezifisch als Ereignisse beschrieben, die im Fall auftreten können. Ereignisse können durch Zustandsänderungen von `CaseFileItem`s oder anderen Elementen wie `Task`s ausgelöst werden und andere Elemente durch ihre Auslösung beeinflussen. Sie können beispielsweise Aktivitäten verfügbar machen oder eine `Stage` terminieren. `UserEventListener` können durch (befugte) Fallbearbeiter ausgelöst werden. So kann beispielsweise ein Fall durch einen Fallbearbeiter direkt abgebrochen werden oder ein Meilenstein gezielt ausgelöst werden. `TimerEventListener` können durch Zustandsänderungen anderer Elemente ausgelöst beziehungsweise gestartet werden und nach Ablauf der definierten Zeit selbst eintreten.

Die Klasse `Milestone` repräsentiert Zwischenziele eines Falls, die bei Erreichung zum Beispiel weitere Elemente verfügbar machen können, wie etwa `HumanTask`s. Einem Meilenstein sind keine Aktivitäten zugeordnet. Meilensteine werden durch Zustandsänderungen von `CaseFileItem`s oder Elementen wie

[5] Siehe https://www.omg.org/spec/DMN/.

`HumanTask` oder `EventListener` erreicht. Im nächsten Abschnitt wird die grafische Repräsentation der beschriebenen CMMN Bausteine gezeigt, bevor die Ausführungssemantik behandelt wird.

2.2.2 Übersicht über die grafischen Elemente

Das in Abbildung 7 dargestellte CMMN-Modell zeigt die visuellen Repräsentationen vieler der im vorherigen Abschnitt beschriebenen CMMN Bausteine und einige neue, bisher nicht beschriebene (grafische) Elemente. Die Namen der Elemente oder Kommentare werden im Text kursiv dargestellt. Eine ausführlichere Zusammenfassung der grafischen CMMN Elemente (der Version 1.0) findet sich in beispielsweise in [31][6]. Weiter eingegangen wird auf neue oder bisher nicht gezeigte (grafische) Elemente im Rahmen der Ausführungssemantik in Abschnitt 2.2.3.

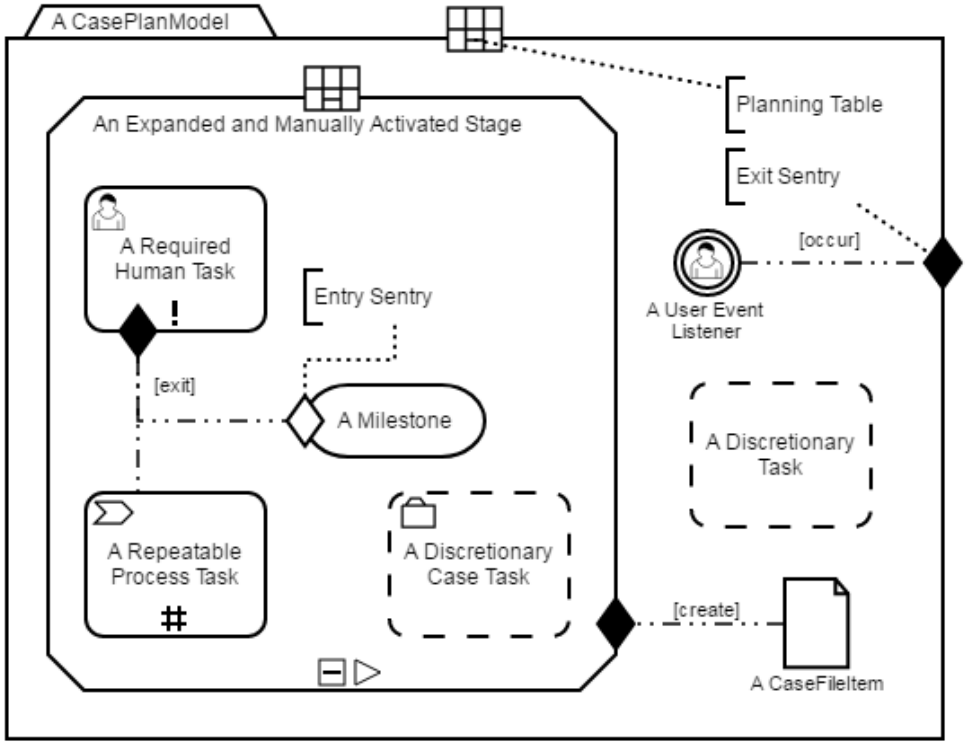

Abbildung 7: Grafische Elemente (eigene Modellierung)

Als äußerster Rahmen ist *A CasePlanModel* dargestellt, was einer äußersten `Stage` entspricht, die im Metamodell über das Attribut `casePlanModel` mit der Klasse `Case` assoziiert ist (vgl. Abbildung 4). Es kann als das Wurzelelement eines Fallmodells betrachtet werden, welches alle Elemente beinhaltet.

A CasePlanModel enthält eine `Stage` *An Expanded and Manually Activated Stage*, einen `UserEventListener` *A User Event Listener*, einen Spezialfall eines `Task` *A Discretionary Task* und ein `CaseFileItem` *A CaseFileItem*. Die `Stage` wiederum enthält den `HumanTask` *A Required Human Task*, den `ProcessTask` *A Repeatable Process Task*, einen `Milestone` *A Milestone* und den `CaseTask` *A Discretionary Case Task*.

[6] Eine auf der referenzierten Veröffentlichung basierende, webbasierte und interaktive Übersicht über CMMN Elemente findet sich unter http://cmmn.byethost4.com/index.html.

Das *Planning Table*-Symbol, welches oben an *A CasePlanModel* und die `Stage` angeheftet ist, gibt Auskunft über enthaltene planbare Elemente. Die sogenannten *discretionary items* werden durch gestrichelte Umrandungen der Darstellungen angezeigt. `Stage`s und `Task`s können *discretionary* sein und werden von einem (befugten) Fallbearbeiter zur Laufzeit geplant (aktiviert). Auf den Planungsmechanismus wird in dieser Arbeit auf Grund der hohen Komplexität nicht weiter eingegangen[7].

Die Spezialisierungen der Elemente werden durch verschiedene Piktogramme gekennzeichnet, wie etwa durch den Oberkörper am `HumanTask` (siehe auch das Zentrum des `UserEventListener`s als Spezialisierung von `EventListener`) und das Prozess-Piktogramm, welches einen `ProcessTask` kennzeichnet.

Grafische Elemente, die bisher nicht als Repräsentation in den Metamodellen dargestellt wurden, sind unter anderem die weißen und schwarzen Diamanten, wie etwa „angeheftet“ an *A Milestone* und *A CasePlanModel* zu sehen. Diese Diamanten sind Teil des Sentry-Konzepts. Visuell verknüpft werden sie mit anderen Elementen über die gestrichelt-gepunkteten Verbindungen. Diese können in eckigen Klammern angeben, auf welche in CMMN definierte Zustandsänderung der verbundene Diamant reagiert (beispielsweise *[occur]* an der Verbindung zum `UserEventListener`). Zustände und Zustandsänderungen werden in Abschnitt 2.2.3.1 näher beschrieben.

Weiße Diamanten überwachen eine oder mehrere Bedingungen, um ein Element zu aktivieren oder auszulösen. So führt der Zustandsübergang *[exit]* des `HumanTask` dazu, *A Milestone* zu erreichen. Schwarze Diamanten, deren Bedingung(en) erfüllt sind, führen hingegen zur Beendigung eines Elements. So kann der gesamte Fall durch Auslösen des `UserEventListener` terminiert werden. Die Diamanten werden im Rahmen des Sentry-Konzepts in Abschnitt 2.2.3.3 weiter beschrieben.

Daneben sind noch sogenannte *decorators* zu sehen, wie das Ausrufezeichen am `HumanTask`, das Rautensymbol am `ProcessTask` oder das Start- oder Play-Symbol an der `Stage`. Sie kennzeichnen Elemente respektiv als benötigt, wiederholbar oder als manuell zu starten und beeinflussen so die Ausführung und die Zusammenstellung der Aktivitäten einer Fall-Instanz. Die *decorators* werden in Abschnitt 2.2.3.2 näher beleuchtet.

Rechts unten im Modell ist die grafische Darstellung eines `CaseFileItem` zu sehen. Es – genauer gesagt sein Zustand – wird von einem schwarzen Diamanten überwacht. Geht es in einen definierten Zustand durch *[create]* über, was anzeigt, dass es erstellt wurde, werden die `Stage` und die darin enthaltenen Elemente terminiert.

Im folgenden Abschnitt wird weiter auf Zustände, *decorators* und das Sentry-Konzept eingegangen. Zusammen stellen diese Möglichkeiten zur Modellierung sehr flexibler Prozesse bereit.

[7] Der Planungsmechanismus stellt eine sehr komplexe Abstraktionsebene dar, die den Rahmen der Arbeit sprengen würde. Beschrieben wird er in Kapiteln 5.4.9, 6.12 und 8.7 der CMMN 1.1 Spezifikation.

2.2.3 Ausführungssemantik

CMMN bietet neben dem in dieser Arbeit nicht weiter beschriebenen Planungsmechanismus durch Zustände und Lebenszyklen, Verhaltensregeln durch sogenannte *decorators* und dem darauf aufbauenden Sentry-Konzept Mechanismen zur Modellierung flexibler Prozesse an. Diese Mechanismen werden in diesem Abschnitt näher betrachtet. Dabei werden grafische Elemente mit Hilfe weiterer Metamodelle und Zustandsdiagramme beschrieben und Beispiele zur Modellierung gezeigt.

2.2.3.1 Zustände und Lebenszyklen

Alle grafisch repräsentierten CMMN Elemente besitzen Zustände und Lebenszyklen, die in der Spezifikation definiert sind. Zu unterscheiden ist hierbei zwischen automatischen und manuellen Zustandsübergängen. Dieser Abschnitt soll eine grundlegende Übersicht geben. Zustandsnamen und Namen von Zustandsübergängen sind im Text kursiv gestellt angegeben.

Dieser Abschnitt zeigt die Definitionen der bereits vorgestellten Klassen `CaseFileItem`, `Stage` und `Task`, sowie `Milestone` und `EventListener`. Nicht gezeigt ist der Spezialfall einer `Stage`, die als `casePlanModel` dient. Jeder Zustand und seine Übergänge sind ausführlich in Kapitel 8 der Spezifikation beschrieben (vgl. [30], S. 107ff.). Zustände und Zustandsübergänge sind im weiteren Text kursiv dargestellt.

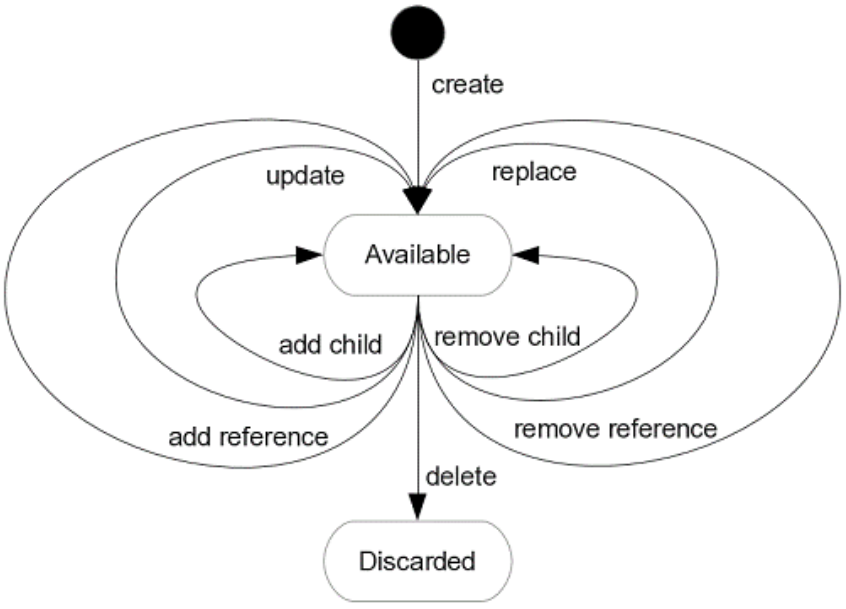

Abbildung 8: Lebenszyklus eines CaseFileItems

Die Zustände und Lebenszyklen einer `CaseFileItem`-Instanz sind in Abbildung 8 dargestellt (s. [30], S.107). Als Zustände sind *Available* und *Discarded* zu sehen. Zu den definierten Übergängen zwischen diesen gehören unter anderem *create*, *update* und *delete*. *Create* führt aus einem Initialzustand zum Zustand *Available* und signalisiert die Erstellung einer `CaseFileItem`-Instanz. Der Übergang *update* zeigt an, dass eine Eigenschaft (optional mit `CaseFileItem`s assoziierte `Property`s) geändert wurde. Die Klasse `Property` kann dazu genutzt werden, Werte verschiedener Datentypen darzustellen und

eine `CaseFileItem`-Instanz weiter zu beschreiben. Während der Übergang *delete* in den Zustand *Discarded* überführt und das Entfernen der `CaseFileItem`-Instanz anzeigt, führen die anderen Übergänge zurück in den Zustand *Available*. Sie können dazu benutzt werden, über den Aufbau von Hierarchien und Verknüpfungen innerhalb dieser zu informieren.

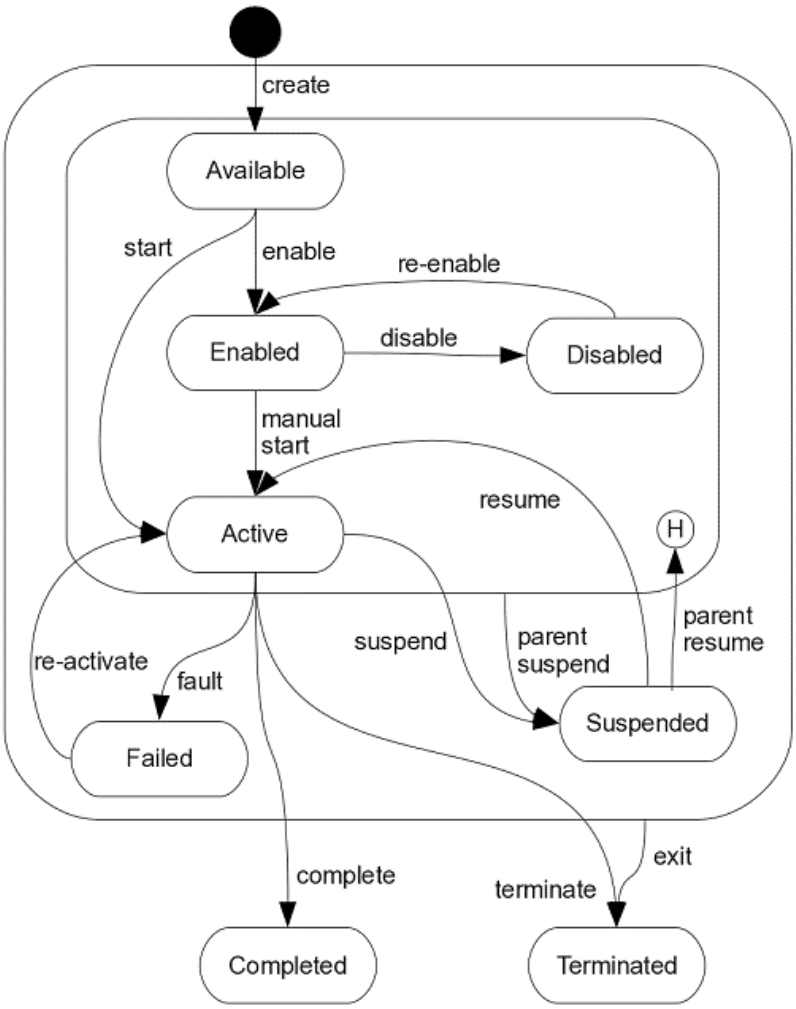

Abbildung 9: Lebenszyklus der Klassen Stage und Task

Instanzen der Klassen (folgend auch als Elemente bezeichnet) `Stage` und `Task` teilen sich den in Abbildung 9 dargestellten Lebenszyklus (s. [30], S. 113). Die Bedeutung ausgewählter Zustände und Übergänge soll kurz beschrieben werden.

Nachdem ein Element aus dem Initialzustand durch *create* in den Zustand *Available* gewechselt ist, wird die `Stage` oder `Task` auf angeheftete weiße Diamanten und *decorators* geprüft, wie etwa die im vorherigen Abschnitt aufgezeigten Start- oder Raute-Symbole. Ist ein weißer Diamant vorhanden, wird dessen Bedingung geprüft. Ist diese erfüllt, kann in den Zustand *Enabled* oder *Active* gewechselt werden – je nachdem, ob ein Start-Symbol *decorator* vorhanden ist, oder nicht. Ansonsten verbleibt das Element im Zustand *Available*, bis die Prüfung eines weißen Diamanten erneut angestoßen wird und erfüllt ist.

Ist ein (gültiges) Start-Symbol vorhanden, geht das Element in den Zustand *Enabled* über (oder wird durch *disable* deaktiviert) und muss manuell durch eine Fallbearbeiterin in den Zustand *Active*

gewechselt werden. Ist der *decorator* nicht gültig oder nicht vorhanden, geht das Element automatisch über *start* in den Zustand *Active* über und zeigt an, dass beispielsweise eine `HumanTask`-Instanz von einem Fallbearbeiter bearbeitet werden kann. Der Zustand *Failed* ist für Fehler während der Ausführung des Modells durch unterstützende IT-Systeme gedacht.

Completed ist der reguläre Endzustand einer `Stage` oder `Task` (sowie auch einer `Milestone-` oder `EventListener`-Instanz), wohingegen der Zustand *Terminated* anzeigt, dass das Element manuell durch einen Fallbearbeiter oder durch das Erfüllen eines schwarzen Diamanten terminiert wurde.

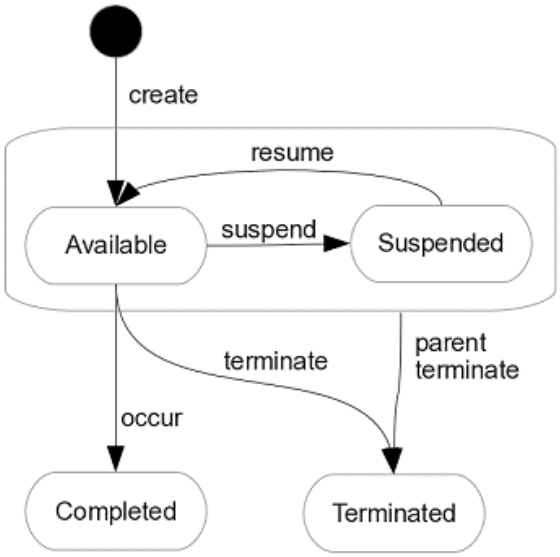

Abbildung 10: Lebenszyklus von Milestone und EventListener

Abschließend wird in Abbildung 10 das Zustandsdiagramm für `Milestone` und `EventListener` gezeigt (s. [30], S. 120). Diese Elemente teilen sich wie `Stage` und `Task` einen Lebenszyklus. Beide können mit *suspend* in den Zustand *Suspended* versetzt werden, wenn sie nicht mehr zu erreichen oder auslösbar sind oder sein sollen. Dies geschieht entweder durch manuelle Interaktion oder wenn eine übergeordnete `Stage` in den Zustand *Suspended* wechselt: Der Zustandsübergang der übergeordneten `Stage` (welche als Spezialfall ebenso ein `Case` sein kann) wird an die enthaltenen Kind-Elemente weiterpropagiert, sodass diese ebenso in den Zustand *Suspended* übergehen. Der vorherige Zustand wird dauerhaft gespeichert und bei einem Wechsel durch den Übergang *resume* als Ausgangspunkt für weitere Übergänge wiederhergestellt. Ebenso wird der Übergang *terminate* einer übergeordneten Stage an die enthaltenen Elemente propagiert, was zu einem Übergang in den finalen Zustand *Terminated* führt.

Weiteren Einfluss auf die Ausführungssemantik und Zustandsübergänge der Elemente haben die im nächsten Abschnitt beschriebenen *decorators*. Die Überwachung der Zustandsübergänge durch Diamanten (`Sentry`s) wird in Abschnitt 2.2.3.3 gezeigt.

2.2.3.2 Verhaltensregeln durch *decorators*

Kapitel 8.6 der Spezifikation (vgl. [30], S. 121ff.) beschreibt die im Folgenden betrachteten Verhaltensregeln. Abbildung 11 zeigt, wie *decorators* den grafischen Elementen zugeordnet werden dürfen (s. [30], S. 79ff.). Neben den im vorherigen Abschnitt gezeigten *decorators* ist auch der bisher nicht gezeigte *decorator AutoComplete* (schwarzes Rechteck, mittlere Spalte der in Abbildung 11 gezeigten *decorators*) dargestellt. Dieser gibt an, ob ein `CasePlanModel` oder eine `Stage` automatisch in den Zustand *Completed* wechselt. Dies geschieht, wenn alle enthaltenen Kindelemente nicht (mehr) im Zustand *Active* sind **und** alle mit einem *required decorator* versehenen Elemente – für die dieser gültig ist – im Zustand *Disabled*, *Completed*, *Terminated* oder *Failed* sind.

Decorator Applicability	Planning Table	Entry Critrion	Exit Criterion	AutoComplete	Manual Activation	Required	Repetition
`CasePlanModel`	☑		☑	☑			
`Stage`	☑	☑	☑	☑	☑	☑	☑
`Task`	`HumanTask` only	☑	☑		☑	☑	☑
`MileStone`		☑				☑	☑

Abbildung 11: Zulässige Kombinationen von Elementen und *decorators*

Im Metamodell entsprechen die *decorators Manual Activation*, *Required* und *Repetition* den mit der Klasse `PlanItemControl` assoziierten Klassen `ManualActivationRule`, `RequiredRule` und `RepetitionRule`, wie in Abbildung 12 (s. [30], S. 52) zu sehen (vgl. Abbildung 6, zu beachten ist die Assoziation zur Klasse `PlanItemDefinition` – diese ist die Basis für Elemente beziehungsweise Klassen wie `Task`).

Die Gültigkeit dieser *decorators* wird zur Laufzeit über die Evaluierung von Ausdrücken (Klasse `Expression`) geprüft. Die Evaluierung wird über boolesche Werte gesteuert. Als Basis für die Evaluierung können von einer Regel-Klasse referenzierte `CaseFileItem`s (und für diese definierte `Property`s) ausgewertet werden. Somit können *decorators* zur Laufzeit ihre Gültigkeit verlieren: Eine zuvor benötigte `Stage` kann entfallen (oder zu einem späteren Zeitpunkt wieder notwendig werden), oder ein `Task` nicht mehr wiederholt werden (oder erst ab einem bestimmten, späteren Zeitpunkt).

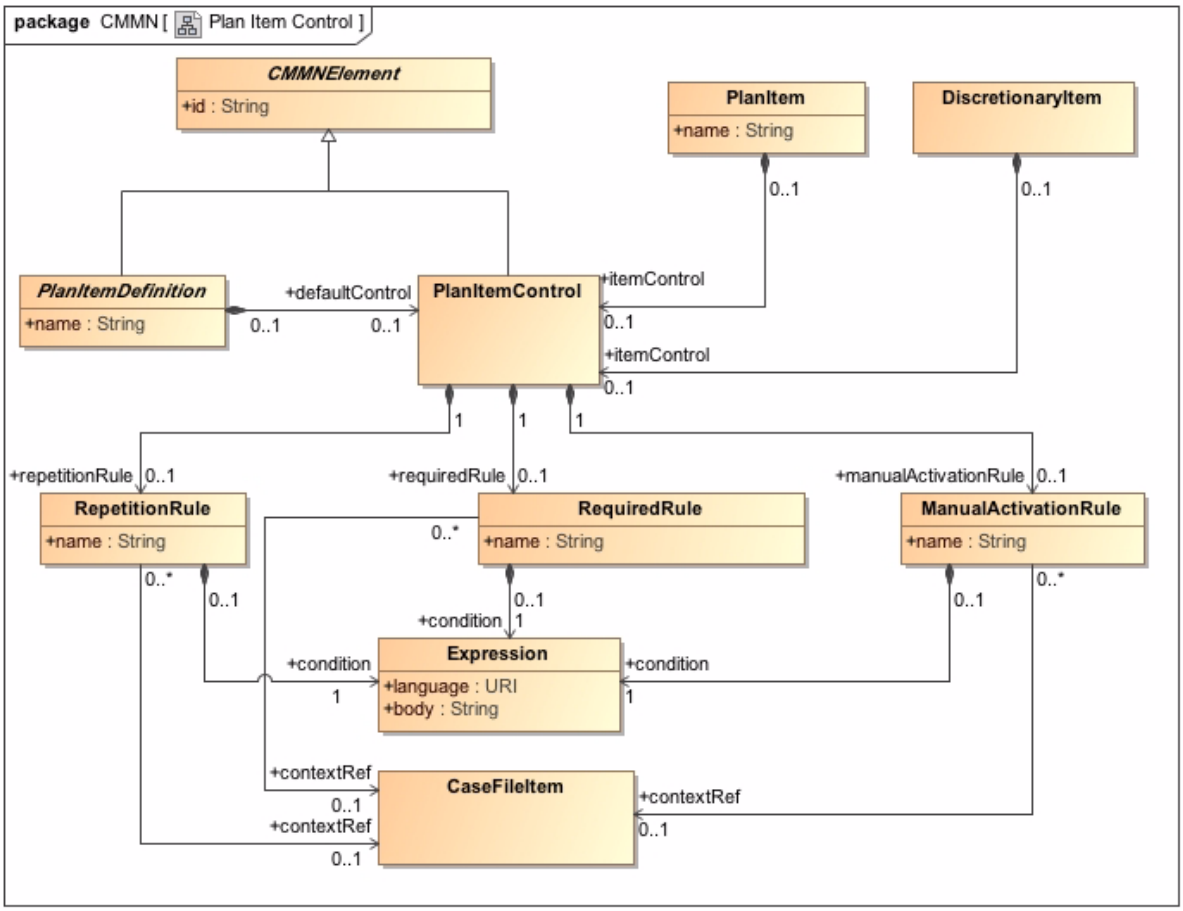

Abbildung 12: Klassendiagramm decorators

Zu beachten ist, dass ein `Milestone` nur *Entry Criterion*s besitzen darf und ein `CasePlanModel` nur *Exit Criterion*s. Auf die zuvor als Diamanten bezeichneten *Entry Criterion* und *Exit Criterion decorators* wird im nächsten Abschnitt eingegangen.

2.2.3.3 Sentry-Konzept

Das Sentry-Konzept, welches in der Spezifikation in Kapitel 8.5 spezifiziert wird, bringt die bisher beschriebenen `CaseFileItem`s, Elemente wie `Task`s und `EventListener` und *decorators* zusammen (vgl. [30], S. 121ff.). Mit ihm können Elemente dynamisch miteinander verknüpft werden und in Abhängigkeit von Zustandsänderungen anderer Elemente oder Daten können diese beispielsweise aktiviert, suspendiert oder terminiert werden.

Die Klasse `Sentry` ist im Zentrum des Klassendiagramms in Abbildung 13 zu sehen (s. [30], S. 32; hier nicht gezeigt sind die beiden Enumeration-Klassen `CaseFileItemTransition` und `PlanItemTransition`, welche die jeweils zulässigen Zustandsübergänge von `CaseFileItem`s und `PlanItem`s enthalten, wie in den vorherigen Abschnitten dargestellt). Sie „wacht“ durch die mit ihr assoziierten `OnParts` und einem (optionalen) `IfPart` über die Zustandsänderungen der beobachteten

`CaseFileItem`s und `PlanItem`s. Letztere können beispielsweise Elemente wie `Stage`, `Task` oder `Milestone` sein (vgl. Abbildung 6).

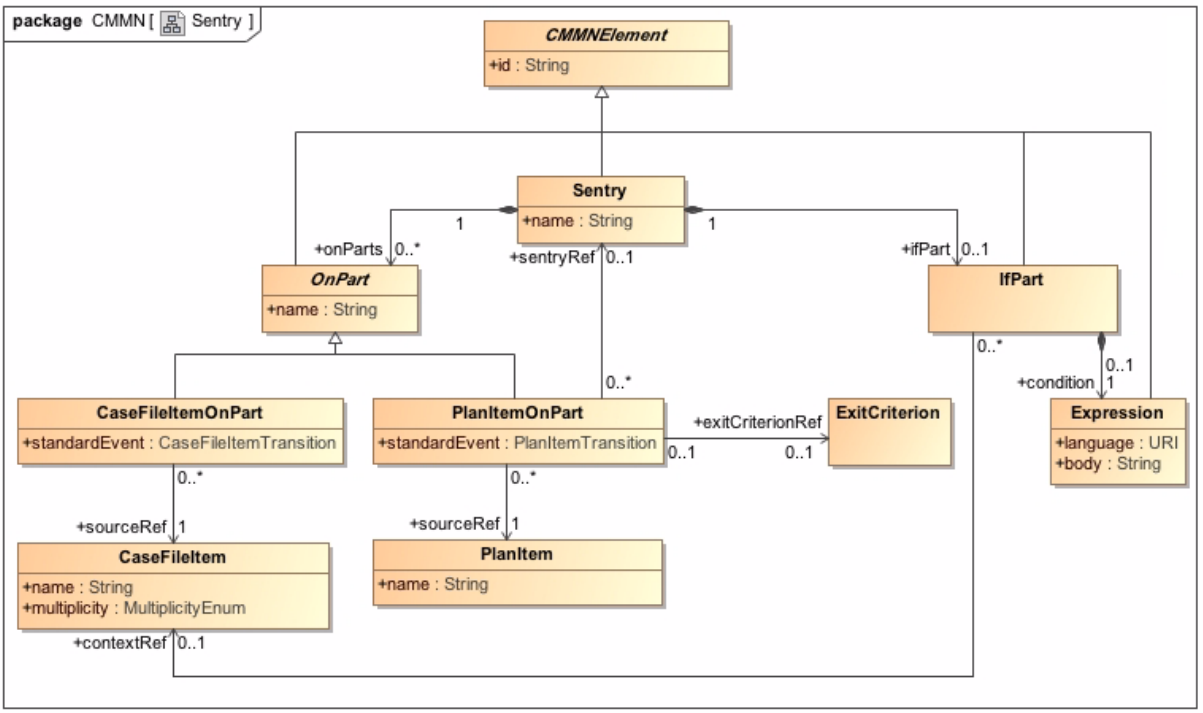

Abbildung 13: Klassendiagramm Sentry-Konzept

Ein `Sentry` stellt eine Kombination aus Ereignis und/oder Bedingung dar. Dabei kann ein `Sentry` entweder nur einen `IfPart` besitzen oder eine Kombination aus einem oder mehreren `OnParts` sowie einem optional `IfPart`. Werden die Bedingungen eines `Sentrys` erfüllt, geht das assoziierte Element (an das ein Diamant geheftet wurde) in den definierten, zulässigen Zustand über.

Ereignisse sind die zuvor beschriebenen und in CMMN definierten Zustandsänderungen, wie etwa der Übergang aus dem Initialzustand eines Elements durch den Zustandsübergang *create*. Diese Übergänge werden durch `OnParts` beobachtet. Bedingungen sind evaluierte Ausdrücke in Form der Klasse `Expression`, die referenzierte `CaseFileItems` (und assoziierte `Propertys`) als Basis zur Auswertung nutzen können. Diese werden durch `IfParts` geprüft.

Das in Abbildung 14 modellierte Beispiel soll das Zusammenspiel zwischen Lebenszyklus, `Sentry`, `CaseFileItem` und *decorators* veranschaulichen: Angenommen ein `HumanTask` ist mit einem *Entry Criterion* und einem *Manual Activation decorator* versehen und wartet im Zustand *Available* auf die Erfüllung des zugrundeliegenden `Sentry`. Das *Entry Criterion* (und somit das `Sentry` im Hintergrund) ist mit dem `EventListener` verbunden und wartet auf dessen Übergang *occur*.

Der `EventListener` wird durch einen nicht grafisch dargestellten `PlanItemOnPart` beobachtetet. Tritt nun der Zustandsübergang *occur* ein, der dem angegebenen Attribut `standardEvent` des `OnParts` entspricht, wird die assoziierte `Sentry`-Instanz benachrichtigt. Anschließend werden dessen Bedingungen geprüft.

Angenommen, dass zu den Bedingungen ein vorhandener `IfPart` gehört, wird die `Expression` zum Beispiel auf Basis des zu sehenden `CaseFileItem`s und dessen boolean `Property`s `satisfied` und `manualActivation` evaluiert. Die Evaluierung des `IfPart` liefert den booleschen Wert *true* zurück und es gilt die `Sentry`-Instanz als erfüllt, da `OnPart` **und** `IfPart` erfüllt sind.

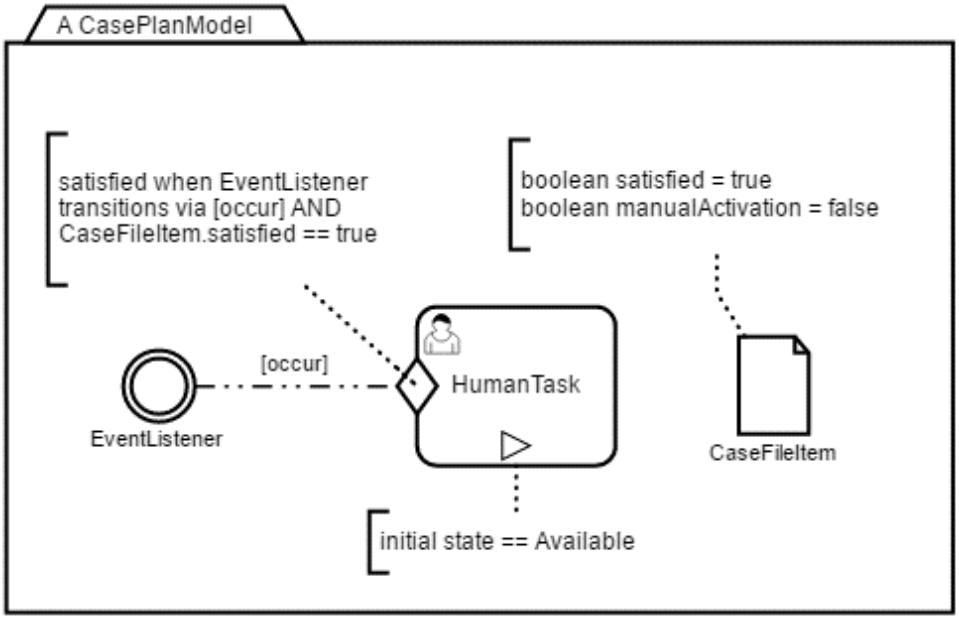

Abbildung 14: Beispiel zur Veranschaulichung des Sentry-Konzepts (eigene Modellierung)

Der vom `Sentry` referenzierte `HumanTask` kann nun vom Zustand *Available* in einen anderen Zustand wechseln. Hierzu werden zunächst *decorators* evaluiert. Angenommen die dem *Manual Activation decorator* zugrundeliegende `ManualActivationRule` evaluiert das Attribut `manualActivation` des `CaseFileItem`. Da diese zum Zeitpunkt der Evaluierung `false` ist, wechselt der `HumanTask` automatisch in den Zustand *Active*.

Das Sentry-Konzept kann auch dazu genutzt werden, um einfache Sequenzen aufeinanderfolgender Aktivitäten zu modellieren, wie etwa die durch BPMN-Gateways bekannten Muster paralleler (AND) oder optionaler Sequenzen (OR) (vgl. auch [30], S. 70ff.).

Abbildung 15 zeigt eine einfache Sequenz zweier aufeinander folgender `Task`s. `Task` B wartet auf den Zustandsübergang von `Task` A durch *complete* und wird bei Erfüllung aktiv (genau gesagt beobachtet ein `PlanItemOnPart` des `Sentry` das `PlanItem` `Task` A).

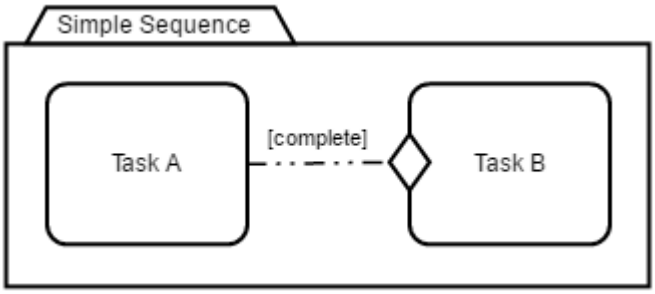

Abbildung 15: Einfache Verknüpfung durch Sentry (eigene Modellierung)

Ein Element kann durch einen Zustandsübergang auch mehrere Elemente beeinflussen, wie in Abbildung 16 zu sehen ist: `Task` B und `Task` C werden aktiv, wenn `Task` A durch *complete* in den Zustand *Completed* wechselt.

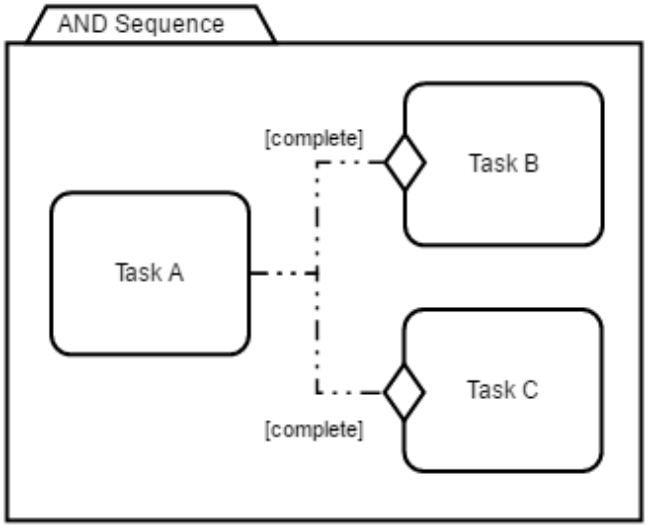

Abbildung 16: AND-Verknüpfung durch Sentrys (eigene Modellierung)

Eine weitere Möglichkeit der Verknüpfung zeigt sich in Abbildung 17. Dargestellt wird, wie mehrere Elemente ein Element beeinflussen: `Task` C wird erst aktiv, wenn `Task`s A **und** B durch den Übergang *complete* jeweils ihren Zustand zu *Completed* gewechselt haben – wann dies für A oder B geschieht, steht vielleicht noch nicht fest.

Das Verhalten von `Sentry`s ist allerdings so spezifiziert, dass es sich merkt, wenn seine Bedingung(en) in der Vergangenheit erfüllt wurde(n). So kann zeitlich gesehen zunächst `Task` A in den beobachteten Zustand übergehen. `Task` B wird vielleicht erst Tage später regulär durch *complete* beendet, führt aber schließlich dazu, dass `Task` C aktiviert wird.

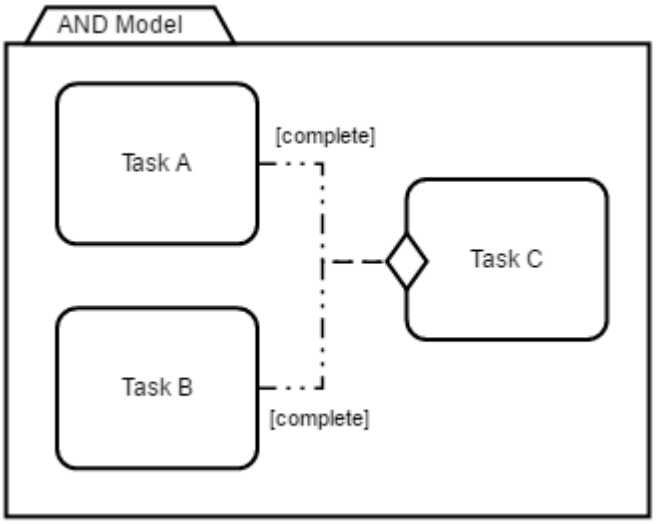

Abbildung 17: AND-Zusammenführung durch ein Sentry (eigene Modellierung)

Abbildung 18 zeigt abschließend, wie eine OR-Struktur modelliert werden kann: `Task` C wird aktiv, wenn entweder `Task` A **oder** B durch den Übergang *complete* seinen Zustand wechselt. Für einen Zustandswechsel aus dem Zustand *Available* heraus genügt die Erfüllung eines überwachenden `Sentry`s and `Task C` durch den angegebenen Zustandsübergang [*complete]*.

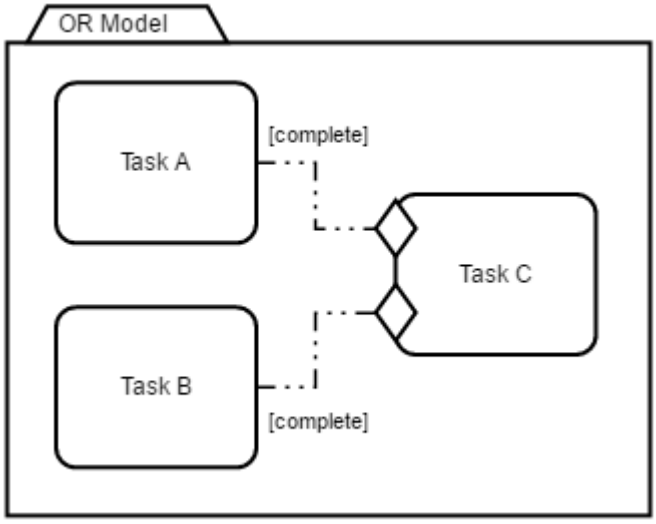

Abbildung 18: OR-Struktur durch Sentrys (eigene Modellierung)

Die Beispiele können beliebig um andere Elemente, wie etwa von Fallbearbeitern auslösbare `UserEventListener`, `CaseFileItem` evaluierende `IfParts` als zusätzliche Bedingung eines `Sentry` und erlaubte *decorator* Kombinationen zur Steuerung der beeinflussten Elemente erweitert werden und zu komplexen Strukturen führen.

Das Sentry-Konzept verbindet alle bisher dargestellten und erläuterten CMMN Strukturen. Es ermöglicht flexible Modelle auf deklarativer Basis zu erstellen. Neben den Lebenszyklen und Zustandsänderungen der einzelnen Elemente und diese beeinflussenden *decorators* nehmen Daten in Form von `CaseFileItem`s eine bedeutende Rolle ein. Sie dienen zur Laufzeit nicht nur als Basis zur Evaluierung der *decorators*, sondern auch zur Evaluierung des `IfParts` eines `Sentrys`. Darüber hinaus können ihre eigenen Zustandsänderungen ebenso andere Elemente beeinflussen.

2.2.4 Modellierungs- und Ausführungsunterstützung

Nachdem verschiedene Aspekte der CMMN behandelt wurden, soll in diesem Abschnitt abschließend eine kleine Übersicht über verfügbare Modellierungswerkzeuge und Ausführungssysteme gegeben werden. Es gibt ein überschaubares Angebot an Modellierungswerkzeugen, die CMMN unterstützen.

Signavio[8] bietet ein nicht ausgereiftes CMMN-Modul an, das bisher nicht kommerziell beworben wird. Das Modul muss per E-Mail-Anfrage freigeschaltet werden. Es unterstützt keinen Im- oder Export von CMMN-Dateien. Es besteht aber die Möglichkeit, CMMN-Modelle online im Repository zu speichern. Enterprise Architect unterstützt ab Version 14 die Modellierung mit CMMN[9]. Das Open Source Projekt bpmn.io aus dem Camunda-Umfeld bietet ein Modellierungswerkzeug (online) auf Basis von JavaScript

[8] Siehe https://www.signavio.com.

[9] Siehe https://www.sparxsystems.eu/enterprisearchitect/newedition/.

an[10]. Das online verfügbare Toolkit bietet allerdings keine Möglichkeiten, Attribute der verschiedenen Elemente anzupassen. So lassen sich beispielsweise Zustandsübergänge nicht frei auswählen. Es stehen aber aktivierbare Module zur Verfügung, die ein Kontextmenü einbinden. Über dieses können dann Attribute wie Zustandsübergange eingestellt werden. Das Modul muss allerdings manuell eingerichtet werden und der erweiterte Editor anschließend in eine (lokale) Webseite eingebunden werden.

Ein (offline) Modellierungswerkzeug inklusive Kontextmenü zur Konfiguration von CMMN Attributen (und Einstellung anbieterspezifischer XML-Elemente) bietet der Camunda Modeler, der augenscheinlich auf dem bpmn.io-Projekt basiert[11].

CMMN-Modelle sollen von der Camunda BPM-Plattform, „flowable“ und „Activiti“ verarbeitet werden können[12]. Hierbei ist allerdings zu beachten, dass manche Elemente nicht unterstützt werden, wie etwa der Planungsmechanismus, der beispielsweise von Camunda nicht unterstützt wird. Da die drei genannten Plattformen ihren Fokus klar auf die Verarbeitung von BPMN (und DMN) legen, ist nach einer Recherche zu diesem Zeitpunkt davon auszugehen, dass die jeweilige CMMN-Implementierung nicht voll und nur teilweise spezifikationskonform ist. Sie machen den Eindruck experimenteller Erweiterungen der genannten BPMN-Plattformen. Zu berücksichtigen ist hierbei, dass CMMN 1.1 relativ jung ist und in der Praxis noch wenig erprobt wurde. Ein kommerzielles Interesse ist im Gegensatz zu BPMN-Funktionalitäten vermutlich geringer.

[10] Siehe https://bpmn.io/toolkit/cmmn-js/.

[11] Siehe https://camunda.com/download/modeler/.

[12] Siehe https://camunda.com, https://flowable.org/ und https://www.activiti.org/.

2.3 Verwandte Arbeiten

Die aktuelle Forschung zum Thema Case Management untersucht die Modellierung von CM Anwendungen und flexiblen Prozessen mit CMMN oder verwendet artefakt- oder objektzentrierte Ansätze, die mitunter auf Graphentheorie fußen. Forschung zu CMMN basierten CM Implementierungen ist weniger vorhanden. In diesem Abschnitt soll eine Übersicht über den Stand der Forschung zu den Themen flexible Prozesse, Modellierung von CM Anwendungen mit CMMN und CM Implementierungen gegeben werden.

2.3.1 Flexible und artefaktzentrierte Prozesse

Um Case Management-Konzepte und die einhergehende Arbeitsweise durch IT-Systeme zu unterstützen, wurden mehrere Ansätze angedacht und entwickelt, die im Folgenden kurz vorgestellt werden sollen. Zu den Ansätzen zählen deklarative Ansätze, die im Gegensatz zu vordefinierten Ausführungspfaden nicht deterministisch sind. Insbesondere für die kollaborative und übergreifend betrachtete Arbeit an einem oder mehreren zentralen Objekten, wie einem Case, haben sich verschiedene Ansätze entwickelt. Die aufgezeigte Forschung zum Thema ist größtenteils vor der Veröffentlichung von CMMN oder der Version 1.1 erschienen oder begonnen worden.

Der Modellierungsstandard BPMN eignet sich nur eingeschränkt zur Modellierung von wissensintensiven Prozessen, insbesondere was die Flexibilität während der Ausführung des Modells und die Darstellung von übergreifenden Informationen betrifft. Eine Modellierung aller möglichen Ausführungspfade und Ausnahmesituationen für einen Prozess würden ein BPMN-Modell schnell überfüllen, nicht mehr nachvollziehbar und wartbar machen und ab einer größeren Komplexität für alle Prozessbeteiligten unhandlich und nutzlos werden (vgl. [32]).

Für flexiblere Ausführungspfade eignen sich ad-hoc-Subprozesse, die Tasks beinhalten. Allerdings ist der ad-hoc-Subprozess in die imperative Kontrollflussstruktur von BPMN eingebettet. Auch ist die Ausführungssemantik für Aktivitäten innerhalb eines ad-hoc-Subprozesses durch ihre eingeschränkten Kontrollmöglichkeiten nur eingeschränkt für CM-Anwendungen geeignet. Es eignet sich eher für PCM-Lösungen, allerdings bestehen ohne Erweiterungen oder Anwendung anderer Ansätze unter Verwendung von BPMN die vier im zweiten Abschnitt benannten Probleme, insbesondere das Problem des *context tunneling*. Erweiterungen wie in [33] versuchen durch neue Elemente den BPMN-Standard (minimal) zu erweitern, um CM-Anwendungen durch konfigurierbare BPMN-Modelle zu ermöglichen.

Ein deklarativer Ansatz, der auf Geschäftsartefakten mit Daten- und Lebenszyklusmodell aufbaut und eine graphische Darstellung beinhaltet, ist der *Guard-Stage-Milestone* (GSM) Ansatz aus [34]. Er bezieht Regeln (in Form von *Event-Condition-Action* (ECA) Regeln) und interne sowie externe Ereignisse mit ein. Organisiert werden können mögliche – das heißt nicht notwendigerweise strikt oder sequenziell auszuführende, aber verfügbare – Aktivitäten frei oder zur Gruppierung in *stages*, die überwacht durch ECA-Regeln (in Form von *guards*) ihre Zustände ändern und weitere Aktionen zugänglich machen. Ausführungspfade können so neben ad-hoc-Aktivitäten bestehen und gezielt zusammenhängende, sequenziell auszuführende Aktivitäten definiert werden. Übergeordnete Ziele, seien diese für eine *stage* oder den gesamten Case definiert, sind in Form von *milestones* festgehalten

und werden durch Regeln bestimmt und können von Ereignissen beeinflusst werden. Dieser Ansatz prägte stark die Entwicklung CMMN, die auch in Kooperation mit Industrieunternehmen entwickelt wurde.

Ein weiterer deklarativer Ansatz für dynamische Prozesse namens *ConDec* verbindet eine erweiterte *Linear Temporal Logic* (LTL) mit einem Mapping auf eine graphische Syntax. Durch LTL-Ausdrücke werden die Relationen zwischen Aktivitäten für dynamische Prozesse definiert. Im einfachsten Fall besteht ein Modell nur aus einer Menge an Aktivitäten und Bedingungen für diese. Zur Prozessausführung werden die LTL-Ausdrücke benutzt, die während der Designzeit aus einer vordefinierten Menge zur Beschreibung der Beziehungen einzelner Aktivitäten gewählt werden können. Regeln sollen die Wissensarbeiter bei der Zielerreichung und Auswahl von Aktivitäten unterstützen (vgl. [35; 36]).

Objekt- [37], daten- und artefakt-zentrierte [38; 39] Ansätze legen ihren Fokus auf prozess-übergreifend modellierte Informationen und Daten in Abgrenzung zu typischen, primär durch Aktivitäten getriebene Ansätze, meist unter Einbezug von Zustandsautomaten. Ihnen ist gemein, dass die in zweiten Abschnitt 2.1.4 aufgeführten vier Punkte (unter *Unzulänglichkeiten „klassischer" Ansätze*) in Teilen als zu lösendes Problem aufgegriffen werden. Im Mittelpunkt steht die Arbeit auf Objekten und Daten in Abhängigkeit ihrer Zustände und Werte, statt Aktivitäten isoliert in einem unflexiblen, imperativen Ausführungspfad mit lediglich den für die Aufgabe benötigten (oder sichtbaren) Daten zu bearbeiten (vgl. [40; 41]).

Auch hybride Ansätze, die mögliche Ausführungspfade durch Regelwerke, Datenzustände und Konfigurationsmöglichkeiten einschränken, wurden entwickelt und setzen beispielsweise auf BPMN-Modellfragmenten auf. Dieser hybride Ansatz kombiniert objekt-zentrierte Modelle mit BPMN-Modellfragmenten und stellt im Datenmodell einen Case in den Mittelpunkt. Datenobjekte werden von Wissensarbeitern bis zu einem gewünschten Zustand der Zielerfüllung bearbeitet. Eine Erweiterung des BPMN-Metamodells, um konfigurierbare kontext-basierte Prozesse gezielt zu instanziieren, wird beispielsweise in [41] vorgestellt. Datenobjekte besitzen als Informationsmodell neben ihren Attributen auch ein Modell ihres Lebenszyklus. Ihr Zustand bestimmt, welche Aktivitäten innerhalb der Prozessfragmente zur Bearbeitung ausgewählt werden können und welche abgeschlossen sind (vgl. [42]). Das Überspringen oder auch eine erneute Bearbeitung einer Aktivität ist so möglich, während beispielsweise ein *Token* in Kombination mit dem Datenzustand zur Ausführungseinschränkung genutzt werden kann. Insgesamt entsteht so ein flexibles Ausführungsmodell und schließlich wird ein Zustand erarbeitet, der durch die Datenobjekte und ihre Zustände das Gesamtziel des Case erfüllt (vgl. [43]). Eine Formalisierung von Aktivitätszuständen, Datenzuständen und ihr Zusammenspiel im Rahmen von BPMN-Fragmenten unter Verwendung weniger Ereignisse findet sich in [25]. Aus diesem Ansatz entwickelte sich das in 2.1.2 beschriebene PCM wie in [16] gezeigt.

Weitere Ansätze fußen auf der Graphentheorie, sollen aber nicht weiter besprochen werden (siehe [44] und [45] für den sogenannten *Dynamic Condition Response Graphs*-Ansatz und [46] für einen

ontologiebasierten Ansatz). Eine weitere Übersicht über Ansätze zur Modellierung von variablen Teilen in Geschäftsprozessmodellen wird in [47] gegeben.

2.3.2 Modellierung von CM Anwendungen mit CMMN

Forschung zur Modellierung mit CMMN behandelt vorrangig die Untersuchung des Standards und vergleicht ihn hierzu beispielsweise mit BPMN oder stellt ihn CM-Konzepten gegenüber. Daneben finden sich in den vergangenen Jahren verstärkt Anwendungsbeispiele von CMMN, um variable und flexible Modelle zu erstellen.

Eine Gegenüberstellung von CMMN und BPMN und eine Literaturrecherche zu verschiedenen Aspekten von CMMN gegenüber BPMN findet sich in [48]. Ein darin vollzogener semantischer Vergleich der Elemente kommt zu ähnlichen Ergebnissen wie in [32]. Die Stärken von BPMN liegen klar in der Modellierung von stark strukturierten Routineprozessen, während CMMN andere Ansätze verfolgt und eine größere Flexibilität insbesondere während der Ausführung unterstützt. Eine Schwäche von CMMN ist die stark reduzierte Palette an Ereigniselementen: So ist nicht klar, wie von außerhalb eines Falls kommende Ereignisse verarbeitet werden sollen.

In [49] werden BPMN und CMMN vor dem Hintergrund der Unterstützung von Routinen in bestehenden Organisationen verglichen. CMMN unterstützt die aufgestellten Anforderungen besser als BPMN. Insbesondere der Aspekt eines erweiterten Kontexts durch den Zugriff auf Daten unabhängig von gerade aktiven Aufgaben wird durch CMMN unterstützt und als vorteilhaft erachtet.

Modellierungen mit CMMN machen vor allem Gebrauch von den flexiblen Strukturen. So werden in [50] Notfallmaßnahmen als Reaktion auf eine Flutkatastrophe in einem Modell vereinheitlicht. Im Zentrum steht eine wiederholbare (also durch den *Repetition decorator* gekennzeichnete) `DecisionTask`, die Wasserstände überwacht und notwendige `Task`-Instanzen auslöst, die in `Stage`s gruppiert sind. Im Vergleich zur Modellierung mit Hilfe von UML Zustandsdiagrammen wird die Modellierung mit CMMN bevorzugt.

Eine weitere Arbeit vereint in [51] mit Hilfe von CMMN die vorgesehenen Behandlungspfade verschiedener Krankenhäuser. Um die bestehenden Unterschiede und Varianten vereinheitlicht zu modellieren, wurden zuvor in BPMN modellierte Modelle in CMMN übertragen und mit Hilfe von *discretionary items* die Unterschiede herausgestellt. So ist es möglich, ein Modell als Basis für verschiedene Ausprägungen „zur Laufzeit“ der Behandlung zu nutzen.

In [52] wird ein Ausschnitt der verfügbaren CMMN Elemente im Rahmen einer erweiterten Wiki-Plattform angewendet, um einfache Endanwender bei der Strukturierung von Wissensarbeit zu unterstützen. Es integriert einen grafischen Editor, um einfache CMMN Modelle mit einer kleinen Auswahl an Elementen zu erstellen und in das Wiki zu integrieren.

Eine Verbindung zu einem Content Management Interoperability Services (CMIS) konformen Repository basierend auf dem Informationsmodell aus CMMN wird in [53] beleuchtet. Es zeigt zwei Alternativen zur Verknüpfung von CMMN-Systemen und CMIS-Repositorys auf. Ein integrierter

Ansatz wird beschrieben, bei dem das Repository in eine CMMN Engine eingebettet ist. Ein weiterer Ansatz, der beschrieben wird, ist die Verbindung zu einem externen CMIS konformen Repository.

2.3.3 Case Management Implementierungen in der Forschung

Um Case Management (und darüber hinaus flexible Prozesse) durch IT-Systeme zu unterstützen, gibt es wenige Implementierungen in der Forschung, die kurz hinsichtlich ihres Ursprungs, des verwendeten und implementierten Ansatzes und der Aktivität betrachtet werden sollen.

Tabelle 1 zeigt ausgewählte Implementierungen aus der Forschung auf. „Chimera“ wurde im Kern in Java implementiert und verwendet eine REST-Architektur für eine webbasierte Oberfläche. Um CM zu unterstützen, verwendet es einen auf BPMN-Fragmenten basierten hybriden Ansatz. Die Fragmente werden durch Fluss- und Datenzustände gesteuert. Zur Modellierung wird das Werkzeug „Gryphon“ benutzt. Es unterstützt kein CMMN. Das Projekt war das letzte Mal im April 2017 verstärkt aktiv, ist aber offen zugänglich[13].

Anders gestaltet es sich mit „PHILHARMONICS Workflows“, das an der Universität Ulm entwickelt wird. Da es in Kooperation mit der Persis GmbH entwickelt wird, ist es nicht öffentlich. Im Rahmen des Projekts wurden auch in diesem Jahr Aufsätze veröffentlicht. Zuletzt wurde die Entwicklung eines grafischen Editors für das Werkzeug vorgestellt[14]. Das Werkzeug unterstützt kein CMMN.

Ein weiteres Werkzeug ist „Barcelona“ (auch BizArtifact genannt), das aus einem Forschungsprojekt mit Unterstützung durch IBM hervorging. Auch dieses Werkzeug unterstützt kein CMMN, orientiert sich aber am Guard-Stage-Milestone-Ansatz (s. Abschnitt 2.3.1). Das in Java implementierte Werkzeug umfasst webbasierte Benutzeroberflächen, ein grafisches Modellierungswerkzeug, sowie eine Ausführungsumgebung. Das Projekt war das letzte Mal im Jahre 2013 aktiv.

Implementierung	Ursprung	Ansatz	CMMN?	Referenz(en)
Chimera[15]	Hasso-Plattner-Institut	Hybrid, BPMN-Fragmente	Nein	[43][16]
PHILHARMONICS Workflows[16]	Universität Ulm	Objektorientier Ansatz	Nein	[54][55]
Barcelona/BizArtifact[17]	IBM	Guard-Stage-Milestone	Nein	[56]
Connecare	EU Projekt; u.a. TU München	u.a. CMMN	Ja	[57; 58]

Tabelle 1: Ausgewählte forschungsorientierte CM-Implementierungen

[13] Siehe hierzu das Projekt auf GitHub https://github.com/bptlab/chimera/graphs/commit-activity .

[14] Siehe http://dbis.eprints.uni-ulm.de/1540/ .

[15] Das Projekt ist zu erreichen unter https://bpt.hpi.uni-potsdam.de/Chimera/ .

[16] Das Projekt ist zu erreichen unter https://www.uni-ulm.de/in/iui-dbis/forschung/laufende-projekte/philharmonic-flows/ .

[17] Die Projektdateien sind unter https://sourceforge.net/projects/bizartifact/ zu erreichen.

Das letzte in der Tabelle aufgelistete Projekt „Connecare“, ein durch die Europäische Union gefördertes Großprojekt, soll im Kern leicht modifizierte CMMN Modelle nutzen, um Pläne zur Patientenversorgung als Fälle zu modellieren. Tasks werden hierzu mit webbasierten Masken verbunden und die Verläufe auf Basis der CMMN Modelle gesteuert. Zwar finden sich Veröffentlichungen, die auch grafisch und semantisch erweiterte CMMN Modelle zeigen, aber wie diese konkret umgesetzt und zur Laufzeit unterstützt werden, ist nicht weiter beschrieben.

3. Konzept eines Case Management Frameworks

In diesem Kapitel wird das Konzept eines Case Management Frameworks vorgestellt. Nachdem das übergeordnete Ziel des Frameworks erläutert wird, werden Anforderungen für Case Management Applikationen aufgeführt und die hieraus abgeleitete Architektur und die enthaltenen Komponenten beschrieben. Kapitel 4 zeigt die prototypische Umsetzung des Konzepts. Als Basis dienen die im Folgenden aufgestellten Anforderungen, sowie die in Abschnitt 2.2 vorgestellten CMMN-Strukturen.

3.1 Übergeordnetes Ziel des Frameworks

Ein wesentliches Ziel des Frameworks ist es, dass es eine leichtgewichtige, schnelle Realisierung von Case Management Anwendungen ermöglicht und die erstellten Applikationen einem einheitlichen Architekturstil folgen. Das Framework ist hierzu einerseits als Referenzarchitektur, andererseits auch als (prototypisches) Software-Framework zu sehen, durch das Fälle entwickelt und schließlich bearbeitet werden können sollen. Im Kern soll das Framework hierzu CM Konzepte umsetzen, wie sie in CMMN Ausdruck finden. Es folgt dabei der Ausführungssemantik der Spezifikation. Auf Basis von CMMN Modellen und der Framework-Implementierung sollen Applikationen erstellt werden können, die sich der von CMMN bekannten Ausführungssemantik entsprechend verhalten. Fallstrukturen sollen mehrfach instanziiert werden und im jeweiligen Kontext bearbeitet werden können.

Wie bereits in der Abgrenzung aufgezeigt, ist das Ziel nicht, eine voll funktionsfähige, schwergewichtige BPM- bzw. CMMN-Plattform zu entwerfen oder zu implementieren, sondern leichtgewichtige Kernstücke zur Bearbeitung von Case Management Anwendungen bereitzustellen, die individuell erweitert und in bestehende Systeme integriert werden können.

3.2 Anforderungen für Case Management Applikationen

In diesem Abschnitt werden grundlegende Anforderungen für CM-Applikationen aufbereitet. Sie sind der Ausgangspunkt für das Konzept und die prototypische Implementierung des Frameworks. Die Liste ist nicht als vollständig anzusehen, sondern umfasst Minimalanforderungen.

Die Anforderungen lassen sich in die fünf Bereiche CM-Kern, aus CMMN abgeleitete Anforderungen, Anforderungen an eine Service- und Persistenzschicht, sowie aus den Problembereichen „traditioneller“ WFMS abgeleitete Anforderungen gliedern.

Die Anforderungen sind vor dem Hintergrund einer leichtgewichtigen, strukturierten Erstellung von CM-Applikationen zu sehen. Einerseits sind sie aus den vorangegangenen Charakteristika von CM abgeleitet, andererseits aus CMMN, welches Kernkonzepte von (P)CM unterstützt.

3.2.1 Basisanforderungen

Basisanforderungen leiten sich aus dem im vorherigen Abschnitt beschriebenen Ziel und der generellen Arbeitsweise des CM ab.

REQ 1. Das Framework soll einen strukturierten Ansatz für die Entwicklung von CM Applikationen anbieten. Entwicklern soll es die systematische Implementierung von CM Applikationen ermöglichen.

REQ 2. Das Framework soll leichtgewichtig sein. Anbieterspezifische Technologien sollen vermieden werden, wie beispielsweise spezifische Applikationsserver oder Datenbankmanagementsysteme. Stattdessen sollen offene Standards genutzt werden.

REQ 3. Fallbearbeiter (Case Workers) sollen durch grafische Benutzerschnittstellen bei der Bearbeitung des Falls unterstützt werden. Fallbearbeitern soll es ermöglicht werden, Daten eines Falls einzusehen und manipulieren zu können. Daten können auch unstrukturierte Dokumente sein.

REQ 4. Fallbearbeiter mit verschiedenen Rollen sollen unterstützt werden. Hierzu wird ein Rollensystem benötigt, um beispielsweise den Zugriff auf bestimmte Funktionen oder sensible Daten zu unterbinden.

REQ 5. Eine Schnittstelle für externe Systeme soll bereitgestellt werden. Webservices werden benötigt, um webbasierten Applikationen die Kommunikation mit der CM-Applikation zu ermöglichen. Beispielsweise müssen Rückmeldungen externer Systeme verarbeiten werden können, nachdem sie aus einem Fall heraus gestartet wurden und ihre Aufgabe erledigt haben.

REQ 6. Lang laufende Fälle müssen unterstützt werden. Dies erfordert einen Persistenzmechanismus, um Fälle dauerhaft zu speichern und bei Bedarf auch Jahre später wieder abrufen zu können.

3.2.2 Anforderungen an einen CM-Kern

Anforderungen an einen CM-Kern umfassen die Struktur der konstituierenden Elemente einer CM-Applikation. Der Kern soll als Herzstück des Software-Frameworks zu sehen sein.

REQ 7. Der Kern soll sich an einem vereinfachten Metamodell der CMMN-Spezifikation orientieren. Eine besondere Berücksichtigung findet hierbei die CMMN zu Grunde liegende Ausführungssemantik wie in der Spezifikation beschrieben.

REQ 8. Der Kern soll Basiselemente bereitstellen, die CM-Konzepte repräsentieren. Diese können entsprechend der CMMN Spezifikation miteinander kombiniert werden, um CM-Applikationen zu erstellen.

3.2.3 Aus CMMN abgeleitete Anforderungen

Diese Anforderungen beziehen sich primär auf das Verhalten der für den Kern geforderten Bausteine.

REQ 9. Durch Basiselemente des Kerns sollen Klassen bereitgestellt werden, die CMMN Elemente repräsentieren, wie beispielsweise `Stage`, `HumanTask` und `Milestone`. Diese bilden die Grundlage für Fallstrukturen und verhalten sich entsprechend der CMMN-Ausführungssemantik.

REQ 10. Das Framework soll die CMMN-Ausführungssemantik umsetzen. Es muss nachvollziehbar sein, wie sich auf Basis des Frameworks erstellte Applikationen und sich darin abgebildete Fallstrukturen verhalten.

REQ 11. Das Sentry-Konzept soll unterstützt werden, um Elemente miteinander verknüpfen zu können. So kann auf Zustandsänderungen von Elementen und Daten reagiert werden, um dynamische Abläufe zu ermöglichen.

3.2.4 Anforderungen an eine Serviceschicht

Services stellen intern und extern Schnittstellen zur Bearbeitung einer Fall-Instanz bereit, aber auch übergreifende Funktionen, um beispielsweise Fälle zu starten.

REQ 12. Das Framework soll eine ausgeprägte Serviceschicht aufweisen. Diese soll verschiedene Interaktionsmöglichkeiten mit dem Kern bereitstellen, um Elemente entsprechend der Spezifikation zu manipulieren und Daten zu bearbeiten.

REQ 13. Die angebotenen Services sollen grundlegende CRUD-Funktionen bereitstellen, die zur Bearbeitung von Fällen benötigt werden, wie beispielsweise durch die gezielte Zustandsänderung einer `HumanTask`.

3.2.5 Anforderungen an eine Persistenzschicht

Eine Persistenzschicht ist nötig, um Fälle und Zustände enthaltener Elemente und Datenwerte dauerhaft speichern zu können.

REQ 14. Eine Persistenzschicht wird zur dauerhaften Speicherung von Fällen, ihren Zuständen und Daten benötigt. Fälle können kurz oder sehr lang sein. Auditoren können Zugriff auf aktive oder bereits abgeschlossene Fälle verlangen.

REQ 15. Eine Persistenzschicht soll eine transparente Verarbeitung des Kerns im Rahmen der Serviceschicht ermöglichen und auf einer objektrelationalen Abbildung aufbauen. Somit kann direkt mit Objekten gearbeitet werden, was die Entwicklung vereinfacht.

3.2.6 Anforderungen abgeleitet aus Problembereichen des traditionellen Prozessmanagement

Diese Anforderungen beziehen sich auf die vier Problembereiche des „traditionellen Prozessmanagements“ (siehe Abschnitte 2.1.1 und 2.1.4) und sollen helfen, diese zu vermeiden.

REQ 16. Um „Kontextblindheit“ zu vermeiden, sollen alle berechtigten Fallbearbeiter alle Daten eines Falls einsehen und für ihre Entscheidungen bezüglich des Verlaufs nutzen können.

REQ 17. Flexible Ausführungspfade sollen möglich sein, um verschiedene Wege zur Erfüllung von Meilensteinen und Zielen anbieten zu können. Ein Fokus soll nicht darauf liegen, was gerade getan werden *muss* (wie bei imperativen Ansätzen), sondern darauf, was getan werden *könnte* (wie bei deklarativen Ansätzen) um ein Ziel zu erreichen.

REQ 18. Arbeitsteilung soll durch ein simples Rollensystem ermöglicht werden. Dieses soll es Fallbearbeitern ermöglichen, Ereignisse auslösen zu können und Aufgaben für sich zu beanspruchen.

REQ 19. Der Zugriff auf die Daten einer Fall-Instanz soll flexibel gestaltet sein und nicht von der gerade angebotenen oder bearbeiteten Aktivität abhängen.

3.3 Architektur und Komponenten einer CM Anwendung

In diesem Abschnitt wird zunächst die grundlegende Architektur einer mit dem Framework erstellten CM Applikation vorgestellt. Anschließend werden die darin eingebetteten Komponenten detaillierter vorgestellt.

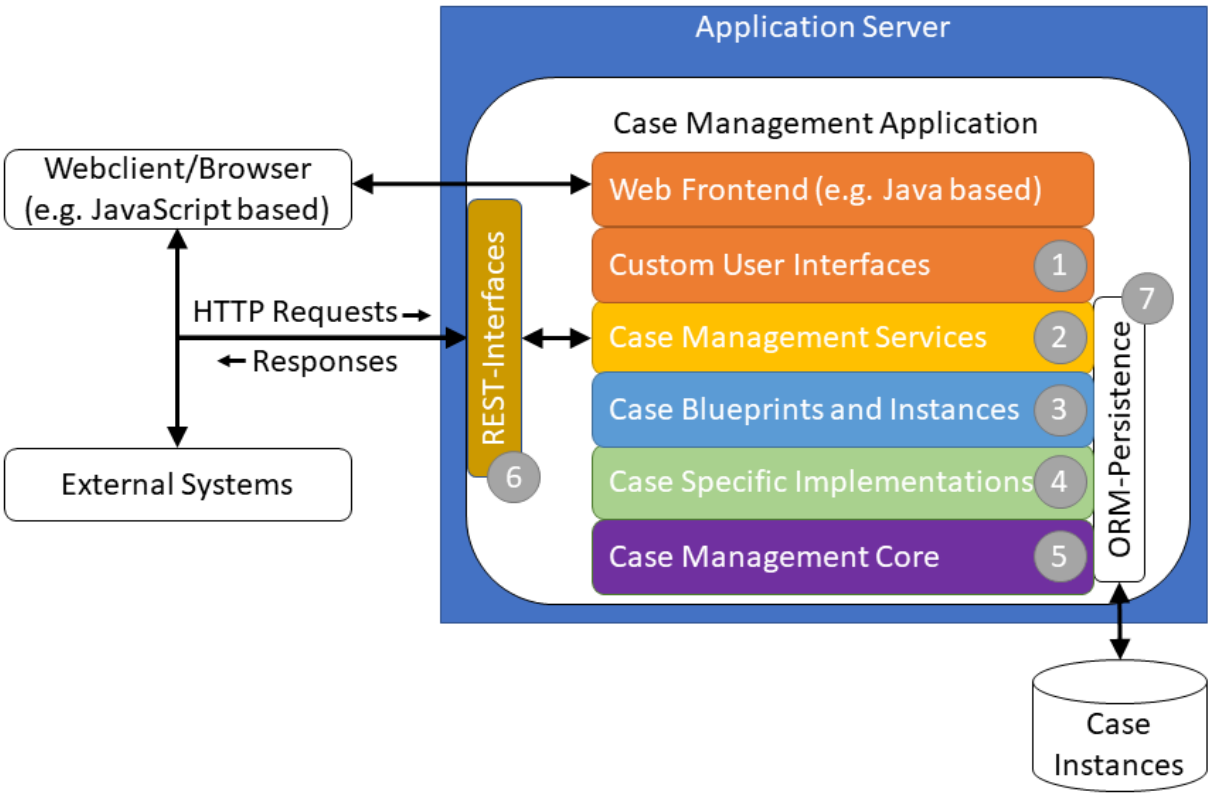

Abbildung 19: Grundlegende Architektur des Frameworks

Abbildung 19 zeigt die Architektur einer Applikation auf Basis des Frameworks, die im Folgenden beschrieben wird. **REQ 1** fordert grundlegend einen strukturierten Ansatz für die Implementierung von CM Applikationen. Aus dieser und weiteren Anforderungen lässt sich eine Schichtenarchitektur ableiten. Eine mit dem Framework erstellte CM-Applikation lässt sich in sieben Schichten gliedern:

(1) Web Frontend/Custom User Interfaces: Sie umfasst Masken zur Interaktion mit dem Framework. Über sie greifen Fallbearbeiter auf das Framework und Fall-Instanzen zu.

(2) Case Management Services: Sie umfasst Schnittstellen, auf deren Basis mit dem Framework interagiert wird, um Fälle zu starten und zu bearbeiten und um Daten zu bearbeiten.

(3) Case Blueprints and Instances: Sie umfasst vorgefertigte Fallstrukturen, die instanziiert werden können oder bereits instanziiert sind. Durch die Services können diese bearbeitet werden.

(4) Case Specific Implementations: Sie umfasst anwendungsspezifische, aus den zugrundeliegenden Modellen abgeleitete Implementierungen und Geschäftsregeln bzw. -logik.

(5) Case Management Core: Sie umfasst wiederverwendbare Basisbausteine des Frameworks und basiert auf vereinfachten Strukturen der CMMN-Spezifikation. Zusammen entstehen aus ihnen die Schichten (4) und (5). Sie bilden die Ausführungssemantik von CMMN ab.

(6) REST-Interfaces: Sie umfasst Schnittstellen für externe Systeme.

(7) ORM-Persistence: Sie umfasst einen objektrelationalen Ansatz zur dauerhaften Speicherung von Fall-Instanzen, die durch Masken der ersten Schicht unter Verwendung der Services bearbeitet werden können.

Die Architektur folgt einer Client-Server-Architektur. Eine mit dem Framework erstellte CM-Applikation läuft auf einem webfähigen (beispielsweise auf Java basierten) Applikationsserver. Fallbearbeiter können mit der Applikation über zwei Wege kommunizieren: Sie können entweder über einen üblichen Browser auf ein (beispielsweise Java basiertes) integriertes Web-Frontend (1) zugreifen, oder mit Hilfe eines (beispielsweise JavaScript basierten) Webclients mit der Applikation über REST-Schnittstellen (6) kommunizieren. Externe Systeme, die zum Beispiel `ProcessTasks` unterstützen, können ebenso über diese REST-Schnittstellen mit der Applikation kommunizieren und Daten austauschen.

REST-Schnittstellen und integriertes Web-Frontend bauen auf internen *Case Management Services* (2) auf. Diese dienen der internen Interaktion mit *Case Blueprints and Instances* (3), *Case Specific Implementations* (4), dem Kernstück des Frameworks in Form des *Case Management Core* (5) und mit der unterstützenden *ORM-Persistence*-Schicht (7). In den folgenden Abschnitten werden die einzelnen Schichten beschrieben und gezeigt, wie diese die Anforderungen des vorherigen Abschnitts aufgreifen.

3.3.1 Präsentationsschicht (*Web Frontend/Custom User Interface*)

In dieser Schicht befinden sich grafische Benutzerschnittstellen für Fallbearbeiter. Sie greift insbesondere Anforderung **REQ 3** auf. Sie umfasst Übersichten und Informationen zu laufenden Fällen, verfügbaren Aufgaben/`HumanTasks` und Daten. Abbildung 20 zeigt schematisch die Inhalte und das Zusammenspiel zwischen der Präsentationsschicht und den Fall-Blaupausen beziehungsweise laufenden Instanzen. Ein Zugriff geschieht über die im nächsten Abschnitt beschriebene Serviceschicht.

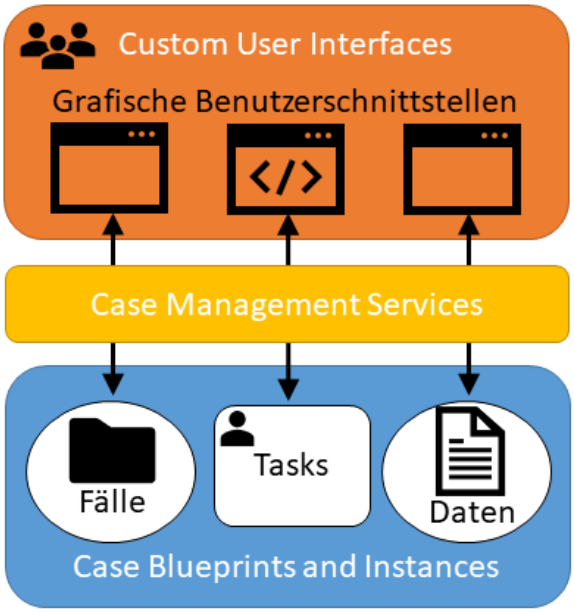

Abbildung 20: Custom User Interfaces

Individuelle Masken, etwa zur Bearbeitung von `HumanTasks`, sind ebenso dieser Schicht zuzuordnen und als Vorlage für verschiedene Fall- und `Task`-Instanzen zu verstehen. Masken zur Darstellung fallweiter Daten und Informationen greifen **REQ 16-19** auf. Sie ermöglichen es, unabhängig von gerade verfügbaren Aufgaben den Fallkontext beziehungsweise enthaltene Daten zu bearbeiten.

3.3.2 Serviceschicht (*Case Management Services*)

REQ 12 und **13** fordern eine ausgeprägte Serviceschicht zur Interaktion mit den verschiedenen Komponenten und Schichten. Die Services (Schicht 2) sind das Bindeglied zwischen Fallbearbeitern/externen Systemen und der CM Applikation. Sie bieten verschiedene Funktionen an, um Informationen über Fallinstanzen, enthaltene Elemente, Aufgaben und Daten abzurufen.

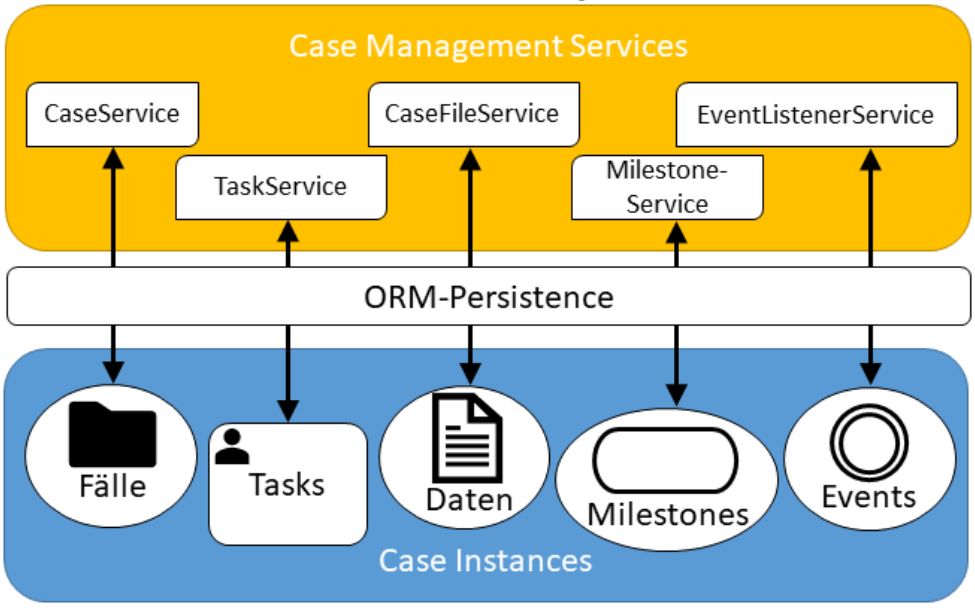

Abbildung 21: Case Management Services

Abbildung 21 zeigt die in dieser Schicht enthaltenen Services `CaseService`, `TaskService`, `CaseFileService`, `MilestoneService` und `EventListenerService`. Sie bieten entsprechend dem namensgebenden Element Funktionen an, um persistierte Fälle zu bearbeiten. Mit Hilfe der ORM-Persistenzschicht können ganze Fallstrukturen als Objekte geladen werden, oder nur einzelne Elemente wie `Milestone`s oder bestimmte `CaseFileItem`s. Diese können anschließend mit Hilfe der Services manipuliert werden.

Intern werden die Services für verschiedene Aufgaben genutzt. Durch sie können bestehende Fallstrukturen geladen, manipuliert oder gelöscht werden. Auch können durch sie neue Fallinstanzen gestartet und persistiert werden. Tasks können geladen, durch einen Fallbearbeiter beansprucht und anschließend durch entsprechende Masken der Präsentationsschicht bearbeitet werden. Außerdem bieten sie an, Elemente ordnungsgemäß in einen anderen Zustand zu überführen, wie etwa durch das Auslösen eines `EventListener`s.

3.3.3 Fall-Blaupausen und Instanzen (*Case Blueprints and Instances*)

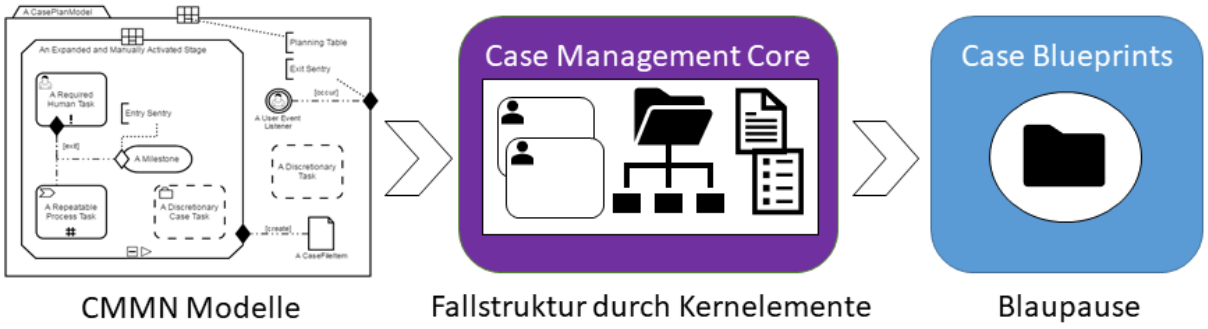

Abbildung 22: Case Blueprints

Dieser Schicht sind neben laufenden Fall-Instanzen vor allem vordefinierte Fallstrukturen zuzuordnen. Auf Basis von CMMN-Modellen werden durch Entwickler mit Hilfe der Framework-Bausteine Fall-Blaupausen erstellt. Die Modelle werden hierzu in entsprechende Elemente des Frameworks übertragen. Das Vorgehen wird schematisch in Abbildung 22 gezeigt. Die entstandenen Blaupausen können durch einen `CaseService` instanziiert werden, um einen neuen Fall zu starten. Auf Blaupausen greifen auch `CaseTask`s zurück, um untergeordnete Fälle zu instanziieren.

3.3.4 Anwendungsspezifische Implementierungen (*Case Specific Implementations*)

Einige Elemente benötigen individuelle Implementierungen zur Konkretisierung ihres Ausführungsverhaltens. Hierzu gehören insbesondere `ProcessTask`s, `CaseTask`s, aber auch Elemente, die *decorators* aufweisen. Die anwendungsspezifischen Implementierungen sind so gestaltet, dass sie für verschiedene Fall-Instanzen in deren Kontext wiederverwendet werden können.

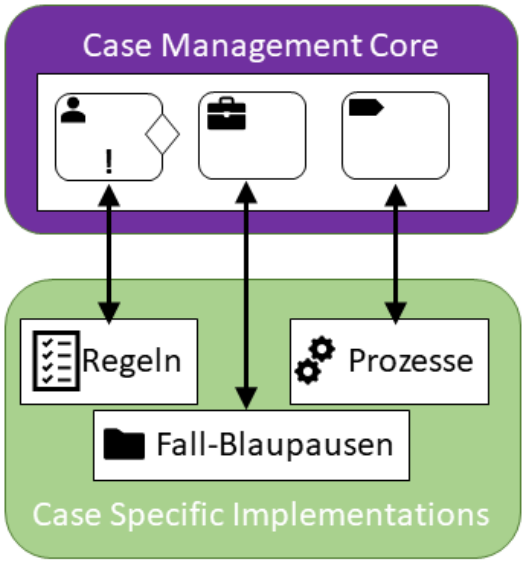

Abbildung 23: Case Specific Implementations

Abbildung 23 zeigt beispielhaft drei Elemente aus CMMN in der Kernschicht (siehe nächster Abschnitt): Einen `HumanTask` mit einem *Required Rule decorator* und einem `Sentry` samt (grafisch nicht darstellbaren) `IfPart`, daneben einen `CaseTask`, sowie einen `ProcessTask`. Daneben wird ihre

Zuordnung zu spezifischen Implementierungen der anwendungsspezifischen Implementierungen gezeigt. Diese Schicht unterstützt **REQ 9** und greift insbesondere die Anforderungen **REQ 10** auf.

Decorators und `IfParts` sind in der Kategorie „Regeln“ zusammengefasst. Zur Laufzeit werden konkrete Implementierungen geladen, die entsprechende Logik enthalten. Auf dieser Basis werden beispielsweise bestimmte Werte referenzierter `CaseFileItem`s evaluiert. Die Ausgestaltung bleibt dem Entwickler allerdings frei, sodass auch andere Quellen als Basis für die Evaluierung dienen können.

Gehen `CaseTask`s in den Zustand *Active* über, benötigen diese Anweisungen darüber, welche Fall-Blaupause in welcher Art instanziiert werden soll. Gegebenenfalls müssen weitere Parameter angegeben werden, um die Blaupause zu instanziieren beziehungsweise zu konfigurieren. Die anwendungsspezifische Implementierung ist ein Bindeglied zwischen Fall-Instanz und Blaupausen.

Werden `ProcessTask`s gestartet (wenn sie also in den Zustand *Active* übergehen), benötigen auch diese Anweisungen darüber, was eigentlich geschehen soll. Eine Prozessimplementierung kann hierbei ein einfacher Algorithmus zum Versenden einer E-Mail sein, oder auch die Kommunikation mit einem externen System, welches den eigentlichen Prozess abbildet und ausführt. Auch hier bleibt die Ausgestaltung der Entwicklerin frei.

3.3.5 Case Management Kern (*Case Management Core*)

Wie bereits durch die Beschreibung der anderen Schichten angedeutet, beinhaltet diese Schicht die CM Kernelemente. Sie werden von allen Schichten verwendet. Wiederverwendbare Klassen repräsentieren CMMN-Elemente und dienen als Basisbausteine für CM Applikationen, die mit dem Framework erstellt werden. Die Kernklassen verhalten sich entsprechend dem durch CMMN spezifizierten Ausführungsverhalten. Abschnitt 4.1 geht weiter auf die Bausteine ein. Diese Schicht greift **REQ 12** und **13** auf und unterstützt **REQ 15**.

3.3.6 REST-Schnittstellen (*REST-Interfaces*)

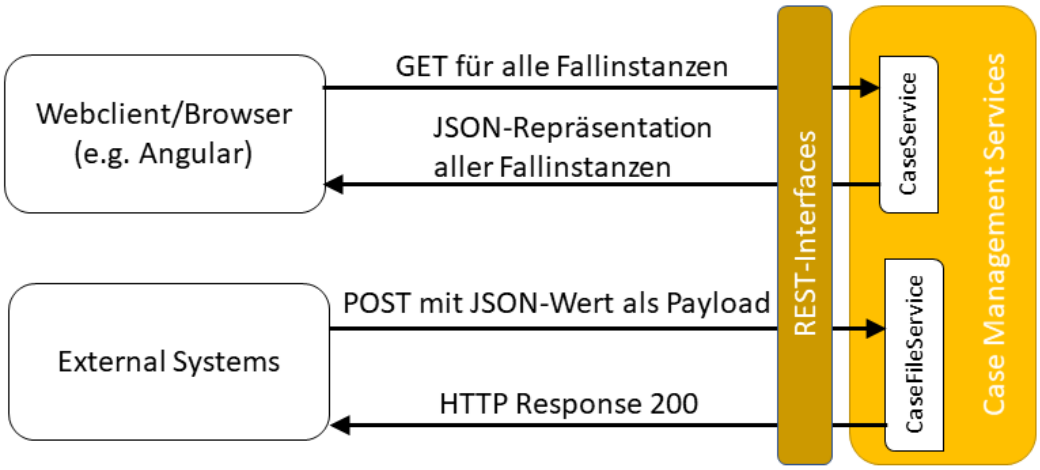

Abbildung 24: REST-Interfaces

Diese Schicht greift insbesondere **REQ 5** auf und unterstützt die Erstellung der in **REQ 3** geforderten grafischen Schnittstellen externer Systeme. Zur Kommunikation mit externen Systemen wird als

leichtgewichtige Alternative zu Simple Object Access Protocol-Webservices (SOAP) auf den Ansatz des Representational State Transfer (REST) zurückgegriffen. Die angebotenen REST-Schnittstellen sind weitestgehend deckungsgleich mit den internen Services. Sie bauen auf diesen auf und stellen eine Schnittstelle für externe Systeme und Webapplikationen bereit. Als Datenaustauschformat wird die Java Script Object Notation (JSON) verwendet, die beispielsweise direkt von JavaScript-Applikationen weiterverwendet werden kann (vgl. [59], insbesondere Kapitel 6; siehe auch [60]).

Je nach Art der Funktion können verschiedene HTTP-Requests verarbeitet werden. GET-Requests dienen dem Abfragen von Informationen wie etwa verfügbarer Tasks einer Fall-Instanz. POST-Requests werden dazu genutzt, um eine `HumanTask` in einen anderen Zustand zu überführen, beispielsweise um diese durch den Übergang *complete* abzuschließen. PUT-Requests werden beispielsweise zur Übertragung und der Erstellung von Dokumenten im Rahmen von `CaseFileItem`s genutzt. Gelöscht werden können diese durch DELETE-Requests. Abbildung 24 zeigt beispielhaft zwei HTTP-Requests: Ein Webclient fragt über GET eine Liste aller aktiven Fall-Instanzen an. Zurückgeliefert wird eine JSON-Repräsentation aller Fallinstanzen. Der andere Request erfolgt durch ein externes System. Es ändert durch einen POST-Request mit einem JSON-Wert eine Property eines `CaseFileItem`s. Ein Erfolg wird durch den HTTP-Statuscode 200 („ok“) quittiert. Zu sehen ist der Rückgriff auf die bestehenden internen Services, in diesem Fall `CaseService` und `CaseFileService`.

3.3.7 Persistenzschicht (*Persistence*)

Diese Schicht greift **REQ 6**, **14** und **15** auf. Unterstützt werden diese durch **REQ 12** und **13**, also durch die zuvor dargestellte Service-Schicht. Fall-Instanzen, die Zustände der enthaltenen Elemente und Daten in `CaseFileItem`s müssen dauerhaft gespeichert werden können. Dokumente und andere Medien können darüber hinaus auf einem Fileserver gespeichert werden.

Die Persistenzschicht besteht aus zwei Teilen: Einer objektrelationalen Abbildung (object-relational mapping, kurz ORM) und einer relationalen Datenbank als Speichermedium. Die Nutzung eines ORM-Frameworks erleichtert die Arbeit mit dem Framework durch die direkte Arbeit mit Objekten. Komplexe Abfragen und Algorithmen, um Objektstrukturen zu laden und von relationalen Strukturen in objektorientierte Strukturen zu transformieren, können weitestgehend durch ein ORM-Framework entfallen oder zumindest stark vereinfacht werden (vgl. [61], S. 215ff.).

4. Prototypische Umsetzung des Frameworks

In diesem Kapitel wird die prototypische Umsetzung des Frameworks vorgestellt. Das Konzept aus dem vorherigen Kapitel wird weiter verfeinert. Zunächst wird die technologische Basis im Kontext des Prototyps vorgestellt. Anschließend werden die Basisbausteine der Referenzarchitektur beschrieben. Die erstellten Klassenstrukturen sind vor dem Hintergrund einer vereinfachten CMMN-Basis entstanden. Anschließend werden adaptierte Entwurfsmuster behandelt, welche insbesondere die Ausführungssemantik von CMMN 1.1 unterstützen. Dieses Kapitel dient auch dazu, einen tieferen Einblick über die Funktionsweise des Frameworks zu ermöglichen. So werden neben Klassendiagrammen auch Sequenzdiagramme und Quellcode-Fragmente gezeigt, um interne Strukturen und Abläufe zu veranschaulichen.

4.1 Technologische Basis der Implementierung

Für die prototypische Implementierung wurden Java 1.8 und die Java Enterprise Edition (JEE) in der siebten Version gewählt[18]. Als Entwicklungsumgebung wurde Eclipse (Oxygen 3) verwendet. Die Inhalte der Präsentationsschicht (Custom User Interfaces der ersten Schicht) wurden mit Hilfe des Java-Webframeworks Vaadin 8 erstellt[19]. Als Datenbank wurde MySQL (MariaDB 10.1.37) verwendet.

4.1.1 Kurzeinführung in die Java Enterprise Edition

JEE bündelt verschiedene Standards und definiert transparente Schnittstellen für die Entwicklung von verteilten, transaktionsbasierten und mehrschichtigen Enterprise-Java-Anwendungen. JEE-konforme Applikationsserver verwenden intern Referenzimplementierungen des Standards zur Umsetzung der spezifizierten Module (vgl. [62], S. 1ff.). Durch den offenen Standard soll **REQ 2** unterstützt werden (vgl. Abschnitt 3.2.1). Die JEE-Architektur besteht im Grunde aus einem oder mehreren Client Systemen, sowie einem JEE Server, der je nach Konformität einen Teil oder alle Aspekte des Standards unterstützt. Clients kommunizieren mit der Applikation, die vom JEE-Server gesteuert ausgeführt wird.

Als **Applikationsserver** für den Prototyp wurde die leichtgewichtige Open Source Implementierung TomEE (PluME 7.1.0 mit Tomcat 8.5) gewählt[20]. TomEE ist ein Tomcat Applikationsserver, der um verschiedene JEE Komponenten erweitert wurde. Die Version „PluME“ wurde gewählt, um eine manuelle Einbindung der verwendeten Standards und ihrer Referenzimplementierungen zu vermeiden. Ein Funktionsüberhang wurde somit bewusst in Kauf genommen. Diesem wird aber durch eine selektive Aktivierung verschiedener Komponenten in Abhängigkeit von vorhandenen Konfigurationsdateien entgegengewirkt.

JEE folgt dem Prinzip „convention over configuration“, das heißt, dass Konventionen des Standards standardmäßig eingesetzt werden und Abweichungen spezifisch deklariert werden (vgl. [63], Abschnitt 4.21). JEE-Standards nutzen primär Annotationen, um Klassen als Teil eines JEE Standards zu markieren. Durch sie wird auch ihr Verhalten definiert. Daneben finden sogenannte *deployment*

[18] Für eine erweiterte Übersicht siehe https://docs.oracle.com/javaee/7/tutorial/overview007.htm .

[19] Die Vaadin-Dokumentation ist unter https://vaadin.com/docs/v8 zu erreichen.

[20] Siehe http://tomee.apache.org/comparison.html für einen Vergleich der verschiedenen Funktionsumfänge der TomEE-Versionen.

descriptors Anwendung. Dies sind Konfigurationsdateien, die beispielsweise Details über Servlet-Konfigurationen oder Datenbankverbindungen beinhalten. Sie können teilweise komplett entfallen.

JEE Server besitzen verschiedene Module (vgl. [63], S. 3ff.): Unter anderem gibt es einen Web Container, der Servlets bereitstellt und HTTP-Requests verarbeitet. Servlets werden von Vaadin genutzt, um grafische Benutzeroberflächen aufzubauen und bereitzustellen. Ferner können REST-Schnittstellen im Grunde als Servlets betrachtet werden.

Ein EJB Container stellt bei Bedarf instanziierte Enterprise Java Beans (EJB) bereit, die einen definierten Lebenszyklus und definierte Anwendungsbereiche haben. Zwei Beispiele sind zustandsbehaftete und zustandslose Beans (vgl. [63] S. 243ff.; [61], S. 125ff.).

Zustandsbehaftete Beans sind Java-Klassen, die eine 1-1 Verbindung zu einem Client im Rahmen eines HTTP-Requests besitzen, um so beispielsweise Login-Details über eine HTTP-Sitzung hinweg speichern zu können. Im Prototyp werden sie für diesen Zweck auf Seite des Frontend genutzt, um den aktuellen Benutzer und Kontext durch beispielsweise zuvor ausgewählte `HumanTask`s zu speichern. Gekennzeichnet werden sie im Standard durch die Annotation *@Stateful*.

Zustandslose Beans werden nach der Benutzung durch einen Client vom Container wieder in einen Pool freigegeben und zwischen etwaigen Clients verteilt. Sie werden für die Implementierungen der Dienste-Schicht benutzt und sind mit der Annotation *@Stateless* versehen. Sie beinhalten jeweils eine injizierte Referenz auf einen `EntityManager` durch die Annotation *@PersistenceContext*, die vom Applikationsserver verwaltet wird und auf eine im Server konfigurierte *DataSource* zugreift. Eine *DataSource* ist eine im Server hinterlegte Konfiguration beziehungsweise Definition, die Zugangsdaten und verwendeten Treiber für eine Verbindung zur Datenbank angibt. Ein `EntityManager` ist ein zentraler Bestandteil des JEE-ORM-Frameworks. Es bietet transparente Schnittstellen für Arbeiten mit der Datenbank an, wie etwa Abfragen und CRUD-Operationen.

4.1.2 Überblick über verwendete Standards

Im Folgenden werden die verwendeten Standards kurz im Kontext ihrer Benutzung im Framework skizziert.

Die Java Transaction API (**JTA**) ist ein Standardinterface zur Markierung von Transaktionsbereichen (vgl. [63], S. 269ff.). Methoden können abweichend von den Standardeinstellungen per Annotationen markiert werden. Der Applikationsserver beziehungsweise Container verwaltet das konfigurierte Transaktionsverhalten. Das Framework überlässt JTA die Verwaltung der Transaktionen, die für Methoden der Dienste und die Nutzung des `EntityManager` genutzt werden (vgl. [61], S. 130ff.).

Enterprise JavaBeans Technology (**EJB**) sind Komponenten, die Geschäftslogik implementieren. Sie können eigenständig oder als Baustein genutzt werden, um Geschäftslogik in einem JEE-Server auszuführen. Das Software-Framework nutzt sie für die Realisierung der Dienste-Schicht. Hier werden zustandslose, wiederverwendbare EJBs genutzt (vgl. [61]).

Contexts and Dependency Injection for Java EE CDI (**CDI**) ermöglicht eine lose Kopplung und vom Applikationsserver verwaltete Erzeugung von Objektinstanzen. Sie wird im Software-Framework beispielsweise für das Injizieren von Services in der Präsentationsschicht genutzt (s. [61], S. 495ff.).

Die Java API for RESTful Web Services (**JAX-RS**) wird für die Realisierung der REST-Schnittstellen eingesetzt. Die REST-Schnittstellen bauen auf den zustandslosen Services auf. Im Rahmen der REST-Schnittstellen wird als Datenformat die JavaScript Object Notation (JSON) genutzt. Die Java API for JSON Processing (**JSON-P**) wird für die Verarbeitung beziehungsweise Transformation von Java-Objekten in JSON-Notation und umgekehrt verwendet. JSON ist ein gängiges Austauschformat, das insbesondere von JavaScript-Applikationen verwendet wird und weniger Metadaten enthält, als beispielsweise XML-Formate (s. [60]; [59], Kapitel 1 und 6).

Die Java Persistence API (**JPA**) (s. [59], Kapitel 2; [61], S. 215ff.; [63], S. 197ff.) ist ein Standard für objektrelationale Abbildungen, um den Bruch zwischen Objektorientierung und relationaler Datenbank zu überbrücken. Zum Standard gehören verschiedene Funktionen zur Abfrage der Daten aus einer relationalen Datenbank, die typsicher in Objekte umgewandelt werden und weiterverarbeitet werden können. Einfache Java-Klassen („Plain Old Java Objects“, POJOs) werden durch die Annotation *@Entity* JPA bekannt gemacht. Weitere Annotationen werden genutzt, um beispielsweise Primärschlüssel (*@Id*) festzulegen, oder Kardinalitäten (zum Beispiel *@OneToMany*) abzubilden. Bidirektionale Assoziationen und Vererbungshierarchien werden ebenso über Annotationen definiert. Aus den annotierten POJOs können durch JPA automatisch Datenbanktabellen und Constraints erzeugt werden.

4.1.3 Vaadin-CDI für grafische Benutzerschnittstellen von Tasks

Im Prototyp wird für die Erstellung grafischer Benutzerschnittstellen (vgl. Abschnitt 3.3.1) das Java-Webframework „Vaadin“ verwendet und durch das „Vaadin CDI Addon“ ergänzt[21]. `Tasks`, insbesondere `HumanTasks`, benötigen eine grafische Benutzerschnittstelle, um die vorgesehene Aufgabe ausführen zu können. Um für eine `Task`-Instanz die korrekte Eingabemaske und Steuerung zu erzeugen, wird das Vaadin CDI Addon genutzt. Es nutzt primär die JEE-Technologien Servlets, CDI und EJBs. Zunächst wird eine Maske als Schablone für `Task`-Instanzen erstellt. Das Funktionsprinzip soll an einem Beispiel veranschaulicht werden.

`Tasks` erben von der Klasse `Element` das für eine Fallstruktur eindeutige Attribut `cmId`. Das Attribut kann aus dem Markup eines erstellten CMMN-Modells übernommen werden, sollte dann aber im Model nicht mehr geändert werden. Angenommen, die `cmId` eines `HumanTasks` ist „MaskeA“.

[21] Ein einführendes Beispiel für das Addon findet sich unter https://vaadin.com/docs/v8/framework/advanced/advanced-cdi.html.

Das Vaadin CDI Addon ermöglicht es, auf Basis von Ereignissen zwischen Masken zu navigieren. Zustandsbehaftete Beans können genutzt werden. Hierzu wird ein Ereignis geworfen und von einem Vaadin Navigator-Service gefangen. Die angegebene Maske wird erzeugt und angezeigt.

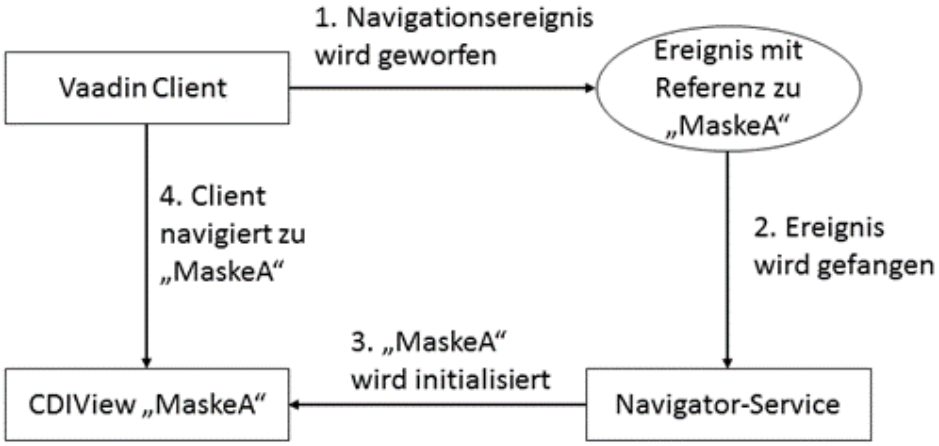

Abbildung 25: Anwendung des Vaadin CDI-Navigator-Mechanismus

Abbildung 25 zeigt schematisch den Ablauf: Ein Benutzer wählt eine verfügbare Aufgabe mit der `cmId` „MaskeA“ aus. Anschließend startet er die Arbeit an der Aufgabe und löst ein (Navigations-)Ereignis mit der Referenz zur „MaskeA“ aus (1), welches vom Vaadin Navigator-Service gefangen wird (2). Das Ereignis enthält die `cmId` des zu bearbeitenden `Task`-Objekts. Der Navigator-Service erstellt anschließend eine Instanz der gewünschten Maske (3) und der Client navigiert zur gewünschten View (4). Kontextinformationen können über eine zustandsbehaftete Bean im Hintergrund hergestellt werden.

4.2 Vereinfachte CMMN-Basis als Grundlage

Dieser Abschnitt zeigt die wichtigsten Unterschiede zur Spezifikation auf, ohne alle Unterschiede zur umfangreichen Spezifikation zu behandeln. Weitere Unterschiede werden im Kontext der vorgestellten Basisbausteine aufgezeigt.

Die Klassenstruktur des Konzepts orientiert sich stark an den Modellen der CMMN-Spezifikation in der Version 1.1. Unter den Voraussetzungen, dass das Framework leichtgewichtig und zugänglich gestaltet sein soll (**REQ 2**), wurden jedoch einige Abstriche vollzogen, die sich insbesondere auf die automatische Verarbeitung von CMMN-XML-Markups auswirken. Einige Elemente der Metamodelle wurden nicht übernommen. Auch ist die Typisierung der `CaseFileItem`s fix und nicht über eine zusätzliche `CaseFileItemDefinition`-Klasse und assoziierte `Import`-Klasse definiert.

Es sollen vier konkrete Beispiele aufgezeigt werden. Das erste Beispiel stellt die Spezialisierung `Definitions` der abstrakten Klasse `CMMNElement` dar[22]: Die Metadaten, die im CMMN-XML-Markup vorhanden sind, wie etwa die bezüglich des Erweiterungskonzept (*extensions*), werden nicht umgesetzt und einbezogen. *Extensions*, also Erweiterungspunkte der Spezifikation, werden in der Praxis primär als Basis für proprietäre Implementierungen genutzt, um produktspezifische Definitionen in das CMMN-Markup einzubinden. So werden beispielsweise Verknüpfungen zu Variablen oder Formular-

[22] Siehe Kapitel 5.1.2 ff. der CMMN 1.1 Spezifikation.

Dateien als selbst definierte Erweiterung hinterlegt. Sie weichen vom Standard ab und sind produktspezifisch.

Das zweite Beispiel für die vereinfachte Basis ist die abstrakte Klasse `PlanItemDefinition` der Spezifikation[23]. Sie kann als Ebene zwischen `CMMNElement` als Basisklasse aller Elemente und diesen Elementen eines CMMN-Modells verstanden werden. `PlanItemDefinition` erbt von `CMMNElement` und beinhaltet instanziiert als Komposition `PlanItemControl`-Objekte. Diese Objekte beinhalten Informationen über die für die Ausführung relevanten *decorators* der CMMN-Elemente, wie etwa *Auto Complete* oder *Manual Start*. Elemente wie `Task` und `Milestone` erben beziehungsweise konkretisieren wiederum `PlanItemDefinition`. Diese Zwischenebene, wie auch die Klasse `PlanFragment`, von der die Klasse `Stage` erbt, sind bewusst nicht in die Implementierung einbezogen, da sie entweder nicht gebrauchte Attribute und Assoziationen enthalten, oder lediglich ein Attribut wie etwa eine Namensattribut, das bereits durch die Basisklasse `Element` mitgegeben ist. *Decorators* sind direkt im jeweiligen Element des Software-Frameworks umgesetzt.

Das dritte Beispiel ähnelt den vorhergehenden und umfasst zwei Aspekte[24]: CMMN-Elemente sind im Software-Framework nicht als von der Klasse `PlanFragment` (die wiederum von der abstrakten Klasse `PlanItemDefinition`) erbend umgesetzt, sondern unmittelbar ohne (Meta-)Zwischenschicht(en). Dieses Konzept ist nicht Bestandteil der Anforderungen an das Framework. Es ist aber denkbar, es mit der bestehenden Architektur umzusetzen: So können Gruppierungen implementiert werden, denen Objekte einer Fall-Instanz zugeordnet werden, die dann erneut instanziiert und persistiert werden können. Es ist durchaus möglich, zur Laufzeit weitere Elemente in eine Fall-Instanz einzubinden. Hierzu müssten die Dienste entsprechend erweitert werden, um gezielt Elemente hinzuzufügen und beispielsweise mit `Sentry`s zu verknüpfen.

Das vierte Beispiel ist die Struktur um die Klasse `Sentry`. Das Framework führt entsprechend der grafischen Notation die von einer abstrakten Klasse `Sentry` erbenden Klassen `EntrySentry` und `ExitSentry` ein. Diese sind im Gegensatz zur Spezifikation im Framework direkt mit dem assoziierten Element verbunden. Ebenso wurden die abstrakte Klasse `Criterion` und ihre Spezialisierungen gestrichen und direkt umgesetzt: Das Framework arbeitet direkt mit Klassen `OnPart` (samt Ausprägungen) und `IfPart`, die Assoziationen zu den jeweiligen `Sentry`-Ausprägungen aufweisen.

Das Konzept des Frameworks stellt somit einen leichtgewichtigen und vereinfachten Teilausschnitt der CMMN-Spezifikation dar (vgl. **REQ 7-10** in Abschnitt 3.2.2), welches aber die Kernelemente und Ausführungssemantik von CMMN 1.1 bereitstellt und umsetzt.

4.3 Paketstruktur der Framework-Implementierung

Um die im Folgenden beschriebenen Basisbausteine, Services und die adaptierten Entwurfsmuster der Framework-Implementierung besser einordnen zu können, wird in Abbildung 26 eine Übersicht der entstandenen Paketstruktur gegeben.

[23] Siehe Kapitel 5.2 der CMMN 1.1 Spezifikation.

[24] Siehe Kapitel 5.4.4 ff. der CMMN 1.1 Spezifikation.

Das Paket `core` (links oben) beinhaltet alle Pakete und Basis-Klassen für grundlegende Fall-Strukturen, wie `Element`, `CaseModel` und `Stage`. Enthalten sind auch Klassen, die Fallbearbeiter und Rollen repräsentieren.

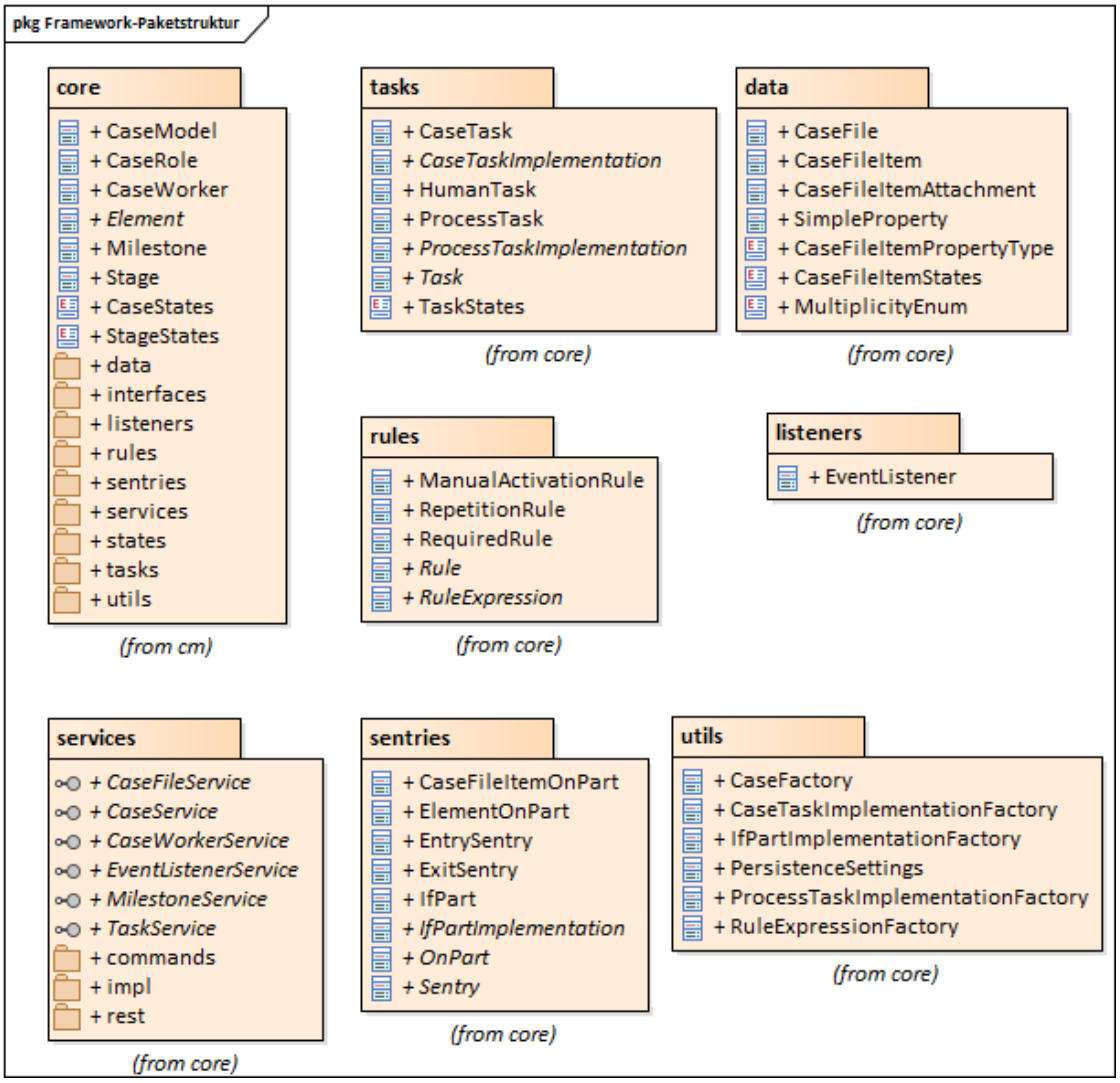

Abbildung 26: Paketstruktur der Framework-Implementierung

Daneben sind ausgewählte, in `core` enthaltene Pakete mit den enthaltenen Klassen dargestellt. Das Paket `tasks` etwa enthält die konkretisierenden Klassen der abstrakten Klasse `Task`. Klassen zur Abbildung von Datenstrukturen befinden sich im Paket `data`. Klassen zur Abbildung der *decorators* befinden sich im Paket `rules`. Das Paket `sentries` enthält Klassen zur Abbildung des Sentry-Konzepts. Im Paket `utils` (rechts unten) sind Factory-Klassen enthalten, die in Abschnitt 4.6.1 näher betrachtet werden.

4.4 Bausteine der Referenzarchitektur

In diesem Abschnitt werden die neu gestalteten Klassenstrukturen vorgestellt, die auf vereinfachten CMMN-Strukturen aufbauen. Vom Framework in der derzeitigen Version nicht unterstützt werden die folgend aufgelisteten (grafischen) CMMN-Elemente.

- Discretionary `Task`/`Stage`
- *Planning Table decorator*
- `PlanItem`/`PlanFragment`

Discretionary `Task`/`Stage`, der *Planning Table decorator* und die Klassen `PlanItem`/`PlanFragment` gehören zum Planungsmechanismus in CMMN, welcher auf Grund seiner hohen Komplexität im Prototyp nicht unterstützt wird. Alternativ können `Sentry`-Strukturen an `Task`s und/oder `Stage`s sowie *Manual Activation decorators* verwendet werden, um Planungsmechanismen und *discretionary items* zu realisieren.

- *Repetition Rule decorator* an `Stage`
- Spezialisierungen von `EventListener`
- `DecisionTask`

Der *Repetition Rule decorator* wird nicht für `Stage`s unterstützt. Die Spezifikation enthält keine Angaben darüber, wie sich eine `Stage` mit enthaltenen Elementen verhält, wenn sie wiederholt werden soll (s. [30], S. 122ff.). Für `Task`s wird dieser *decorator* unterstützt, indem eine Klon-Instanz des zu wiederholenden `Task` durch einen `TaskService` erstellt und dem `CaseModel` hinzugefügt wird.

Spezialisierungen von `EventListener` werden im Prototyp nicht unterstützt. `TimerEventListener` benötigen weitere Technologien und Implementierungen, die eine korrekte Verarbeitung abgelaufener Zeiträume unterstützen. Statt eines `UserEventListener` kann ein regulärer `EventListener` genutzt werden.

`DecisionTask`s werden zum Zeitpunkt nicht unterstützt, können aber durch `ProcessTask`s ersetzt werden. Eine zugrundeliegende individuelle Implementierung kann entsprechend genutzt werden.

Die Farbgebung der folgend gezeigten Bausteine entspricht den Farben der in Kapitel 3 vorgestellten Schichten *Case Management Core* (lila gefärbte Elemente) und *Case Specific Implementations* (grün gefärbte Elemente).

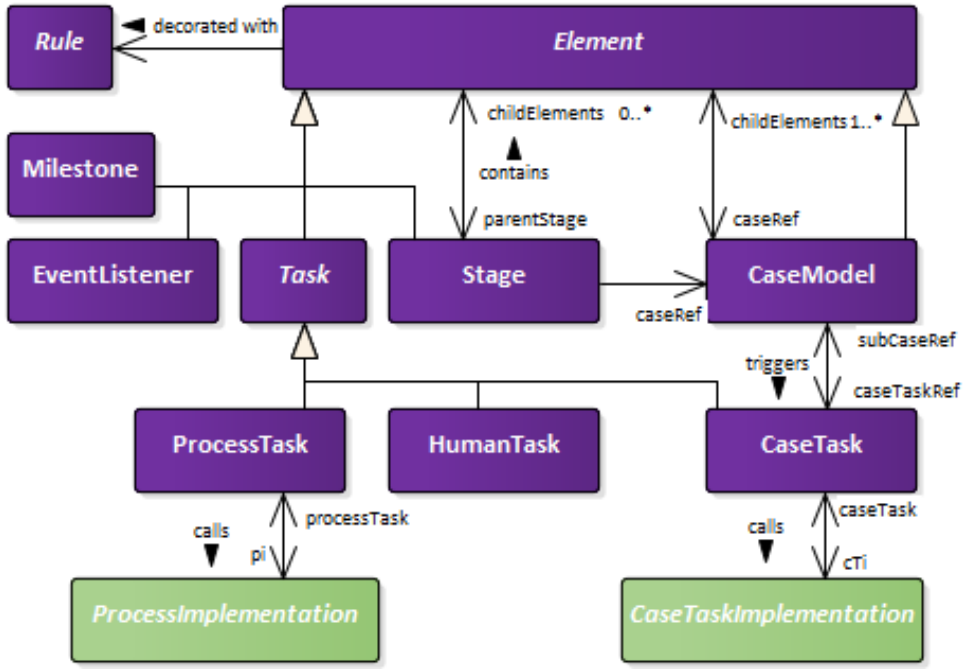

Abbildung 27: Framework Basiselemente

4.4.1 Case Management Basiselemente

Abbildung 27 zeigt die Basiselemente des Frameworks. Durch sie können einfache und grundlegende Fallstrukturen erstellt werden, wie in **REQ 8** gefordert ist (s. Abschnitt 3.2.2). Die Basiselemente erben von der abstrakten Klasse `Element`, welche in etwa der CMMN-Klasse `CMMNElement` entspricht. Von ihr geerbt werden identifizierende Attribute: Eine numerische ID für die Persistenz, eine Zeichenkette als CM-ID, sowie ein für menschliche Benutzer lesbarer Name. Die CM-ID entspricht dem ID-Attribut der Klasse `CMMNElement` und ist im Fall-Modell (und XML-Markup einer CMMN-Datei) eindeutig. Außerdem wird der Zustand des erbenden Elements als Zeichenkette gespeichert.

Die Klasse `CaseModel` repräsentiert einen Fall und dient als Wurzelelement einer Fallstruktur. Als Container-Element kann es alle anderen Elemente enthalten. Auf die in der Spezifikation verwendete äußerste `Stage` als Attribut `casePlanModel` eines eigentlichen `Case` wurde verzichtet.

Die abstrakte Klasse `Rule` dient der Abbildung der *decorators*, wie sie in Abschnitt 2.2.3.2 beschrieben sind. Abschnitt 4.4.3 geht genauer auf sie ein und zeigt im Rahmen der erstellten Klassen für Datenstrukturen entsprechende Spezialisierungen.

In der Abbildung zu sehen sind außerdem Klassen für die bereits vorgestellten Elemente `Stage`, `Milestone`, `EventListener`, `Task` und ihre Spezialisierungen, welche den gleichnamigen CMMN-Elementen entsprechen. Diese können einem `CaseModel` direkt zugeordnet werden, oder in einer `Stage` enthalten sein.

Basiselemente referenzieren neben dem Container (`CaseModel` oder `Stage`) in dem sie enthalten sind auch das Wurzel-`CaseModel`. Durch dieses Attribut können alle einem Fall zugehörigen Elemente unabhängig ihrer internen Struktur und Zugehörigkeit zu einem `Stage`-Container-Element abgefragt werden.

Die zwei `Task`-Spezialisierungen `ProcessTask` und `CaseTask` weisen Assoziationen zu den abstrakten Klassen `ProcessImplementation` und `CaseTaskImplementation` auf. Diese dienen der individuellen Ausgestaltung der jeweiligen Logik und werden zur Laufzeit der `Task`-Instanz geladen und ausgeführt. Sie werden näher in Abschnitt 4.6.1 beschrieben.

4.4.2 Rollensystem

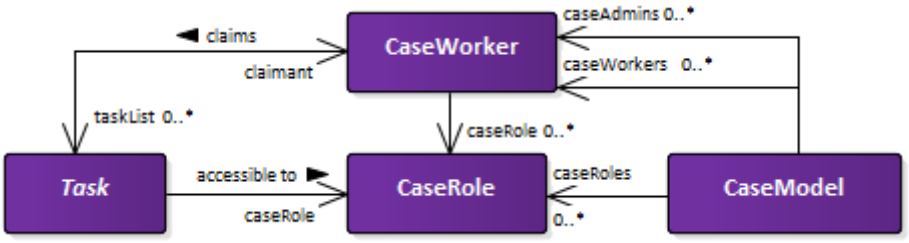

Abbildung 28: Framework Rollensystem

In diesem Abschnitt wird ein einfaches Rollensystem beschrieben, wie es in **REQ 4** gefordert ist (vgl. Abschnitt 3.2.1). Abbildung 28 zeigt die Klassen der einfachen Rollenstruktur. Eine Repräsentation von Benutzern ist in der CMMN-Spezifikation nicht vorgesehen, aber durchaus sinnvoll, um Basiszugang zu einer mit dem Framework erstellten Applikation bereitzustellen. Hierzu dient die Klasse `CaseWorker`. Sie kann einem `CaseModel` als reguläre Fallbearbeiter oder Admin-Fallbearbeiter zugeordnet werden (siehe Assoziation zwischen `CaseModel` und `CaseWorker` über Attribute `caseAdmins` und `caseWorkers`).

Berechtigungen für einen Fall und den darin enthaltenen Aufgaben können so über die Klasse `CaseRole` gesteuert werden. Eine `CaseRole` entspricht hierbei dem sehr einfach gestalteten Rollensystem der Spezifikation und basieren auf einem simplen Rollennamen.

Um ein `CaseModel` oder `Task`s einer bestimmten Rolle zugänglich zu machen, kann eine `CaseRole` angegeben werden. `Task`s können so bestimmten Rollen und Fallbearbeitern zugeordnet werden, beziehungsweise von diesen beansprucht und anschließend als (die Aufgabe leitender) Fallbearbeiter bearbeitet werden. Beim Zugriff durch die Services des Frameworks auf `Task`s werden `CaseRole`s mit denen des jeweiligen `CaseWorker`s verglichen. Bei einer Übereinstimmung wird ein Zugriff erlaubt (beziehungsweise überhaupt erst für den Fallbearbeiter angezeigt, sofern verfügbar).

4.4.3 Datenstrukturen

Dieser Abschnitt beschreibt die entstandenen Datenstrukturen für `CaseModel`s. Abbildung 29 zeigt die vereinfachte Datenstruktur des Frameworks. Einem `CaseModel` ist die Klasse `CaseFile` zugeordnet. Sie ist als Fall-Akte anzusehen und kann mehrere `CaseFileItem`s enthalten. Über sie kann auf enthaltene Daten eines `CaseModel`s zugegriffen werden.

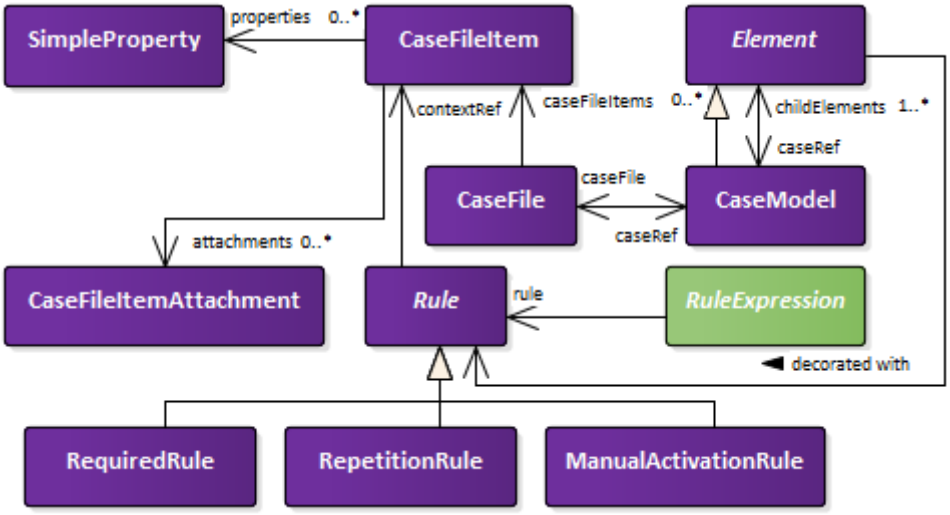

Abbildung 29: Framework Datenstrukturen

Im Gegensatz zur CMMN-Spezifikation (vgl. Abschnitt 2.2.1) werden Daten in Form primitiver Datentypen eines `CaseFileItem`s in der assoziierten Klasse `SimpleProperty` gespeichert. Ein `CaseFileItem` kann mehrere `SimpleProperty`s besitzen. Die Metaklassen der Spezifikation, wie etwa `CaseFileItemDefinition`, entfallen. `SimpleProperty`s können als Grundlage zur Evaluierung von *decorators* dienen, oder auch Daten speichern, die in `HumanTask`s oder anderen `Task`s verwendet werden. Sie dienen außerdem der Prüfung des `IfPart`s, das mit einem `Sentry` assoziiert ist.

Neu hinzugekommen ist die Klasse `CaseFileItemAttachment`, welche mit `CaseFileItem` assoziiert ist und nicht in der Spezifikation enthalten ist. Sie kann auf zwei Arten verwendet werden: Erstens können Daten und Dokumente direkt in der Datenbank gespeichert werden. Zweitens kann ein Dateipfad gespeichert werden, auf den zugegriffen werden kann. So können Verbindungen zu beispielsweise sehr großen Dokumenten hergestellt werden, die nicht in der Datenbank abgelegt werden sollen.

Zu sehen sind auch die Spezialisierungen der Klasse `Rule`, welche *decorators* repräsentieren: `RequiredRule`, `RepetitionRule` und `ManualActivationRule`. Die abstrakte Klasse `RuleExpression` enthält die individuell implementierte Logik der Spezialisierungen. Zusammen stellen sie eine Vereinfachung der Strukturen um die Klasse `PlanItemControl` der Spezifikation dar. `RuleExpression` ähnelt der Klasse `Expression` der Spezifikation. Neben einem identifizierenden Namen haben sie als Attribut eine Referenz auf ein `CaseFileItem`, dessen `SimpleProperty`s zur Evaluierung genutzt werden.

4.4.4 Sentry-Strukturen

Dieser Abschnitt beleuchtet die entstandene Klassenstruktur, um das Sentry-Konzept umzusetzen (vgl. **REQ 7**, **10** und insbesondere **11** in Abschnitt 3.2.2f.). Abbildung 30 zeigt die vereinfachte Struktur des in Abschnitt 2.2.3.3 beschriebenen Sentry-Konzepts der Spezifikation: Die abstrakte Klasse `Sentry` bleibt erhalten, spezialisiert sich aber in die zwei (nicht dargestellten) Klassen `EntrySentry` und `ExitSentry`, um die in Abschnitt 2.2.3.2 beschriebenen weißen und schwarzen Diamanten abzubilden. Die Klasse `ExitCriterion` der Spezifikation entfällt und wird durch die Klasse `ExitSentry` ersetzt. `Sentry`s können mit `Element` assoziiert werden, was dem „Anheften" eines Diamanten an ein Element im grafischen CMMN-Modell entspricht.

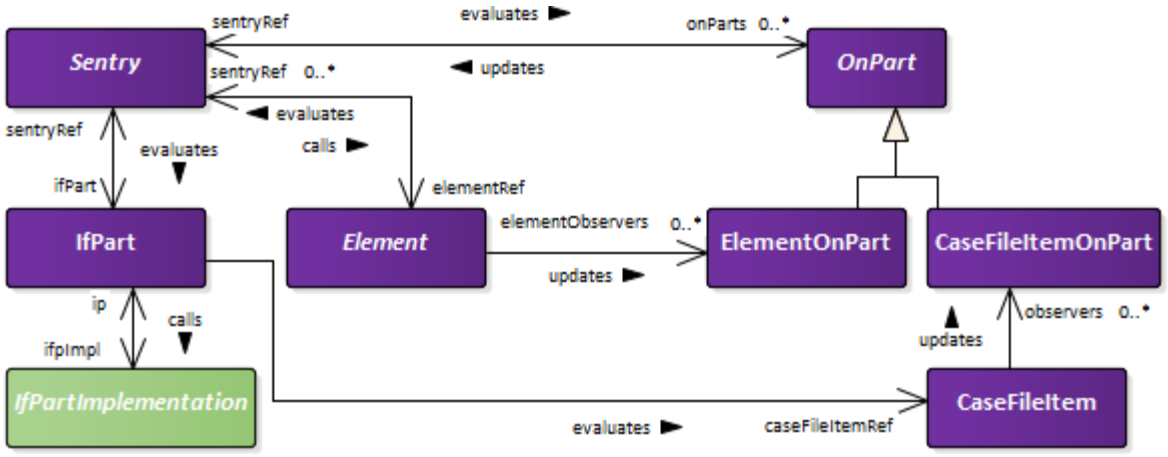

Abbildung 30: Framework Sentry-Strukturen

Die Spezialisierungen der Klasse `OnPart` überwachen Zustandsänderungen von Fall-Elementen. `ElementOnPart`s überwachen die Zustandsänderungen von referenzierten Spezialisierungen der abstrakten Klasse `Element` (beispielsweise eines `Task`s oder `Milestone`s). `CaseFileItemOnPart` überwacht die Zustandsänderungen referenzierter `CaseFileItem`s. `OnPart`s sind einem `Sentry` zugeordnet und benachrichtigen dieses über Zustandsänderungen.

Die Bedingungen eines `Sentry`s setzen sich aus `OnPart` und – wenn vorhanden – `IfPart` zusammen. `OnPart`s werden durch definierte Zustandsübergänge erfüllt. Ist beispielsweise *complete* definiert und ein überwachtes Element geht durch *complete* in den zulässigen Zustand *Completed* über, ist der `OnPart` erfüllt. Wird ein `Sentry` über eine Zustandsänderung benachrichtigt, wird auch der assoziierte `IfPart` evaluiert. Die Evaluierung erfolgt auf Basis einer individuellen Implementierung, welche als von `IfPartImplementation` erbende Klasse zu erstellen ist und die Prüflogik enthält. Ein referenziertes `CaseFileItem` und enthaltene `SimpleProperty`s können als Basis für die Evaluierung dienen. Ist auch der `IfPart` erfüllt, ist die zusammengesetzte Bedingung des `Sentry`s erfüllt. Wenn die Bedingungen eines `Sentry`s erfüllt sind, wird eine Zustandsänderung des referenzierten `Element`s – typischerweise vom Zustand *Available* zu *Enabled* oder *Active* – erwirkt.

4.5 Case Management Services

Dieser Abschnitt soll eine Übersicht über die Implementierung der in **REQ 12** und **13** (vgl. Abschnitt 3.2.4) geforderten Services geben. Abbildung 31 zeigt die Paketstruktur der Services. Enthalten sind die Interfaces der in Abschnitt 3.3.2 konkretisierten Services. Implementierungen finden sich im Paket `impl`. Schnittstellen und Implementierungen der REST-Schnittstellen sind im Paket `rest` enthalten. Die Services bieten CRUD-Operationen an, die überwiegend dazu dienen, Elemente abzurufen und zu manipulieren.

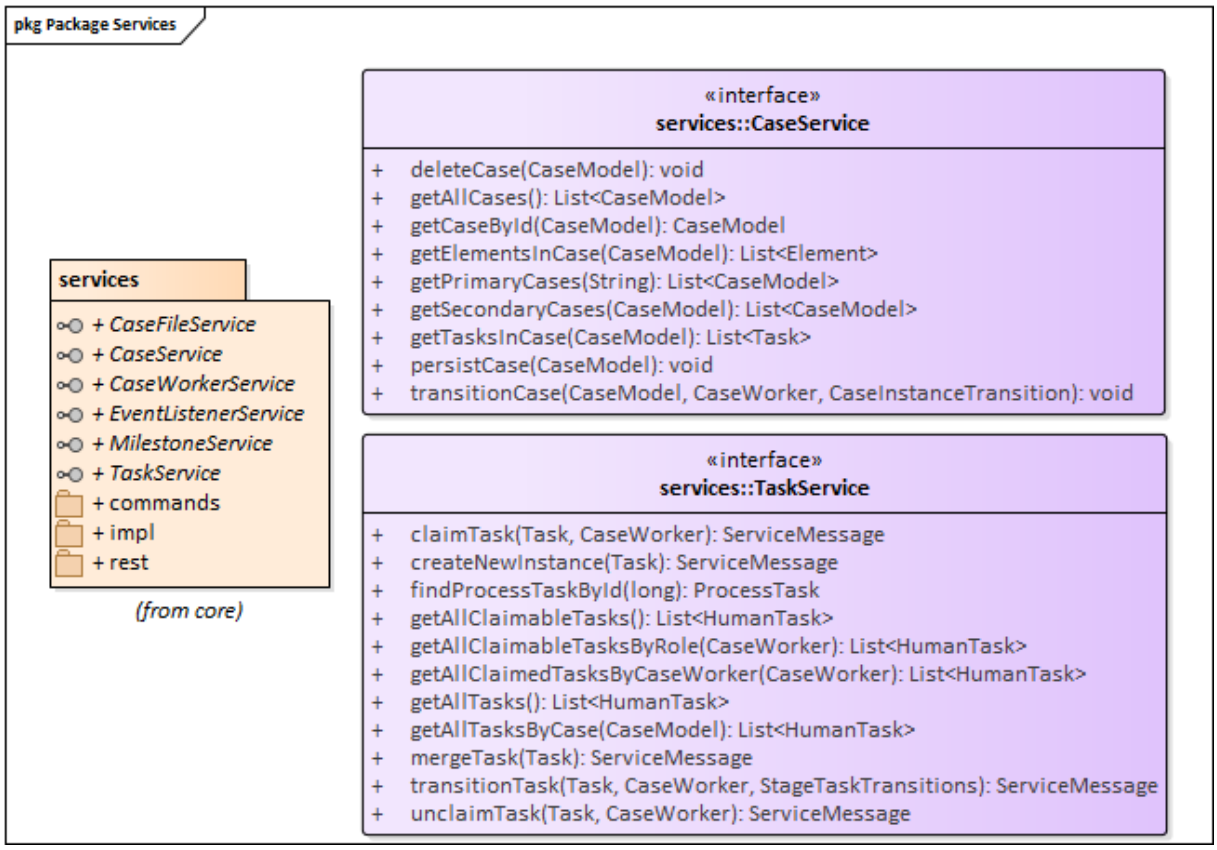

Abbildung 31: Service Implementierungen

Beispielhaft sollen ausgewählte Methoden der zwei detaillierter gezeigten Interfaces `CaseService` und `TaskService` erläutert werden. Die Methoden `getPrimaryCases(String)` und `getSecondaryCases(CaseModel)` können dazu genutzt werden, verfügbare Fall-Instanzen und untergeordnete Fälle abzurufen, die durch `CaseTask`s des „primary cases“ verknüpft sind. Ein „primary case“ ist ein Wurzel-Fall. „Secondary cases“ sind durch `CaseTask`s mit diesem verknüpfte Fall-Instanzen. Der String-Parameter der ersten Methode kann zur Filterung des Fall-Zustands genutzt werden; standardmäßig werden Fälle im Zustand „*ACTIVE*“ abgerufen. Der zweiten Methode wird ein `CaseModel` beziehungsweise ein „shallow `CaseModel`“ übergeben. „Shallow“ bedeutet, dass das Objekt nicht vollständig initialisiert wurde, aber mindestens die Persistenz-ID aufweist. So kann typensicher direkt mit den jeweiligen Objekten gearbeitet werden.

Die Methode `transitionCase(CaseModel, CaseWorker, CaseInstanceTransition)` des `CaseService` dient dazu, einen Fall durch eine Benutzerinteraktion beispielsweise in den Zustand

„*SUSPENDED“* zu versetzen. Der übergebene `CaseWorker` dient der Prüfung, ob dieser berechtigt ist, die Zustandsänderung zu veranlassen. Ähnlich verhält es sich mit der Methode `transitionTask(Task, CaseWorker, StageTaskTransitions)` des `TaskService`. Tasks (primär `HumanTask`s) können durch `CaseWorker` beansprucht („ge-claimed“) werden (siehe Methoden `claimTask(Task, CaseWorker)` und `unclaimTask(Task, CaseWorker)`). Zustandsänderungen können vom `CaseWorker` veranlasst werden, der einer `Task` zugeordnet wurde.

Die Methode `createNewInstance(Task)` wird im Rahmen des *Repetition Rule decorators* genutzt (siehe Abschnitt 2.2.3.2), um einen Klon des zu wiederholenden Tasks zu erzeugen, Bestandteile des `Sentry`-Konzepts zu konfigurieren und den Klon zu persistieren.

„Shallow“-Objekte werden auch in den anderen Services genutzt und dienen beispielsweise den REST-Schnittstellen, die mitunter auf Pfadparametern basieren und die Persistenz-ID nutzen. So wird beispielsweise für einen Pfad `/rest/cases/{id}` zum Abrufen eines bestimmten CaseModels der Pfadparameter `{id}` genutzt, um ein „shallow `CaseModel`“ zu erzeugen. Mit Hilfe des `CaseService`s wird dieses anschließend aus der Datenbank abgerufen und als JSON über die REST-Schnittstelle zurückzugeben. Die Implementierung der REST-Schnittstellen ist im Prototyp nicht abgeschlossen, bietet aber rudimentäre Funktionen zum Abrufen von `CaseModel`-Objekten und enthaltenen Elementen an, wie auch Schnittstellen für die Arbeit mit `CaseFileItem`s.

4.6 Adaptierte Entwurfsmuster für CM Anwendungen

Dieser Abschnitt stellt etablierte Entwurfsmuster vor (vgl. [64] und [65]), die für das Framework ausgewählt und für dieses angepasst wurden. Die angewendeten Muster gehören zu den zwei Gruppen der Erzeugungs- und Verhaltensmuster. Sie dienen als Basis für die Umsetzung verschiedener Aspekte des Frameworks, die im Folgenden genauer beschrieben werden.

4.6.1 Erzeugungsmuster Fabrik-Methode (*Factory Pattern*) in verschiedenen Anwendungsbereichen

Fabrik-Methoden (vgl. [64], S. 101ff.; [65], S. 440ff.) werden im Framework dazu genutzt, konkrete Ausprägungen von Klassen oder Interface implementierende Klassen zur Laufzeit zu erzeugen. Diese konkretisieren eine abstrakte Klasse oder implementieren ein gemeinsames Interface. Die Fabrik-Methode erzeugt zur Laufzeit die entsprechende Ausprägung und gibt diese über definierte Schnittstellen zurück. Eine Auswahl erfolgt durch Kontrollstrukturen wie If-Anweisungen oder Switch-Case-Verzweigungen (ein Beispiel ist in Abbildung 36 des Abschnitts 4.6.1.5 zu sehen). Als Typ wird der abstrakte Typ oder das gemeinsame Interface verwendet, sodass die konkrete Ausprägung verborgen bleibt.

Genutzt wird dieses Verfahren vor allem zur Verarbeitung der anwendungsspezifischen Implementierungen, wie etwa der Klassen `IfPart`, `Rule` beziehungsweise `RuleExpression`, `ProcessTask` und `CaseTask` (vgl. Abschnitt 3.3.4). Auch Zustandsausprägungen der Klassen `Element` und `CaseFileItem` werden auf diese Weise erzeugt.

4.6.1.1 Erzeugung von Fall-Instanzen aus Blaupausen

Fall-Blaupausen, also vordefinierte und durch Entwickler auf Basis der Framework-Klassen erstellte Fallstrukturen, werden durch die Klasse `CaseFactory` gekapselt, die diese als `CaseModel`-Objekte zurückgibt. Die erzeugten Objekte können anschließend durch die Klasse `CaseService` verarbeitet werden, um diese zu starten und zu persistieren. Die erzeugten `CaseModel`-Objekte werden auch von `CaseTask`s benutzt.

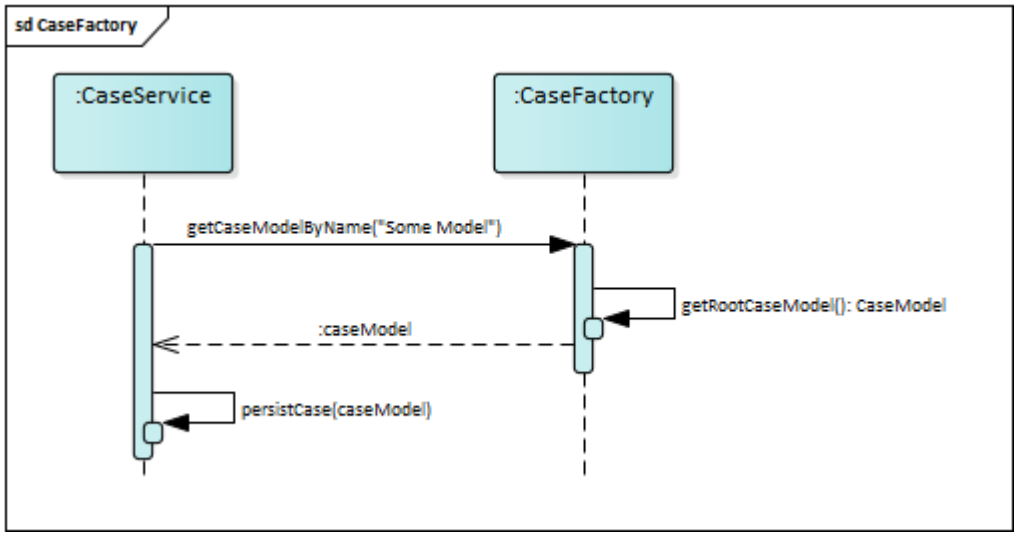

Abbildung 32: Sequenzdiagramm Zugriff auf CaseFactory

Abbildung 32 zeigt, wie ein `CaseService` auf die Methode `getCaseModelByName()` der `CaseFactory` zugreift. Als Parameter wird der Fall-Name „Some Model“ als String übergeben. Intern wird in `CaseFactory` ein `CaseModel` dem übergebenen Namen entsprechend erstellt und an den Service zurückgegeben. Im Beispiel entspricht der übergebene Name einem Aufruf der Methode `getRootCaseModel()`, welche die Fallstruktur zurückgibt. Anschließend persistiert der Service den Fall. Die instanziierte Fall-Blaupause steht zur Bearbeitung bereit. CaseFactory kann auch Methoden für einen direkten Zugriff auf Blaupausen bereitstellen, wie etwa durch eine `getSomeCase()`-Methode.

4.6.1.2 Erzeugung von IfPart-Implementierungen

Wird ein `Sentry` über Zustandsänderungen informiert und besitzt einen `IfPart`, so wird dieser geprüft, um festzustellen, ob auch diese Bedingung erfüllt wurde, um das referenzierte Element in den entsprechenden Zustand übergehen zu lassen. In Abbildung 33 wird die Verwendung der `IfPartImplementationFactory` gezeigt. Durch sie können die individuellen Implementierungen erstellt werden, welche die abstrakte Klasse `IfPartImplementation` konkretisieren.

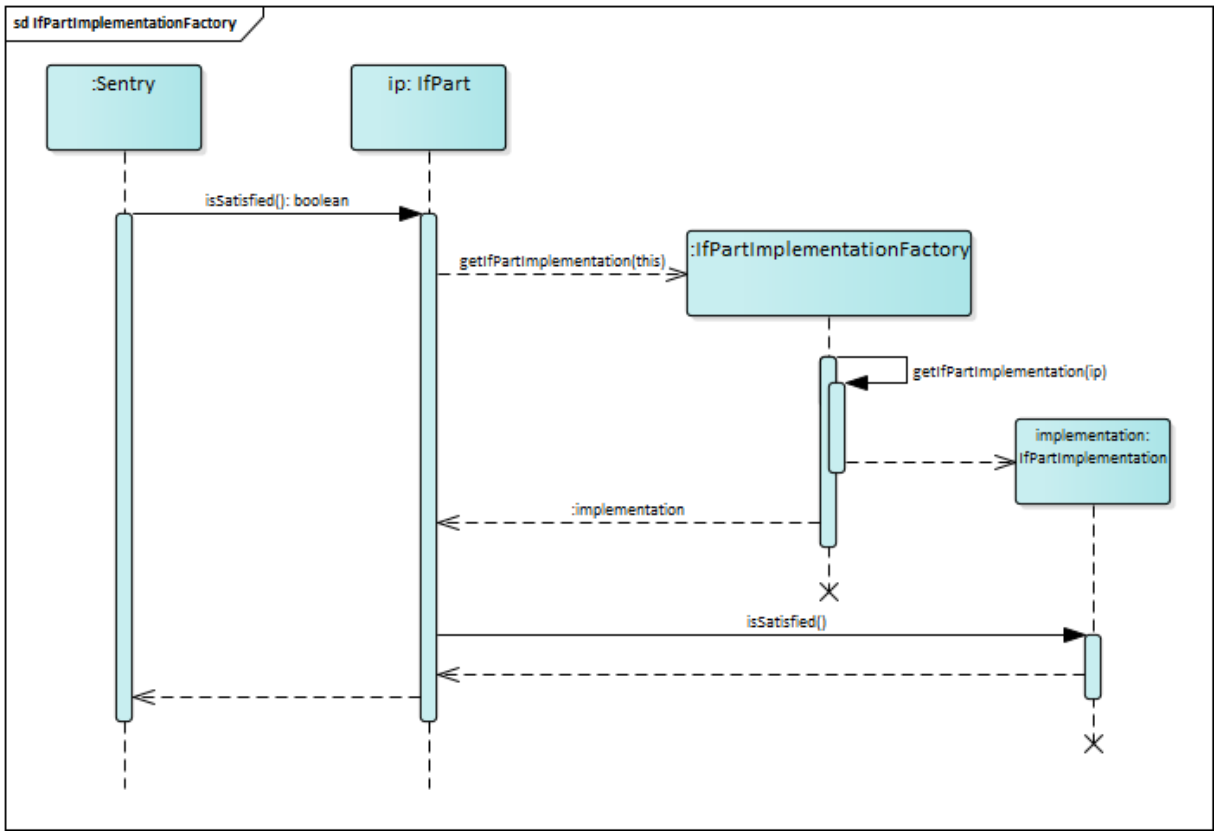

Abbildung 33: Sequenzdiagramm IfPart und IfPartImplementationFactory

Ein `Sentry` ruft die Methode `isSatisfied()` des assoziierten `IfParts` auf. Das `IfPart`-Objekt beschafft sich durch die Methode `getIfPartImplementation()` der `IfPartImplementationFactory` die korrekte Ausprägung der Implementierung. Es übergibt sich hierzu selbst als Parameter. Auf Basis der CM-ID des `IfPart` wird die entsprechende Ausprägung der abstrakten `IfPartImplementation` erzeugt. Anschließend prüft der `IfPart` durch die Methode `isSatisfied()` der konkreten Klasse, ob die `IfPart`-Bedingung erfüllt ist und liefert einen booleschen Wert an `Sentry` zurück. Nicht gezeigt ist der Zugriff auf das in `IfPart` referenzierte `CaseFileItem`, dessen `SimpleProperty` geprüft wird.

4.6.1.3 Erzeugung von ProcessTask-Implementierungen

Konkretisierungen der abstrakten Klasse `ProcessTaskImplementation` werden durch die Klasse `ProcessTaskImplementationFactory` erzeugt. Wird die Methode `startProcess()` eines `ProcessTask`s aufgerufen, ruft dieser die `ProcessTaskImplementationFactory` auf und übergibt sich selbst als Parameter. Auf Basis dessen CM-ID erzeugt die Fabrik-Methode die korrekte Konkretisierung und gibt diese zurück an den `ProcessTask`. Anschließend wird die Methode `startProcess()` der zurückgegebenen `ProcessTaskImplementation` ausgeführt und der Algorithmus, welcher den Prozess repräsentiert, durch die Methode `startProcess()` gestartet. Abbildung 34 zeigt diesen Ablauf.

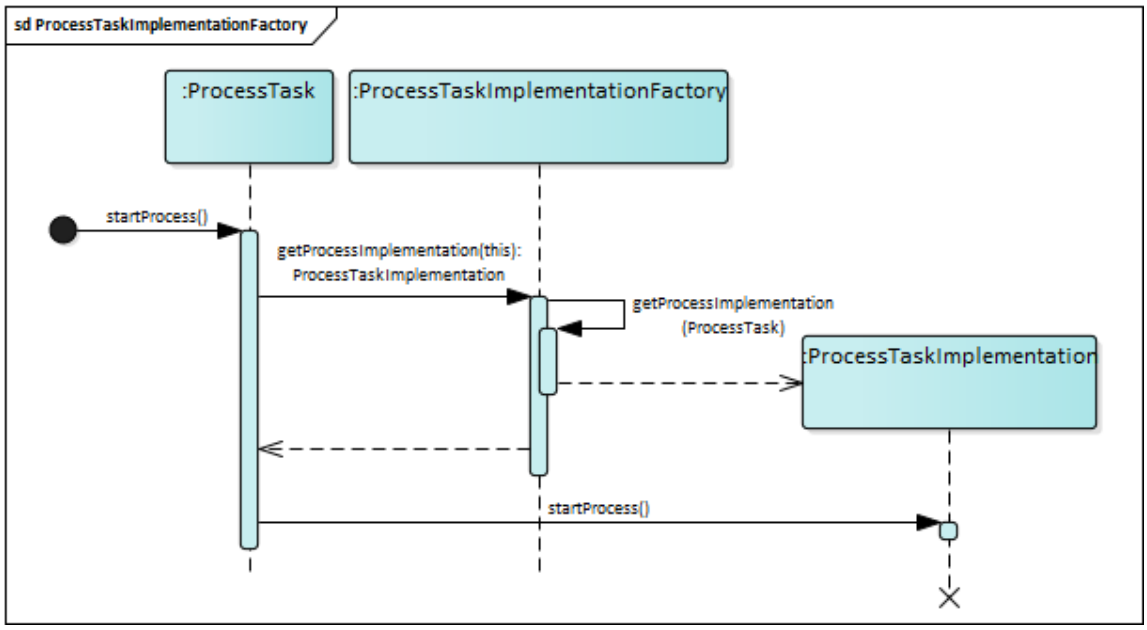

Abbildung 34: Sequenzdiagramm ProcessTask und ProcessTaskImplementationFactory

4.6.1.4 Erzeugung von *decorator* Rules

Die *decorators* in Form der Klassen `ManualActivationRule`, `RequiredRule` und `RepetitionRule` konkretisieren die abstrakte Klasse `Rule`. Zur Laufzeit wird die entsprechende konkretisierende Klasse mit Hilfe der `RuleExpressionFactory` auf Basis des Regelnamens erzeugt und zurückgegeben. Abbildung 35 zeigt dieses Vorgehen: Ausgehend von der – beispielsweise durch ein erfülltes `Sentry` angestoßenen – Zustandsänderung eines sich im Zustand „*AVAILABLE*“ befindlichen `HumanTask`s wird geprüft, ob deren *Manual Activation decorator* gültig ist. Die Fabrik erzeugt die entsprechende Konkretisierung, die `HumanTask` evaluiert durch die Methode `evaluate()` der Klasse `Rule` die Ausprägung und liefert einen booleschen Wert zurück. Nicht gezeigt ist der Zugriff auf das in `Rule` referenzierte `CaseFileItem`, dessen `SimpleProperty` geprüft wird

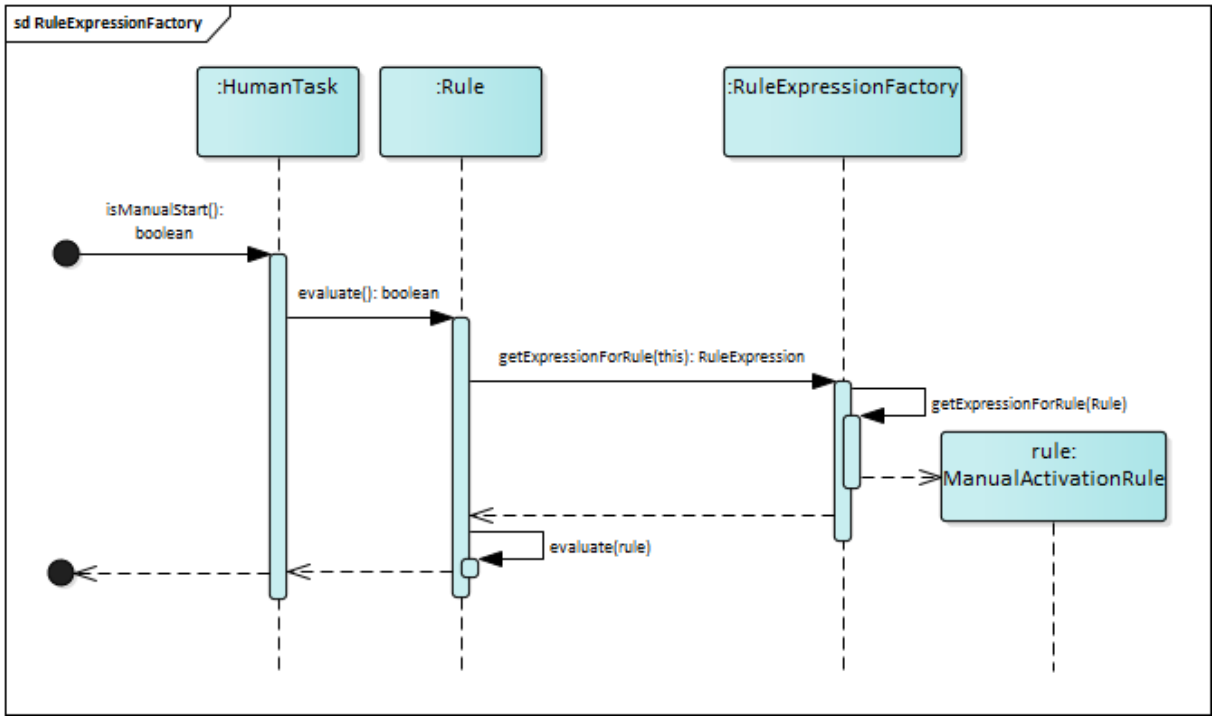

Abbildung 35: Sequenzdiagramm Rule und RuleExpressionFactory

4.6.1.5 Erzeugung von Zustandsausprägungen

Methoden zum Laden der aktuellen Zustandsausprägung eines Elements sind direkt in den von `Element` erbenden Klassen, wie `HumanTask` oder `Milestone` und `CaseFileItem` implementiert. Die Methode `loadContextState()` (welche in `Element` abstrakt ist) initialisiert das jeweilige Klassenattribut `contextState` des zugehörigen Interface-Typs (wie etwa `IStageTaskState` für `Stage` und `Task`, oder `ICaseFileItemState` für `CaseFileItem`) mit der korrekten Zustandsausprägung, Das Attribut wird nicht persistiert und ist dementsprechend mit der Annotation *@Transient* gekennzeichnet.

```
@Transient
private ICaseFileItemState contextState;
private void loadContextState() {
    if (state.equals(CaseFileItemStates.INITIAL.toString())) {
        setContextState(new CaseFileItemInitial(this));
    }
    if (state.equals(CaseFileItemStates.AVAILABLE.toString())) {
        setContextState(new CaseFileItemAvailable(this));
    }
    if (state.equals(CaseFileItemStates.DISCARDED.toString())) {
        setContextState(new CaseFileItemDiscarded(this));
    }
}
```

Abbildung 36: Fabrik-Methode loadContextState der Klasse CaseFileItem

Abbildung 36 zeigt beispielhaft die If-Anweisungen zum Laden der Zustandsausprägung eines `CaseFileItems`: Der aktuelle Zustand, der als String in Attribut `state` persistiert ist, wird mit den Werten der `Enumeration`-Klasse `CaseFileItemStates` verglichen. Entsprechend wird das Attribut `contextState` zur Laufzeit beispielsweise mit einer Referenz auf ein erzeugtes Objekt des Typs `CaseFileItemAvailable` initialisiert, welches das Interface `ICaseFileItemState` implementiert.

4.6.1.6 Erzeugung von CaseTask-Implementierungen

Um `CaseTasks` auszuführen, also um ein Kind-`CaseModel` zu erzeugen, wird die Klasse `CaseTaskImplementationFactory` benutzt. Dieser wird der `CaseTask` übergeben und auf Basis dessen CM-ID erzeugt sie eine entsprechende `CaseTaskImplementation`, deren Methode `startCase()` ausgeführt wird. In dieser Methode ist die Logik zum Erzeugen eines `CaseModel` mit Hilfe der Klasse `CaseFactory` enthalten.

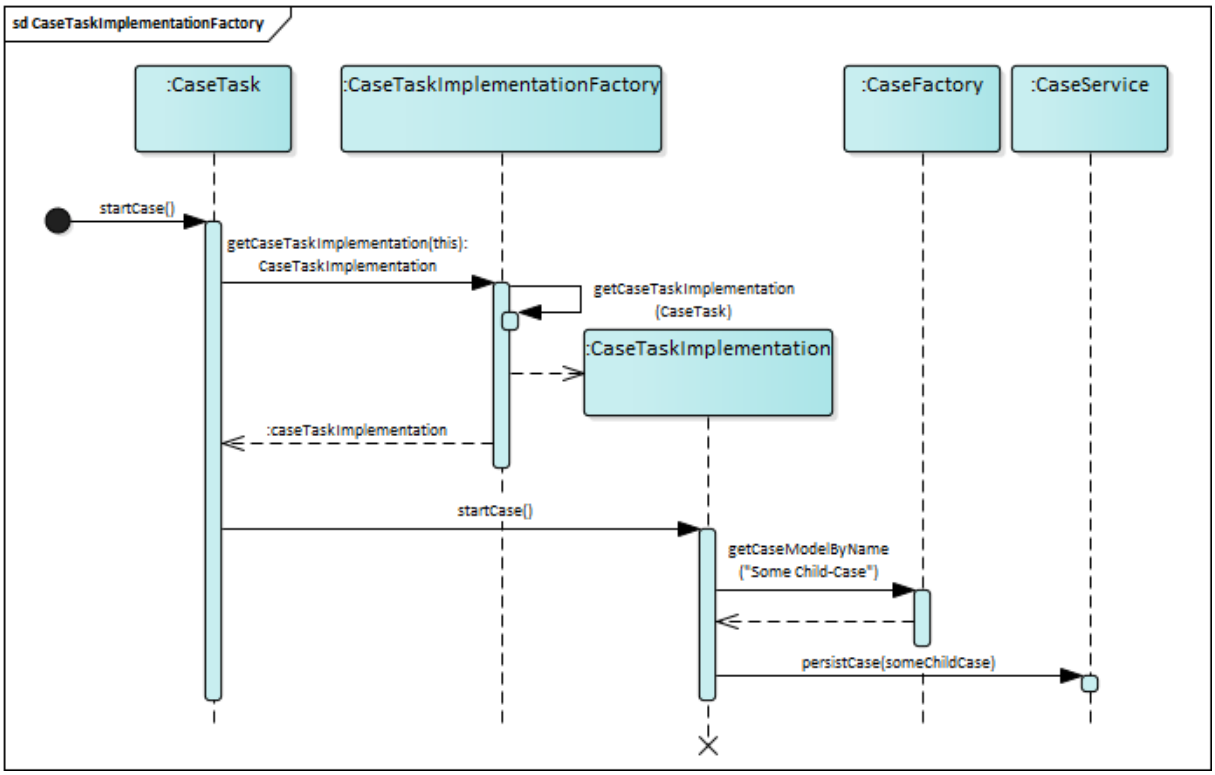

Abbildung 37: Sequenzdiagramm CaseTask und CaseTaskImplementationFactory

Abbildung 37 zeigt dieses Vorgehen und die Interaktion mit der `CaseFactory` und des `CaseService`. Nicht gezeigt sind weitere Schritte, um das neu erzeugte `CaseModel`-Objekt dem `CaseTask`-Attribut `subCaseRef` zuzuordnen und eine Referenz zum übergeordneten `CaseTask` im `CaseModel`-Objekt seinem Attribut `caseTaskRef` zuzuweisen.

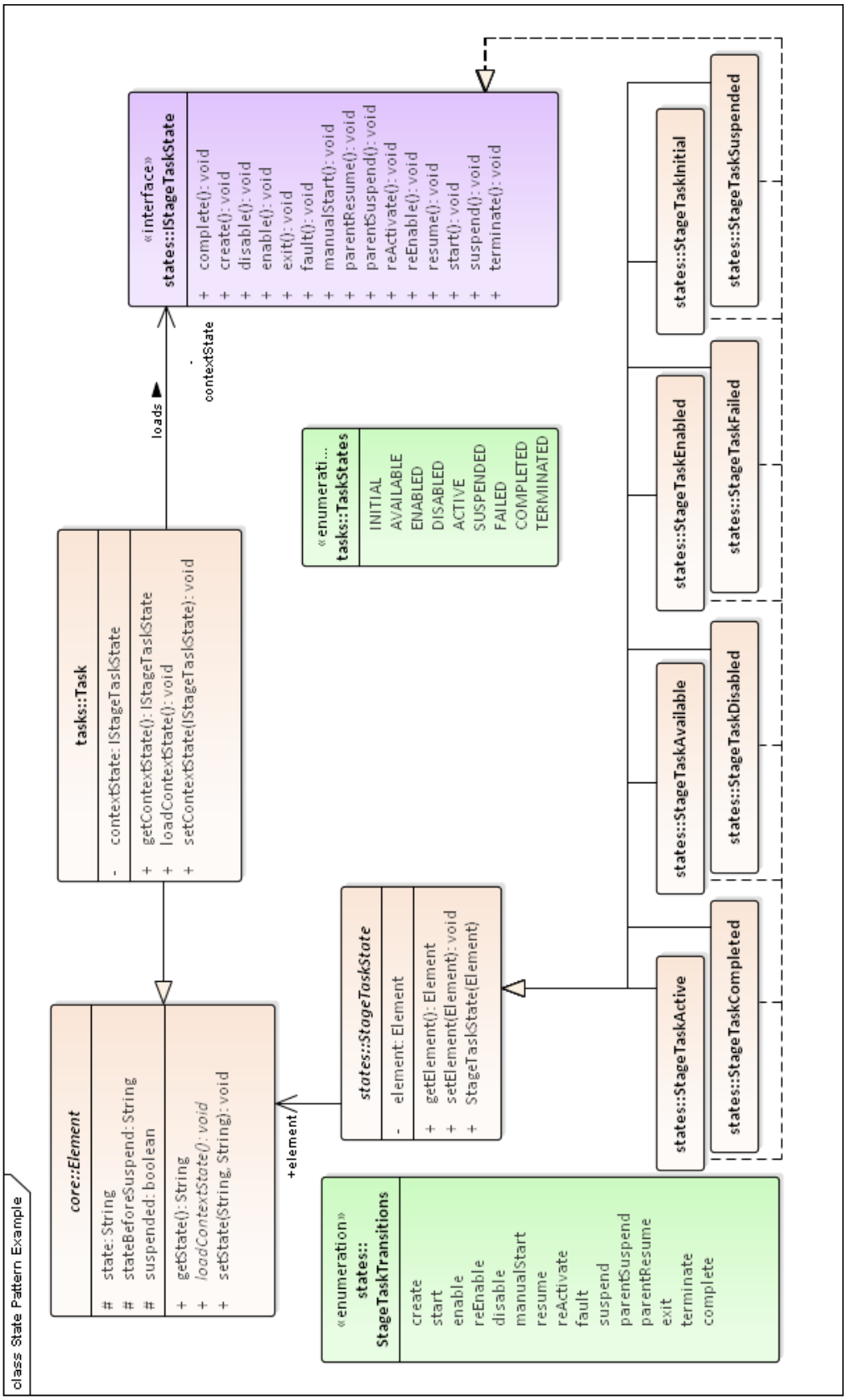

Abbildung 38: Klassendiagramm der Zustände für Stage und Task

4.6.2 Entwurfsmuster Zustand (*State Pattern*) zur Abbildung der Zustände und Lebenszyklen

Um die in Abschnitt 2.2.3.1 beschriebenen Zustände und Lebenszyklen abzubilden, wird das Zustand-Muster verwendet (vgl. [64], S. 398ff.; [65], S. 191, S. 486ff. und S. 638). Abbildung 38 zeigt

beispielhaft die Klassenstruktur für die Elemente `Stage` und `Task`. Diese Elemente teilen sich nach der Spezifikation die gleichen Zustände und Zustandsübergänge. Dargestellt ist die Assoziation mit der Klasse `Task`. Zustandsübergänge sind in der Enumeration-Klasse `StageTaskTransition` enthalten. Die Zustände für `Tasks` sind in der Enumeration-Klasse `TaskStates` in Großbuchstaben enthalten. Die Bezeichnungen der Zustände und Zustandsübergänge entsprechen der CMMN 1.1 Spezifikation.

Dieses Muster wird verwendet, um die einzelnen Zustände und das jeweilige Verhalten zu kapseln und Zustandsänderungen in Abhängigkeit des aktuellen Zustands zu definieren. Somit können zulässige Zustandsänderungen implementiert werden. Beim Aufruf einer nicht zulässigen Zustandsänderung kann beispielsweise ein Fehler geworfen werden. Der aktuelle Zustand, sowie der Zustand vor der Zustandsänderung hin zum Zustand „*SUSPENDED*“ werden in den Attributen `state` und `stateBeforeSuspend` der Klasse `Element` gespeichert und persistiert.

Jeder Zustand ist als konkrete Klasse realisiert, die alle von der abstrakten Klasse `StageTaskState` erben und die Schnittstelle `IStageTaskState` implementieren. Die abstrakte Klasse dient als Verbindung zum eigentlichen Element. Dieses Element wird im Konstruktor übergeben und als Referenz gehalten. Die Elemente selbst, wie etwa `Stage` oder `Task`, implementieren die abstrakte Methode `loadContextState()` der abstrakten Klasse `Element`, die sie konkretisieren. Über diese konkrete Methode wird durch eine Abfrage des aktuellen Zustands die entsprechende Implementierung der Schnittstelle `IStageTaskState` geladen. Sie drücken die Zustandsübergänge aus (siehe auch Abschnitt 4.6.1.5).

Das Attribut `contextState` ist je nach definiertem Element und seinen Zuständen im spezialisierten Element wie etwa `Stage` oder `Task` deklariert, aber für JPA als *transient* gekennzeichnet und wird nicht persistiert. Es wird zur Laufzeit geladen und entspricht der aus den Musterbeschreibungen bekannten Klasse „Kontext“. Auf eine Speicherung wurde verzichtet, um nötige Verbundoperationen auf Datenbankseite zu vermeiden und das Datenmodell schlank zu halten.

Die konkreten Klassen, welche die Zustände abbilden, implementieren alle Methoden für Zustandsübergänge, wie etwa die Methode `complete()`. Es sind aber nicht alle Methoden zulässig, sodass „unzulässige“ Methoden lediglich als Rumpf implementiert sind und keine Auswirkungen haben.

Wird beispielsweise versucht, den Zustand eines Elements mit dem aktuellem Zustand „*ACTIVE*“ in den Zustand „*AVAILABLE*“ durch die Transition *create* entsprechend durch die Methode `create()` zu wechseln, wird zwar die korrekte Implementierung von `IStageTaskState` geladen, aber die Methode ist entsprechend wirkungslos, da die Transition nicht zulässig ist.

Abbildung 39 zeigt beispielhaft den Ablauf für die Transition *create*, um den Zustand der `Task`-Instanz vom Zustand „*INITIAL*“ in den Zustand „*ACTIVE*“ zu überführen. Als Akteur wird ein `TaskService` dargestellt. Dieser ruft den kontextuellen Zustand einer `Task`-Instanz ab. Die `Task`-Instanz ruft eine interne Methode auf, um die korrekte Ausprägung der Klasse `StageTaskState` zurückzuliefern. Der Service ruft die Methode *create()* des Interface `IStageTaskState` auf, um in den Zustand „*ACTIVE*“ zu wechseln[25]. Anschließend ruft die Ausprägung von `StageTaskState`, in diesem Falle dem initialen Zustand entsprechend eine Instanz der Klasse `StageTaskInitial`, die Methode `setState(String state, String transition)` der `Task`-Instanz zum Ändern des Zustandes auf. Intern werden im Rahmen dieser Methode etwaige Beobachter, in diesem Falle `ElementOnParts`, benachrichtigt.

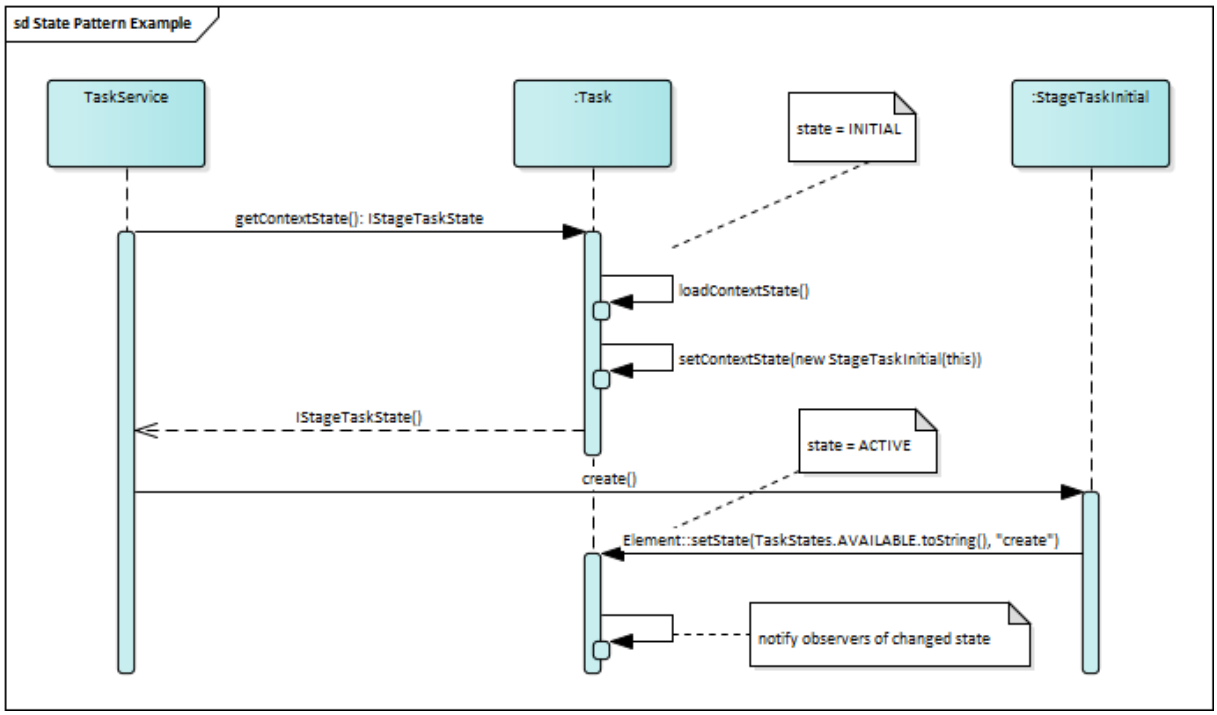

Abbildung 39: Sequenzdiagramm State Pattern

Das Muster findet Verwendung für alle Elemente, einschließlich der Zustandsänderungen von `CaseFileItem`-Instanzen. Unterschiede ergeben sich in den konkreten Ausprägungen, ihren Implementierungen, den verwendeten Zuständen und Transitionen und den benachrichtigen Beobachtern. Im Falle der Klasse `CaseFileItem` sind die Beobachter beispielsweise `CaseFileItemOnPart` statt `ElementOnPart`.

Die Anwendung des Musters in dieser Weise hat den Nebeneffekt, dass Zustandswechsel, die über ein externes System über die REST-Schnittstellen bewirkt werden sollen, keine unzulässigen Zustandswechsel bewirken. Ein Beispiel hierfür ist eine `Task`-Instanz, die in den Zustand „*SUSPENDED*“ überführt wurde. Es sind keine Zustandsänderungen zulässig, außer der, die zurück in

[25] In diesem Fall hat die *Task*-Instanz keine Beobachter. Der Zustand wechselt nach einer Prüfung direkt auf „ACTIVE“, durchläuft intern aber zunächst den Zustand „AVAILABLE“, um etwaige Beobachter zu beeinflussen, die auf die Transition *start* warten.

den Zustand vor dem Wechsel auf „*SUSPENDED*“ führt. Hierzu muss die Transition *resume* erfolgen. Dem externen System ist der letzte Zustand „*ACTIVE*“ bekannt und es soll die Transition *complete* erfolgen, die aber auf Seiten des Servers diesen unzulässigen Übergang nicht erlaubt.

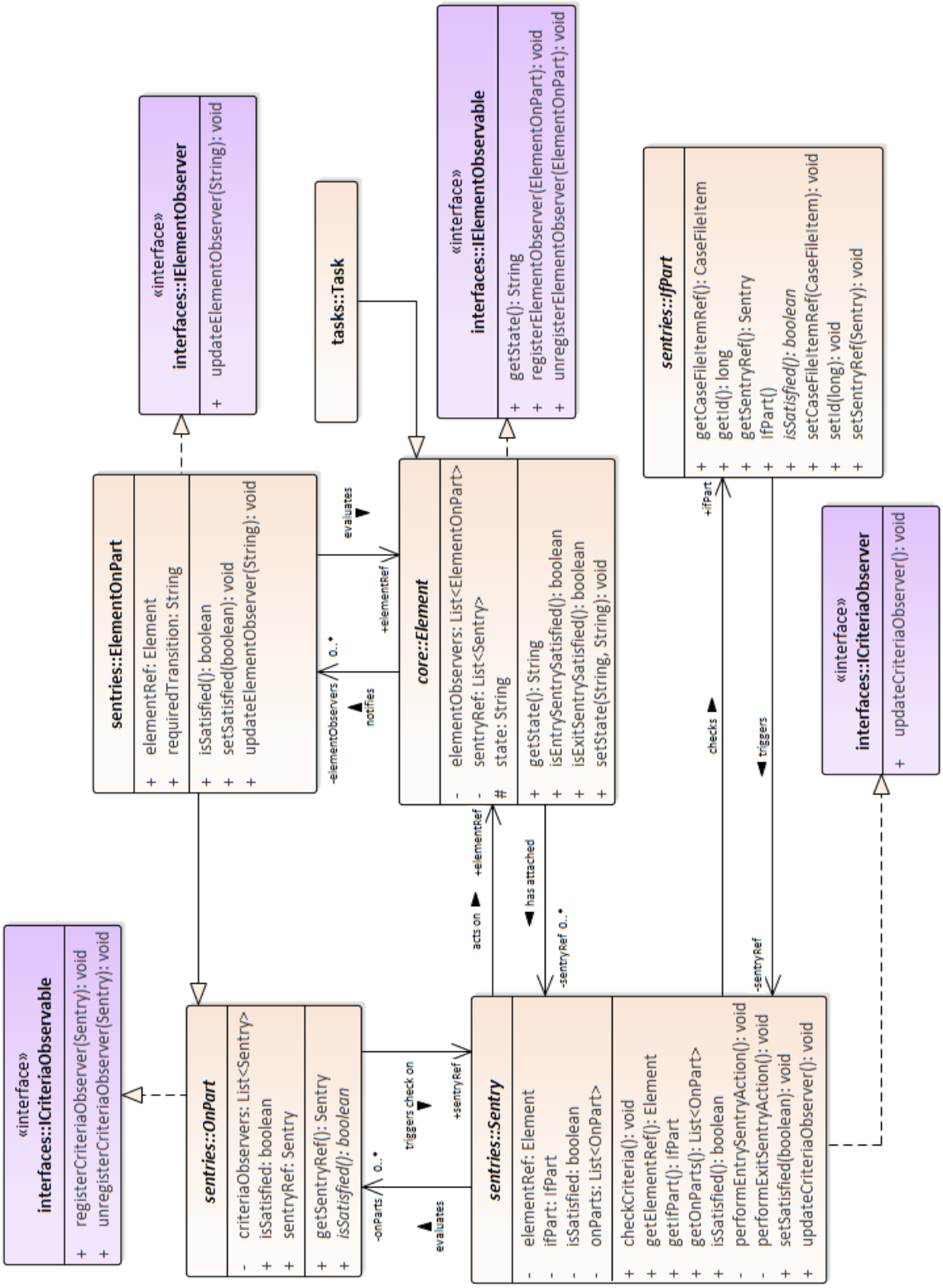

Abbildung 40: Klassendiagramm Beobachter-Muster im Element-Kontext

4.6.3 Entwurfsmuster Beobachter (*Observer Pattern*) zur Abbildung des Sentry-Konzepts

Für die Realisierung des Sentry-Konzeptes wird das Muster Beobachter verwendet (vgl. [64], S. 287ff.; [65], S. 463). Das Muster wird in verschiedenen Kontexten des Sentry-Konzeptes verwendet: Im Kontext der Klasse `Element`, im Kontext der Klasse `CaseFileItem` und im Kontext der Beziehung zwischen `Sentry`s und `OnPart`s. In Beschreibungen des Musters ist eine Referenz auf Schnittstellen angezeigt, statt auf konkrete Klassentypen. Dies ist auf Grund des verwendeten JPA-Standards nicht möglich. JPA-Referenzimplementierungen bieten eingeschränkte Funktionen hierfür an, die aber vom Standard abweichen. Um im Rahmen des Standards zu verbleiben, werden konkretisierende Klassen benutzt. Dies stellt in diesem Kontext kein Problem dar, da keine anderen Subtypen verwendet werden. Die Attribute `criteriaObservers` der Klassen, die `OnPart` konkretisieren, `onParts` der Klasse `Sentry` und `elementObservers` der Klasse `Element` sind mit den abstrakten Klassen typisiert und referenzieren nicht, wie in Musterbeschreibungen vorgeschlagen, die entsprechenden Interface-Typen.

Abbildung 40 zeigt beispielhaft die Klassenstruktur im ersten Kontext, der sich auf die Klasse `Element` bezieht (und alle konkretisierenden Klassen, wie etwa `HumanTask`). Im Zentrum steht die Klasse `Element`, welche durch `ElementOnPart` überwacht wird. Überwachende `OnPart`s sind im Attribut `elementObservers` der Klasse `Element` referenziert. Eine Referenz auf das überwachte Element ist im Attribut `elementRef` der Klasse `ElementOnPart` gespeichert. Eine Referenz zum `Sentry`, zu dem ein `OnPart` gehört, ist im Attribut `sentryRef` der Klasse `OnPart` gespeichert. `Sentry`s wiederum besitzen durch die Attribute `onParts` und `ifPart` Referenzen zu den assoziierten `OnPart`(s) und (optionalen) `IfPart`. Die gezeigten Interfaces stellen Methoden bereit, um Beobachter zu verwalten und zu benachrichtigen, wenn eine Zustandsänderung eingetreten ist.

Ein `Element` ändert seinen Zustand mit der Methode `setState(String, String)`. Die Parameter entsprechen den Namen des neuen Zustands und dem des durchlaufenen Übergangs. Auf einem überwachenden `ElementOnPart` wird die Methode `updateElementObserver(String)` ausgeführt. Der Parameter entspricht dem Namen des durchlaufenen Übergangs des Elements und wird mit dem im `ElementOnPart` gespeicherten Attribut `requiredTransition` verglichen. Stimmen sie überein, informiert der nun erfüllte `OnPart` sein referenziertes `Sentry`. Dieses führt seine Methode `checkCriteria()` aus, welches die Bedingungen inklusive vorhandener `IfPart`s prüft. Ist es beispielsweise ein `EntrySentry` und dieses ist erfüllt, wird die Methode `performEntrySentryAction()` ausgeführt und das durch das Attribut `elementRef` referenzierte Element aus dem Zustand „*AVAILABLE*“ in den Zustand „*ENABLED*“ oder „*ACTIVE*“ überführt.

Abbildung 41 veranschaulicht diesen Ablauf: `Task` A geht durch *complete* in den Zustand „*COMPLETED*“ über, benachrichtigt hierdurch einen überwachenden `OnPart`, der wiederum sein `Sentry` benachrichtigt. Das `Sentry` prüft seine Bedingungen, die alle erfüllt sind. Es führt die Methode `performEntrySentryAction()` aus, wodurch `Task` B, an die das `Sentry` „angeheftet“ ist, aus dem Zustand „*AVAILABLE*“ heraus in den Zustand „*ACTIVE*“ geführt wird. Hierzu wird der korrekte Zustandskontext geladen. Anschließend wird die Methode `start()` der Klasse

`StageTaskAvailable` ausgeführt. Diese konkretisiert die abstrakte Klasse `StageTaskState` (siehe Abschnitt 4.6.1.5).

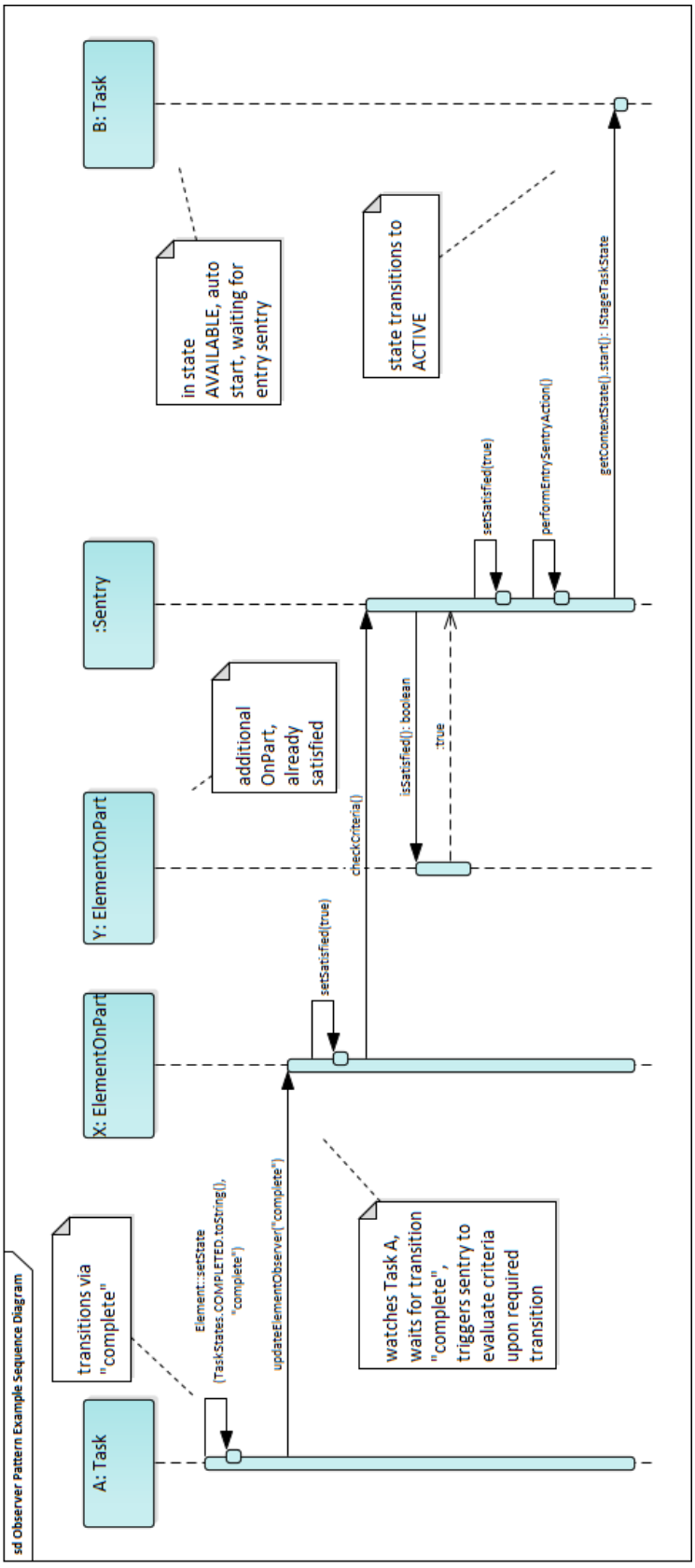

Abbildung 41: Sequenzdiagramm Beobachter-Muster im Element-Kontext

4.6.4 Entwurfsmuster Befehl (*Command Pattern*) für die Befehlsverarbeitung der Services

Dieses Muster kann eingesetzt werden, um Befehle zu kapseln und gezielt erweiterbar zu gestalten (vgl. [64], S. 273ff.; [65], S. 618, S. 641ff.). Im Prototyp wird das Muster im `TaskService` verwendet: Beispielsweise kann jede Anweisung eines Zustandsübgergangs eines Elements, wie etwa *suspend* für die Klassen `Stage` und `Task`, in einer entsprechenden Klasse als Befehl verarbeitet werden. Diese kapseln die Methode `loadContext()` von `Element` und `CaseFileItem`. Die jeweilige Klasse implementiert das Interface `TaskTransitionCommand` und die Methode `execute()` zur Ausführung des eigentlichen Befehls (vgl. Abschnitt 4.6.1.5). Jede Klasse hält eine Referenz auf das Element, auf dem der Befehl ausgeführt werden soll, wie etwa eine Instanz der Klasse `Task`.

```
191 StageTaskTransitionController sttctrl =
192         new StageTaskTransitionController();
193 TaskTransitionCommand sttcomm =
194         StageTaskTransitionCommandFactory
195         .getCommand(transition, taskInEm);
196 sttctrl.saveCommand(sttcomm);
197 sttctrl.executeCommand();
```

Abbildung 42: Befehlsmuster im TaskService

Zur Steuerung und Ausführung der Befehle bedient sich der Service eines `TransitionController`s. Dieser bezieht den entsprechenden Befehl durch eine `TransitionCommandFactory`, speichert den Befehl und führt ihn aus. Somit können beispielsweise Logging-Funktionen auf Ebene der Services angesiedelt werden, statt direkt in Methoden, die Zustandsänderungen repräsentieren. Abbildung 42 zeigt den entsprechenden Algorithmus. Zunächst wird dem `StageTaskTransitionController` ein `TaskTransitionCommand` zugewiesen, das von einer `StageTaskTransitionCommandFactory` bezogen wird (Z. 191-196). Abschließend wird der Befehl ausgeführt (Z.197) und der `Task` in den entsprechenden Zustand überführt, der im Parameter `transition` (Z. 195) angegeben wurde.

5. Praktische Anwendung des Frameworks

In diesem Abschnitt wird in Grundzügen gezeigt, wie eine einfache Applikation auf Basis des Frameworks erstellt werden kann. Ein CMMN-Modell wird in eine Blaupause mit Hilfe der Bausteine des Frameworks übertragen. Grafische Benutzerschnittstellen unterstützen `HumanTask`s, die auf `Property`s eines `CaseFileItem`s zugreifen. Individuelle Implementierungen werden für `ProcessTask`s, einen `IfPart` und einen *Required Rule decorator* erstellt. Die erstellte Blaupause in Form eines `CaseModel`s kann anschließend mit Hilfe der `CaseFactory` und Services des Frameworks instanziiert und bearbeitet werden. Die Instanziierung und Bearbeitung der erstellten Fall-Blaupause wird abschließend in Abschnitt 5.4 ausschnittsweise gezeigt.

5.1 Fallbeispiel Urlaubsantrag

Die Beispielapplikation soll die Bearbeitung von Urlaubsanträgen unterstützen. Zunächst werden durch einen Bearbeiter die Daten eines eingereichten Urlaubsantrags erfasst. Der Antragsteller, Anfang und Ende des gewünschten Zeitraums und die Anzahl der Tage werden in die Maske eingegeben. Anschließend prüft ein Vorgesetzter den Antrag auf Terminüberschneidungen. Wurde er genehmigt, aktualisiert das System das Urlaubskonto des Antragstellers. Der Antragsteller wird über den Ausgang der Prüfung automatisch per E-Mail informiert.

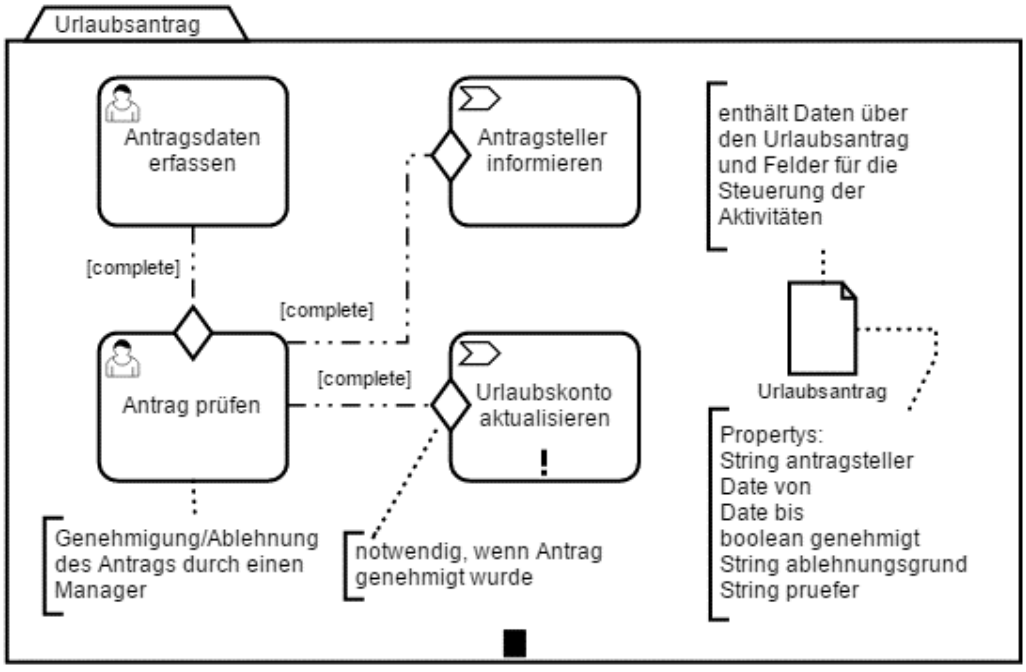

Abbildung 43: CMMN-Modell "Urlaubsantrag"

5.1.1 Modellierung in CMMN

Abbildung 43 zeigt ein mit einem *Auto Complete decorator* versehenes CMMN-Modell des beschriebenen Ablaufs mit vier `Task`s und einem `CaseFileItem`. Die Aufgaben sind durch zwei `HumanTask`s dargestellt. `HumanTask` *Antragsdaten erfassen* speichert die eingegebenen Daten in den `Property`s des `CaseFileItem`s *Urlaubsantrag* (siehe auch Kommentar im Modell zu den enthaltenen `Property`s). `HumanTask` *Antrag prüfen* greift auf diese Daten zurück, um dem Prüfer eine

Entscheidungsgrundlage zu geben. Der Prüfer kann den Antrag genehmigen oder ablehnen. Dies wird im `CaseFileItem` vermerkt.

Die `ProcessTask` *Antragsteller informieren* kapselt das Versenden einer E-Mail an den Antragsteller, um diesen über die getroffene Entscheidung zu informieren. Die `ProcessTask` *Antrag speichern* speichert den Antrag und aktualisiert das Urlaubskonto des Antragstellers. Die Speicherung des Antrags und Aktualisierung des Urlaubskontos wird nur ausgeführt, wenn der Antrag geprüft und genehmigt wurde.

Verknüpft sind die `Task`s durch `EntrySentry`s, die Zustandsänderungen überwachen. Im Beispiel ist der für alle `Sentry`s überwachte Übergang als *complete* angegeben. Dem `EntrySentry` des `ProcessTask` *Urlaubskonto aktualisieren* ist ein `IfPart` zugeordnet. Dieser `IfPart` soll prüfen, ob der Antrag genehmigt wurde. Außerdem weist *Urlaubskonto aktualisieren* einen *Required Rule decorator* auf. Diese Regel gilt, solange der Antrag als genehmigt vermerkt ist.

5.1.2 Analyse des Modells

Bevor das CMMN-Modell aus Abbildung 43 in die Strukturen des Frameworks übertragen wird, soll die zu erstellende Fallstruktur analysiert werden. Es wird ermittelt, welche individuellen Bestandteile entwickelt werden müssen. Insbesondere Datenstrukturen müssen analysiert werden.

Für das Beispiel lässt sich folgende grundlegende Struktur der zu erstellenden Fall-Blaupause festlegen: Ein `CaseModel` beinhaltet als Kind-Elemente zwei `HumanTask`s und zwei `ProcessTask`s. Für die `HumanTask`s müssen Vaadin-Masken erstellt werden. Als Rollen werden „Bearbeiter“ und „Prüfer“ identifiziert. Die `ProcessTask`s benötigen individuelle `ProcessTaskImplementations`. Außerdem werden für einen `IfPart` eine `IfPartImplementation` und für einen *Required Rule decorator* eine `RuleExpression` benötigt. Verknüpfungen werden durch `Sentry`s mit `ElementOnPart`s abgebildet, die alle auf den Zustandsübergang *complete* reagieren.

Die Datenstruktur für das `CaseFileItem` enthält vor allem Daten, die vom Benutzer manipuliert werden. Sie sind in Tabelle 2 aufgeführt. Die Variable „genehmigt“ wird zur Prüfung des `IfPart`s genutzt. Die Variable „required“ wird für den *Required Rule decorator* genutzt. Wird der Antrag abgelehnt, wird der Grund hierfür in Variable „ablehnungsgrund“ gespeichert.

Variable/Property	Datentyp	Beschreibung
antragsteller	Long	ID des Antragstellers
von	Date	Beginn des Urlaubs
bis	Date	Ende des Urlaubs
tage	int	Anzahl der Urlaubstage
genehmigt	boolean	Wurde der Antrag genehmigt?
required	boolean	Ist die Speicherung notwendig?
ablehnungsgrund	String	Was ist der Ablehnungsgrund?
pruefer	String	Wer war der Prüfer des Antrags?

Tabelle 2: Datenstruktur des CaseFileItems Urlaubsantrag

5.2 Übertragung des Modells in die Strukturen des Frameworks

Die analysierten Modellbestandteile werden nun in die Strukturen des Frameworks übertragen. Zunächst wird gezeigt, wie eine Blaupause mit der grundlegenden Fallstruktur manuell erstellt wird. Diese wird durch eine Methode der `CaseFactory` als ein `CaseModel` zurückgegeben. Anschließend wird exemplarisch gezeigt, wie eine `IfPart`- und `ProcessTask`-Implementierung erstellt und diese durch Factory-Klassen verknüpft werden. Außerdem wird ein Auszug aus dem Quellcode einer Maske für einen `HumanTask` gezeigt. Er zeigt, wie ein injizierter `TaskService` genutzt wird, um die `SimpleProperty`s eines `CaseFileItem`s abzurufen und zu bearbeiten.

5.2.1 Wurzelelement „Urlaubsantrag“

Abbildung 44 zeigt den ersten Teil der Blaupause. Als Wurzelelement wird in Zeile 127 ein `CaseModel` mit der `cmId` „urlaubsantrag“ instanziiert. Der zweite Parameter des Konstruktors ist der aus dem grafischen Modell abzulesende Name des Falles. In Zeile 128 wird der *Auto Complete decorator* eingestellt.

```
127 CaseModel model = new CaseModel("urlaubsantrag", "Urlaubsantrag");
128 model.setAutoComplete(true);
129 CaseFileItem urlaubsantrag = new CaseFileItem("urlaubsantrag",
130         MultiplicityEnum.ExactlyOne.toString(), "Urlaubsantrag");
131 model.getCaseFile().addCaseFileItem(urlaubsantrag);
```

Abbildung 44: CaseModel der Fall-Blaupause

Anschließend wird in Zeile 129f. das `CaseFileItem` mit der `cmId` „urlaubsantrag“ und dem Namen „Urlaubsantrag“ instanziiert. Das Attribut `multiplicity` wird mit „ExactlyOne“ initialisiert, da es nur eine Version des Urlaubsantrags gibt. Das `CaseFileItem` wird zunächst ohne `SimpleProperty`s in Zeile 131 dem `CaseFile` hinzugefügt. Das `CaseFile` wird durch den Aufruf der Methode `getCaseFile()` der Klasse `CaseModel` automatisch erstellt, wenn es nicht vorhanden ist.

5.2.2 Hinzufügen von Human- und ProcessTasks

Abbildung 45 zeigt, wie dem `CaseModel` seine Kind-Elemente hinzugefügt werden können. Hierzu werden entsprechend Objekte der Klasse `HumanTask` (Z.133-138) und `ProcessTask` (Z.139ff.) instanziiert. Ihren Konstruktoren wird eine `cmId`, ein durch Menschen lesbarer Name wie im CMMN-Modell, sowie eine Referenz zum `CaseModel`-Wurzelelement übergeben. Die verwendeten Konstruktormethoden fügen das Element automatisch dem angegebenen `CaseModel` hinzu.

Die `cmId`s der `HumanTask`s werden später durch das Vaadin-CDI Addon zur korrekten Instanziierung der entsprechenden grafischen Maske verwendet. Z.135 und Z.138 zeigen einen Weg, wie `CaseRoles` konfiguriert werden können, um Zugriffe auf die `Task`s auf bestimmte Rollen zu beschränken.

```
133 HumanTask antragsdatenErfassen = new HumanTask("antragsdatenErfassen",
134         "Daten für Urlaubsantrag erfassen", model);
135 antragsdatenErfassen.setCaseRole(new CaseRole("bearbeiter"));
136 HumanTask antragPruefen = new HumanTask("antragPruefen",
137         "Urlaubsantrag prüfen", model);
138 antragsdatenErfassen.setCaseRole(new CaseRole("pruefer"));
139 ProcessTask antragstellerInformieren =
140         new ProcessTask("antragstellerInformieren",
141                 "Antragsteller Informieren", model);
142 ProcessTask urlaubskontoAktualisieren =
143         new ProcessTask("urlaubskontoAktualisieren",
144                 "Urlaubskonto aktualisieren", model);
145 RequiredRule aktualisierenRequired =
146         new RequiredRule("required", urlaubsantrag);
147 urlaubskontoAktualisieren.setRequiredRule(aktualisierenRequired);
```

Abbildung 45: Task-Elemente der Fall-Blaupause

Die übergebenen `cmId`s der `ProcessTask`s werden zur Laufzeit benutzt, um durch eine Fabrik-Methode die korrekten Implementierungen zu erzeugen und auszuführen. Zeilen 145ff. zeigen, wie für den `ProcessTask` *Urlaubskonto Aktualisieren* der *Required Rule decorator* konfiguriert wird. Eine Instanz der Klasse `RequiredRule` mit dem Namen „required“ wird dem `ProcessTask` durch deren Methode `setRequiredRule()` hinzugefügt. Der im Konstruktor übergebene Regelname wird zur Laufzeit dazu genutzt, die entsprechende Implementierung auszuführen. Die übergebene Referenz des `CaseFileItem`s wird als Grundlage für die Prüfung der Regel genutzt.

5.2.3 Verknüpfung durch Sentry-Strukturen

Abbildung 46 zeigt, wie `Sentry`-Strukturen die Elemente miteinander verknüpfen. Einem `EntrySentry` wird im Konstruktor als erste zwei Parameter eine optionale `cmId` und ein optionaler Name übergeben. Der dritte Parameter ist eine Referenz auf das Element, dem das `Sentry` zugeordnet ist und welches von diesem gesteuert wird. Es soll das an den `ProcessTask` *Urlaubskonto aktualisieren* angehefte `Sentry` näher betrachtet werden.

```
149 EntrySentry enterAntragPruefen = new EntrySentry("", "", antragPruefen);
150 ElementOnPart enterAntragPruefenOn =
151         new ElementOnPart(enterAntragPruefen, antragsdatenErfassen,
152                 StageTaskTransitions.complete.toString());
153 EntrySentry enterAntragstellerInformieren =
154         new EntrySentry("", "", antragstellerInformieren);
155 ElementOnPart enterAntragstellerInformierenOn =
156         new ElementOnPart(enterAntragstellerInformieren, antragPruefen,
157                 StageTaskTransitions.complete.toString());
158 EntrySentry enterUrlaubskontoAktualisieren =
159         new EntrySentry("", "", urlaubskontoAktualisieren);
160 ElementOnPart enterUrlaubskontoAktualisierenOn =
161         new ElementOnPart(enterUrlaubskontoAktualisieren,
162                 antragPruefen, StageTaskTransitions.complete.toString());
163 IfPart urlaubskontoAktualisierenIf = new IfPart("kontoAktualisieren",
164         enterUrlaubskontoAktualisieren, urlaubsantrag);
```

Abbildung 46: Sentry-Strukturen der Fall-Blaupause

Zunächst wird in Z.158f. ein neues `EntrySentry` instanziiert. Dem Konstruktor wird die zuvor deklarierte Variable `urlaubskontoAktualisieren` als Referenz auf den `ProcessTask` übergeben. Der `ProcessTask`-Instanz wird durch die Konstruktormethode das `Sentry` automatisch zu seiner `Sentry`-Liste hinzugefügt. Anschließend wird in Zeile 160ff. ein `ElementOnPart` instanziiert. Diesem werden im Konstruktor drei Parameter übergeben: Der erste ist die Referenz auf das `EntrySentry`, zu dem der `OnPart` gehört. Der zweite Parameter ist eine Referenz auf das zu überwachende Element, also der zuvor deklarierte `HumanTask` *Antrag Prüfen*. Der dritte Parameter gibt an, welcher Zustandsübergang diesen `OnPart` erfüllt. In diesem Falle ist dies *complete*. Für die Angabe wird in Zeile 162 die Enumeration-Klasse `StageTaskTransitions` genutzt, welche alle Zustandsübergänge der Klassen `Stage` und `Task` definiert.

Ein `IfPart` des `EntrySentry`s wird in Z.163f. mit drei Parametern instanziiert. Der erste ist die `cmId` des `IfPart`s, welche zur Laufzeit für die korrekte Verarbeitung der entsprechend erstellten `IfPart`-Implementierung genutzt wird. Der zweite Parameter ist eine Referenz auf das `Sentry` zu dem der `IfPart` gehört. Der `IfPart` wird durch die Konstruktormethode automatisch dem `Sentry` hinzugefügt. Der dritte Parameter ist eine Referenz zum `CaseFileItem`, welche als Basis für die Evaluierung durch die `IfPart`-Implementierung dient.

5.2.4 Datenstruktur des CaseFileItems „Urlaubsantrag“

Die zuvor im Rahmen der Analyse festgelegten Daten werden durch `SimpleProperty`s ausgedrückt, die einem `CaseFileItem` zugeordnet werden. Abbildung 47 zeigt, wie in Z.166-172 `SimpleProperty`-Instanzen erstellt werden. Dem Konstruktor einer `SimpleProperty` wird ein Schlüssel-Wert-Paar übergeben. Z.173f. zeigt, wie die Instanzen dem `CaseFileItem` hinzugefügt werden. Die Methode `addProperty()` akzeptiert mehrere Referenzen auf `SimpleProperty`s.

```
166 SimpleProperty antragsteller = new SimpleProperty("antragsteller", "");
167 SimpleProperty von = new SimpleProperty("von", "");
168 SimpleProperty bis = new SimpleProperty("bis", "");
169 SimpleProperty tage = new SimpleProperty("tage", "");
170 SimpleProperty genehmigt = new SimpleProperty("genehmigt", "false");
171 SimpleProperty ablehnungsgrund = new SimpleProperty("ablehnungsgrund", "");
172 SimpleProperty pruefer = new SimpleProperty("pruefer", "");
173 urlaubsantrag.addProperty(antragsteller, von, bis, tage, genehmigt,
174         ablehnungsgrund, pruefer);
```

Abbildung 47: CaseFileItem-Struktur der Fall-Blaupause

Zu beachten ist, dass `SimpleProperty`s Werte als Zeichenkette speichern: So wird in Z.170 der boolesche Wert als String „false“ gespeichert. Bei der Manipulation von `Property`s müssen Werte entsprechend konvertiert werden. Zuvor wurde versucht, generische Klassen mit Typ-Inferenz zu verwenden, um verschiedene Typen abzubilden. Dies war aber mit JPA nicht fehlerfrei möglich, sodass für den Prototyp auf einfache String-Repräsentationen zurückgegriffen wird.

5.2.5 Initialisierung des CaseModels und enthaltener Elemente

Abschließend werden das `CaseModel` und die enthaltenen Elemente initialisiert, bevor die erstellte Fallstruktur zurückgegeben wird und persistiert werden kann. Abbildung 48 zeigt den verketteten Aufruf der Methoden `getContextState()` und `create()` in Zeile 176. Hierdurch wird die `CaseModel`-Instanz aus dem Zustand *INITIAL* in den Zustand *ACTIVE* überführt.

```
176 model.getContextState().create();
177 return model;
```

Abbildung 48: Initialisierung des CaseModels der Fall-Blaupause

Die Aktivierung wird an alle Kind-Elemente, im Beispiel also an die vier `Task`s, propagiert. Sie werden aus dem Initialzustand in zulässige Zustände überführt. Somit wird der `HumanTask` *Antragsdaten erfassen* in den Zustand *ACTIVE* überführt und kann von einem `CaseWorker` bearbeitet werden, wohingegen alle anderen Tasks in den Zustand *AVAILABLE* überführt werden. Auch `CaseFileItem`s können durch den Aufruf gleichnamiger Methoden aus ihrem Initialzustand überführt werden.

Hierbei ist zu beachten, dass der Zustandsübergang *create* nur einmal durchgeführt werden kann. Überwachen `CaseFileItemOnPart`s von `Sentry`s genau diesen Übergang des `CaseFileItem`s und wird dieser am Anfang eines Falles durchgeführt, kann dieser folglich nur initial genutzt werden.

Außerdem muss beachtet werden, dass die Initialisierung gegebenenfalls Elemente aktiviert, welche Gebrauch von Services machen. So können eine `IfPart`-Implementierung oder die Implementierung eines `ProcessTask`s beispielsweise auf `CaseFileItem`s zugreifen. Folglich sollte die Initialisierung im Zweifel erst nach der Persistierung der Fall-Instanz aus der Blaupause angestoßen werden, damit benötigte Referenzen bereits in der Datenbank vorhanden sind und gefunden werden können.

5.3 Individuelle Implementierungen

Individuelle Implementierungen werden auf Basis von abstrakten Klassen des Frameworks erstellt und durch `Factory`s zur Laufzeit mit den Elementen des `CaseModel`s verknüpft. Die Erstellung anderer individueller Implementierungen, wie etwa für `CaseTask`s und `Rule`s für *decorators*, folgt dem gleichen Schema. In diesem Abschnitt soll exemplarisch gezeigt werden, wie die individuellen Implementierungen für einen `IfPart` und einen `ProcessTask` erstellt und mit den Factory-Klassen verknüpft werden. Es werden die Erstellung einer `IfPartImplementation` und einer `ProcessTaskImplementation` gezeigt. Außerdem wird beispielhaft ein Auszug aus dem Quellcode der Maske für den `HumanTask` *Antrag prüfen* gezeigt.

5.3.1 IfPart, IfPartImplementation und IfPartImplementationFactory

```
14  @Override
15  public boolean isSatisfied() {
16      boolean response = false;
17      CaseFileItem urlaubsantrag = ip.getCaseFileItemRef();
18      SimpleProperty lookingFor = urlaubsantrag.getProperty("genehmigt");
19      if (lookingFor != null) {
20          if (lookingFor.getValue() != null) {
21              if (Boolean.parseBoolean(lookingFor.getValue())) {
22                  response = true;
23              }
24          }
25      }
26      return response;
27  }
```

Abbildung 49: Individuelle IfPart-Implementierung

Für die Implementierung des `IfPart`s wird eine neue Klasse erstellt, welche die abstrakte Klasse `IfPartImplementation` konkretisiert. Abbildung 49 zeigt die überschriebene Methode `isSatisfied()`. Diese Methode wird später vom assoziierten `Sentry` aufgerufen, um ihre Bedingungen zu prüfen. In Z.17f. wird durch das im `IfPart` referenzierte `CaseFileItem` dessen `SimpleProperty` „genehmigt“ initialisiert. Z.21 zeigt die Konvertierung des String-Wertes in einen booleschen Wert, der in Z.26 als boolesche Variable `response` zurückgegeben wird, sofern der Wert vorhanden ist.

Abbildung 50 zeigt die Konfiguration der Klasse `IfPartImplementationFactory`. In einer `Switch-Case`-Verzweigung wird der in Abbildung 45 in Z.163 angegebene Name der `IfPart`-Instanz als `case` definiert und im `Switch`-Teil mit der `cmId` des übergebenen `IfPart`s verglichen. Entspricht er dem Namen (hier „kontoAktualisieren“), wird eine Instanz der erstellten Klasse zurückgegeben.

```
public static IfPartImplementation getIfPartImplementation(IfPart ip) {
    switch (ip.getCmId()) {
    case "kontoAktualisieren":
        return new AntragSpeichernIfPartImplementation(ip);
    default:
        return null;
    }
}
```

Abbildung 50: Konfiguration der IfPartImplementationFactory

5.3.2 ProcessTask, ProcessTaskImplementation und ProcessTaskImplementationFactory

Für die Implementierung des `ProcessTask` Antragsteller informieren wird eine Klasse erstellt, welche die abstrakte Klasse `ProcessTaskImplementation` konkretisiert. Die abstrakte Methode `startProcess()` wird überschrieben. Dies ist in Abbildung 51 dargestellt: In Zeile 23 wird mit Hilfe vom CDI-Container des Applikationsservers ein zustandsloser `CaseFileService` initialisiert[26]. In der folgenden Zeile wird durch den Kontext der `ProcessTask`-Instanz eine Referenz zum übergeordneten `CaseModel` erstellt und in Z.27 das `CaseFileItem` „Urlaubsantrag“ initialisiert. Anschließend wird der Wert der `SimpleProperty` „genehmigt“ ausgelesen und der Antragsteller entsprechend informiert. Z.31ff. zeigt, wie der Ablehnungsgrund geladen wird, wenn der Antrag nicht genehmigt wurde. Abschließend wird der Zustand der `ProcessTask`-Instanz durch Aufruf der verketteten Methoden `getContextState().complete()` in den Zustand *COMPLETED* überführt.

Die erstellte Klasse wird anschließend – wie bei der vorgestellten individuellen Implementierung des `IfPart`s – in der Klasse `ProcessTaskImplementationFactory` im Rahmen einer `Switch-Case`-Verzweigung referenziert. Zur Laufzeit kann auf Basis der `cmId` der übergebenen `ProcessTask`-Instanz die Implementierung erzeugt und ausgeführt werden zu.

[26] Die Verwaltung der zustandslosen Service-Instanz wird so vom Applikationsserver gesteuert. Sie kann nach Gebrauch wieder freigegeben und anderen Clients zugeteilt werden.

```
20@Override
21 public void startProcess() {
22     // get CaseFileService via CDI-context, since @Inject does not work in a non-managed bean
23     CaseFileService cfService = CDI.current().select(CaseFileService.class).get();
24     Long caseModelRefId = this.processTask.getCaseRef().getId();
25     CaseModel shallowCase = new CaseModel();
26     shallowCase.setId(caseModelRefId);
27     CaseFileItem urlaubsantrag = cfService.getCaseFileItem(shallowCase, "urlaubsantrag");
28     boolean genehmigt = Boolean.parseBoolean(urlaubsantrag.getProperty("genehmigt").getValue());
29     if(genehmigt) {
30         // Antragsteller per E-Mail informieren
31     } else {
32         String grund = urlaubsantrag.getProperty("ablehnungsgrund").getValue();
33         // Antragsteller mit Grund der Ablehnung informieren
34     }
35     this.processTask.getContextState().complete();
36 }
```

Abbildung 51: Individuelle ProcessTask-Implementation

Wie auch für die anderen individuellen Implementierungen, wird für die in Abbildung 45 in Z.145ff. erstellte `RequiredRule` eine Klasse erstellt. Diese konkretisiert die abstrakte Klasse `RuleExpression` und überschreibt dessen abstrakte Methode `evaluate()`. Anschließend wird die konkretisierende Klasse in einer `Switch-Case`-Verzweigung der Factory-Klasse `RuleExpressionFactory` verknüpft. Auf Basis des im Konstruktur angegebenen Regelnamen „required“ wird zur Laufzeit bei Bedarf die korrekte Regel erzeugt und ausgeführt.

5.3.3 Maske für `HumanTask` „Antrag prüfen“

```
40 @CDIView("antragPruefen")
41 public class UrlaubsantragPruefenView extends CustomComponent implements View {
   // ..
43     @Inject private CaseWorkerService cwService;
44     @Inject private TaskService taskService;
45     @Inject private CaseWorkerInfo caseWorkerInfo;
46     @Inject private CaseFileService caseFileService;
47     @Inject private TaskInfo taskInfo;
48
49     private Button btnComplete;
50
51     private Button generateCompleteButton() {
52         Button completeButton = new Button("Antrag genehmigen");
53         completeButton.addClickListener(e -> {
54             urlaubsantrag.getProperty("genehmigt").setValue(String.valueOf(true));
55             caseFileService.updateCaseFileItem(urlaubsantrag);
56             taskService.transitionTask(taskInfo.getTask(), caseWorkerInfo.getUser(),
57                     StageTaskTransitions.complete);
58             // ..
```

Abbildung 52: Vaadin-CDI-View und injizierte Services

Abbildung 52 zeigt einen Auszug des Quellcodes einer mit Vaadin erstellten Maske. Die Annotation *`@CDIView`* teilt dem Addon mit, dass es sich um eine CDI-Maske handelt. Masken, die mit dem Vaadin-CDI Addon erstellt werden, können auf zwei Arten vom Vaadin-Navigator (vgl. Abschnitt 4.1.3) angesprochen werden, die in Z.40f. dargestellt sind.

Die erste Art ist die Parametrisierung der Annotation. Der angegebene String-Parameter „antragPruefen“ entspricht der `cmId` des in Abbildung 45 Z.136 instanziierten `HumanTask` *Antrag prüfen*. So kann – auf Basis einer durch einen `CaseWorker` ausgewählten `Task`-Instanz und ihrer `cmId` – zur entsprechenden Maske navigiert werden. Ein korrekter Kontext kann anschließend über Attribute der `Task`-Instanz hergestellt werden, wie etwa durch das in ihr referenzierte `CaseModel`.

Die zweite Art ist die Namensgebung der Klasse. Ist die Klasse annotiert und als CDI-View registriert, aber kein Parameter angegeben, kann zur Maske über ihren transformierten Klassennamen navigiert werden. Der Klassenname „UrlaubsantragPruefenView“ würde durch das Vaadin Addon zu „urlaubsantrag-pruefen“ werden und könnte über diesen angesprochen werden.

Zeilen 43-47 zeigen, wie durch die Annotation `@Inject` verschiedene Services und die Info-Beans `CaseWorkerInfo` und `TaskInfo` in die Maske injiziert werden. `CaseWorkerInfo` enthält Informationen über den angemeldeten Benutzer der Applikation. `TaskInfo` enthält Informationen über eine zuvor durch den angemeldeten `CaseWorker` ausgewählte Task-Instanz. Das `CaseFileItem` `urlaubsantrag` wird an nicht gezeigter Stelle über den injizierten `CaseFileService` geladen.

Der Auszug des Quellcodes zeigt in Z.49ff. die Erstellung eines `Button`s, der den Antrag genehmigt und die Arbeit der `HumanTask` abschließt. Hierzu wird zunächst der `SimpleProperty` „genehmigt“ der Wert „true“ zugewiesen (Z.54). Anschließend wird der `urlaubsantrag` durch den `CaseFileService` aktualisiert (Z.55). Abschließend wird der `TaskService` genutzt, um die in der Info-Bean referenzierte `Task`-Instanz in den Zustand *COMPLETED* zu überführen (Z.56f.).

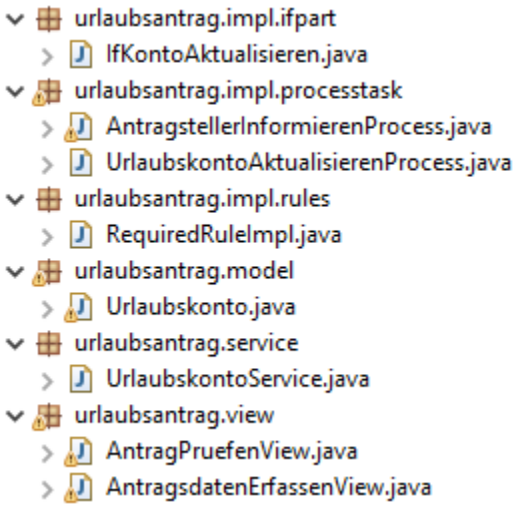

Abbildung 53: Paketstruktur und Klassen der individuellen Implementierungen

5.3.4 Übersicht über Paketstruktur und erstellte Klassen des Fallbeispiels

Abbildung 53 zeigt alle für das Fallbeispiel erstellten Klassen im übergeordneten Paket `urlaubsantrag`. Im Paket `urlaubsantrag.impl` befinden sich in den enthaltenen Paketen die auf Basis des Frameworks erstellten individuellen Implementierungen für den `IfPart`, die `ProcessTask`s, die `RequiredRule`. Im Paket `urlaubsantrag.view` sind zwei mit Vaadin erstellte Masken für die `HumanTasks` zu sehen.

Bisher nicht angesprochen wurden die Klassen in den Paketen `urlaubsantrag.model` und `urlaubsantrag.service`. Sie dienen im Fallbeispiel der Repräsentation der Urlaubskonten der Anwender der Applikation. Die Klasse `Urlaubskonto` ist ein für JPA annotiertes POJO. Sie enthält die ID eines `CaseWorkers` und einen Integer-Wert für die verbleibenden Tage. Die Klasse `UrlaubskontoService` ist ein zustandsloser und injizierbarer Service, der zwei Methoden anbietet, um das Urlaubskonto eines `CaseWorkers` abzurufen und dessen Tage anzupassen. Er simuliert den Zugriff aus der Framework-Applikation heraus auf die Daten eines externen Systems über einen Service.

5.4 Ausführung und Bearbeitung einer Fall-Instanz

In diesem Abschnitt wird ausschnittsweise die Bearbeitung einer Instanz des zuvor erstellten Fallbeispiels gezeigt. Für das Beispiel wird mit Hilfe von JPA auf Basis der annotierten Klassen zunächst automatisch ein Datenbankschema erstellt. Anschließend werden zwei Benutzer mit den Rollen aus der Fall-Blaupause (vgl. Abschnitt 5.2.2) persistiert: „Jane Worker“ mit der Rolle „bearbeiter“ (Login mit worker als Benutzername und Passwort) und „John Admin“ mit der Rolle „pruefer“ (Login mit admin als Benutzername und Passwort). Zusätzlich wird für beide `CaseWorker` jeweils ein Urlaubskonto-Eintrag mit je 25 und 30 Urlaubstagen erstellt.

Der Framework-Prototyp stellt neben einer Login-Maske zwei einfache Übersichtsmasken bereit: Case-List und Task-List. In der Case-List können Fallinstanzen gestartet, angezeigt und gelöscht werden. Die Task-List zeigt verfügbare `HumanTasks` entsprechend der Rolle des angemeldeten `CaseWorkers` an. Über sie können `HumanTasks` beansprucht und anschließend bearbeitet werden. Beide Übersichtsmasken und die individuell erstellten Masken für die `HumanTasks` des Fallbeispiels benutzen injizierbare Services des Frameworks (vgl. Abschnitt 5.3.3).

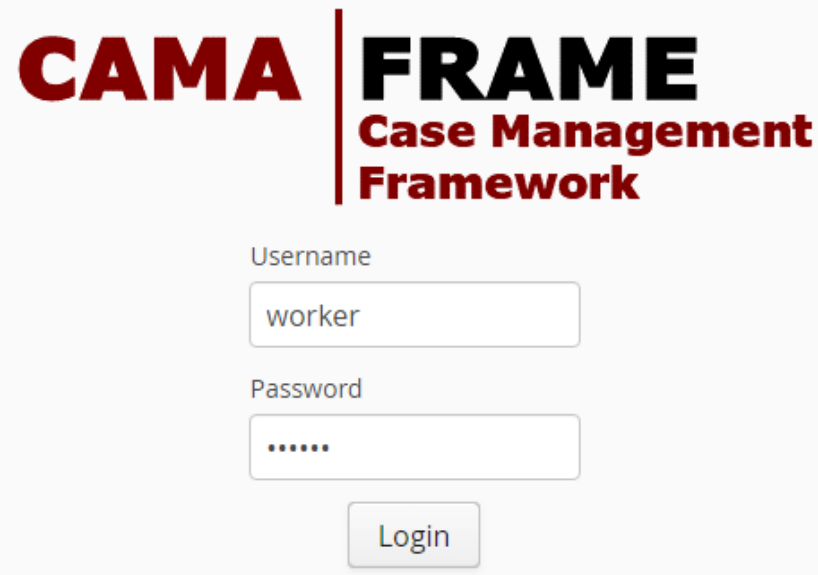

Abbildung 54: Login als Jane Worker

5.4.1 Login, Fall-Instanziierung und Task-Beanspruchung

Zunächst wird sich wie in Abbildung 54 zu sehen als „Jane Worker“ durch eine im Framework enthaltene Login-Maske angemeldet. Im Hintergrund wird ein `CaseWorkerService` genutzt, um die Anmeldedaten zu prüfen. Nach dem Login wird automatisch die Case-List angezeigt.

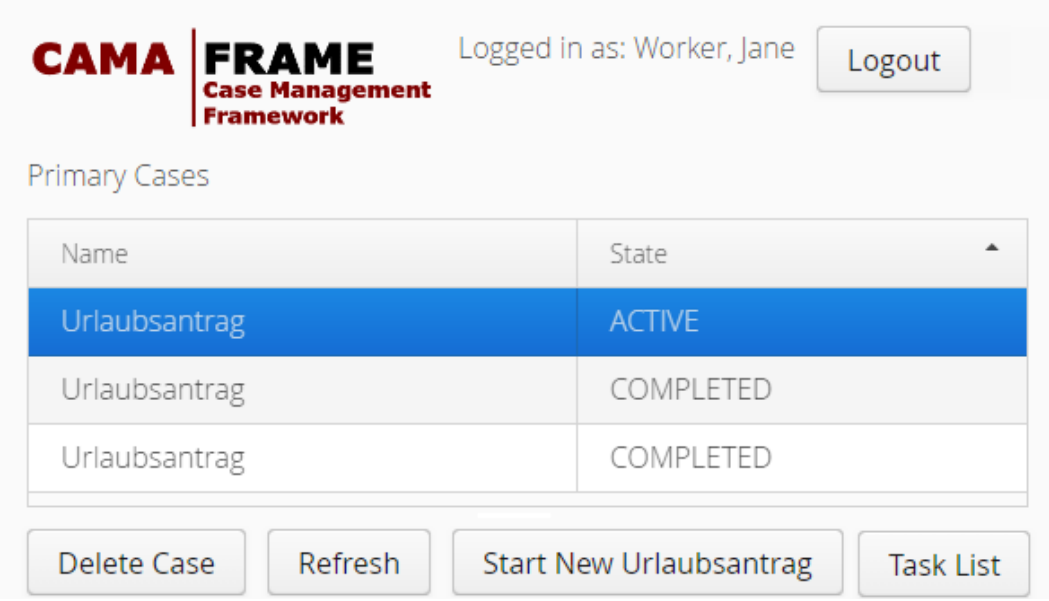

Abbildung 55: Case-List mit neuer Fall-Instanz

Die in Abbildung 55 gezeigte Case-List nutzt im Hintergrund einen `CaseService` zum Abrufen persistierter Fall-Instanzen aus der Datenbank. Durch die Schaltfläche „Start New Urlaubsantrag“ wird mit Hilfe des `CaseService` eine Fall-Blaupause neu instanziiert. In der Case-List sind anschließend drei Fall-Instanzen zu sehen, von denen zwei ältere im abgeschlossenen Zustand „COMPLETED“ sind. Die neue Fall-Instanz ist im aktiven Zustand „ACTIVE“ und soll bearbeitet werden. Hierzu wird mit Hilfe der Schaltfläche „Task List“ zur Task-List navigiert.

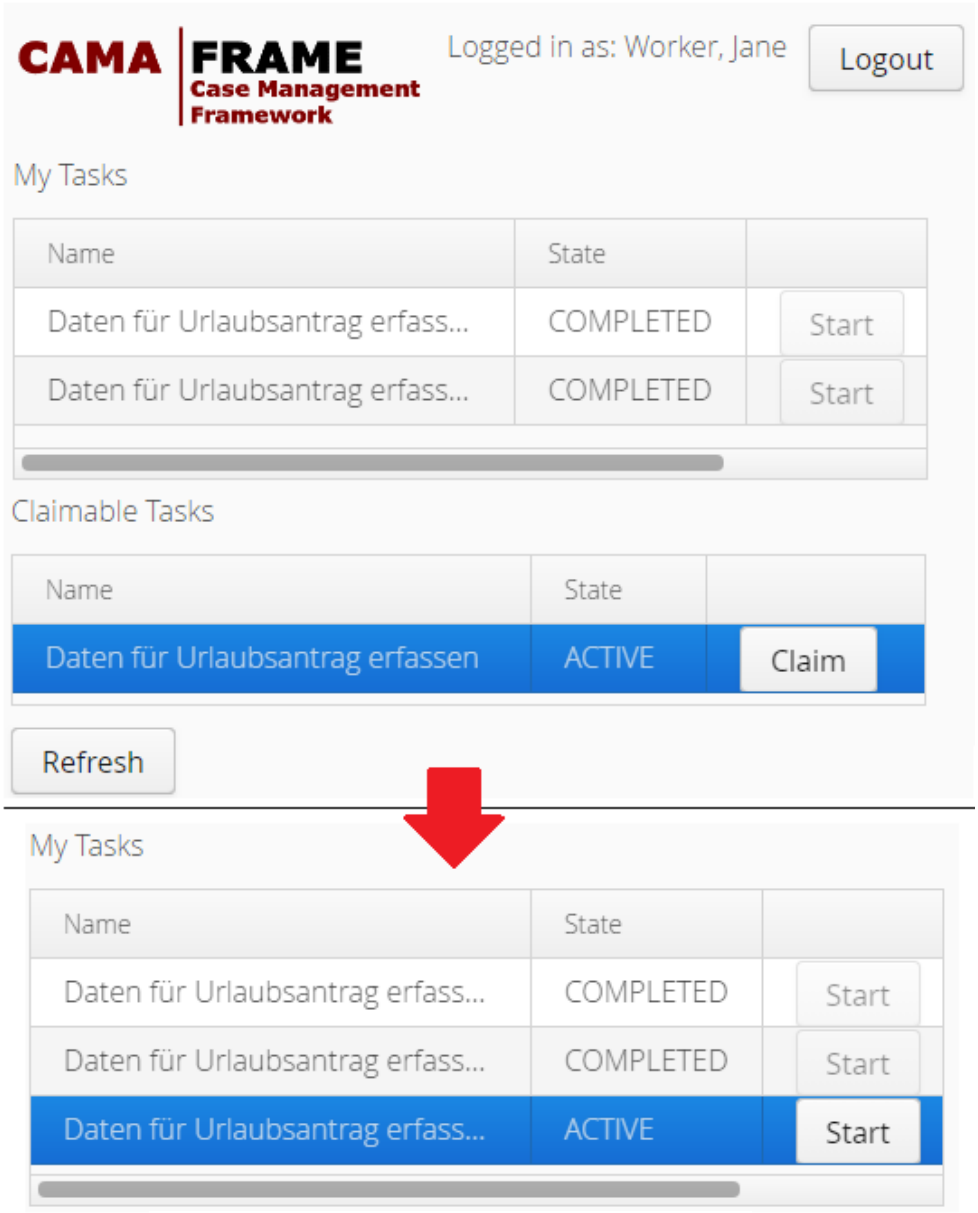

Abbildung 56: Task-List für "Jane Worker"

Abbildung 56 zeigt die Task-List für „Jane Worker“. Unter „My Tasks“ im oberen Teil der Abbildung sind die bereits bearbeiteten `HumanTasks` der zwei vergangenen Fall-Instanzen mit dem Status „COMPLETED“ zu sehen. Unter „Claimable Tasks“ ist der `HumanTask` „Daten für Urlaubsantrag erfassen“ der neuen Fall-Instanz zu sehen. Er wird durch die Schaltfläche „Claim“ durch „Jane Worker“ beansprucht und taucht anschließend unter „My Tasks“ auf (siehe Pfeil). Dies ist im unteren Teil der Abbildung dargestellt. Im Hintergrund wird ein `TaskService` genutzt. Durch die Schaltfläche „Start“ des ausgewählten Tasks wird zu einer durch das Vaadin Addon erzeugte Maske für den `HumanTask` „Antragsdaten erfassen“ navigiert (vgl. Abschnitt 4.1.3).

5.4.2 Erfassung der Antragsdaten

Abbildung 57 zeigt die Maske zur Erfassung der Antragsdaten. Als Antragsteller wird „John Admin“ ausgewählt und ein Zeitraum über zwei Tage gewählt. Die Maske (also auch der zugrundeliegende `HumanTask` *Antragsdaten erfassen*) wird durch die Schaltfläche „Fertig“ beendet.

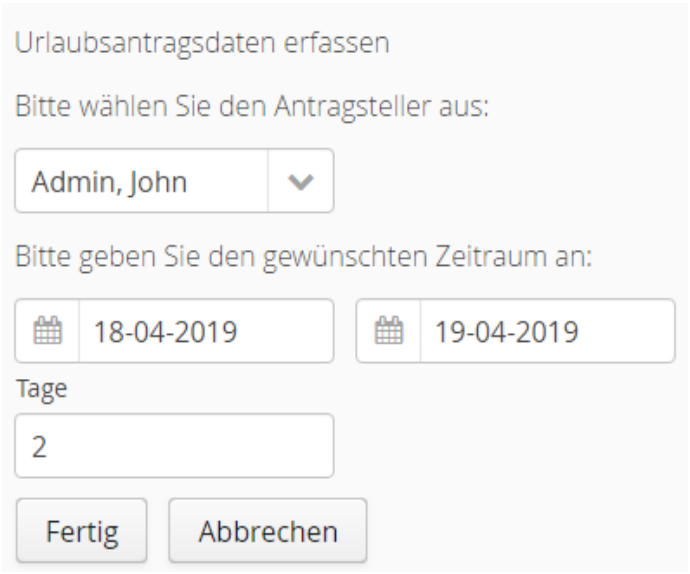

Abbildung 57: Erfassung der Antragsdaten

Im Hintergrund werden ein in die Maske injizierter `CaseWorkerService` zur Auswahl des Antragstellers, ein `CaseFileService` zur Speicherung der erfassten Daten und ein `TaskService` für den Zustandsübergang des abgeschlossenen `HumanTask`s genutzt. Der Zustandsübergang erfüllt das `Sentry` des `HumanTask` *Antrag prüfen* und wird in einen aktiven Zustand überführt.

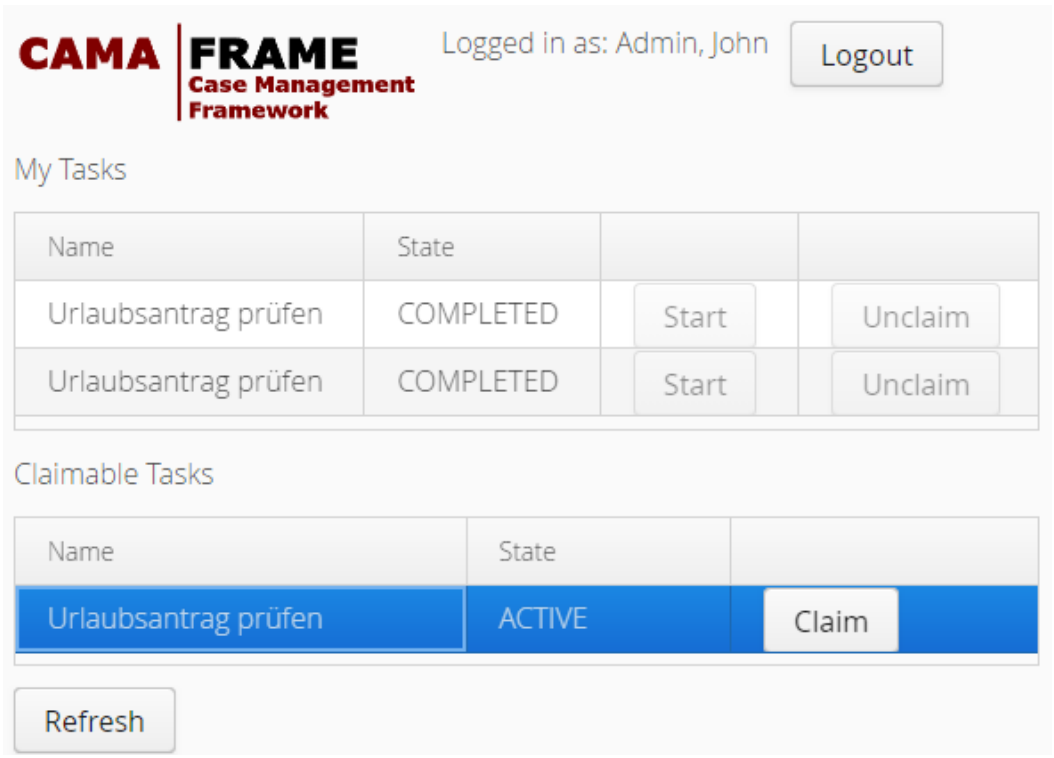

Abbildung 58: Task-List für "John Admin" als Prüfer

5.4.3 Antrag prüfen als Benutzer „John Admin“

Abbildung 58 zeigt die Task-List des nun angemeldeten Nutzers „John Admin“. Er sieht nun im Gegensatz zu „Jane Worker“ in seiner Task-List unter „Claimable Tasks“ eine aktive Instanz des `HumanTasks` *Antrag prüfen*. Über die Task-List beansprucht er die Aufgabe für sich und navigiert anschließend durch die Schaltfläche „Start“ wie zuvor beschrieben zur Maske der Aufgabe.

Er prüft durch die in Abbildung 59 gezeigte Maske des `HumanTask` *Antrag prüfen* den Antrag. Will er den Antrag ablehnen, muss er im gezeigten Textfeld einen Ablehnungsgrund eingeben und die Aufgabe durch die Schaltfläche „Antrag ablehnen“ beenden. Anschließend wird der Antragsteller über die Ablehnung und den angeführten Grund informiert. Genehmigt er den Antrag, wird der Antragsteller über die Genehmigung und den aktualisierten Stand seines Urlaubskontos informiert.

Im Hintergrund werden erneut Services genutzt, die in die Maske injiziert wurden. Ein `CaseWorkerService` wird zur Anzeige des Namens des Antragstellers genutzt. Ein `CaseFileService` wird zum Laden der Antragsdaten und Speichern des Ablehnungsgrundes eingesetzt. Ein `TaskService` überführt den der Maske zugrundeliegenden `HumanTask` bei Abschluss in den nächsten Zustand. Wird der Antrag genehmigt, wird in der `ProcessTask`-Implementierung ein `UrlaubskontoService` zur Aktualisierung der Urlaubstage eingesetzt. Als Parameter wird diesem der Wert der `SimpleProperty` „tage“ übergeben, welcher zuvor vom `CaseFileService` geladen wurde.

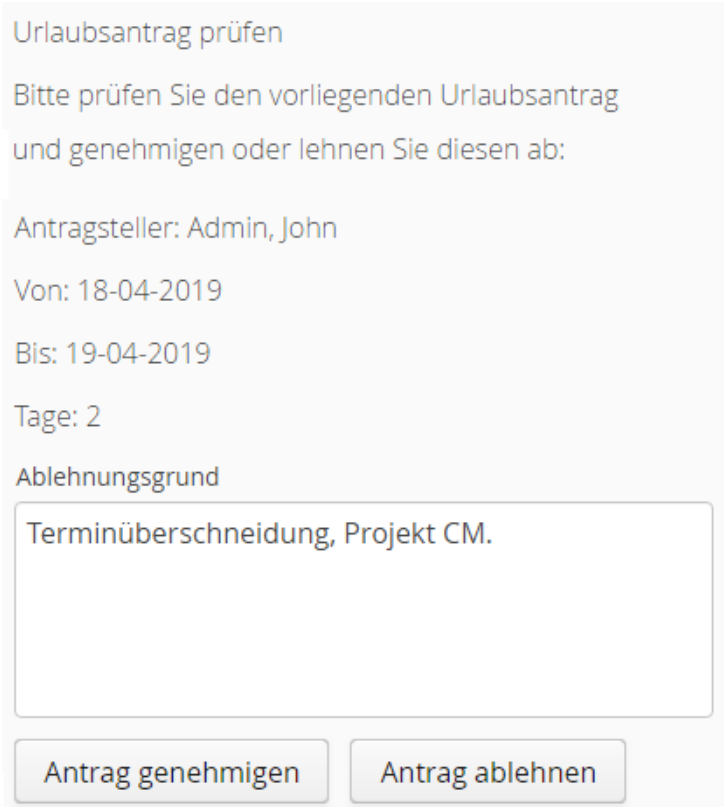

Abbildung 59: Maske für HumanTask "Antrag prüfen"

5.4.4 Ausführung von ProcessTask-Implementierungen

Abbildung 60 zeigt zwei Log-Einträge der in Abhängigkeit der getroffenen Entscheidung angestoßenen `ProcessTask`-Implementierungen: Der obere zeigt, wie der Antragsteller über seinen abgelehnten Antrag mit dem zuvor eingetragenen Ablehnungsgrund informiert wurde. Zu sehen ist, dass dies durch die Klasse `AntragstellerInformierenProcess` aus dem Paket `urlaubsantrag.impl.processtask` geschah.

```
Apr 18, 2019 6:17:14 PM urlaubsantrag.impl.processtask.AntragstellerInformierenProcess startProcess
INFORMATION: Der Antragsteller mit der ID 2 wurde über seinen abgelehnten Urlaub informiert.
             Der Ablehnungsgrund lautet:Terminüberschneidung, Projekt CM.

Apr 18, 2019 6:21:38 PM urlaubsantrag.impl.processtask.UrlaubskontoAktualisierenProcess startProcess
INFORMATION: Der Urlaubsantragvon Antragsteller mit der ID 2 vom 2019-04-18 bis zum 2019-04-19
             wurde genehmigt. Es verbleiben 28 Tage.
```

Abbildung 60: Log-Einträge der ProcessTask-Implementierungen

Der untere Log-Eintrag hingegen zeigt die Ausführung der Klasse `UrlaubskontoAktualisierenProcess`; der Antrag wurde also genehmigt. Beide `ProcessTask`-Implementierungen gebrauchen einen `CaseFileService` zum Laden der entsprechenden Daten. Der Kontext wird durch die in der `Task`-Instanz gespeicherten Referenz zum `CaseModel` hergestellt.

Ein abschließender Blick in die Datenbank, wie in Abbildung 61 dargestellt, zeigt die aktualisierten Tage des Urlaubskontos für „John Admin“: Der Anfangswert von 30 Tagen wurde um 2 Tage reduziert und weist nun 28 Tage aus.

```
SELECT cw.ID, cw.FIRSTNAME, cw.LASTNAME, uk.URLAUBSTAGE
FROM caseworker cw, urlaubskonto uk
WHERE cw.ID = uk.ANTRAGSTELLERID AND cw.ID = 2
```

ID	FIRSTNAME	LASTNAME	URLAUBSTAGE
2	John	Admin	28

Abbildung 61: Blick auf das aktualisierte Urlaubskonto

5.4.5 Blick auf die Elemente der Instanz mit Hilfe einer REST-Schnittstelle

Abbildung 62 zeigt die Antwort einer REST-Schnittstelle zum Abfragen der Elemente in einer Fall-Instanz. Zu sehen ist eine gekürzte Response im JSON-Format, welche Informationen über die `HumanTasks` und `ProcessTasks` enthält. In diesem Falle wurde der Antrag genehmigt, was sich am Zustand „COMPLETED“ beider `ProcessTasks` in den Zeilen 2-13 zeigt.

Der Pfad für die genutzte REST-Schnittstelle, auf die durch einen GET-Request zugegriffen wurde, lautete in diesem Beispiel `Urlaubsantrag/rest/cases/10/elements`.

```
1  [
2    {
3      "cmId": "antragstellerInformieren",
4      "id": 19,
5      "name": "Antragsteller Informieren",
6      "state": "COMPLETED"
7    },
8    {
9      "cmId": "urlaubskontoAktualisieren",
10     "id": 20,
11     "name": "Urlaubskonto aktualisieren",
12     "state": "COMPLETED"
13   },
14   {
15     "cmId": "antragPruefen",
16     "id": 19,
17     "name": "Urlaubsantrag prüfen",
18     "state": "COMPLETED"
19   },
20   {
21     "cmId": "antragsdatenErfassen",
22     "id": 20,
23     "name": "Daten für Urlaubsantrag erfassen",
24     "state": "COMPLETED"
25   }
26 ]
```

Abbildung 62: Elemente einer Fall-Instanz im JSON-Format

6. Kritische Betrachtung

In diesem Kapitel soll das Framework kritisch betrachtet werden. Ausgehend von dem mit Hilfe des Frameworks erstellten Fallbeispiel wird zunächst eine Aufwandsschätzung aufgezeigt. Sie soll dazu dienen, den Entwicklungsaufwand für Case Management Lösungen auf Basis des Frameworks abschätzen zu können. Zur Analyse des Fallbeispiels und des Frameworks wurde das Eclipse-Plugin „STAN - Structure Analysis for Java“ genutzt.[27] Klassenrümpfe oder Methodensignaturen werden in der Zählung nicht mit einbezogen, sondern nur die individuell benötigten Zeilen ohne Leerzeilen gezählt. Anschließend werden Vorteile und Nachteile des Frameworks aufgeführt. Abschließend wird das Framework der camunda Plattform als Alternative zum Framework gegenübergestellt.

6.1 Aufwandsschätzung für Entwicklungen mit dem Framework

Das Fallbeispiel im vorherigen Kapitel hat grundlegend aufgezeigt, welche individuellen Implementierungen nötig sind, um eine Case Management Anwendung auf Basis des Framework-Prototyp zu realisieren.

Zu den individuell zu entwickelnden Teilen einer CM Anwendung gehören neben der zum Zeitpunkt noch händischen Erstellung einer Fall-Blaupause die Implementierung anwendungsspezifischer Logik für die Verarbeitung verschiedener Elemente, wie etwa für `ProcessTask`s. Daneben müssen grafische Benutzerschnittstellen erstellt werden, insbesondere für die Arbeit an `HumanTask`s.

Für eine Aufwandsschätzung werden die „Lines of Code“ (LoC) herangezogen, also die Anzahl der benötigten Zeilen Quellcode, um beispielsweise eine Fall-Blaupause mit Hilfe des Frameworks zu erstellen. Der Quellcode für die Konfigurationen in den entsprechenden Factory-Klassen wird nicht mit einbezogen. Dieser umfasst üblicherweise circa zwei Zeilen.

Artefakt	**LoC**
Fall-Blaupause „Urlaubsantrag“ mit einem `CaseFileItem` und 8 `SimpleProperty`s	35
Einfache Regel-Implementierung für einen *Required Rule decorator*	5
Zwei einfache ProcessTask-Implementierungen	70
Eine IfPart-Implementierung auf Basis des CaseFileItem der Fall-Blaupause	7
Gesamt	**117**

Tabelle 3: Lines of Code für das Fallbeispiel (ohne Masken)

Eine Analyse des Quellcodes des Fallbeispiels wird in Tabelle 3 aufgeführt. Nicht aufgeführt sind LoCs für die erstellten Benutzermasken der beiden `HumanTask`s des Fallbeispiels. Diese enthalten viel Standardcode, der nicht direkt die Implementierung der eigentlichen Logik und Masken betrifft. Im

[27] „STAN“ kann über den Eclipse-Marketplace unter https://marketplace.eclipse.org/content/stan-structure-analysis-java bezogen werden.

Durschnitt betragen die LoC für die erstellten Masken jeweils circa 115 Zeilen. Die reine Logik für den Abschluss einer Maske mit Hilfe der Framework-Services beträgt beispielsweise circa 20 Zeilen.

Je nach Komplexität und Interaktion mit externen Systemen können sich die für eine Anwendung benötigten LoC erhöhen. Es lässt sich aber in etwa abschätzen, wie viele LoC für ein Modell mit einfacher Logik nötig sind. Tabelle 4 führt Schätzwerte für die verschiedenen Elemente des Frameworks auf. Diese sind bis auf das `CaseTask`-Element aus dem Fallbeispiel abgeleitet worden. Die zusätzlichen Services zur Bearbeitung von Urlaubskonten werden hier nicht mitbetrachtet, da solche Erweiterungen in ihrer Komplexität nicht abschätzbar und im Beispiel sehr einfach gehalten sind.

Framework-Element	**LoC**
`Task`-Definition	1
`ProcessTask`-Implementierung (einfache Verarbeitung eines `CaseFileItem`)	35
`CaseTask`-Implementierung[28]	10
`HumanTask`-Maske (auf Basis von Vaadin)	115
Regel-Definition für einen *decorator*	1
Regel-Implementierung für einen *decorator*	5
`Sentry`-Definition mit einem `OnPart` und einem `IfPart`	3
`IfPart`-Implementierung (einfache Prüfung eines `SimpleProperty`s)	7
`SimpleProperty` eines `CaseFileItem`s	1

Tabelle 4: Schätzwerte für LoC der Framework-Elemente

Es zeigt sich, dass der größte Aufwand für die Erstellung der grafischen Benutzerschnittstellen etwaiger `HumanTask`s benötigt wird. Auch `ProcessTask`- und `CaseTask`-Implementierungen erfordern je nach Komplexität 10-35 oder mehr Zeilen Code. LoC, um Elemente in einer Fall-Blaupause zu definieren, belaufen sich auf überschaubare 1-3 Zeilen und steigen in Abhängigkeit ihrer Anzahl linear an. Es ist denkbar, dass die Erstellung von Fall-Blaupausen zukünftig durch eingelesene CMMN-Modelldateien erfolgen kann. Ausgehend von den erstellten `CaseModel`-Strukturen können weitergehend anschließend die Klassen- und Methodenrümpfe der individuell zu implementierenden Komponenten erzeugt werden.

6.2 Vorteile und Nachteile des erstellten Frameworks

Der größte Vorteil des Frameworks ist die starke Orientierung an CMMN, insbesondere die Umsetzung der zugrundeliegenden Ausführungssemantik und Strukturen der (grafischen) Elemente. So sind die Elemente zur Umsetzung von CMMN-Anwendungen gegeben und können in verschiedenen Fall-

[28] CaseTasks werden im Fallbeispiel nicht eingesetzt, aber vom Framework-Prototyp unterstützt. Der Schätzwert wurde aus einem anderen Fallbeispiel abgeleitet.

Instanz wiederverwendet werden. Einmal erstellte Strukturen können beliebig oft instanziiert und in verschiedenen Kontexten bearbeitet werden.

Ein weiterer Vorteil ist die überschaubare Architektur des Frameworks und Anzahl der erstellten Klassen. Der Framework-Prototyp besteht aus rund 140 Klassen und knapp 5.800 Zeilen Quellcode. Praktisch entwickelt wird aber auf Basis von circa 30 Klassen und 6 Services. Die Umsetzung der Ausführungssemantik durch angepasste, etablierte Entwurfsmuster bietet einen guten Ausgangspunkt für weitere Entwicklungen des Frameworks, um beispielsweise im Prototyp nicht unterstützte Elemente wie `DecisionTask`s zu implementieren.

Zu den Nachteilen gehört, dass die Implementierung vereinfachte CMMN-Strukturen nutzt: Da keine spezifikationskonforme Umsetzung erfolgt ist, können CMMN-Modelle nicht in die vorgesehenen Strukturen gebracht werden, was beispielsweise ein maschinelles Lesen und Verarbeiten des Austauschformats erschwert. Zusätzliche Technologien, wie etwa der Einsatz einer Ausdruckssprache zur Auswertung von `IfPart`- oder Regel-Bedingungen, können hier Ansatzpunkte für eine konforme und vereinfachte Verarbeitung bieten.

Auch bietet der Prototyp keine ausgeprägten grafischen Benutzerschnittstellen, um beispielsweise `CaseWorker` und ihre Berechtigungen zu verwalten, oder weitere Kontextinformationen anzuzeigen. Standardschnittstellen zu anderen Systemen sind bisher nicht konzipiert oder implementiert worden, sind aber durchaus notwendig, um mit anderen (Standard-)Systemen interagieren zu können.

Ein den Prototyp betreffender Nachteil ist seine Komplexität hinsichtlich der technologischen Basis: Der JEE-Standard ist sehr umfangreich und benötigt Erfahrung in dessen Anwendung. Das Zusammenspiel mit einem JEE-Applikationsservern erhöht diese Komplexität nochmals; während der Entwicklung des Prototyps traten verschiedene Konfigurationsprobleme auf, deren Behebung viel Zeit beanspruchte.

Bezüglich der Wahl eines Applikationsservers ergibt sich außerdem doch eine gewisse Herstellerabhängigkeit, da der JEE-Standard manche Aspekte nicht unterstützt, beziehungsweise aufwendige Eigenentwicklungen nötig macht – sollte man keine vom Standard abweichende Referenzimplementierung nutzen. Ein Beispiel hierfür ist die Verarbeitung von HTTP-Requests über REST-Schnittstellen. Referenzimplementierungen vereinfachen die Arbeit mit Parametern, benötigen dafür aber die Entwicklung gegen anbieterspezifische APIs. Ein weiteres Beispiel in diesem Rahmen ist die Verarbeitung von Entitätsklassen und Transformation in JSON-Formate. Auch hier ist der Standard eingeschränkt, sodass gegen anbieterspezifische und vom Applikationsserver verwendete APIs entwickelt werden muss.

6.3 Camunda im Kurzvergleich

Abschließend soll in diesem Abschnitt zum Vergleich mit dem Framework der durch camunda verfolgte Ansatz kurz aufgezeigt werden.

Camunda hat sich als Anbieter einer open source BPM-Plattform mit Fokus auf die Unterstützung von BPMN 2.0 etabliert und unterstützt die Ausführung von CMMN 1.1-Modellen, wenn auch nur eingeschränkt[29]. Camunda setzt hierzu den größten Teil der Spezifikation und Ausführungssemantik um, mit Ausnahme von `ManualTask`s (dies ist nachvollziehbar, da solcher an sich in der Ausführung keine Funktion hat und nicht blockiert), *discretionary items*, *plan fragments* und *listeners* (dies ist überraschend, da die Verarbeitung von *listeners* durch die Implementierung von BPMN 2.0 erprobt sein sollte). Die Ausgestaltung etwaiger `CaseFileItem`s wird nicht unterstützt.

Es entsteht der Eindruck, als würde sich die Implementierung stark an der bereits umgesetzten Implementierung von BPMN 2.0 orientieren und intern viele Konzepte dieser Implementierung wiederverwenden. So ist es nicht überraschend, dass sehr flexible und nicht in BPMN vorhandene Konstrukte im Rahmen des *planning tables*, also *discretionary items*, bisher nicht umgesetzt sind.

Das in der Implementierung verwendete Klassenmodell basiert auf der CMMN-Spezifikation und den dargestellten Modellen und Klassendefinitionen[30]. Camunda nutzt diese, um CMMN-Dateien im XML-Format einzulesen und hieraus Fall-Instanzen zu generieren. Die Bearbeitung der Fall-Aufgaben erfolgt wie die Bearbeitung von Aufgaben eines BPMN-Modells, das durch camunda ausgeführt wird.

Vorteile	Nachteile
Strikte Orientierung an CMMN 1.1	Undurchsichtige Architektur
Funktion zum Einlesen von CMMN Modellen	Komplexe Implementierung
Ausführung der Modelle durch die Engine und Bearbeitung der Aufgaben durch mitgelieferte Web-Apps wie „Task List“	Bindung an die camunda-Plattform, insbesondere durch plattformabhängige Erweiterungen

Tabelle 5: Vor- und Nachteile der camunda Plattform

Tabelle 5 zeigt Vor- und Nachteile von camunda. Neben den Vorzügen, wie das standardisierte Austauschformat der CMMN-Modelle im XML-Format (bis auf wenige camunda-spezifische Erweiterungen) ausführen zu können, sowie mitgelieferter Verwaltungsfunktionen und Übersichtsmasken, gibt es auch Nachteile.

[29] Siehe https://docs.camunda.org/manual/7.7/reference/cmmn11/

[30] Siehe https://github.com/camunda/camunda-cmmn-model/tree/master/src/main/java/org/camunda/bpm/model/cmmn/impl/instance für die konkreten Implementierungen.

Die Architektur ist insgesamt mit über 10.000 Klassen sehr umfangreich und basiert auf verschiedenen (Web-)Technologien, wie etwa auch der Java Enterprise Edition. Um beispielsweise auf Basis von camunda ein Framework zu erstellen, müssten die CMMN-Spezifikation, die camunda-Implementierung und technische Basis, sowie die Einbettung in das camunda-Umfeld und die BPM-Plattform verstanden werden. Die Implementierung ist auf Grund einer strikten Orientierung an der CMMN-Spezifikation sehr komplex und ohne tiefergehende Studien nicht nachvollziehbar. Demgegenüber ist das in dieser Arbeit erstellte Framework mit rund 140 Klassen sehr schlank gehalten.

Camunda stellt in der kostenfreien Version rudimentäre grafische Übersichten zur Verwaltung von Fall-Instanzen bereit, um beispielsweise Aufgaben anzuzeigen und zu starten. Für Aufgaben, die durch eine grafische Schnittstelle unterstützt werden sollen, müssen wie im Framework dieser Arbeit entsprechende Masken erstellt werden. Die Logik für CMMN-Elemente, wie etwa die eines eines `CaseTask`s oder `ProcessTask`s, muss implementiert und anschließend durch camunda-spezifischen Erweiterungen des CMMN-Modells konfiguriert werden. Im Gegensatz zum Framework können für `IfPart`s Ausdrücke mit Hilfe einer Expression Language direkt ausgewertet werden.

Für die reine Benutzung von camunda ist ein nachteiliger Punkt die Bindung an die camunda-Plattform. Zwar ist camunda als Open Source Produkt konzipiert, aber das Geschäftsmodell liegt im professionellen Support durch camunda, das den gesamten BPM-Zyklus von der Modellierung bis hin zur Ausführung abdeckt. Eine Herstellerabhängigkeit erzeugt camunda spätestens durch die verschiedenen Erweiterungen von CMMN-Modellen, um beispielsweise Datenfelder oder Klassen zu definieren, die durch camunda zur Laufzeit ausgeführt werden sollen. Diese modifizierten Modelle können nicht auf die gleiche Weise durch andere Plattformen verarbeitet werden, da sie außerhalb der Spezifikation liegen.

6.4 Ausgewählte Forschungsimplementierungen im Vergleich

Verglichen mit dem Framework bieten die in Abschnitt 2.3.3 aufgeführten CM-Implementierungen bis auf „Connecare“ keine Unterstützung von CMMN. Die Implementierungen von „Connecare“ und „PHILHARMONICS Workflows“ sind nicht öffentlich verfügbar.

„Chimera“ rückt Daten in den Vordergrund von Modell-Fragmenten, die auf der BPMN basieren. Im Gegensatz zum Framework werden keine CMMN-Elemente unterstützt, womit die in CMMN spezifizierte Ausführungssemantik und damit verbundene Flexibilität und CM-spezifische Elemente wie etwa `Milestone`s entfallen. Dafür sollen die verschiedenen BPMN-Ereignistypen unterstützt werden können. „Barcelona“ setzt den GSM-Ansatz um (vgl. Abschnitt 2.3.1). Im Gegensatz zur CMMN-Ausführungssemantik wird eine OCL[31] auf definierte Daten angewendet. Es bietet keine spezialisierten Tasks. Es stehen lediglich Stages zur Verfügung, die atomare Aktivitäten darstellen können, oder als Container-Element für weitere (atomare) Stages genutzt werden können. Stages sind fest mit `Sentry`s („Guards“ in GSM) verknüpft. Als eine der Grundlagen von CMMN ähnelt es dem Kern des Frameworks, ist aber komplexer in der Anwendung. Auch verhält es sich in der Ausführung anders und bietet im Kern nur drei Elemente zur Modellierung von CM-Anwendungen an.

[31] Für die angewendete „Object Constraint Language“, siehe https://www.omg.org/spec/OCL/About-OCL/.

7. Zusammenfassung und Ausblick

Case Management Applikationen zu entwerfen und von Grund auf zu entwickeln ist eine herausfordernde und komplexe Aufgabe. In dieser Arbeit wurde zu diesem Zweck ein Framework konzipiert, welches eine systematische, schnelle und leichtgewichtige Entwicklung von Case Management Lösungen ermöglicht. Entwickler konzentrieren sich primär auf die Geschäftslogik und die Erstellung von unterstützenden grafischen Benutzerschnittstellen einer Anwendung, während das Framework die korrekte Ausführung dieser steuert. Das Framework baut hierzu auf gängigen Case Management Charakteristika auf, die sich in der Notationssprache CMMN ausdrücken.

Vereinfachte CMMN-Strukturen bilden die Basis für das Framework, behalten aber die spezifizierte Ausführungssemantik von CMMN bei. So können CMMN-Modelle als standardisierte Grundlage der Entwicklung genutzt werden. Das Ausführungsverhalten einer Anwendung entspricht dem der CMMN Spezifikation, welches flexible Prozessabläufe ermöglicht.

Ausgehend von verschiedenen aufgestellten Anforderungen ist ein Konzept entwickelt worden, das eine mehrschichtige Architektur und in diese eingebetteten Komponenten einer Case Management Applikation vorsieht. Das Framework zeigt auf, wie eine Case Management Applikation strukturiert entwickelt werden kann. Der Prototyp stellt diese Strukturen bereits größtenteils bereit, um leichtgewichtige Applikationen zu erstellen. Kernklassen und Services sind ausgiebig getestet worden, weisen aber sicherlich noch Fehler auf. Insbesondere das Testen komplexer Modelle mit allen möglichen und dynamischen Ausführungspfaden ist sehr aufwendig und schwierig.

Der Entwurf der umzusetzenden CMMN Ausführungssemantik baut auf adaptierten Mustern des Software Engineerings auf. Etablierte Muster aus dem Software Engineering wurden für die Verwendung im Framework angepasst und schaffen eine solide und ausbaufähige Grundlage für weitere Entwicklungen.

Um Herstellerabhängigkeiten zu vermeiden, wird der offene Java Enterprise Edition Standard genutzt: Er ermöglicht, aus verschiedenen Angeboten an Applikationsservern zu wählen, die mitunter kostenlos sind und verschiedene Datenbankmanagementsysteme unterstützen. Nachteilig ist die Wahl der Programmiersprache; die Ausführungen in dieser Arbeit hinsichtlich des Konzepts und insbesondere der Implementierung können aber in andere Sprachen übertragen werden, wie etwa C#. Ein nicht zu unterschätzender Aspekt ist die technologische Komplexität des JEE Standards.

Das Framework unterstützt Entwickler durch vorgefertigte Komponenten, Strukturen und bereitgestellte Services bei der Entwicklung einer Case Management Lösung: Im Kern der Applikation werden wiederverwendbare Basisbausteine benutzt, die CMMN-Elemente beziehungsweise CMMN-Modelle repräsentieren. Diese werden zu sogenannten Fall-Blaupausen zusammengestellt. Erstellte Fall-Blaupausen werden durch anwendungsspezifische Logik in Form individueller Implementierungen auf Basis abstrakter Klassen des Frameworks ergänzt, um Geschäftslogik und Regeln einzubinden. Diese können ebenso wiederverwendet werden. Die vorgefertigten Fall-Blaupausen können anschließend mit Hilfe des Frameworks und der enthaltenen Services instanziiert und bearbeitet werden.

Daten rücken ins Zentrum der Anwendung und ihrer Ausführungssemantik. Sie können entsprechend der CMMN Spezifikation den Ablauf einer Fall-Instanz beeinflussen. Zur Laufzeit einer Fall-Instanz werden die Framework-Bausteine und individuell entwickelten Teile im korrekten Kontext verknüpft und verhalten sich entsprechend der CMMN Spezifikation. Fallbearbeiter können den Ablauf einer Fall-Instanz beeinflussen. Sei es durch die bewusste Auswahl von zu erledigenden Aufgaben oder durch die Manipulation von Fall-Daten: Beides kann Auswirkungen auf andere Elemente nach sich ziehen. So werden dynamische Abläufe im jeweiligen Kontext ermöglicht.

Das Konzept sieht vor, dass Fallbearbeiter durch grafische Benutzerschnittstellen unterstützt werden, um durch sie ausgewählte Arbeiten auszuführen oder Daten zu manipulieren. Die prototypische Implementierung zeigt auf, wie hierzu eine Präsentationsschicht mit Hilfe eines Java Webframework implementiert werden kann. Der Kontext der jeweiligen Fall-Instanz, beziehungsweise der in der Instanz enthaltenen Aufgabe, kann durch die Framework-Bausteine und die enthaltenen Services hergestellt werden. Der Prototyp bietet Fallbearbeitern rudimentäre Übersichtsmasken zur Verwaltung von Fall-Instanzen und in diesen enthaltenen Aufgaben an. Erstellte Masken können direkten Gebrauch der Services machen.

Ein erstelltes Fallbeispiel validiert den Einsatz des Frameworks. Ausgehend von einer Analyse des erstellten CMMN-Models wird gezeigt, wie eine Fall-Blaupause und nötige Individualimplementierungen erstellt und mit dem Framework zur Verarbeitung verknüpft werden. Anschließend wird die Bearbeitung durch zugewiesene Fallbearbeiter in verschiedenen Rollen gezeigt.

Aus dem Beispiel abgeleitete Schätzwerte für benötigte Zeilen Quellcode zeigen, dass bereits mit relativ wenig Aufwand eine (einfache) Case Management Anwendungen erstellt werden kann. Der größte Aufwand fällt für die Erstellung grafischer Benutzerschnittstellen und die Implementierung der Geschäftslogik an. Somit können sich Entwickler auf diese Punkte konzentrieren und die korrekte Ausführung dem Framework überlassen.

Auch wenn der Prototyp bereits viele Funktionen bereitstellt, konnten mehrere Teile nur ansatzweise oder auf Grund der Komplexität nicht implementiert werden. Hierzu zählen die REST-Schnittstellen des Prototyps und der bisher ausgeklammerte Planungsmechanismus von CMMN. Die unausgereiften REST-Schnittstellen kapseln derzeit lediglich die Funktionen des `CaseService` und `CaseFileService`. Diese gilt es weiterzuentwickeln, um beispielsweise eine umfängliche Kommunikation mit externen Systemen zu ermöglichen, wie etwa einer Applikation auf Basis eines JavaScript-Frameworks zur Implementierung von grafischen Benutzerschnittstellen.

Wie der im Prototyp nicht unterstützte, sehr komplexe Planungsmechanismus von CMMN in das Framework eingebettet werden kann und ob die vereinfachten Strukturen des Frameworks ausreichen, muss tiefergehend geprüft werden. Auch die im Prototyp nicht unterstützten Elemente DecisionTask und Spezialisierungen des EventListeners, welche weitere Technologien benötigen, sollten konzipiert und in das Framework implementiert werden.

Ein weiterer Punkt ist, Sicherheitsvorkehrungen zu implementieren, um den Zugang zu einer mit dem Framework erstellten Applikation und Kommunikation zwischen dieser und externen Systemen

abzusichern. Außerdem sollte das entwickelte Rollensystem geprüft werden, ob dies ausreichend ist, oder erweitert werden sollte, um feingranulare Berechtigungen konfigurieren zu können.

Es ist geplant, das Framework einem breiteren Publikum zur Verfügung zu stellen, um dieses weiter zu entwickeln. Wünschenswert wäre hierbei, den Einsatz in einem Unternehmen zu testen, um Unzulänglichkeiten und Schwachpunkte des Konzepts und der Implementierung aufzuzeigen.

Literaturverzeichnis

[1] North, K, Güldenberg, S (2008): Produktive Wissensarbeit(er). Antworten auf die Management-Herausforderung des 21. Jahrhunderts: Performance messen Produktivität steigern Wissensarbeiter entwickeln. 1. Auflage. Gabler Verlag.

[2] Buck-Emden, R, Alda, S (2017): Systemunterstützung für wissensintensive Geschäftsprozesse – Konzepte und Implementierungsansätze. In: Barton, T, Müller, C, Seel, C (Hrsg), *Geschäftsprozesse.* Springer Fachmedien Wiesbaden, Wiesbaden.

[3] Marin, MA, Hauder, M, Matthes, F (2016): Case Management: An Evaluation of Existing Approaches for Knowledge-Intensive Processes. In: Reichert, M, Reijers, HA (Hrsg), *Business Process Management Workshops. BPM 2015, 13th International Workshops, Innsbruck, Austria, August 31 - September 3, 2015, Revised Papers.* Springer International Publishing, Cham.

[4] Henk de Man: Case Management: A Review of Modeling Approaches. https://www.bptrends.com/case-management-a-review-of-modeling-approaches/. Abgerufen am 01.04.2019.

[5] Michael White (2009): Case Management: Combining Knowledge with Process, bptrends.com.

[6] Di Ciccio, C, Marrella, A, Russo, A (2015): Knowledge-Intensive Processes. Characteristics, Requirements and Analysis of Contemporary Approaches. Journal on Data Semantics, 4(1):29–57.

[7] Palmer, N (2011): BPM and ACM. In: Fischer, L, Koulopoulos, T (Hrsg), *Taming the unpredictable. Real world adaptive case management: case studies and practical guidance.* Future Strategies, Lighthouse Point, Fla.

[8] Reibnitz, C von (2015): Case Management. Praktisch und effizient. Springer Berlin Heidelberg, Berlin, Heidelberg.

[9] Swenson, KD (2013): State of the Art In Case Management, fujitsu.com.

[10] Fischer, L (Hrsg) (2012): How knowledge workers get things done. Real-world adaptive case management. Future Strategies, Lighthouse Point, Fla.

[11] Rooze, EJ, Paapst, M, Sombekke, J (2007): eCase Management. An international study in judicial organisations., https://www.rechtspraak.nl/.

[12] Schuerman, D, Schwarz, K, Williams, B (2014): Dynamic Case Management for Dummies. John Wiley & Sons, Inc., Hoboken, New Jersey.

[13] Tran, TTK, Pucher, MJ, Mendling, J, Ruhsam, C (2013): Setup and Maintenance Factors of ACM Systems. In: Hutchison, D, Kanade, T, Kittler, J, Kleinberg, JM, Mattern, F, Mitchell, JC, Naor, M, Nierstrasz, O, Pandu Rangan, C, Steffen, B, Sudan, M, Terzopoulos, D, Tygar, D, Vardi, MY, Weikum, G, Demey, YT, Panetto, H (Hrsg), *On the Move to Meaningful Internet Systems: OTM 2013 Workshops.* Springer Berlin Heidelberg, Berlin, Heidelberg.

[14] Motahari-Nezhad, HR, Swenson, KD (2013): Adaptive Case Management. Overview and Research Challenges. In: IEEE Computer Society (Hrsg), *2013 IEEE 15th Conference on Business Informatics.* IEEE.

[15] Herrmann, C, Kurz, M (2011): Adaptive Case Management: Supporting Knowledge Intensive Processes with IT Systems. In: Schmidt, W (Hrsg), *S-BPM ONE - Learning by Doing - Doing by Learning. Third International Conference, S-BPM ONE 2011, Ingolstadt, Germany, September 29 - 30, 2011. Proceedings.* Springer-Verlag GmbH Berlin Heidelberg, Berlin, Heidelberg.

[16] Meyer, A, Herzberg, N, Puhlmann, F, Weske, M (2014): Implementation Framework for Production Case Management. Modeling and Execution. In: IEEE Computer Society (Hrsg), *2014 IEEE 18th International Enterprise Distributed Object Computing Conference.* IEEE.

[17] Brinkmann, V (2010): Case Management. Gabler, Wiesbaden.

[18] Rausch, A, Zhang, H, Richardson, I, Münch, J, Kuhrmann, M (Hrsg) (2016): Managing software process evolution. Traditional, agile and beyond - how to handle process change. Springer, Switzerland.

[19] Hauder, M, Munch, D, Michel, F, Utz, A, Matthes, F (2014): Examining Adaptive Case Management to Support Processes for Enterprise Architecture Management. In: IEEE Computer Society (Hrsg), *2014 IEEE 18th International Enterprise Distributed Object Computing Conference Workshops and Demonstrations.* IEEE.

[20] Y., A, A., M (2017): Adaptive Case Management Framework to Develop Case-based Emergency Response System. International Journal of Advanced Computer Science and Applications, 8(4).

[21] Bodendorf, F (2014): Constructivistic and connectivistic e-learning by collaborative case management. In: IEEE Computer Society (Hrsg), *2014 International Conference on Web and Open Access to Learning (ICWOAL).* IEEE.

[22] Sem, HF, Carlsen, S, Coll, GJ (2013): On Two Approaches to ACM. In: van der Aalst, W, Mylopoulos, J, Rosemann, M, Shaw, MJ, Szyperski, C, La Rosa, M, Soffer, P (Hrsg), *Business Process Management Workshops.* Springer Berlin Heidelberg, Berlin, Heidelberg.

[23] Fischer, L, Koulopoulos, T (Hrsg) (2011): Taming the unpredictable. Real world adaptive case management: case studies and practical guidance. Future Strategies, Lighthouse Point, Fla.

[24] Elliott, MS, King, JL (2005): A Common Information Space in Criminal Courts. Computer-Supported Cooperative Work (CSCW) Case Management Systems. In: IEEE Computer Society (Hrsg), *Proceedings of the 38th Annual Hawaii International Conference on System Sciences.* IEEE.

[25] van der Aalst, WMP, Weske, M, Grünbauer, D (2005): Case handling. A new paradigm for business process support. Data & Knowledge Engineering, 53(2):129–162.

[26] Object Management Group (OMG) (2011): Business Process Modeling Notation (BPMN). http://www.omg.org/spec/BPMN/2.0/.

[27] Benner-Wickner, M, Book, M, Gruhn, V (2016): Adapting Case Management Techniques to Achieve Software Process Flexibility. In: Kuhrmann, M, Münch, J, Richardson, I, Rausch, A, Zhang, H (Hrsg), *Managing Software Process Evolution.* Springer International Publishing, Cham.

[28] van der Aalst, WMP, Berens, PJS (2001): Beyond workflow management. In: Ellis, C(S), Rodden, T, Zigurs, I (Hrsg), *Proceedings of the 2001 International ACM SIGGROUP Conference on Supporting Group Work - GROUP '01.* ACM Press, New York, New York, USA.

[29] van der Aalst, WMP, Pesic, M, Schonenberg, H (2009): Declarative workflows. Balancing between flexibility and support. Computer Science - Research and Development, 23(2):99–113.

[30] Object Management Group (OMG) (2016): Case Management Model and Notation (CMMN). http://www.omg.org/spec/CMMN/1.1/.

[31] Marin, MA (2016): Introduction to the Case Management Model and Notation (CMMN). https://arxiv.org/abs/1608.05011.

[32] Zensen, A, Küster, J (2018): A Comparison of Flexible BPMN and CMMN in Practice. A Case Study on Component Release Processes. In: IEEE Computer Society (Hrsg), *2018 IEEE 22nd International Enterprise Distributed Object Computing Conference (EDOC).* IEEE.

[33] Rychkova, I, Nurcan, S (2011): The Old Therapy for the New Problem. Declarative Configurable Process Specifications for the Adaptive Case Management Support. In: Zur Muehlen, M, Su, J (Hrsg), *Business Process Management Workshops: BPM 2010 International Workshops and Education Track, Hoboken, NJ, USA, September 13-15, 2010, Revised Selected Papers.* Springer Berlin Heidelberg, Berlin, Heidelberg.

[34] Hull, R, Damaggio, E, Fournier, F, Gupta, M, Heath, F, Hobson, S, Linehan, M, Maradugu, S, Nigam, A, Sukaviriya, P, Vaculin, R (2011): Introducing the Guard-Stage-Milestone Approach for Specifying Business Entity Lifecycles. In: Bravetti, M, Bultan, T (Hrsg), *Web Services and Formal Methods: 7th International Workshop, WS-FM 2010, Hoboken, NJ, USA, September 16-17, 2010. Revised Selected Papers.* Springer Berlin Heidelberg, Berlin, Heidelberg.

[35] Pesic, M, van der Aalst, WMP (2006): A Declarative Approach for Flexible Business Processes Management. In: Hutchison, D, Kanade, T, Kittler, J, Kleinberg, JM, Mattern, F, Mitchell, JC, Naor, M, Nierstrasz, O, Pandu Rangan, C, Steffen, B, Sudan, M, Terzopoulos, D, Tygar, D, Vardi, MY, Weikum, G, Eder, J, Dustdar, S (Hrsg), *Business Process Management Workshops.* Springer Berlin Heidelberg, Berlin, Heidelberg.

[36] Jalali, A, Bider, I (2014): Towards Aspect Oriented Adaptive Case Management. In: IEEE Computer Society (Hrsg), *2014 IEEE 18th International Enterprise Distributed Object Computing Conference Workshops and Demonstrations.* IEEE.

[37] Henriques, R, Rito Silva, A (2011): Object-Centered Process Modeling. Principles to Model Data-Intensive Systems. In: Zur Muehlen, M, Su, J (Hrsg), *Business Process Management Workshops:*

BPM 2010 International Workshops and Education Track, Hoboken, NJ, USA, September 13-15, 2010, Revised Selected Papers. Springer Berlin Heidelberg, Berlin, Heidelberg.

[38] Meyer, A, Weske, M (2014): Activity-Centric and Artifact-Centric Process Model Roundtrip. In: Lohmann, N, Song, M, Wohed, P (Hrsg), *Business Process Management Workshops.* Springer International Publishing, Cham.

[39] Cohn, D, Hull, R (2009): Business Artifacts. A Data-centric Approach to Modeling Business Operations and Processes. IEEE Data Eng. Bull., 32:3–9.

[40] Kirsch-Pinheiro, M, Rychkova, I (2013): Dynamic Context Modeling for Agile Case Management. In: Hutchison, D, Kanade, T, Kittler, J, Kleinberg, JM, Mattern, F, Mitchell, JC, Naor, M, Nierstrasz, O, Pandu Rangan, C, Steffen, B, Sudan, M, Terzopoulos, D, Tygar, D, Vardi, MY, Weikum, G, Demey, YT, Panetto, H (Hrsg), *On the Move to Meaningful Internet Systems: OTM 2013 Workshops.* Springer Berlin Heidelberg, Berlin, Heidelberg.

[41] Ben Said, I, Chaabane, MA, Andonoff, E, Bouaziz, R (2014): Context-Aware Adaptive Process Information Systems. The Context-BPMN4V Meta-Model. In: Manolopoulos, Y, Trajcevski, G, Kon-Popovska, M (Hrsg), *Advances in Databases and Information Systems.* Springer International Publishing, Cham.

[42] Beck, H, Hewelt, M, Pufahl, L (2017): Extending Fragment-Based Case Management with State Variables. In: Dumas, M, Fantinato, M (Hrsg), *Business Process Management Workshops: BPM 2016 International Workshops, Rio de Janeiro, Brazil, September 19, 2016, Revised Papers.* Springer International Publishing, Cham.

[43] Marcin Hewelt, Mathias Weske (2016): A Hybrid Approach for Flexible Case Modeling and Execution. In: La Rosa, M, Loos, P, Pastor, O (Hrsg), *Business Process Management Forum.* Springer International Publishing, Cham.

[44] Hildebrandt, T, Mukkamala, RR, Slaats, T (2011): Designing a Cross-Organizational Case Management System Using Dynamic Condition Response Graphs. In: IEEE Computer Society (Hrsg), *2011 IEEE 15th International Enterprise Distributed Object Computing Conference.* IEEE.

[45] Hildebrandt, T, Marquard, M, Mukkamala, RR, Slaats, T (2013): Dynamic Condition Response Graphs for Trustworthy Adaptive Case Management. In: Hutchison, D, Kanade, T, Kittler, J, Kleinberg, JM, Mattern, F, Mitchell, JC, Naor, M, Nierstrasz, O, Pandu Rangan, C, Steffen, B, Sudan, M, Terzopoulos, D, Tygar, D, Vardi, MY, Weikum, G, Demey, YT, Panetto, H (Hrsg), *On the Move to Meaningful Internet Systems: OTM 2013 Workshops.* Springer Berlin Heidelberg, Berlin, Heidelberg.

[46] Tran Thi Kim, T, Weiss, E, Adensamer, A, Ruhsam, C, Czepa, C, Tran, H, Zdun, U (2016): An Ontology-Based Approach for Defining Compliance Rules by Knowledge Workers in Adaptive Case Management - A Repair Service Management Case. In: IEEE Computer Society (Hrsg),

2016 IEEE 20th International Enterprise Distributed Object Computing Workshop (EDOCW). IEEE.

[47] Arnold, O, Laue, R (2014): Überblick über Ansätze zur Modellierung von Variabilität in Geschäftsprozessmodellen. HMD Praxis der Wirtschaftsinformatik, 51(5):627–642.

[48] Breitenmoser, R, Keller, T (2015): Case management model and notation - a showcase. European Scientific Journal, 11(25).

[49] Blaukopf, S, Mendling, J (2018): An Organizational Routines Perspective on Process Requirements. In: Teniente, E, Weidlich, M (Hrsg), *Business Process Management Workshops.* Springer International Publishing, Cham.

[50] Shahrah, AY, Al-Mashari, MA (2017): Modelling emergency response process using case management model and notation. IET Software, 11(6):301–308.

[51] Kirchner, K, Herzberg, N (2017): Ein CMMN-basierter Ansatz für Modellierung und Monitoring flexibler Prozesse am Beispiel von medizinischen Behandlungsabläufen. In: Barton, T, Müller, C, Seel, C (Hrsg), *Geschäftsprozesse.* Springer Fachmedien Wiesbaden, Wiesbaden.

[52] Hauder, M, Kazman, R, Matthes, F (2015): Empowering End-Users to Collaboratively Structure Processes for Knowledge Work. In: Abramowicz, W (Hrsg), *Business information systems. 18th International Conference, BIS 2015, Poznań, Poland.* Springer International Publishing, Cham.

[53] Marin, MA, Brown, JA (2015): Implementing a Case Management Modeling and Notation (CMMN) System using a Content Management Interoperability Services (CMIS) compliant repository. https://arxiv.org/pdf/1504.06778.pdf.

[54] Künzle V., RM (2011): A Modeling Paradigm for Integrating Processes and Data at the Micro Level. In: Halpin, T, Nurcan, S, Krogstie, J, Soffer, P, Proper, E, Schmidt, R, Bider, I (Hrsg), *Enterprise, Business-Process and Information Systems Modeling. 12th International Conference, BPMDS 2011, and 16th International Conference, EMMSAD 2011, held at CAiSE 2011, London, UK, June 20-21, 2011. Proceedings.* Springer-Verlag GmbH Berlin Heidelberg, Berlin, Heidelberg.

[55] Andrews, K, Steinau, S, Reichert, M (2015): A Runtime Environment for Object-Aware Processes. http://ceur-ws.org/Vol-1418/paper2.pdf.

[56] Heath, F, Boaz, D, Gupta, M, Vaculín, R, Sun, Y, Hull, R, Limonad, L (2013): Barcelona. A Design and Runtime Environment for Declarative Artifact-Centric BPM. In: Hutchison, D, Kanade, T, Kittler, J, Kleinberg, JM, Mattern, F, Mitchell, JC, Naor, M, Nierstrasz, O, Pandu Rangan, C, Steffen, B, Sudan, M, Terzopoulos, D, Tygar, D, Vardi, MY, Weikum, G, Basu, S, Pautasso, C, Zhang, L, Fu, X (Hrsg), *Service-Oriented Computing.* Springer Berlin Heidelberg, Berlin, Heidelberg.

[57] Michel, F, Matthes, F (2018): A Holistic Model-Based Adaptive Case Management Approach for Healthcare. In: IEEE Computer Society (Hrsg), *2018 IEEE 22nd International Enterprise Distributed Object Computing Conference workshops. EDOCW 2018.* IEEE, Piscataway, NJ.

[58] Michel, F, Hernandez-Mendez, A, Matthes, F (2018): An Overview of Tools for an Integrated and Adaptive Healthcare Approach. In: IEEE Computer Society (Hrsg), *2018 IEEE 22nd International Enterprise Distributed Object Computing Conference workshops. EDOCW 2018.* IEEE, Piscataway, NJ.

[59] Inden, M (2016): Der Java-Profi: Persistenzlösungen und REST-Services. Datenaustauschformate, Datenbankentwicklung und verteilte Anwendungen. 1. Auflage. dpunkt.Verlag, Heidelberg.

[60] Purushothaman, J: RESTful Java web services. Design scalable and robust RESTful web services with JAX-RS and Jersey extension APIs.

[61] Ihns, O, Heldt, SM, Koschek, H, Ehm, J (2011): EJB 3.1 professionell. Grundlagen- und Expertenwissen zu Enterprise JavaBeans 3.1 - inkl. JPA 2.0. dpunkt.Verlag, Heidelberg.

[62] Müller-Hofmann, F, Hiller, M, Wanner, G (2015): Programmierung von verteilten Systemen und Webanwendungen mit Java EE. Erste Schritte in der Java Enterprise Edition. Springer Vieweg, Wiesbaden.

[63] Schießer, M, Schmollinger, M (2015): Workshop Java EE 7. Ein praktischer Einstieg in die Java Enterprise Edition mit dem Web Profile. 2. Auflage. dpunkt, Heidelberg.

[64] Gamma, E, Riehle, D (2008): Entwurfsmuster. Elemente wiederverwendbarer objektorientierter Software. Addison-Wesley, München.

[65] Larman, C (2009): Applying UML and patterns. An introduction to object-oriented analysis and design and iterative development. 3. Auflage. Prentice Hall, Upper Saddle River, NJ.

Versicherung

„Ich versichere, dass ich die vorstehende Arbeit selbständig angefertigt und mich fremder Hilfe nicht bedient habe. Alle Stellen, die wörtlich oder sinngemäß veröffentlichtem oder nicht veröffentlichtem Schrifttum entnommen sind, habe ich als solche kenntlich gemacht.“

_____

André Zensen