ulm university universität
uulm

Fakultät für Ingenieurwissenschaften und Informatik
Institut für Datenbanken und Informationssysteme

Diplomarbeit

Implementierung einer Komponente zur Ausführung von Mikro-Prozessen in einem datenorientierten Prozess-Management-System

Stefan Schultz
31. März 2012

1. Gutachter: Prof. Dr. Manfred Reichert
2. Gutachter: Prof. Dr. Peter Dadam
Betreuerin: Vera Künzle

Danksagung

Ich danke allen, die mir bei der Erstellung dieser Diplomarbeit geholfen haben. Allen voran Vera Künzle, für ihren Rat und die konstruktive Kritik an meiner Arbeit. Des Weiteren danke ich Rüdiger Pryss für seine Hilfe, besonders in der letzten Phase. Außerdem danke ich meiner Familie für ihre Unterstützung. Ein besonderer Dank gilt meiner Freundin Anastasia für ihre Geduld und die Unterstützung während meines Studiums, insbesondere während meiner Diplomarbeit.

Inhaltsverzeichnis

1 Einführung

1.1 Motivation

Herkömmliche Prozess-Management-Systeme (PrMS) bieten Anwendern die Möglichkeit zur Modellierung, Ausführung und Überwachung von Geschäftsprozessen [1]. Die einzige Bedingung, die vor der Ausführung einer Aktivität erfüllt sein muss ist, dass die vorangegangene Aktivität ausgeführt worden ist oder nicht mehr aktiviert werden kann. Innerhalb von Aktivitäten kann auf beliebige Daten zugegriffen werden, dies geschieht jedoch ohne Einbezug des PrMS. Zusätzlich können innerhalb des Datenflusses nur atomare Datenelemente verwaltet werden. Diese Form der Modellierung führt insbesondere bei datenintensiven Prozesse bzw. Prozessen innerhalb datenbankbasierten Anwendungen, zu einer inadäquaten Abbildung von Geschäftsvorgängen auf Prozesse. Sie macht Prozesse unflexibel und sieht keine ausreichende Integration der Daten vor, was zu Ineffizienz bei der Ausführung oder fehlerhaften Ergebnissen führt [1]. In der Praxis kommt es daher oft vor, dass die Prozesslogik hart im Quellcode einer Anwendung verdrahtet ist. Dies führt zu einem unverhältnismäßig hohen Wartungsaufwand bei Änderungen und schränkt die Flexibilität weiter ein.

Ein datenorientiertes PrMS soll diese Unzulänglichkeiten umgehen. Es wird davon ausgegangen, dass ein Prozess nicht länger aktivitätsgetrieben sondern datengetrieben ausgeführt wird [1]. Der Fortschritt eines Prozesses definiert sich nicht über die tatsächlich ausgeführten Aktivitäten, sondern anhand der zur Verfügung stehenden Daten. Ein Benutzer ist bei der Eingabe von Daten zusätzlich nicht an den Fortschritt des Prozesses gebunden. Somit kann er dem Prozess bereits Daten zur Verfügung stellen, obwohl diese für die Ausführung des Prozesses noch nicht unbedingt benötigt werden.

Bestimmte Prozesse erfordern zusätzlich Entscheidungen des Benutzers um eine korrekte Ausführung zu gewährleisten. Diese Entscheidungen erfordern eine vom Prozessfortschritt unabhängige, integrierte Sicht auf prozessrelevante Daten zur Unterstützung des Benutzers. Um dies zu gewährleisten, werden Daten zum Einen nicht länger nur als atomare Datenelemente verwaltet und zum Anderen bei der Modellierung des Prozesses stärker integriert. Im folgenden Abschnitt wird das Konzept PHILharmonicFlows vorgestellt, welches den Ansatz eines datenorientierten PrMS verfolgt.

1.2 PHILharmonicFlows

PHILharmonicFlows ist ein Konzept für ein datenorientiertes PrMS welches zur Zeit am Institut für Datenbanken und Informationssysteme (DBIS) der Universität Ulm entwickelt wird. Es umfasst neben ausführlichen Konzepten zur Modellierung von datenorientierten Prozessen auch eine umfangreiche operationale Semantik für deren Ausführung [3, 4]. Beides soll anhand einer Implementierung eines entsprechenden Prototyps eva-

luiert werden. Dafür wurden zunächst zwei Usability-Konzepte vorgestellt, welche das Aussehen und die Funktionalität der Modellierungs- und der Ausführungskomponente beschreiben [5, 7]. Letzteres umfasst eine Menge von Anforderungen an die Laufzeitkomponente basierend auf dem fachlichen Konzept, Benutzerstudien und Designstandards für Webanwendungen. Der letzte Schritt ist nun die Implementierung eines Prototyps der beiden Komponenten auf Basis der genannten Arbeiten.

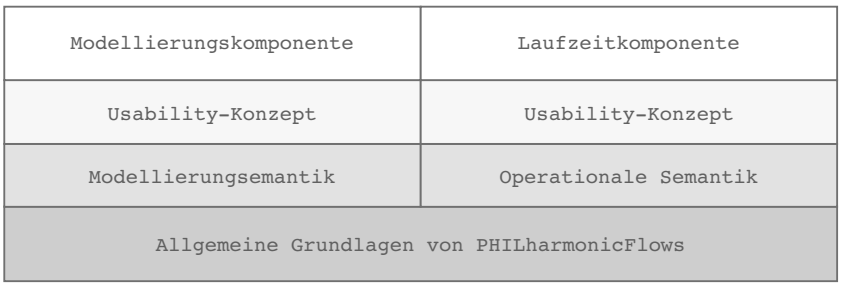

Abbildung 1: Vorgehen bei der Umsetzung des PHILharmonicFlows Konzepts

Das Framework bietet umfangreiche Möglichkeiten zur Modellierung von Daten und Prozessen. Es wird zunächst ein Datenmodell erstellt und darauf basierend die Prozesse modelliert. Des Weiteren können Benutzerrechte und Rollen für den Zugriff auf bestimmte Daten angelegt werden. Diese Informationen werden in einer speziellen Modellierungsdatenbank gespeichert und bilden einen Teil der Grundlage für die Ausführung. Den restlichen Teil liefert eine ausführliche operationale Semantik, die das Verhalten der einzelnen Prozesselemente bei der Ausführung beschreibt.

Während der Ausführung ist es dem Benutzer möglich, neue Prozesse zu starten, über eine Arbeitsliste die Zustände aller laufenden Prozesse zu überblicken und Aktivitäten auf ihnen auszuführen. Er hat außerdem die Möglichkeit die Daten der jeweiligen Prozesse einzusehen und zu verändern. Dies geschieht über automatisch erzeugte Formulare, die neben den Daten auch prozessspezifische Informationen anzeigen können, um den Benutzer bei der Eingabe zu unterstützen. Die Instanzdaten die zur Laufzeit anfallen werden ebenfalls in einer Datenbank gespeichert.

1.3 Aufgabenstellung

Ziel dieser Arbeit ist die Implementierung einer Komponente zur Ausführung von Mikro-Prozessen. Diese stellen einen Teil der Prozessstruktur von PHILharmonicFlows dar und werden im nächsten Kapitel genauer vorgestellt. Die Umsetzung baut auf dem fachlichem Konzept (insbesondere der operationalen Semantik) [3, 4] sowie auf dem Usability-Konzept [5] auf, welches die Oberfläche und die benötigte Funktionalität beschreibt, sowie gewisse Rahmenbedingungen und Anforderungen festlegt.

Neben einer performanten Ausführung soll die Komponente als Webanwendung realisiert werden. Dabei wird die Anwendung in einem Web-Browser ausgeführt. Diese Technik findet immer häufiger Anwendung, da sich eine Ausführung im Browser kaum noch von der Ausführung herkömmlicher Programme auf einem Computer unterscheidet. Zudem hat es den Vorteil, dass es auf beliebigen Architekturen ausgeführt werden kann, solange man einen Rechner mit Internetzugang besitzt und einen Browser installiert hat, welcher die verwendeten Techniken unterstützt.

1.4 Herausforderungen

Bei der Implementierung musste auf eine Vielzahl von Problemen eingegangen werden, die sich aus dem Konzept in Verbindung mit den Eigenschaften einer Webanwendungen ergeben haben. Die wichtigsten Punkte werden im folgenden Abschnitt erläutert.

1.4.1 Datenstruktur

Prozesse basieren auf beliebigen Datenstrukturen die anhand eines relationalen Metamodells definiert werden. Durch die unterschiedlichen Signaturen ist es daher nicht möglich, die anfallenden Daten unterschiedlicher Prozesse zur Laufzeit in derselben Tabelle abzulegen. Dies ist vergleichbar mit dem Versuch die Daten eines Webshops zusammen mit den Daten einer Bewerberverwaltung in einer Tabelle abzulegen, welche grundsätzlich verschieden sind. Des Weiteren muss beachtet werden, dass zur Laufzeit beliebig viele Instanzen eines Prozesses gestartet werden können. Bei einer normalisierten relationalen Datenbank, deren Daten über mehrere Tabellen verteilt liegen, würde dies zu hohen Einbußen bei der Performanz führen da ständig Anfragen über mehrere Tabellen erfolgen würden. Die Struktur der Laufzeitdatenbank spielt daher eine sehr große Rolle. Bei ihrer Erzeugung muss ein Kompromiss zwischen der Granularität der Daten, der Komplexität von Anfragen und dem Informationsgehalt des Ergebnisses gefunden werden.

Die generische Struktur der Daten führt zu einem weiteren Problem. Zur Laufzeit muss gewährleistet werden, dass alle vorhandenen und alle in Zukunft modellierten Prozesse ausgeführt werden können, egal wie die Daten aufgebaut sind. Das sorgt dafür, dass häufig auf das Metamodell zugegriffen werden muss um strukturelle Informationen abzufragen.

1.4.2 Eigenschaften von Webanwendungen

Neben den Herausforderungen, die sich durch die generische Struktur der Prozesse ergeben, gibt es auch allgemeinere Punkte die sich durch die Struktur einer Webanwendungen ergeben.

Die größte Herausforderung ist die Performanz des Systems. Eine Webanwendung stellt eine Anfrage an den Server und dieser schickt das Ergebnis an die Instanz zurück, welche es dann in geeigneter Form darstellt. Die Größe des Ergebnisses hat einen starken Einfluss auf die Geschwindigkeit des Systems, insbesondere wenn die Ressourcen zum Empfang dieser Ergebnisse in seiner Geschwindigkeit eingeschränkt sind. Es ist somit erstrebenswert mit wenigen Datenbankaufrufen und einfachen Ergebnissen auszukommen.

Ein allgemein bekanntes Problem ist die Darstellung von Webseiten, da sie in unterschiedlichen Browsern nicht immer gleich dargestellt werden oder die Funktionalität eingeschränkt sein kann, da bestimmte Techniken nicht unterstützt werden. Das spielt bei der serverseitigen Implementierung keine tragende Rolle, da die Darstellung vom Browser des Anwenders übernommen wird. Eine Anpassung für unterschiedliche Browser hat somit keinen großen Einfluss auf grundlegende Design-Entscheidungen. Die clientseitige Darstellung von Steuerelementen wie Textfeldern oder Checkboxen dagegen wird in jedem Browser anders behandelt und führt oft zu unterschiedlichen Ergebnissen. Es mussten somit Alternativen für die visuelle Hervorhebung von Steuerelementen gefunden werden. Die Implementierung des Prototyps begrenzt sich vorerst auf die Darstellung in Mozillas Firefox in Version 8.

Eine weitere Herausforderung stellt die Zustandslosigkeit einer Webseite dar. Das Hypertext Transfer Protocol (HTTP) sieht hierfür keine Mechanismen vor [10]. Sobald eine Anfrage vom Client gestellt wurde und eine Antwort vom Server gesendet wurde, ist die Transaktion beendet. Eine herkömmliche Webseite ist somit zustandslos und vergisst alles was vor einem Aufruf passiert ist. Um dem Benutzer das Verhalten einer herkömmlichen Anwendung zu simulieren muss dafür gesorgt werden, dass die Webanwendung einen Zustand besitzt.

1.5 Struktur der Arbeit

Die Ausarbeitung gliedert sich in sechs Kapitel auf. Im folgenden Kapitel werden die Grundlagen von PHILharmonicFlows erläutert und dabei auf die bei der Implementierung umgesetzten Elemente eingegangen. In Kapitel 3 wird die zugrundeliegende technische Infrastruktur erläutert und die Werkzeuge zur Implementierung vorgestellt. In Kapitel 4 wird auf die Architektur und die Implementierung des Prototyps eingegangen und die dafür erarbeiteten Lösungen vorgestellt. Die Arbeit endet mit einer Diskussion in Kapitel 5 und der Zusammenfassung und einem Ausblick für zukünftige Arbeiten in Kapitel 6.

2 Fachliche Grundlagen

In diesem Kapitel werden die fachlichen Grundlagen von PHILharmonicFlows evaluiert. Nach der Vorstellung und Erläuterung der einzelnen Komponenten und deren Elemente wird noch kurz die im Usability-Konzept erarbeitete Benutzeroberfläche vorgestellt.

2.1 Aufbau von PHILharmonicFlows

PHILharmonicFlows besteht grundlegend aus drei Komponenten: der Datenstruktur, der Prozessstruktur und der Benutzerintegration (Abbildung 2). Sie bilden die Basis, um die genannten Eigenschaften eines datenorientierten PrMS erfüllen zu können.

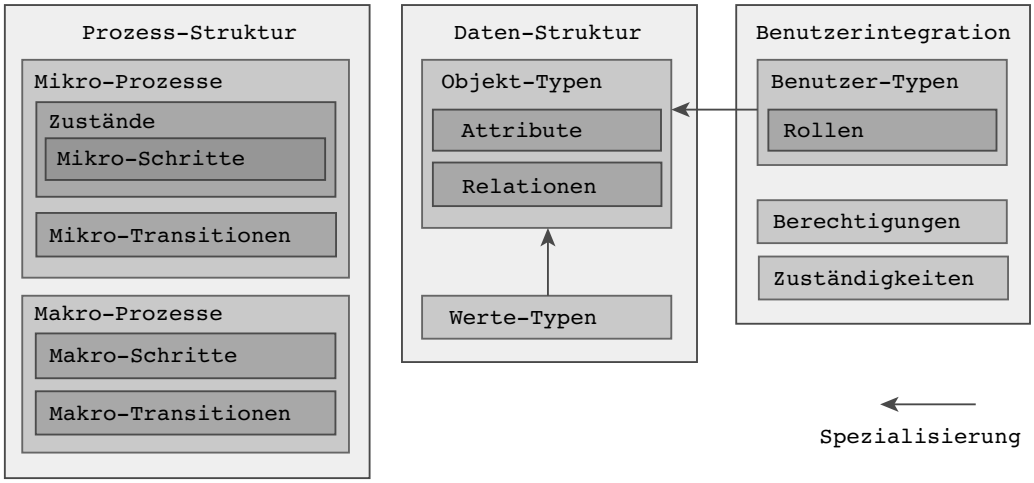

Abbildung 2: Schema der drei Basiskomponenten von PHILharmonicFlows [5]

Zur besseren Übersicht wird im Folgenden von *Typen* gesprochen wenn es um die Modellierung von Prozessen oder Daten geht. Zur Laufzeit werden beliebig viele *Instanzen* von diesen *Typen* angelegt und Werte dafür angegeben (Abbildung 3). Ein Prozess-Typ beschreibt also den modellierten Prozess, eine Prozessinstanz eine Ausprägung dieses Typs bei der Ausführung.

Ein Benutzer modelliert zunächst ein relationales Datenmodell, bestehend aus einer Menge von Objekt-Typen die zueinander in Relation stehen können und beliebige Attribute besitzen. Sie bestimmen die Struktur der Daten die zur Laufzeit für die Objekt-Instanzen gespeichert werden können. Für jeden Objekt-Typ wird anschließend ein Mikro-Prozess-Typ (kurz: Mikro-Prozess) modelliert. Dieser legt das Verhalten des eigentlichen Geschäftsprozesses bezüglich eines Objekt-Typs fest. Anschließend wird ein Makro-Prozess-Typ (kurz: Makro-Prozess) modelliert, welcher für die Koordination zwischen Mikro-Prozessen verantwortlich ist, da deren Instanzen in sich abgeschlossen sind und asynchron

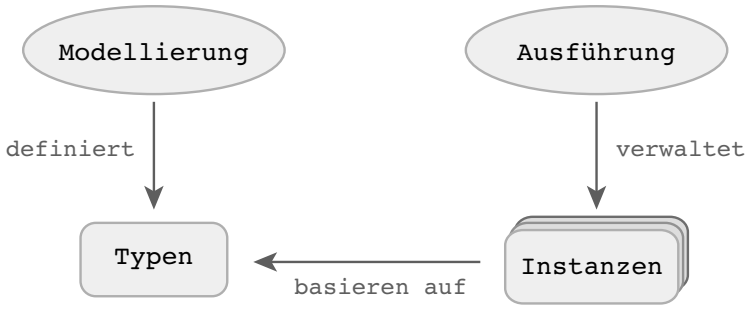

Abbildung 3: Schematischer Zusammenhang zwischen Modellierung und Ausführung

ablaufen. Es ist somit nicht nur möglich das Verhalten einzelner Mikro-Prozess-Instanzen (kurz: Prozess-Instanzen), sondern auch die Interaktion zwischen Instanzen unterschiedlicher Mikro-Prozesse zu steuern.

Die Integration von Benutzern erfolgt in der Datenstruktur durch die Hinzunahme von benutzerspezifischen Objekt-Typen (Benutzer-Typen) in das Datenmodell, sowie durch die Vergabe von Rollen und Benutzerrechten für einzelne Attribute. In der Prozessstruktur erfolgt die Integration durch die Zuordnung von Rollen und Zuständigkeiten für den Mikro-Prozess-Typ. Über die Benutzerintegration lässt sich zur Laufzeit bestimmen ob ein Benutzer zum Beispiel ein Attribut einer Objekt-Instanz schreiben darf oder ob er Aktionen auf einer Prozess-Instanz durchführen darf.

Im folgenden Kapitel sollen die einzelnen Bestandteile der Datenstruktur, der Prozessstruktur und der Benutzerintegration erläutert werden, sowie die konzipierte Benutzeroberfläche vorgestellt werden. Der Fokus liegt dabei auf den Mikro-Prozessen als Teil der Prozessstruktur. Die koordinierenden Makro-Prozesse sind nicht Teil dieser Arbeit. Folgendes Anwendungsszenario ist ein stark vereinfachter Untersuchungsprozess eines Patienten in einem Krankenhaus und dient als Grundlage für alle weiteren Kapitel. Es wird zunächst ein Patient eingeliefert und von einem Mitarbeiter eine Fallakte erstellt und mit den Basisdaten des Patienten gefüllt. Anschließend wird der Patient von einem Arzt untersucht, es wird eine Diagnose erstellt und eine Behandlung vorgeschlagen. Stimmt der Patient zu, wird ein Behandlungstermin vereinbart, der Patient behandelt und anschließend die Fallakte geschlossen.

2.2 Datenstruktur

Das Datenmodell stellt die Datengrundlage eines Prozesses dar. In ihm wird die Struktur der Laufzeitdaten festgelegt.

Das Basiselement ist der *Objekt-Typ* (OT) welcher aus einer Menge von Attributen besteht. Durch die Erzeugung von Objekt-Instanzen zur Laufzeit können dann Daten für

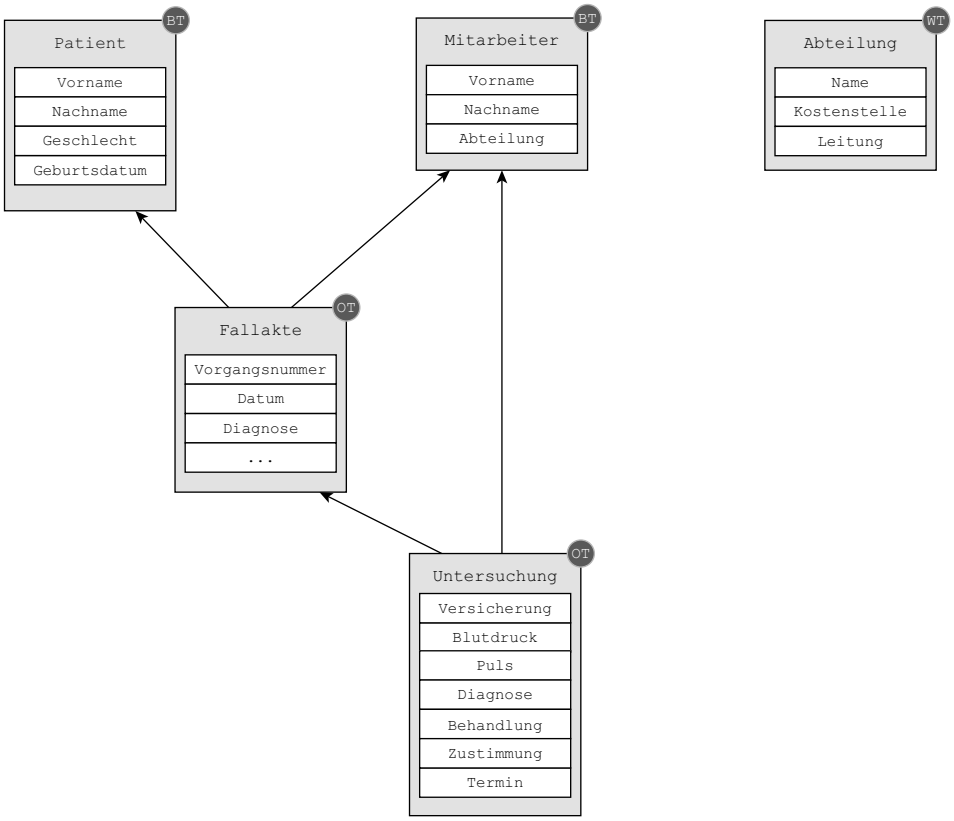

Abbildung 4: Datenmodell für die Aufnahme und Untersuchung eines Patienten im Krankenhaus

die Attribute eingegeben werden. Jeder Objekt-Typ kann in Relation zu weiteren Objekt-Typen stehen. Neben Objekt-Typen gibt es noch zwei Spezialisierungen. *Benutzer-Typen* (BT) und *Werte-Typen* (WT). Ein Werte-Typ dient dazu eine Wertemenge für ein Attribut zu definieren. Es ist somit zur Laufzeit nur möglich Werte eines bestimmten Werte-Typs zuzulassen. Ein Anwendungsfall wäre ein Attribut bei dem der Name einer Abteilung angegeben werden soll. Um Schreibfehler oder unterschiedliche Schreibweisen zu unterbinden, kann es das Attribut *Name* des Werte-Typs *Abteilung* referenzieren. Für jede Abteilung gibt es dann eine Werte-Instanz und der Benutzer kann dann den entsprechenden Abteilungsnamen aus einer Auswahlliste wählen (Abbildung 4). Ein anderes Attribut könnte beispielsweise die Kostenstelle einer Abteilung referenzieren. Werte-Typen beziehen sich auf keine anderen Objekt-Typen und werden somit auch nur indirekt in das Datenmodell eingeschlossen. Benutzer werden über Benutzer-Typen direkt ins Datenmodell integriert. Diese beinhalten zusätzlich einen *Benutzernamen* und ein *Passwort* Attribut zur Autorisierung und besitzen vordefinierte Benutzerrollen. Somit lassen sich auch komplexe Organisationsmodelle adäquat abbilden.

Das vorliegende Datenmodell (Abbildung 4) besteht aus fünf Objekt-Typen, wobei es sich bei *Patient* und *Mitarbeiter* um Benutzer-Typen handelt und bei *Abteilung* um einen Werte-Typ. Man beachte, dass es für *Mitarbeiter* keine konkrete Bezeichnung wie Arzt oder Krankenschwester gibt. Dies erfolgt durch die Zuordnung von Benutzerrollen.

2.3 Prozessstruktur

Nach der Modellierung der Struktur eines Prozesses in Form seiner Daten, wird nun das Verhalten des Prozesses festgelegt. Dafür wird für jeden Objekt-Typ ein Mikro-Prozess definiert. Dieser besteht aus *Zuständen* (`MicroStates`) welche unterschiedliche *Mikro-Schritte* (`MicroSteps`) zusammenfassen. Ein *atomarer Mikro-Schritt* bezieht sich auf genau ein Attribut des zugehörigen Objekt-Typs. Im Gegensatz dazu gibt es noch *leere Schritte* die keine Attribute referenzieren. *Werteschritte* (`MicroValueSteps`) sind für die Bildung von Wertebereichen innerhalb eines *Mikro-Schrittes* vorgesehen und erlauben es alternative Ausführungspfade abhängig von Attributwerten zu definieren. *Mikro-Transitionen* (`MicroTransitions`) bilden den Kontrollfluss zwischen unterschiedlichen *(Werte-)Schritten*. Das Prozessmodell beschreibt aber lediglich die Struktur eines *Mikro-Prozess*. Zusätzlich wird die operationale Semantik benötigt die dessen Ausführung beschreibt. Durch sie ist zum Beispiel festgelegt, dass Pfade mit einer höheren Priorität vorrangig aktiviert werden oder dass eine Prozess-Instanz bei der Eingabe eines Wertes fortgesetzt werden kann, wenn alle Bedingungen dafür erfüllt sind. Zur Umsetzung erhält jedes Prozesselement eine Laufzeitmarkierung.

Die operationale Semantik beschreibt wie sich Änderungen der Laufzeitmarkierungen auf die Markierung angrenzender Elemente auswirken und so den Prozess vorantreiben. Zusätzlich werden Ausnahmebehandlungen festgelegt, um auf Ausnahmesituationen reagieren zu können. Sie sorgt außerdem dafür, dass nur bestimmte Kombinationen von

Laufzeitmarkierungen zwischen betroffenen Elementen auftreten können. Dadurch wird sichergestellt, dass alle auftretenden Fälle abgedeckt sind und eine korrekte Ausführung gewährleistet ist.

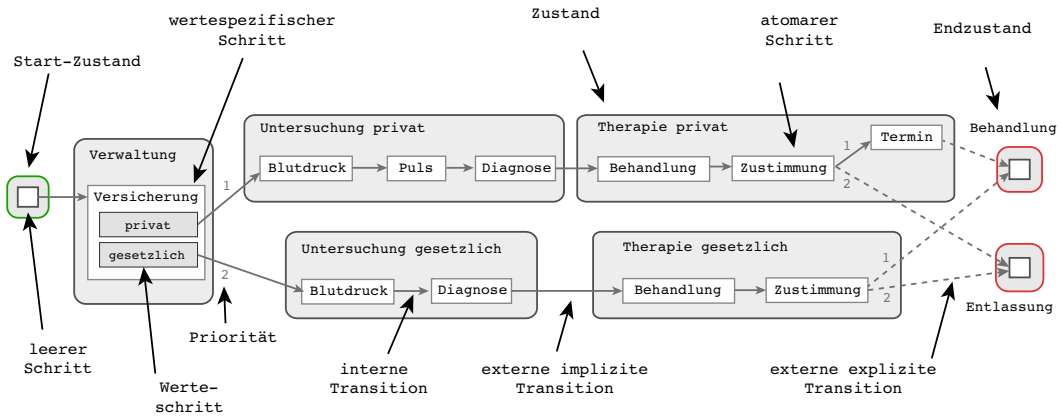

Abbildung 5: Ein möglicher Mikro-Prozess für den Objekt-Typ *Untersuchung*

Der vorgestellte Mikro-Prozess für den Objekt-Typ *Untersuchung* (Abbildung 5) besteht aus acht Zuständen, vierzehn Schritten, zwei Werteschritten und fünfzehn Transitionen. Nach dem Start einer Instanz muss zunächst die *Versicherungsart* des Patienten angegeben werden. In einem der folgenden Zustände (entsprechend ob der Patient *gesetzlich* oder *privat* versichert ist) müssen dann die Werte für *Blutdruck*, ggf. *Puls* und *Diagnose* eingegeben werden. In den Folgezuständen kann dann eine Behandlung angegeben werden und die Zustimmung des Patienten für die Behandlung eingeholt werden. Der Privatpatient kann nach seiner Zustimmung noch zusätzlich einen Vorzugstermin für die Behandlung angeben. Wird die Zustimmung nicht erteilt wird der Patient entlassen, ansonsten wird er behandelt. Im folgenden Abschnitt werden die einzelnen Elemente, deren Funktion und ihre Laufzeitmarkierungen erläutert.

2.3.1 Mikro-Prozess

Ein Mikro-Prozess-Typ steuert das Verhalten eines bestimmten Objekt-Typs. Dazu wird bei der Erzeugung einer neuen Objekt-Instanz automatisch eine neue Mikro-Prozess-Instanz initiiert und gestartet. Durch die Eingabe von Werten in das Datenmodell ändert die Prozessinstanz ihren Zustand und endet, wenn ein Endzustand erreicht wird. Während der Ausführung kann eine Mikro-Prozess-Instanz unterschiedliche Laufzeitmarkierungen besitzen (Tabelle 1). Nach dem Start wird sie als RUNNING markiert, nach der Beendigung als FINISHED. Weitere Markierungen sind zwischen zwei verschiedenen Mikro-Prozess-Instanzen von Bedeutung. So kann ein einzelner Mikro-Prozess im Kontext eines Makro-Prozess als BYPASSED, SKIPPED, BLOCKED oder DEADLOCKED markiert sein. Diese Markierungen werden hier aber außer Acht gelassen.

Zustand_{Mark}	**Erklärung**
RUNNING	Die Instanz wird im Moment ausgeführt.
FINISHED	Die Prozess-Instanz hat einen Endzustand erreicht.

Tabelle 1: Die Laufzeitmarkierungen eines Mikro-Prozesses

2.3.2 Zustände

Ein Zustand hat die Funktion unterschiedliche Mikro-Schritte zu gruppieren und den Zugriff zwischen unterschiedlichen Benutzern zu koordinieren. Mit ihm und der Zuordnung von Benutzerollen ist es möglich, dass ein Benutzer für den Zustand *Untersuchung* Schreibrechte besitzt, jedoch für den Zustand *Therapie* keine Rechte besitzt. Ein Zustand ist entweder ein Startzustand, ein Endzustand oder ein regulärer Zustand. Ein Mikro-Prozess besitzt genau einen Startzustand und mindestens einen Endzustand.

Für einen Zustand existiert ebenfalls eine Menge von Laufzeitmarkierungen (Tabelle 2). Initial hat ein Zustand die Markierung WAITING, es sei denn es handelt sich um den Startzustand, dann hat er die Markierung ACTIVATED. Ein Markierungswechsel (Abbildung 6) erfolgt durch den Übergang einer Prozessinstanz von einem Zustand in einen Folgezustand. Dabei wird der aktivierte Zustand als CONFIRMED markiert und der Folgezustand als ACTIVATED. Wenn es alternative Folgezustände gibt müssen diese als SKIPPED markiert werden, da immer nur ein Pfad des Mikro-Prozess auf einmal durchlaufen werden darf. Der Übergang erfolgt dabei durch eine Ausnahmebehandlung die alternative Pfade eliminiert die nicht mehr ausgeführt werden können (externe Dead-Path-Elimination).

Zustand_{Mark}	Erklärung
WAITING	Der Zustand wurde noch nicht aktiviert, kann aber zu einem späteren Zeitpunkt noch aktiviert werden.
ACTIVATED	Der Zustand ist aktiviert.
CONFIRMED	Die Ausführung befindet sich nach dem Zustand und der Zustand wurde aktiviert.
SKIPPED	Der Zustand kann nicht mehr aktiviert werden, da er auf einem abgewählten, alternativen Pfad der Ausführung liegt.

Tabelle 2: Die Laufzeitmarkierungen eines Zustands

Ein Mikro-Prozess erkennt einen Zustandswechsel daran, wenn eine externe Transition erreicht wird. Wenn deren Laufzeitmarkierung auf READY gesetzt wird, wird der Vorgängerzustand abgeschlossen und der Nachfolgezustand aktiviert. Im vorgestellten Prozessmodell wäre dies zum Beispiel der Übergang vom Zustand *Verwaltung* in den Zustand *Untersuchung gesetzlich*.

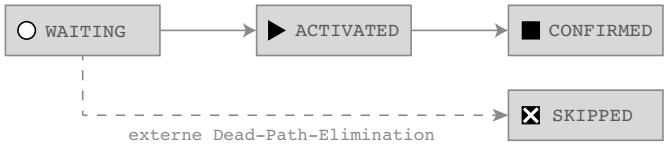

Abbildung 6: Markierungswechsel eines Zustands [3]

2.3.3 Mikro-Schritte

Ein Mikro-Schritt (kurz: Schritt) bezieht sich im Allgemeinen auf genau ein Attribut des zugehörigen Objekt-Typs und stellt somit die Verbindung zwischen den Daten und dem Mikro-Prozess dar. Wenn ein Wert für ein Attribut einer Objekt-Instanz eingegeben wird, kann der zugehörige Schritt aktiviert und die Ausführung fortgesetzt werden. Neben einem atomaren Schritt, der sich auf genau ein Attribut bezieht, gibt es noch zwei weitere Arten. Ein wertespezifischer Schritt referenziert ebenfalls ein Attribut, beinhaltet aber zusätzlich Werteschritte die unterschiedliche Wertebereiche definieren können. Für unterschiedliche Werte des Attributs können somit unterschiedliche Werteschritte und somit alternative Pfade aktiviert werden. Ein leerer Schritt bezieht sich auf kein Attribut und wird automatisch aktiviert, wenn er von der Ausführung erreicht wird. Er wird zum Beispiel in einem Endzustand eingesetzt weil hier keine weiteren Daten mehr eingegeben werden müssen, da der Mikro-Prozess beendet ist. In Abbildung 5 ist dies zum Beispiel der Fall, wenn die Ausführung vom Zustand *Therapie privat* in den Zustand *Behandlung* übergeht. An dieser Stelle sind bereits alle nötigen Daten eingegeben worden.

Ein Schritt sorgt zusätzlich für die Konsistenz zwischen dem Prozess- und dem Datenzustand. Wenn ein Attribut nachträglich geändert wird, kann es passieren dass die Prozessausführung nicht mehr konsistent ist, da sich durch den neuen Wert vielleicht eine alternative Ausführung ergeben hätte, als mit dem alten Wert. Um dies zu korrigieren, werden weitere Ausnahmebehandlungen vorgestellt. Falls der zugehörige Zustand noch aktiviert ist, muss er zurückgesetzt werden und mit dem neuen Wert evaluiert werden (interne Rücksetzung des Zustands). Falls er bereits abgeschlossen ist, muss der Schritt als inkonsistent markiert werden und der Benutzer muss eingreifen (Inkonsistenzbehandlung).

Das Verhalten eines Schrittes wird ebenfalls durch Laufzeitmarkierungen gesteuert. Insgesamt verfügt er über neun verschiedene Markierungen (Tabelle 3). Initial werden alle Schritte als WAITING markiert, es sei denn sie liegen im Startzustand dann werden sie als READY markiert. Der Startschritt (der erste Schritt im Zustand) wird wiederum als ENABLED markiert was bedeutet, dass für diesen Schritt als nächstes ein Wert eingegeben werden muss um den Prozess fortzusetzen.

Eine Änderung der Laufzeitmarkierung ist bei einem Schritt von unterschiedlichen Prozesselementen abhängig. Wenn ein Wert für das referenzierte Attribut eingegeben wird,

Schritt_{Mark}	Erklärung
WAITING	Der Zustand in dem der Schritt liegt wurde noch nicht aktiviert.
READY	Der Zustand wurde aktiviert, der Schritt kann im Moment aber noch nicht erreicht werden.
ENABLED	Der Schritt kann durch Eingabe eines Wertes aktiviert werden.
ACTIVATED	Der Schritt ist aktiviert. Ein Wert wurde eingegeben.
BLOCKED	Der eingegebene Wert muss verändert werden, da er in keinen Wertebereich fällt der durch Werteschritte definiert wurde.
UNCONFIRMED	Der Schritt wurde besucht aber der dazugehörige Zustand wurde noch nicht komplett durchlaufen.
CONFIRMED	Der Zustand wurde verlassen und der Schritt wurde aktiviert.
BYPASSED	Der Schritt liegt in einem aktivierten Zustand, der Pfad auf dem er liegt wurde aber abgewählt.
SKIPPED	Der Schritt wurde nicht aktiviert.

Tabelle 3: Laufzeitmarkierungen eines Schrittes

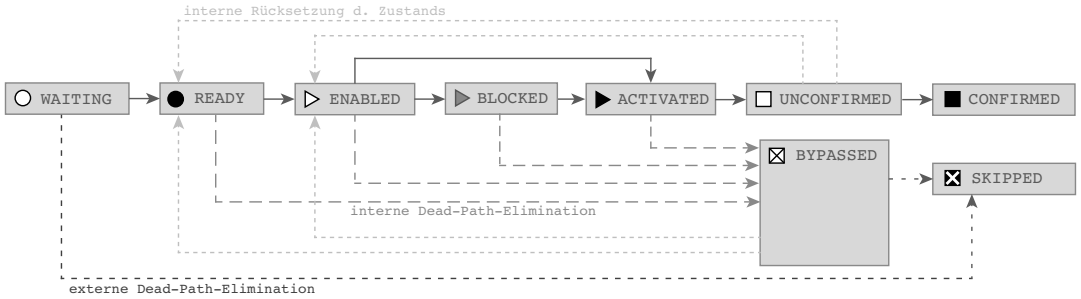

Abbildung 7: Markierungswechsel eines Schrittes [3]

wenn sich die Laufzeitmarkierung des zugehörigen Zustands verändert oder wenn sich die Markierungen der eingehenden oder ausgehenden Transitionen verändern.

Die Eingabe eines Wertes hat auf einen Schritt genau dann eine Auswirkung, wenn er als ENABLED markiert ist, d.h. dass er aktiviert werden kann. Dabei geht die Markierung in ACTIVATED über und gibt an dass der Schritt aktiviert wurde. Dabei muss zusätzlich berücksichtigt werden, ob der Mikro-Schritt einen oder mehrere Werteschritte beinhaltet. Falls dies zutrifft, muss geprüft werden ob der eingegebene Wert in mindestens einem Wertebereich eines Werteschritts fällt oder nicht. Bei Letzterem muss der Schritt vorläufig als BLOCKED markiert werden bis ein gültiger Wert eingegeben wurde. Außerdem muss unterschieden werden, ob ein Wert gesetzt wurde oder gelöscht wurde, da Markierungen zum Beispiel auch von BLOCKED auf ENABLED zurückgesetzt werden müssen, wenn der Wert entfernt wurde. Durch die Veränderung eines Wertes können zudem Ausnahmebehandlungen angestoßen werden.

Diese Übergänge sind in Tabelle 4 dargestellt. Die erste Spalte gibt die Ausgangsmarkierung des Schrittes an. Die zweite Spalte gibt an, ob der Schritt einen oder mehrere Werteschritte beinhaltet. Die dritte Spalte gibt an, ob der Attributwert verändert wurde und die letzte Spalte beinhaltet die neue Markierung des Schrittes, wenn die Bedingungen in den vorherigen Spalten erfüllt sind. Dadurch erhält ein Schritt mit der Ausgangsmarkierung ENABLED die neue Markierung ACTIVATED, wenn kein Werteschritt vorhanden ist und ein Wert gesetzt wurde (vgl. Tabelle 4, Zeile 2).

Schritt_{Mark}	$\exists Werteschritt$	$\exists Wert$	$\text{Schritt}'_{Mark}$
ENABLED	Ja	Ja	BLOCKED
ENABLED	Nein	Ja	ACTIVATED
BLOCKED	-	Nein	ENABLED
BLOCKED	-	Ja	ACTIVATED
UNCONFIRMED	-	-	Ausnahmebehandlung
BYPASSED	-	-	Ausnahmebehandlung

Tabelle 4: Markierungsänderungen, die durch die Veränderung des Attributwerts ausgelöst werden. Eine Veränderung kann das Löschen, Eingeben oder Ändern eines Wertes bedeuten.

Eine weitere Änderung der Laufzeitmarkierung wird durch die Änderung des zugehörigen Zustands hervorgerufen. Wenn ein Zustand aktiviert wird, müssen auch alle Schritte verändert werden. Dafür werden zunächst alle Schritte im Zustand von ihrer aktuellen Belegung WAITING auf ENABLED bzw. READY gesetzt. Es muss außerdem darauf reagiert werden, ob bereits Werte für Attribute eingegeben wurden, denn dann kann der Prozess weitergeschaltet werden. Das gleiche gilt wenn ein Zustand abgeschlossen wird. Dabei müssen alle aktivierten Schritte als CONFIRMED und alle übersprungenen Schritte in alternativen Pfaden als SKIPPED markiert werden.

Bei diesen Übergängen (Tabelle 5) muss auf die aktuelle Markierung des Zustands geachtet werden und ob es sich dabei um den Endzustand handelt, da dieser nur einen leeren Schritt beinhaltet. Außerdem muss geprüft werden, ob mindestens eine eingehende Transition auf READY gesetzt ist, weil der Schritt dann die Markierung READY überspringen und mit ENABLED markiert werden kann. Wenn der Mikro-Schritt einen Werteschritt beinhaltet, muss zusätzlich geprüft werden, ob mindestens ein Wertebereich getroffen wird. Zuletzt muss noch beachtet werden, ob ein Wert gesetzt wurde oder nicht. Diese Eigenschaften führen zu unterschiedlichen Folgemarkierungen eines Schrittes.

So erhält ein Schritt mit der Markierung WAITING die neue Markierung ENABLED wenn der übergeordnete Zustand die Markierung ACTIVATED besitzt, es sich dabei um keinen Endzustand handelt, keine Werteschritte vorhanden sind und kein Wert gesetzt ist. Zusätzlich muss gelten, dass mindestens eine eingehende Transition die Markierung READY besitzt (vgl. Tabelle 5, Zeile 2).

Schritt_{Mark}	Zustand_{Mark}	Zustand_{Typ}	$\exists Werteschritt$	$\exists Wert$	$\text{Schritt}'_{Mark}$
WAITING	ACTIVATED	End	-	-	CONFIRMED
WAITING	ACTIVATED	$\neg End$	Nein	Nein	ENABLED (*)
WAITING	ACTIVATED	$\neg End$	Nein	Ja	ACTIVATED (*)
WAITING	ACTIVATED	$\neg End$	Ja	Nein	ENABLED (*)
WAITING	ACTIVATED	$\neg End$	Ja	Ja	BLOCKED (*)
WAITING	ACTIVATED	$\neg End$	-	-	READY (+)
READY	ACTIVATED	$\neg End$	Nein	Nein	ENABLED (*)
READY	ACTIVATED	$\neg End$	Nein	Ja	ACTIVATED (*)
UNCONFIRM.	CONFIRMED	-	-	-	CONFIRMED
BYPASSED	CONFIRMED	-	-	-	SKIPPED

Tabelle 5: Markierungsänderungen durch Zustandsmarkierung ausgelöst. (*) Mindestens eine eingehende Transition muss als READY markiert sein. (+) Wenn keine eingehende Transition READY ist.

Die häufigste Änderung der Laufzeitmarkierung lösen Markierungswechsel von eingehenden und ausgehenden Transitionen aus. Durch sie erkennt ein Schritt zum Beispiel, ob er aktiviert werden kann oder nicht. Durch das Wechselspiel der Laufzeitmarkierungen zwischen Schritten und Transitionen ist eine korrekte Ausführung der Prozessinstanz gewährleistet. Dabei müssen ebenfalls mehrere Faktoren berücksichtigt werden. Zunächst spielt die Markierung der auslösenden Transition eine Rolle. Außerdem muss beachtet werden, ob der Schritt einen Werteschritt beinhaltet (wegen der Markierung BLOCKED) und ob ein Wert gesetzt wurde oder nicht. So erhält ein Schritt mit der Markierung WAITING die neue Markierung SKIPPED wenn die eingehende Transition die Markierung SKIPPED besitzt. Ob der Schritt dabei Werteschritte beinhaltet oder einen Wert besitzt, ist dabei unerheblich (vgl. Tabelle 6, Zeile 1).

Schritt_{Mark}	Transition_{Mark}	$\exists Werteschritt$	$\exists Wert$	$\text{Schritt}'_{Mark}$
WAITING	SKIPPED	-	-	SKIPPED
WAITING	-	Nein	Ja	ACTIVATED
WAITING	-	Ja	Ja	BLOCKED
READY	READY	Nein	Ja	ACTIVATED
READY	READY	Ja	Ja	BLOCKED
READY	READY	-	Nein	ENABLED
ENABLED	-	Nein	Ja	ACTIVATED
ENABLED	-	Ja	Ja	BLOCKED
ACTIVATED	ACTIVATED	-	-	UNCONFIRMED
BYPASSED	-	-	-	SKIPPED

Tabelle 6: Änderungen durch eine Transitionsmarkierung ausgelöst.

Die letzte Möglichkeit der Änderung einer Laufzeitmarkierung ist die Ausführung von Ausnahmebehandlungen. Diese treten auf wenn:

- ein alternativer Pfad in einem aktivierten Zustand, der nicht ausgewählt wurde eliminiert wird (interne Dead-Path-Elimination)
- wenn beim Verlassen eines Zustands alternative Pfade die nicht aktiviert wurden zustandsübergreifend eliminiert werden müssen (externe Dead-Path-Elimination) weil sie nicht mehr erreichbar sind
- ein Attributwert innerhalb eines aktivierten Zustands nachträglich geändert wird und der Zustand erneut evaluiert werden muss (interne Rücksetzung des Zustands)

Bei Ausnahmebehandlungen in einem Schritt (Tabelle 7) muss zum Einen auf Abbruchkriterien geachtet werden und ob es sich um einen Startschritt handelt, da dieser bei einer Rücksetzung eine andere Markierung erhält als die anderen Schritte. Die interne Dead-Path-Elimination darf nur angewendet werden, wenn alle eingehenden Transitionen des Schrittes als BYPASSED markiert sind. Der Schritt kann also nicht mehr aktiviert werden und wird ebenfalls als BYPASSED markiert. Bei der externen Dead-Path-Elimination muss geprüft werden ob alle eingehenden Transitionen bereits als SKIPPED markiert sind um sicherzustellen, dass der Schritt von keinem alternativen Pfad aus mehr erreicht werden kann.

Somit hat ein Schritt mit der Markierung READY nach einer internen Dead-Path-Elimination die Markierung BYPASSED wenn die eingehende Transition als BYPASSED markiert ist. Um welche Art von Schritt es sich dabei handelt ist egal (vgl. Tabelle 7, Zeile 1).

Schritt_{Mark}	Schritt_{Typ}	Transition_{Mark}	Ausnahmebehandlung	$\text{Schritt}'_{Mark}$
READY	-	BYPASSED	int. Dead-Path-Elimination	BYPASSED
ENABLED	-	BYPASSED	int. Dead-Path-Elimination	BYPASSED
BLOCKED	-	BYPASSED	int. Dead-Path-Elimination	BYPASSED
ACTIVATED	-	BYPASSED	int. Dead-Path-Elimination	BYPASSED
BYPASSED	-	SKIPPED	ext. Dead-Path-Elimination	SKIPPED
WAITING	-	SKIPPED	ext. Dead-Path-Elimination	SKIPPED
-	Start	-	int. Rücksetzung d. Zustands	ENABLED
-	$\neg Start$	-	int. Rücksetzung d. Zustands	READY

Tabelle 7: Änderungen eines Schrittes durch Ausnahmebehandlungen.

2.3.4 Werteschritte

Ein Werteschritt ist ein Element zur Definition von Wertebereichen innerhalb eines Schrittes. Ein Schritt kann beliebig viele Werteschritte besitzen und die Wertebereiche können sich zudem überschneiden. Jeder Werteschritt kann dabei mehrere ausgehende Transitionen besitzen. Somit lassen sich unterschiedliche Pfade für unterschiedliche Attributwerte anhand eines Prädikats aktivieren.

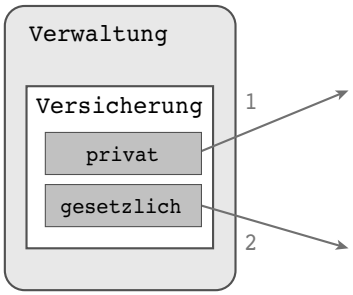

Abbildung 8: Ein wertespezifischer Mikro-Schritt mit zwei Werteschritten.

Das Prädikat besteht aus einem oder mehreren UND-verknüpften einfachen Prädikaten. Ein einfaches Prädikat bezieht sich auf genau ein Attribut. Es besitzt einen Vergleichsoperator und einen Referenzwert und wird wahr wenn der Ausdruck durch das Einsetzten des Attributwerts wahr wird. Das komplexe Prädikat wird wahr, wenn jedes der einfachen Prädikate wahr wird. Ein Werteschritt kann genau dann aktiviert werden, wenn sein Prädikat wahr ist. Falls mehrere Werteschritte gleichzeitig aktiviert werden können, was durch die überlappenden Wertebereiche möglich ist, wird über die Prioritäten der ausgehenden Transitionen bestimmt welcher Pfad den Vorrang hat. Eingehende Transitionen sind nicht vorgesehen, der Prozessfluss geht grundsätzlich beim übergeordneten Schritt

ein (siehe Abbildung 5 bei dem der Startschritt mit dem Schritt *Versicherung* verbunden ist und nicht mit den Werteschritten darin).

Abbildung 8 zeigt einen Schritt mit zwei Werteschritten. Beiden Werteschritten wurde ein Prädikat der Form *Versicherung==Wert* zugeordnet, wobei der Wert einmal "*gesetzlich*" ist und einmal "*privat*". Dadurch definieren die Werteschritte jeweils eine Wertemenge mit genau einem Wert für den der Werteschritt aktiviert werden kann. Bei allen anderen Werten würde der übergeordnete Schritt als BLOCKED markiert werden.

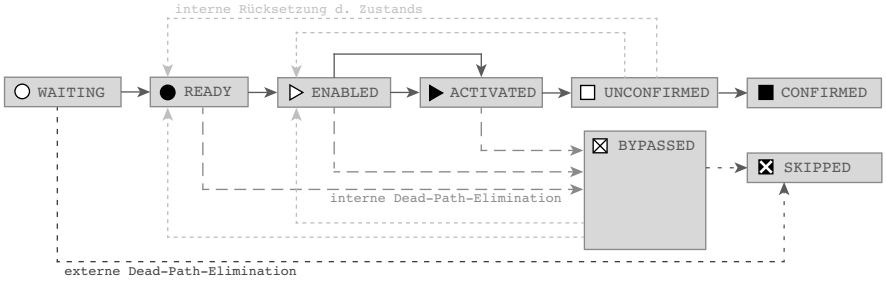

Abbildung 9: Markierungswechsel eines Werteschrittes [3]

Da ein Werteschritt grundsätzlich die gleiche Aufgabe wie ein Schritt hat (nur auf einen Wertebereich beschränkt) besitzt er sehr ähnliche Laufzeitmarkierungen wie ein Schritt und verhält sich auch ähnlich. Eine Markierungsänderung wird jedoch vom übergeordneten Schritt und ausgehenden Transitionen ausgelöst. Dabei wird vorausgesetzt, dass ein Werteschritt nur dann aktiviert werden kann wenn das zugehörige Prädikat *wahr* wird.

Initial sind alle Werteschritte mit der Markierung des übergeordneten Schrittes belegt. Bei der Eingabe oder Änderung eines Wertes im Schritt werden alle Prädikate der Werteschritte evaluiert und der Pfad aktiviert für den das Prädikat wahr wurde und die ausgehende Transition die höchste Priorität besitzt. Alle anderen Werteschritte werden anschließend als BYPASSED markiert und alle ausgehenden Pfade deaktiviert.

In Tabelle 8 sind die unterschiedlichen Übergänge, ausgelöst durch den übergeordneten Schritt, dargestellt. Dabei wird dessen Markierung berücksichtigt und geprüft, ob der Werteschritt die höchste Priorität bezüglich der anderen aktivierbaren Werteschritte besitzt. So erhält ein Werteschritt mit der Markierung WAITING die neue Markierung READY, wenn der übergeordnete Schritt als READY markiert wurde (vgl. Tabelle 8, Zeile 1).

2.3.5 Mikro-Transitionen

Eine Mikro-Transition (kurz: Transition) stellt den Kontrollfluss innerhalb eines Mikro-Prozesses dar. Sie besitzt genau einen Quell- und einen Zielschritt und ist standardgemäß

Werteschritt$_{Mark}$	Schritt$_{Mark}$	Werteschritt$'_{Mark}$
WAITING	READY	READY
READY	ENABLED	ENABLED
READY	ACTIVATED	ACTIVATED (*)
READY	BYPASSED	BYPASSED
READY	UNCONFIRMED	BYPASSED
ENABLED	ACTIVATED	ACTIVATED (*)
ENABLED	BYPASSED	BYPASSED
ENABLED	UNCONFIRMED	BYPASSED
ACTIVATED	UNCONFIRMED	UNCONFIRMED
BYPASSED	CONFIRMED	SKIPPED
UNCONFIRMED	CONFIRMED	CONFIRMED

Tabelle 8: Änderungen eines Werteschritts durch den übergeordneten Schritt. (*) Bei einem Übergang zu ACTIVATED muss stets gelten, dass die Priorität der ausgehenden Transition die höchste aktivierbare ist.

mit einer Priorität versehen. Über diese lässt sich im Zweifelsfall entschieden, welcher Pfad aktiviert werden darf falls mehrere Alternativen aktivierbar sind. Transitionen die vom selben Schritt ausgehen müssen unterschiedliche Prioritäten aufweisen.

Transitionen lassen sich unterschiedlichen Kategorien zuordnen, abhängig davon, wo sie im Modell platziert werden und welche Aufgabe sie haben. Interne Transitionen verbinden Schritte innerhalb desselben Zustands und der Übergang erfolgt immer implizit. Externe Transitionen dagegen verbinden Schritte über zwei Zustände hinweg und können sowohl implizit als auch explizit schalten. Bei letzterem wird zusätzlich eine Entscheidung des Benutzers erwartet.

Eine Transition besitzt neun Laufzeitmarkierungen (Tabelle 9) und wird initial als WAITING markiert, es sei denn sie geht vom Startschritt aus, dann wird sie als READY markiert. Eine externe explizite Transitionen besitzt zusätzlich eine Markierung CONFIRMABLE, in der sie nach der Aktvierung des Vorgängerschrittes so lange verbleibt, bis sie durch den Benutzer aktiviert wird. So lassen sich auch explizite Zustandsübergänge modellieren.

Ein Markierungswechsel einer Transition wird durch einen Zustandswechsel oder die Änderung der Markierung von (Werte-)Schritten (Tabelle 10 und 11) ausgelöst. Dabei wird zwischen den drei Arten von Transitionen unterschieden, da diese unterschiedliche Übergänge für die einzelnen Markierungen besitzen. Alle externe Transitionen werden beispielsweise automatisch von ACTIVATED auf CONFIRMED gesetzt ohne den Weg über UNCONFIRMED zu gehen. Das kommt daher, dass diese Markierung den Zustand

Transition_{Mark}	Erklärung
WAITING	Die Transition wurde noch nicht, kann aber noch aktiviert werden.
CONFIRMABLE	Die Transition muss vom Benutzer weitergeschaltet werden.
READY	Die Transition kann aktiviert werden.
ENABLED	Die Transition wird aktiviert wenn sie die höchste Priorität besitzt.
ACTIVATED	Die Transition hat die höchste Priorität und ist aktiviert.
UNCONFIRMED	Die Transition wurde aktiviert, der Zustand ist aber noch nicht abgeschlossen.
CONFIRMED	Der übergeordnete Zustand wurde abgeschlossen.
BYPASSED	Die Transition wurde deaktiviert, kann aber durch eine Ausnahmebehandlung noch aktiviert werden.
SKIPPED	Die Transition wurde nicht aktiviert.

Tabelle 9: Laufzeitmarkierungen einer Transition

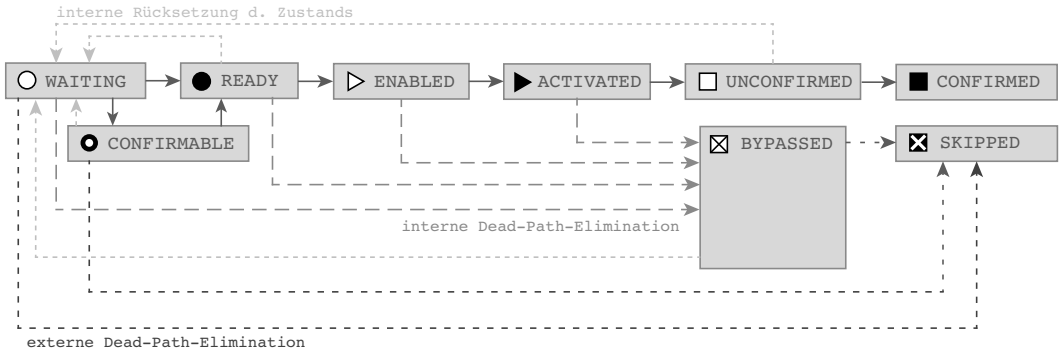

Abbildung 10: Markierungswechsel einer Transition [3]

einer Transition beschreibt die sich genau einem Zustand zuordnen lässt. Bei expliziten Transitionen ist dies nicht möglich. Zudem werden sie nie als BYPASSED sondern direkt als SKIPPED markiert. Das liegt daran, dass sie nur von einer externen Dead-Path-Elimination betroffen sind und nicht von einer internen, da sie sich keinem Zustand zuordnen lassen. Es muss außerdem bei allen Transitionen beachtet werden, dass immer nur die ausgehende Transition eines (Werte-)Schrittes aktiviert werden darf, die die höchste Priorität von allen besitzt. Für alle anderen Transitionen muss zunächst eine interne Dead-Path-Elimination durchgeführt werden um zu garantieren, dass nur ein Pfad weiter verfolgt wird.

Eine interne Transition mit der Markierung ACTIVATED erhält beispielsweise die Folgemarkierung UNCONFIRMED wenn ihr Ziel-Schritt die Markierung CONFIRMED erhält. Eine externe Transition würde im gleichen Szenario dagegen die Markierung CONFIRMED erhalten (vgl. Tabelle 10, Zeile 5 und Tabelle 11, Zeile 9).

Transition_{Mark}	$\text{(Werte-)Schritt}_{Mark}$	$\text{Schritt}_{Position}$	$\text{Transition}'_{Mark}$
WAITING	UNCONFIRMED	Quelle	READY
WAITING	CONFIRMED	Quelle	READY
READY	ACTIVATED	Quelle	ENABLED
ENABLED	-	-	ACTIVATED (*)
ACTIVATED	UNCONFIRMED	Senke	UNCONFIRMED
UNCONFIRMED	CONFIRMED	Quelle	CONFIRMED

Tabelle 10: Änderungen einer internen Transition durch den Vorgänger und Nachfolgerschritt. Die Senke kann sowohl ein Schritt als auch ein Werteschritt sein. (*) Bei einem Übergang von ENABLED zu ACTIVATED muss stets gelten, dass die Priorität der Transition die höchste aktivierbare ist.

2.3.6 Ausnahmebehandlungen

Um auf bestimmte Ausnahmen im Prozessablauf reagieren zu können, gibt es gewisse Ausnahmebehandlungen. Die *interne Dead-Path-Elimination* kommt zum Einsatz, wenn ein Pfad *innerhalb* eines Zustands nicht mehr aktiviert werden kann. Dies geschieht dann, wenn eine Transition mit einer höheren Priorität aktiviert wird bzw. ein bestimmter Werteschritt aktiviert wird und alle Alternativen nicht. Alle (Werte-)Schritte und Transitionen auf den alternativen Pfaden werden dabei als BYPASSED markiert, dabei müssen jedoch alle eingehenden Transitionen eines Schrittes ebenfalls als BYPASSED markiert sein, da der Schritt sonst noch von einem weiteren Pfad aktiviert werden könnte. Einbezogen werden dabei Schritte mit der Markierung READY, ENABLED, BLOCKED und ACTIVATED. Alle anderen Markierung indizieren, dass sich der Schritt in einem nicht (mehr) aktivierten Zustand befindet und somit keine interne Dead-Path-Elimination auftreten kann.

Transition_{Mark}	(Werte-)Schritt_{Mark}	$\text{Schritt}_{Position}$	Transition_{Typ}	$\text{Transition}'_{Mark}$
WAITING	UNCONFIRMED	Quelle	impl.	READY
WAITING	UNCONFIRMED	Quelle	expl.	CONFIRMABLE
CONFIRMABLE	-	-	expl.	ENABLED (*)
READY	CONFIRMED	Senke	impl.	CONFIRMED
READY	ACTIVATED	Senke	impl.	ENABLED
READY	CONFIRMED	Quelle	impl.	READY
READY	-	-	impl.	ENABLED
ENABLED	-	-	-	ACTIVATED (+)
ACTIVATED	UNCONFIRMED	Senke	-	CONFIRMED
BYPASSED	-	-	expl.	SKIPPED
-	CONFIRMED	-	-	SKIPPED
-	SKIPPED	-	-	SKIPPED

Tabelle 11: Änderungen einer externen Transition durch Vorgänger und Nachfolgerschritte. Die Senke kann sowohl ein Schritt als auch ein Werteschritt sein. (*) Durch Interaktion des Benutzers. (+) Bei einem Übergang von ENABLED zu ACTIVATED muss stets gelten, dass die Priorität der Transition die höchste aktivierbare ist.

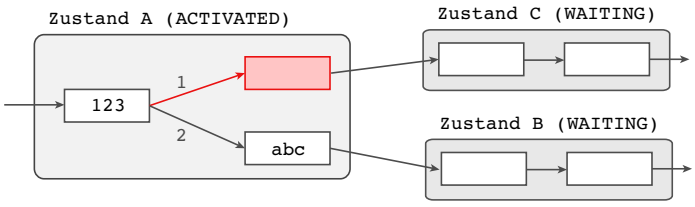

Abbildung 11: Prozess-Instanz nach einer internen Dead-Path-Elimination

Die *externe Dead-Path-Elimination* findet Anwendung, wenn ein Zustand abgeschlossen wird in dem zuvor eine interne Dead-Path-Elimination durchgeführt wurde. Sobald auf die erste Transition getroffen wird, die als BYPASSED markiert wurde, wird sie gestartet. Alle Pfade die während der Ausführung nicht aktiviert wurden werden dabei als SKIPPED markiert. Die Besonderheit ist, dass die externe Dead-Path-Elimination nicht am Ende des Zustands endet, sondern den Pfad solange verfolgt und eliminiert bis er auf einen Schritt trifft der mehrere Pfade zusammenfügt und davon mindestens ein Pfad noch aktiviert werden kann.

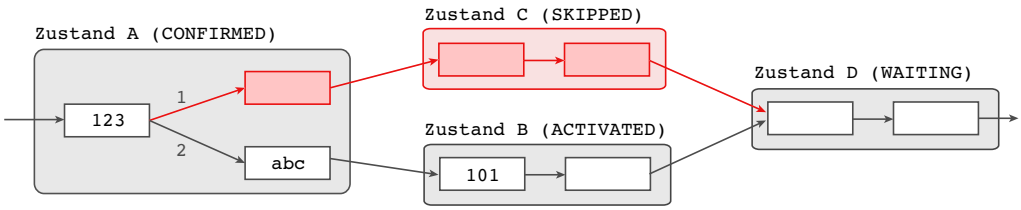

Abbildung 12: Prozess-Instanz nach einer externen Dead-Path-Elimination

Ein *State-Internal-Reset* wird ausgelöst, wenn die Änderung eines Attributes in einem *aktivierten* Zustand einen alternativen Ausführungspfad aktivieren würde, der zuvor von der internen Dead-Path-Elimination deaktiviert wurde. Dies kann zum einen passieren, wenn die Änderung des Werts einen alternativen Pfad aktivieren würde oder wenn ein Wert für einen Schritt gesetzt wird, dessen eingehende Transition eine höhere Priorität besitzt als die ursprünglich aktivierte. In diesem Fall werden alle Schritte und Transitionen innerhalb des aktivierten Zustands auf ihre initiale Markierung zurückgesetzt und der Zustand mit dem neuen Wert erneut evaluiert. Somit ist es möglich alternative Pfade nachträglich zu aktivieren, die durch die interne Dead-Path-Elimination bereits deaktiviert wurden. Dies ist nur möglich, solange der Zustand aktiviert ist.

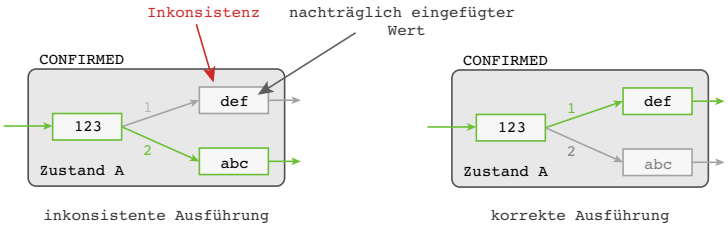

Abbildung 13: Inkonsistenz bei der nachträglichen Änderung eines Attributwerts

Inkonsistenzen treten auf, wenn ein Attributwert nachträglich geändert wird. Dies passiert entweder direkt, wenn im Mikro-Prozess genau ein Schritt dieses Attribut referenziert oder wenn mehrere Schritte in unterschiedlichen Zuständen das Attribut referenzieren. Die Ausführung wird dabei normal weitergeführt, dem Benutzer wird die Inkonsistenz aber im Formular und in der Übersichtsliste aller Prozesse angezeigt. Der

Benutzer kann zur Auflösung der Inkonsistenz auswählen, ob er sie ignorieren will, ob er alle Schritte auf den ursprünglichen konsistenten Wert zurücksetzen möchte oder ob die Ausführung zurückgesetzt werden soll bis zu dem Punkt, an dem die Ausführung noch konsistent war.

2.3.7 Beispiel für eine Prozessausführung

Im folgenden Beispiel (Abbildung 14) soll der Markierungsübergang zwischen Schritten und Transitionen an einem einfachen Beispiel erläutert werden. Da eine Veränderung eines Schrittes und dazugehöriger Transitionen einher geht, werden diese zusammen in einer Abbildung angezeigt.

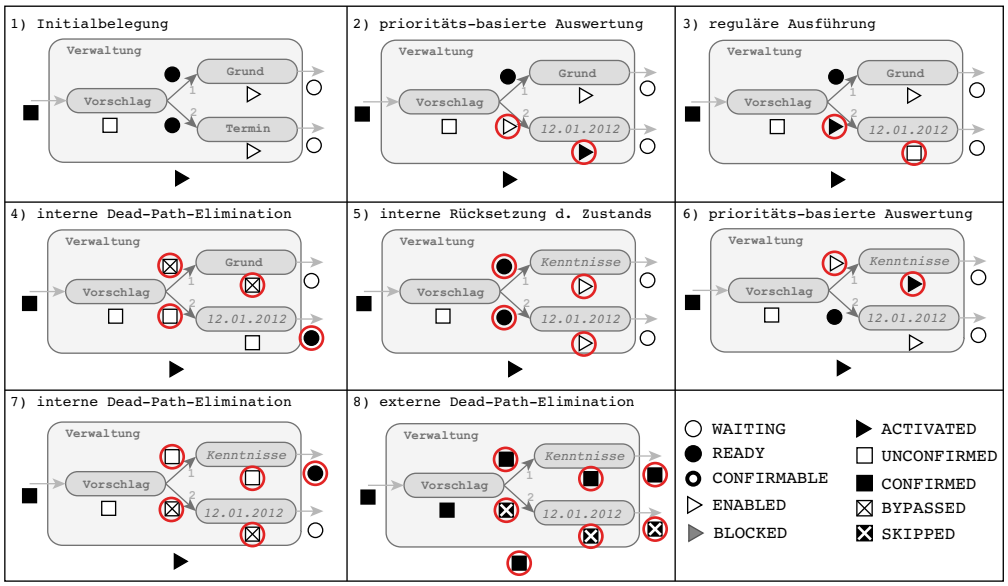

Abbildung 14: Ablauf eines Markierungswechsels innerhalb eines Zustands.

Die Ausgangslage ist, dass ein Wert für eines der beiden Attribute *Grund* und *Termin* angegeben werden soll (Feld 1). Nach der Eingabe eines Datums in *Termin* schaltet der Schritt auf ACTIVATED und daraufhin die eingehende Transition auf ENABLED. Nach einer Prioritätsprüfung darf die Transition auf ACTIVATED schalten, da keine Transition mit höherer Priorität aktiviert kann. Nun schaltet der Schritt auf UNCONFIRMED, weil die eingehende Transition aktiviert ist (Feld 3) und alle Transitionen, die bei der Prioritätsprüfung verloren haben werden als BYPASSED markiert (Feld 4). Wenn nun nachträglich ein Wert für *Grund* angegeben wird, muss eine interne Rücksetzung des Zustands durchgeführt werden der alle Schritte auf ihre Initialmarkierung zurücksetzt (Feld 5). Anschließend wird der Zustand erneut evaluiert. Da nun auch ein Wert für *Grund* existiert und die eingehende Transition die höchste Priorität (1) besitzt, wird der dazuge-

hörige Schritt und die Transition selbst aktiviert und alle alternativen Pfade durch eine interne Dead-Path-Elimination als BYPASSED markiert (Feld 6 und 7). Anschließend wird der aktuelle Zustand abgeschlossen, alle aktivierten Elemente auf CONFIRMED gesetzt und die restlichen Elemente durch die externe Dead-Path-Elimination als SKIPPED markiert. Anschließend wird der Folgezustand aktiviert.

2.4 Benutzerintegration

Bei der Modellierung des Datenmodells können allen Benutzer-Typen unterschiedliche Benutzerrollen zugeordnet werden. Diesen Benutzerrollen sind bestimmte Berechtigungen und Zuständigkeiten zugeordnet und sorgen zur Laufzeit dafür, dass beim Zugriff auf Attribute zwischen unterschiedlichen Benutzergruppen unterschieden werden kann.

Berechtigungen erlauben das Erzeugen, Löschen und Bearbeiten von Instanzdaten in Abhängigkeit des Zustands der dazugehörigen Mikro-Prozess-Instanz. Eine Erzeugungsberechtigung erlaubt es einem Benutzer Instanzen eines bestimmten Objekt-Typs zu erzeugen. Die dazugehörige Mikro-Prozess-Instanz wird automatisch dazu erzeugt. Eine Löschberechtigung bestimmt, ob der Benutzer eine Instanz löschen darf. Dabei wird auch beachtet in welchen Zustand sich der dazugehörige Mikro-Prozess der Instanz befindet. Es kann also sein, dass ein Benutzer eine Instanz nur Löschen darf bevor sich die Prozessinstanz im Endzustand befindet. Attributberechtigungen geben an ob ein Benutzer für ein Attribut eines Objekt-Typs schreibend oder lesend zugreifen darf. Dabei wird ebenfalls der Zustand der Prozessinstanz berücksichtigt. Die Schreibrechte lassen sich zusätzlich in obligatorische und optionale Schreibrechte unterteilen. Wenn ein Benutzer eine Berechtigung zur Weiterschaltung einer Transition besitzt, darf er die explizite Transition aktivieren und den Prozess in einen Folgezustand weiterschalten.

Aus den Berechtigungen für Attribute lässt sich eine Rechtetabelle (Abbildung 15) für jeden Objekt-Typ ableiten. Anhand dieser Rechtetabelle können zur Laufzeit dann automatisch Formulare zur Eingabe der Daten erzeugt werden. Es muss lediglich geprüft werden, um welchen Objekt-Typ es sich handelt, in welchem Zustand sich der zugehörige Mikro-Prozess befindet, welcher Rolle der Benutzer zugeordnet ist und für welche Attribute er Schreib- und Leserechte besitzt. In der Abbildung ist eine Zuordnung von Rollen dargestellt. Es wird davon ausgegangen, dass es drei Rollen gibt: Krankenschwester (KS), Arzt und Chefarzt (CA). Ein Attribut kann gelesen werden (R), optional geschrieben werden (OW) oder muss obligatorisch geschrieben werden (MW). Mit Schreibrechten besitzt man automatisch auch Leserechte. Im Zustand *Verwaltung* beispielsweise muss die Krankenschwester die Versicherung des Patienten abgeben. Alle anderen Attribute dürfen nur von Ihr gelesen werden. Ein Arzt darf alle Attribute optional schreiben. Im Zustand *Untersuchung gesetzlich* muss die Krankenschwester den Blutdruck angeben und der Arzt muss eine Diagnose eintragen. Der Chefarzt darf dies alles optional durchführen. Im Zustand *Untersuchung privat* hat eine Krankenschwester keine Rechte.

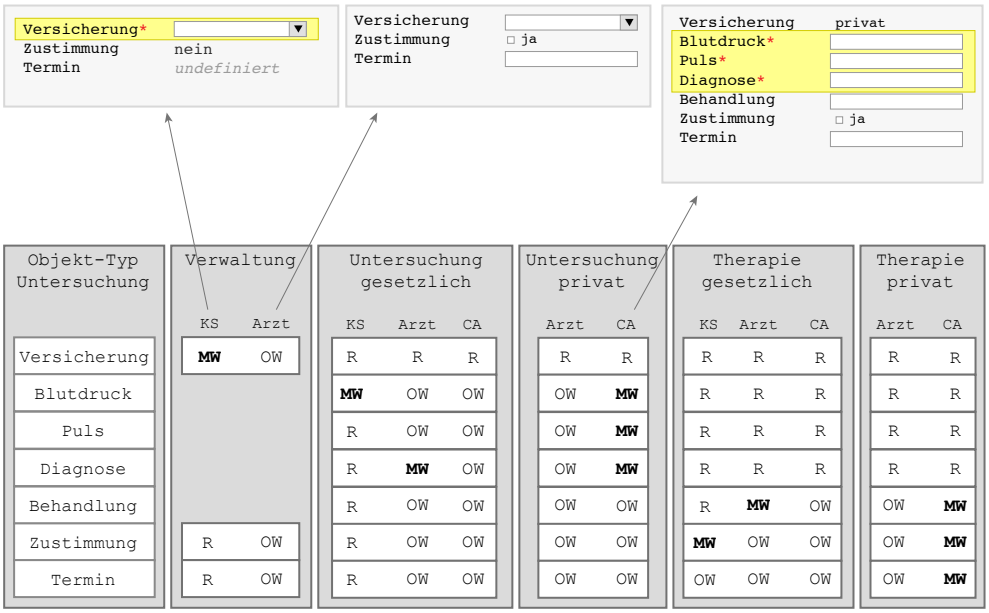

Abbildung 15: Eine Rechte-Tabelle mit drei Rollen für den Objekt-Typ *Untersuchung*

Um dem Benutzer zu signalisieren, dass er ein Attribut obligatorisch schreiben muss, wird im Formular visuell darauf hingewiesen. Wenn die Ausführung eines Mikro-Prozesses zum Beispiel auf einen Schritt trifft, dessen Attribut von ein Benutzer obligatorisch ausgefüllt werden muss, so wird die Formularzeile farblich hervorgehoben. Nach Eingabe des Wertes schaltet der Mikro-Prozess weiter und ein anderes Feld wird hervorgehoben falls die entsprechenden Rechte dafür vorliegen. Da ein Benutzer durch einen Zustandswechsel andere Schreibrechte besitzen kann, werden Zustandswechsel immer explizit durchgeführt um den Benutzer nicht zu verwirren.

Zuständigkeiten werden unterschiedlichen Benutzerrollen zugeordnet und legen fest, wer in einer bestimmten Situation für eine Objekt-Instanz beziehungsweise dessen Prozessinstanz eine Aktivität ausführen muss. Eine Zuständigkeit für einen Zustand bestimmt, welcher Benutzer für Objekt-Instanzen zuständig ist deren Mikro-Prozess-Instanz sich in diesem Zustand befindet. Dies ist notwendig wenn zum Beispiel ein Attribut der Objekt-Instanz zwingend geschrieben werden muss um die Prozessinstanz weiter zu schalten. Eine Zuständigkeit für die Weiterschaltung eines Mikro-Prozess bestimmt den Benutzer der eine explizite Weiterschaltung einer Transition im Prozess durchführen darf, wenn diese die Markierung CONFIRMABLE besitzt.

2.5 Benutzeroberfläche

Nach den fachlichen Details zum Aufbau von PHILharmonicFlows soll nun die Benutzeroberfläche basierend auf dem Usability-Konzept [5] erläutert werden. Es wird der Zugang für einen Benutzer sowie die verschiedenen Arbeitsbereiche und Dialoge erläutert. Das Konzept sieht zunächst vor, dass sich ein Benutzer mit seinen Zugangsdaten anmelden muss. Nach dem Login wird der Benutzer auf eine Übersichtsseite geführt, die unterschiedliche Bereiche zur Betrachtung und Verwaltung von Objekt-Instanzen und Mikro-Prozess-Instanzen vorsieht. Die Bearbeitung von Objekt-Instanzen erfolgt über automatisch erzeugte Formulare auf Basis der vergebenen Rollen, Zuständigkeiten und Berechtigungen.

Nach dem Aufruf der Startseite, wird der Benutzer automatisch zu einem Login-Formular weitergeleitet. Dort hat er die Möglichkeit seine bereits vorhandenen Nutzerdaten einzugeben und sich anzumelden. Bei einer fehlerhaften Eingabe meldet das System einen entsprechenden Fehler und bittet um die erneute Eingabe der Daten. Ein neuer Benutzer hat die Möglichkeit sich über ein Formular für das System zu registrieren.

Die Hauptseite des Systems ist in drei Bereiche unterteilt. Der Aufgabenbereich (*Tasks*) liefert eine Übersicht über alle laufenden Prozessinstanzen sowie über anstehende Aufgaben und Inkonsistenzen in den Daten. Der Datenbereich (*Data*) lieferte eine integrierte Sicht auf die Daten aller Objekt-Instanzen. Der Überwachungs-Bereich (*Monitoring*) soll eine graphische Sicht einer ausgewählten Prozessinstanz beinhalten und Werkzeuge zur Auflösung von Konflikten anbieten. Dieser Bereich ist nicht Teil dieser Arbeit und wird nur der Vollständigkeit halber angegeben.

Der Aufgabenbereich (Abbildung 16) enthält eine Übersicht über alle laufenden Mikro-Prozess-Instanzen in Form von aggregierten Zählern und nach drei Kategorien sortiert: anstehende Aktivitäten (*Todo*), die Zuständigkeiten des Benutzers (*Responsible*) und Inkonsistenzen in den Daten (*Error*). Die Sicht ist zeilenweise nach Objekt-Typen gruppiert und lässt sich um eine weitere Aufgliederung nach Zuständen erweitern.

Für anstehende Aktivitäten werden alle Mikro-Prozess-Instanzen gezählt für die der Benutzer eine Zuständigkeit für den aktuellen Zustand der Instanz besitzt oder eine Zuständigkeit für die explizite Weiterschaltung einer aktivierten externen expliziten Transition (Todo). Für Zuständigkeiten des Benutzers (Responsible) werden alle Instanzen gezählt, für deren Mikro-Prozess-Typ der Benutzer eine Zuständigkeit besitzt. Die Inkonsistenzen setzen sich aus allen Mikro-Schritten zusammen für die eine Inkonsistenz vorliegt (Error).

Der Datenbereich liefert im Gegenzug eine integrierte Sicht auf die Daten eines Prozesses (Abbildung 17). Hier können Werte der einzelnen Attribute von Instanzen eines Objekt-Typs eingesehen werden und Aktivitäten wie das Erstellen, Bearbeiten oder Löschen von Instanzen ausgeführt werden. Für diese Aktivitäten müssen jedoch entsprechende Berechtigungen für den Benutzer hinterlegt werden. Des Weiteren werden Aktivitäten angebo-

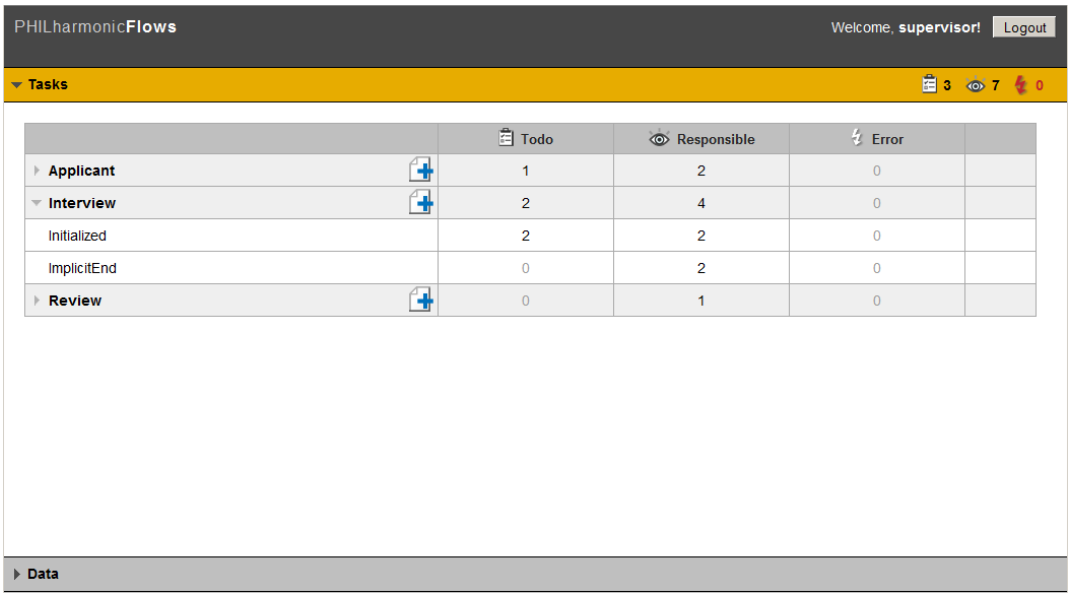

Abbildung 16: Screenshot des Aufgabenbereichs auf der Hauptseite

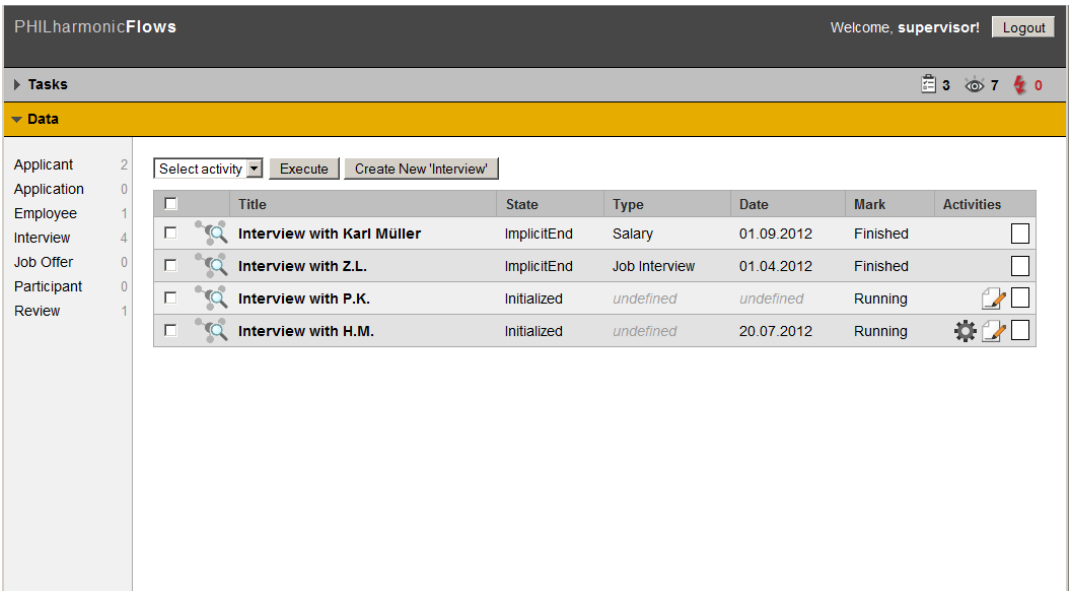

Abbildung 17: Screenshot des Datenbereichs auf der Hauptseite

ten, um den Prozessfluss zu steuern, wie das Weiterschalten expliziter Transitionen oder die Auflösung von Inkonsistenzen. Im Usability-Konzept sind noch weitere Aktivitäten vorgesehen, welche an dieser Stelle aber nicht aufgezählt wurden. Auf der Formularseite (Abbildung 18) wird entsprechend der Benutzerrolle, dem zugehörigen Objekt-Typ sowie dem Zustand der Prozess-Instanz ein Formular erzeugt. Jedes Attribut des zugehörigen Objekt-Typs erhält je nach Rechten ein Eingabefeld, in das der Benutzer Daten eingeben kann [5].

- Für Attribute vom Typ *String* wird standardgemäß eine Textbox angezeigt. Wenn das Attribut eine Wertemenge referenziert, werden die Werte zur Auswahl einzeln aufgelistet. Bei mehr als acht Werten werden sie in einem Auswahlmenü zusammengefasst. Attribute vom Typ *Date* werden ebenfalls durch eine Textbox dargestellt.
- Für Attribute vom Typ *Integer* oder *Decimal* wird ein schmaleres Textfeld angezeigt.
- Für Attribute die *Wahrheitswerte* beinhalten, wird eine Auswahlbox angezeigt. Diese ist standardgemäß nicht angehakt.

Die Darstellung von prozessrelevanten Informationen einer Mikro-Prozess-Instanz erfolgt durch visuelle Markierungen. Ein Attribut für das obligatorisch ein Wert eingegeben werden muss wird mit einem roten Stern hinter dem Attributnamen und einer gelben Umrahmung markiert. Ein inkonsistentes Attribut wird durch eine rot hinterlegte Zeile markiert. Ein Attributwert, welcher außerhalb aller vordefinierten Wertebereiche liegt, wird mit einem roten Ausrufezeichen und einem roten Rahmen um die Zeile markiert. Hinter den Eingabefeldern wird eine Laufzeitmarkierung angezeigt, welche später genauer erläutert wird.

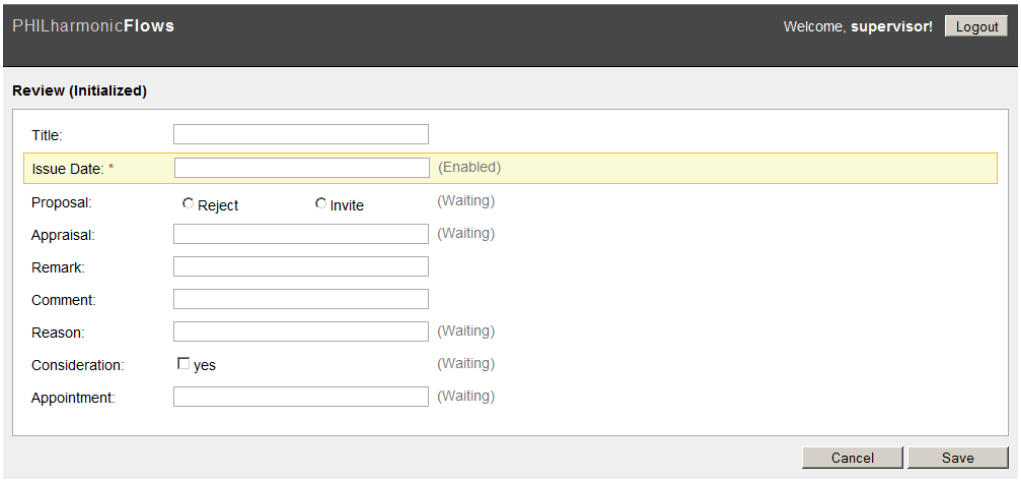

Abbildung 18: Screenshot der Formularseite.

Durch die Eingabe eines Wertes in ein Feld wird die Prozess-Instanz automatisch weitergeschaltet. Dies erkennt man durch den Wechsel der Markierungen beim Verlassen eines Feldes. Änderungen werden nur übernommen, wenn das Formular explizit gespeichert wird. Ein Zustandswechsel erfolgt nur explizit nach dem Speichern des Formulars und nicht automatisch bei der Dateneingabe, da der Benutzer in einem anderen Zustand auch andere Rechte besitzen kann und das resultierende Formular anders aussehen kann, was zu Verwirrungen führen kann.

Für bestimmte Aktivitäten ist eine zusätzliche Interaktion des Benutzers notwendig. So soll der Benutzer explizit bestätigen, dass er eine oder mehrere Instanzen löschen möchte oder eine Auswahl treffen, in welchen Zustand eine Mikro-Prozess-Instanz übergehen soll, wenn es mehrere Möglichkeiten im Prozessfluss gibt. Dafür wird dem Benutzer ein Dialogfenster angezeigt.

- Wenn der Benutzer Instanzen löschen möchte, wird er zuvor gefragt, ob er diese Aktion wirklich durchführen möchte. Nach der Bestätigung schließt sich das Fenster und die entsprechende Instanz wird aus der Datenbank gelöscht.

- Wenn ein Benutzer eine Mikro-Prozess-Instanz weiterschalten möchte weil eine oder mehrere externe explizite Transition aktiviert wurden, wird ihm ein Dialogfenster mit allen möglichen Folgezuständen angezeigt. Nach der Auswahl eines Zustands und der Bestätigung der Aktivität wird die Instanz weitergeschaltet.

- Wenn der Benutzer Inkonsistenzen in den Attributen behandeln möchte, wird ihm ein Fenster mit möglichen Optionen angezeigt. Der Benutzer wählt die passende Option aus und bestätigt die Aktivität.

Es gibt noch weitere Möglichkeiten, die in dieser Arbeit jedoch noch nicht implementiert wurden. Diese sind im Usability-Konzept von Christian Scheb jedoch dokumentiert.

3 Technische Grundlagen

Bei der Erstellung einer Webanwendung wird im Normalfall auf ein Framework zurückgegriffen. *Spring* im Bereich Java, *Microsoft .NET* für C# oder Visual Basic und das *Zend Framework* für PHP sind bekannte Vertreter dafür. Sie stellen umfangreiche Bibliotheken und Werkzeuge zur Verfügung, die den Benutzer bei der Erstellung von Anwendungen unterstützen. Um eine Webanwendung ausführen zu können, muss diese auf einem Server zur Verfügung gestellt werden. Um anfallende Daten persistent speichern zu können, muss zusätzlich eine Datenbanken angebunden werden.

Im folgenden Kapitel wird die technische Infrastruktur von PHILharmonicFlows erläutert sowie einzelne Konzepte des verwendeten Frameworks vorgestellt, die für die Implementierung verwendet wurden.

3.1 PHILharmonicFlows

Bei der Modellierung der Daten- und der Prozessstruktur sowie bei der Integration von Benutzern fällt eine große Menge an Daten an. In vorangegangenen Arbeiten bezüglich der Modellierung wurde festgelegt, dass einzelne Elemente wie Mikro-Prozess-Typen oder deren Mikro-Schritte und Zustände sowie Objekt-Typen und Attribute in jeweils eigene Tabellen abgelegt und über Fremdschlüssel referenziert werden (3. Normalform). So lassen sich Redundanzen vermeiden und die Konsistenz der Daten besser gewährleisten. Für diese Daten wurde bereits eine umfangreiche Typdatenbank evaluiert [6].

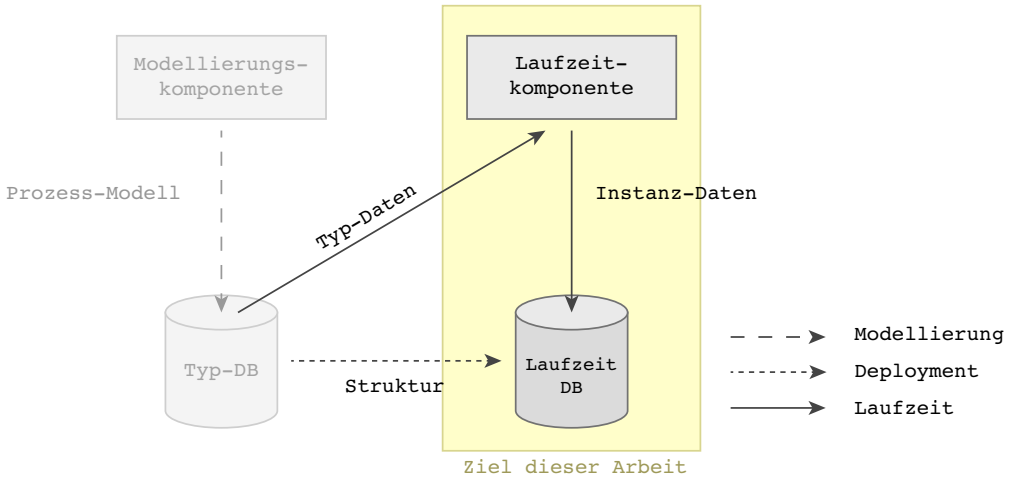

Abbildung 19: Schema der Infrastruktur von PHILharmonicFlows.

Zur Laufzeit fallen ebenfalls Daten an, die nun entsprechend gespeichert werden sollen. Die Menge der Daten ist jedoch weit größer als bei der Modellierung, da zur Laufzeit für

jeden modellierten Typ beliebige viele Instanzen erzeugt werden können. Die Struktur dieser Daten ergibt sich aus den Werten in der Typdatenbank. Bei der Modellierung wird zum Beispiel festgelegt, dass es einen Objekt-Typ *Untersuchung* gibt und dass er die Attribute *Versicherung*, *Blutdruck*, *Puls*, *Diagnose*, *Behandlung*, *Zustimmung* und *Termin* besitzt. Dafür wird in der Typdatenbank ein Eintrag in der Tabelle *Objekt-Typ* erzeugt und für jedes Attribut ein Eintrag in der Tabelle *Attribut* mit einer Referenz auf den Objekt-Typ. Für einen Mikro-Prozess-Typ werden Zustände, deren Schritte und Werteschritte sowie die Transitionen festgelegt. Zur Laufzeit müssen dann Datenbanktabellen existieren, welche die Ausprägungen aller erzeugten Instanzen adäquat speichern können. Die Struktur der Datenbank soll zudem für einen schnellen Zugriff auf die Daten sorgen.

Um den Zugriff auf Instanzdaten eines Objekt-Typs zu beschleunigen, ist es sinnvoll, sie in einer Tabelle abzulegen, statt sie wie die Daten der Typdatenbank normalisiert abzulegen. Dadurch beschleunigt sich der Zugriff, da keine komplexen Anfragen über mehrere Tabellen benötigt werden. Die Tabelle trägt den Namen des Objekt-Typs und die Spalten der Tabelle speichern die unterschiedlichen Ausprägungen der Attribute. Da die Erzeugung einer Objekt-Instanz immer mit der Erzeugung einer Mikro-Prozess-Instanz einhergeht, werden darin zusätzlich Ausprägungen der zugehörigen Mikro-Prozess-Instanz gespeichert. Dabei handelt es sich um die aktuelle Laufzeitmarkierung und den aktuell aktivierten Zustand des Mikro-Prozesses. Reine Typinformationen der Elemente werden nicht in der Laufzeitdatenbank gehalten, sondern aus der Typdatenbank abgefragt wenn sie benötigt werden. Die Erzeugung (*Deployment*) der *Laufzeitdatenbank* muss zeitlich gesehen zwischen Modellierung und der Laufzeit erfolgen, da ihre Struktur erst nach der Modellierung bekannt ist.

Da ein Zugriff auf Daten anderer Modelle nicht vorgesehen ist und somit ein Prozessmodell und dessen Daten in sich abgeschlossen ist, können Instanzen unterschiedlicher Modelle in unterschiedlichen Datenbanken abgelegt werden. So lässt sich die Zahl der erzeugten Instanzen innerhalb einer Datenbank zumindest auf ein einzelnes Datenmodell reduzieren. Ein weiterer Vorteil ist, dass sich somit einzelne Modelle auf unterschiedlichen Datenbankservern verteilen lassen, anstatt eine Datenbank mit allen Datenmodellen zu verteilen.

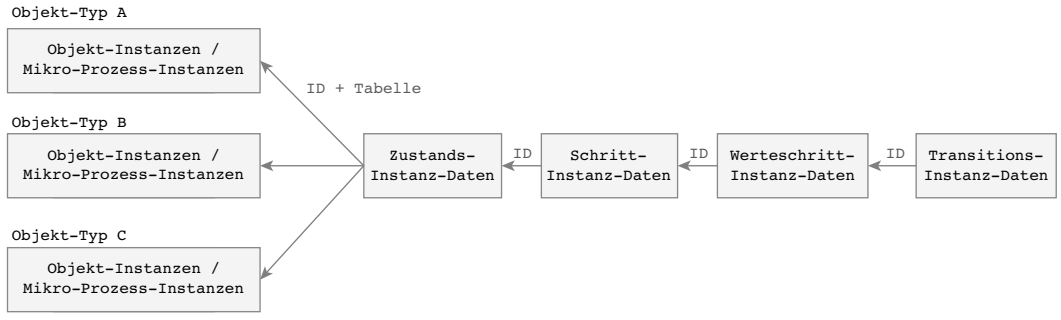

Abbildung 20: Struktur der Instanztabellen zur Laufzeit

Da sich die Struktur der Instanzdaten aller Prozesselemente wie Zustände oder Mikro-Schritte für unterschiedliche Mikro-Prozess-Typen nicht verändert, können deren Ausprägungen prozessübergreifend in einer Tabelle gespeichert werden. Es gibt also eine Tabelle für Ausprägungen von Zuständen, Schritten, Werteschritten und Transitionen, egal von welchem Mikro-Prozess-Typ sie abstammen (Abbildung 20). Sie speichern neben der aktuellen Laufzeitmarkierung auch eine Referenz zu ihrem Typ in der Typdatenbank. Zusätzlich müssen Ausprägungen eines untergeordneten Elements das übergeordnete Element über dessen Instanz-ID referenzieren um die Zugehörigkeit der Elemente zur Laufzeit bestimmen zu können. Eine Ausprägung eines Schrittes speichert zum Beispiel immer die ID einer Ausprägung des zugehörigen Zustands. Um einen Zustand der korrekten Prozess-Instanz zuordnen zu können, muss zusätzlich zur ID der Name des Objekt-Typs gespeichert werden, um zur Laufzeit auf die korrekte Tabelle zugreifen zu können.

3.2 Datenbank

Die Datengrundlage bildet das relationale Datenbankmanagementssystem Microsoft SQL Server 2008. Mit ihm ist es möglich, eine große Menge an Datenbanken zu verwalten und mit Anfragen oder gespeicherten Prozeduren Daten aus der Datenbank abzufragen, zu löschen, zu ändern oder neue Datensätze einzufügen. Es bietet zudem eine nahtlose Integration in Visual Studio und das .NET-Framework.

3.2.1 Datenbankindex

Ein Index ist eine Datenstruktur, die den Zugriff und das Sortieren von Datensätzen in einer Datenbank beschleunigt. Sie besteht aus einer Menge von Verweisen auf eine Datenbanktabelle und sorgt dafür, dass die Tabelle bei einer Anfrage nicht sequentiell durchsucht werden muss. In der Laufzeitdatenbank kommen ebenfalls Indexe zum Einsatz. Dabei werden jeweils die Primärschlüssel (ID) der Datenbanktabellen indiziert, um den Zugriff zu beschleunigen.

3.2.2 Prepared Statements

Für die Eingabe und das Abrufen von Daten aus der Datenbank kommen ausschließlich vordefinierte Statements (*Prepared Statements*) zur Anwendung. Dabei handelt es sich um Templates, in denen statt einer Variable, ein Parameter angegeben ist. Zur Laufzeit werden den Parametern dann die tatsächlichen Variablenwerte zugeordnet und an den Datenbankserver geschickt. Dieser setzt die Werte ein und führt die Anfrage aus und liefert gegebenenfalls das Ergebnis zurück.

Der Vorteil dabei ist der erhöhte Schutz vor sogenannten SQL-Injections, mit denen es möglich ist weitere Daten aus der Datenbank auszulesen was natürlich nicht erwünscht ist. Normale Anfragen werden als einfache Zeichenkette an den Datenbankserver übertragen und ausgeführt. In ihr können neben eingegebenen Werten aus einem Formular aber auch noch weitere, unerwünschte SQL-Befehle hinterlegt werden. Der Datenbankserver führt die Anfrage einfach aus, egal ob sie modifiziert wurde oder nicht. Wenn die Anfrage jedoch interpretiert auf dem Server liegt, kann sie auch nicht ohne Weiteres verändert werden. *Prepared Statements* können außerdem zu einem Geschwindigkeitsvorteil führen, wenn die Abfrage häufig ausgeführt wird.

3.3 Framework

Als Framework für die Erstellung der webbasierten Komponente wurde das auf dem Microsoft .NET basierende Framework ASP.NET gewählt. Die Entscheidung ergab sich daraus, dass die Modellierungskomponente bereits in Visual Studio 2010, basierend auf einem SQL-Server zur Datenhaltung implementiert wurde. Das Framework stellt eine umfangreiche Klassenbibliothek (*Framework Class Library* [8, 9]) für Webanwendungen zur Verfügung, die zum Beispiel vorgefertigte Elemente zur Gestaltung dynamischer und benutzerfreundlicher Anwendungen bietet. Das sind unter anderem Steuerelemente, die auf der AJAX-Technologie (Asynchronous Javascript and XML) basieren und es erlauben nur bestimmte Teile einer Seite neu zu laden.

Visual Studio 2010 ist eine Entwicklungsumgebung von Microsoft für unterschiedliche Programmiersprachen wie C# oder Visual Basic. Dabei können neben klassischen Desktopanwendungen auch Webapplikationen entwickelt werden. Für ASP.NET Anwendungen wird zusätzlich ein integrierter Webserver für Debugging-Aufgaben geliefert. Der Zugriff auf eine Datenbank im zugehörigen Datenbankserver ist ebenfalls in die Oberfläche integriert.

ASP.NET (Active Server Pages .NET) ist eine serverseitige Technologie zur Erstellung von komplexen Webanwendungen, basierend auf dem Microsoft .NET-Framework, und wird aktuell in der Version 4.0 verwendet. Die Grundlage sind `WebForms`, welche einzelne Seiten für Inhalte repräsentieren. Abgeleitet werden sie von einer Basisklasse `Page`. In ihnen können `WebControls` oder `UserControls` sowie regulärer (X)HTML-Code eingebunden werden, die mit auf dem Server hinterlegter Logik verarbeitet und durch eine Rendering-Funktion der Page-Klasse als (X)HTML-Code im Browser dargestellt werden (Abbildung 21). Ein `WebControl` ist ein vordefiniertes Steuerelemente für eine Webseite. Dabei kann es sich um eine Textbox für ein Formular oder kompliziertere Konstrukte wie eine Tabelle handeln, die bestimmte Daten aus einer Datenbank anzeigt und Basisfunktionen zum Erstellen, Bearbeiten und Löschen zur Verfügung stellt.

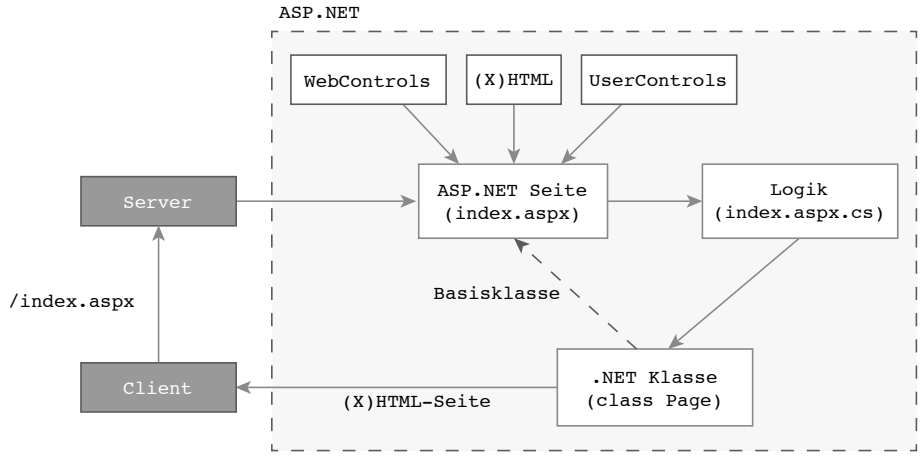

Abbildung 21: Schematische Funktionsweise von ASP.NET

3.4 Verwendete Konzepte

Im Folgenden sollen die Konzepte vorgestellt werden, die bei der Implementierung des Prototyps der Laufzeitkomponente zum Einsatz gekommen sind.

3.4.1 Steuerelemente

Micorsoft .NET bietet eine umfangreiche Sammlung von Elementen an, um eine Anwendung zu steuern. Diese leiten sich von der Klasse `Control` ab und sind vom Typ `WebControl` (Abbildung 22). Die Besonderheit daran ist, dass sie neben Standardattributen wie ID, Name oder Größe auch Attribute enthalten können die für eine Webanwendung relevant sind, wie die Angabe einer Stylesheet-Klasse (CSS) oder einer URL für Verlinkungen oder Aktionen.

Einfache `WebControls` definieren Elemente wie Container (die wiederrum `WebControls` enthalten können), Formularfelder, Schaltflächen, Text oder Hyperlinks. Komplexere Steuerelemente wie eine `GridView` dagegen basieren auf einer Datenquelle (wie zum Beispiel einer bestimmten Datenbanktabelle). Dafür wird dem `GridView` eine `SqlDataSource` zugeordnet, welche ebenfalls ein `WebControl` darstellt. Dieser `SqlDataSource` wird eine Datenbankverbindung und Befehle für SELECT, INSERT, UPDATE und DELETE zugeordnet. Nach dem Binden der Daten an die `GridView`, kann diese die Daten in einer Tabelle darstellen und die gewünschten Operationen auf den Daten ausführen, ohne dass dafür weiterer Code geschrieben werden muss. Durch die Verwendung von Stylesheets (CSS) kann das Aussehen der `WebControls` beliebig angepasst werden. In Tabelle 12 befindet sich eine Auflistung der am häufigsten verwendeten Elemente.

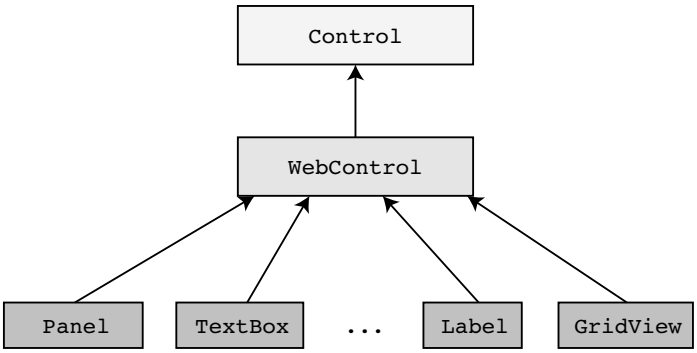

Abbildung 22: Klassenhierarchie einer `WebControl` [8].

WebControl	Erklärung
`Label`	Ein einfacher Text
`TextBox`	Ein Eingabefeld für beliebige Zeichenketten.
`Panel`	Ein einfacher Container zur Gruppierung von Elementen.
`DropDownList`	Eine Auswahlliste, aus der ein Element ausgewählt werden kann.
`Button`	Eine normale Schaltfläche zum Auslösen eines Events auf dem Server.
`ImageButton`	Eine Schaltfläche in Form eines Bildes.
`LinkButton`	Eine Schaltfläche in Form eines Textes.
`ListView`	Eine Listenansicht, basierend auf einer Datenbank oder einem Datenelement.
`GridView`	Eine Tabellenansicht, basierend auf einer Datenbank oder einer anderen Datenquelle.
`SqlDataSource`	Regelt die Kommunikation zwischen einer Datenbank und einem datengebundenen WebControl.

Tabelle 12: Standard-Steuerelemente von ASP.NET [8, 9]

Ein benutzerdefiniertes Steuerelement ist ein `UserControl` und dient dazu, eine Menge von `WebControls`, die eine logische Einheit bilden (zum Beispiel ein Formular mit zwei Schaltflächen) zu kapseln, und als einzelnes Steuerelement zu verwalten. Somit lassen sich komplexe Strukturen mit einer benutzerdefinierten Logik aufbauen und beliebig oft wiederverwenden.

Bei einer Interaktion des Benutzers mit einem Steuerelement, bei dem ein Aufruf zum Server erfolgt (zum Beispiel einem `Button`), wird ein entsprechender `EventHandler` aufgerufen der dann die entsprechend hinterlegte Logik ausführt. Bei bestimmten `WebControls`, wie zum Beispiel einer `GridView` wird dabei auf ein sogenanntes `Command` zurückgegriffen (Abbildung 23). Diese bestehen aus einem Namen (`CommandName`) und optional aus einem Parameter (`CommandArgument`). Das Besondere an einem `Command` ist, dass es bezüglich eines `WebControls` von einem `EventHandler` bzw. `CommandHandler` aufgefangen wird und anhand des Namens unterschiedliche Programmlogik ausgeführt werden kann. Das `GridView` besitzt bereits fertige `Commands` zum Bearbeiten und Löschen von einzelnen Zeilen. Bei komplexeren Aufgaben ist es jedoch nötig, benutzerdefinierte `Commands` zu erstellen, wenn man zum Beispiel beim Erzeugen eines neuen Eintrags in der `GridView` ein benutzerdefiniertes Formular verwenden möchte. Das `CommandArgument` könnte zum Beispiel der Index der Zeile sein, für die das `Command` ausgeführt werden soll.

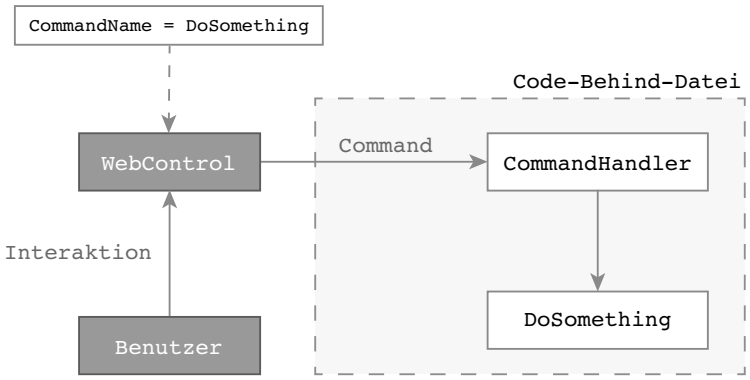

Abbildung 23: Schema eines `Command` in einem Steuerelement

3.4.2 Code-Behind-Dateien

ASP.NET verwendet ein sogenanntes Code-Behind-Modell. Dabei wird für jede Seite eine Datei erzeugt, in der dann benutzerdefinierte Logik hinterlegt werden kann. Es können `EventHandler` für unterschiedliche Events eingeführt werden oder die Anzeige der Seite verändert werden, indem zum Beispiel ein neues `WebControl` hinzugefügt wird oder vorhandene verändert werden. Der Vorteil dabei ist, dass Fehler früh entdeckt werden können, da der Code bereits vor Aufruf der Seite kompiliert wird und nicht erst zur Laufzeit. Dieser Ansatz ist ähnlich zum Modell-View-Controller (MVC) Konzept, wobei der

Controller die Code-Behind-Datei darstellt und die Seite eine View. Die in MVC übliche Umleitung von einer URL auf einen Controller (Routing) ist hier leicht abgewandelt umgesetzt, indem eine Interaktion mit einem Steuerelement zum `EventHandler` in der Code-Behind-Datei umgeleitet wird. Dieser eventbasierte Ansatz wird als Model-View-ViewModel (MVVM) bezeichnet [8, 9].

Das Prinzip wird in Abbildung 24 verdeutlicht. Beim Laden der Seite wird das Label in der Code-Behind-Datei mit dem Text *Title* belegt. Bei einem Klick auf den Button in der Seite wird etwas mit dem Wert in der Textbox berechnet.

demo.aspx

```
<html>
<head>
   <title>Demo</title>
</head>
<body>
  <form>
    <asp:Label ID="Title" /><br/>
    <asp:TextBox ID="Box" /><br/>
    <asp:Button ID="Button"
      OnClick="Button_Click" />
  </form>
</body>
</html>
```

demo.cs

```
public class Demo : Page {

  private void On_Load()
  {
    Title.Text = "Title";
  }

  public void Button_Click()
  {
    do_something(Box.Value);
  }
}
```

Abbildung 24: Eine Inhaltsseite mit zugehöriger Code-Behind-Datei

3.4.3 AJAXControl-Toolkit

AJAX (dt.: Asynchrones Javascript und XML) ist ein Konzept, welches die asynchrone Übertragung von Informationen zwischen einem Browser und einem Webserver beschreibt. Mit ihm ist es möglich, Daten zu senden oder zu empfangen, ohne dafür die Webseite selbst erneut zu laden, wie es bei einem regulären Aufruf des Servers der Fall ist (Abbildung 25). Mit AJAX ist es außerdem möglich, nur bestimmte Bereiche einer Webseite zu aktualisieren.

Auf diesem Konzept basiert auch das `AJAXControlToolkit`. Es stellt unterschiedliche AJAX-basierte `WebControls` zur Verfügung um Standard-WebControls um Funktionalität für AJAX zu erweitern (Tabelle 13). Dafür muss lediglich ein `ToolScriptManager` in die Seite eingebaut werden, welcher unter anderem für die Verwaltung der anfallenden asynchronen Aufrufe des Servers zuständig ist [9].

Um eine benutzerfreundliche Webseite mit dynamischem Inhalt zu implementieren, ist es oft sinnvoll, nur bestimmte Teile einer Seite zu aktualisieren, da sich oft nur geringe Anteile einer Webseite verändern. Dies ist mit einem `UpdatePanel` aus dem AJAX-

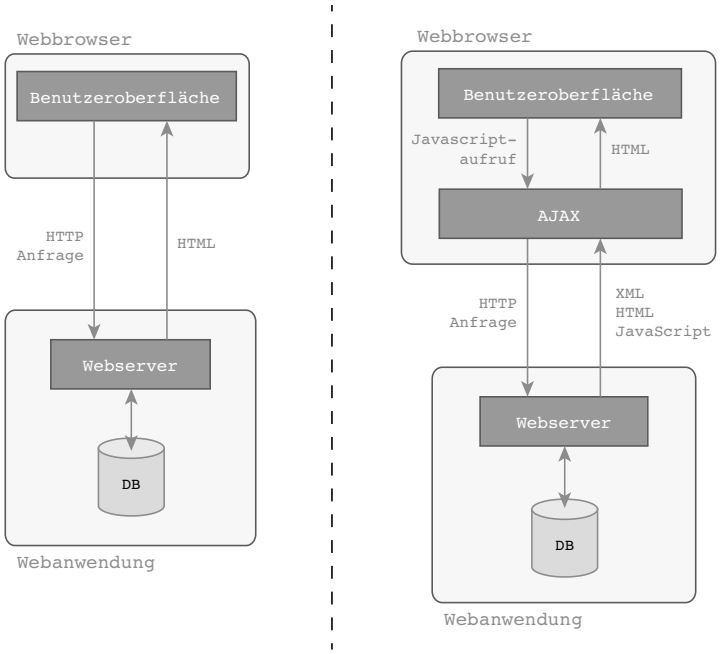

Abbildung 25: Schema einer Webanwendung ohne bzw. mit AJAX-Unterstützung

ControlToolkit möglich. Ein `UpdatePanel` ist im Allgemeinen ein normales `Panel`, hier jedoch mit AJAX-Funktionalität. Durch einen Trigger in Form eines Timers oder einer `WebControl`, wie einem Button, ist es möglich die Inhalte eines `UpdatePanel` asynchron zu aktualisieren. Dafür wird zwar die komplette Seite erneut auf dem Server erzeugt, jedoch nur der Teil gesendet, der sich innerhalb des `UpdatePanels` befindet und aktualisiert werden soll. Das reduziert die Menge an gesendeten Daten über das Netzwerk.

Für die Anzeige von Meldungsfenstern gibt es die Möglichkeit, einen ModalPopUpExtender zu verwenden. Dieser ermöglicht es, die komplette Webseite visuell und funktional zu deaktivieren und ein Fenster mit beliebigem Inhalt anzuzeigen. Während das Fenster geöffnet ist, kann die restliche Webseite nicht bedient werden. Durch das Bestätigen des Dialogs wird das Fenster geschlossen und die Seite wieder aktiviert ohne die Seite neu zu laden.

Zur besseren Strukturierung von Inhalten können bestimmte Bereiche ausgeblendet werden und erst durch einen Mausklick angezeigt werden. Dieses Prinzip findet auch in einem AccordionControl Anwendung. Es besteht aus unterschiedlichen untereinander angeordneten Panels. Jedes `Panel` besteht aus einem Kopf- und einem Inhaltsbereich. Durch einen Klick auf den Kopfbereich wird der zugehörige Inhaltsbereich geschlossen oder geöffnet. Das Besondere an einem `AccordionControl` ist, dass sich immer nur ein Inhaltsbereich auf einmal öffnen lässt. Durch das Öffnen eines anderen Bereichs schließt sich der aktuell geöffnete Bereich automatisch.

AJAX WebControl	Erklärung
`UpdatePanel`	Erweitertes `Panel` zum partiellen Aktualisieren von Inhaltsseiten.
`ModalPopUpExtender`	Erweiterte Funktionalität für ein `Panel` zur Verwendung als Dialog. Dabei wird die komplette Webseite deaktiviert und nur der Dialog angezeigt.
`AccordionControl`	Darstellung von Inhalten in ausklappbaren Panels mit Kopf- und Inhaltsbereich. Dabei wird immer nur ein `Panel` gleichzeitig geöffnet.

Tabelle 13: WebControls aus dem AJAXControl-Toolkit [8, 9]

3.4.4 LINQ

Bei LINQ (*Language Integrated Query*) handelt es sich um eine Komponente zum Zugriff auf Elemente aus einer Datenquelle. Dazu gehören neben XML-Dateien oder Datenbanken auch Arrays und Listen. Der Zugriff kann dabei zum Beispiel in Form einer Abfrage, einer Sortierung, einer Gruppierung oder Aggregation erfolgen. Durch die an SQL angelehnte Syntax, sind LINQ-Abfragen besser lesbar als zum Beispiel ein Konstrukt mit einer *foreach*-Schleife. Jede *foreach*-Schleife lässt sich in eine äquivalente LINQ-Anfrage überführen. Bei einer statischen Datenquelle mit einem festen Schema, ist ein Mapping von Klassen auf Tabellen möglich. Dabei entspricht eine Klasse einer Tabelle und eine Instanzvariable je einer Spalte. Es kann für beliebige Datenbankschemas automatisch eine Klassenstruktur mit den benötigten Methoden erzeugt werden (*LINQ2SQL*) und anschließend mit LINQ-Abfragen auf den Elementen gearbeitet werden. Das Mapping kann in diesem Kontext leider nicht angewandt werden, da das Datenbankschema bei der Erzeugung noch unbekannt ist. Durch den Einsatz von LINQ-Ausdrücken erhöht sich aber Lesbarkeit des Codes und die Geschwindigkeit beim Zugriff auf Datenstrukturen wie einem Array (vgl. Listing 1 und 2).

Listing 1: geschachtelte foreach-Schleife

```
1  var cnt = 0;
2
3  #durchlaufe aeusseres Array
4  foreach(var outer in Array.Values)
5  {
6      #falls mehr als 5 Elemente, weitersuchen
7      if(outer.size >= 5) continue;
8
9      #sonst, durchlaufe inneres Array
10     foreach(var inner in outer)
11     {
12         #erhoehe Zaehler falls Schluessel ungleich 73
```

```
13              if(inner.Key != 73) count++;
14          }
15      }
```

Listing 2: aquivalenter LINQ-Ausdruck

```
1 #suche alle Elemente mit weniger als 5 Kindelementen
2 var cnt = Array.Values.Where(outer=>outer.Count<5)
3
4 #zaehle alle Kindelemente mit Schluessel ungleich 73
5 cnt = cnt.SelectMany(outer=>outer).Count(inner=>inner.Key != 73);
```

Die Datenbasis für die Operation bildet ein zwei-dimensionales Array mit unterschiedlichen Einträgen. Es soll gezählt werden, wie viele Einträge in der zweiten Dimension nicht den Schlüssel `73` besitzen. Außerdem sollen nur Elemente in der zweiten Dimension beachtet werden, wenn das übergeordnete Element weniger als fünf Einträge enthält. In der regulären Umsetzung wird zunächst die erste Dimension des Arrays mit einer for-Schleife durchlaufen und geprüft ob weniger als fünf Elemente vorhanden sind. Ist dies der Fall, werden alle Elemente in der zweiten Dimension in einer weiteren for-Schleife gezählt wenn sie nicht den Schlüssel `foo` besitzen. Der LINQ-Ausdruck reduziert die Menge der Einträge zunächst auf die Elemente mit weniger als fünf Einträgen. Anschließend zählt er alle Elemente, die nicht den Schlüssel `73` besitzen. Der LINQ-Ausdruck musste aus Platzgründen in zwei Zeilen verteilt werden.

4 Implementierung

Nach den fachlichen und technischen Grundlagen von PHILharmonicFlows, folgen nun Details zur Implementierung des Prototyps. Im Anschluss an die Einführung der Architektur werden einzelne Komponenten sowie deren Umsetzung erläutert.

4.1 Architektur

Der Prototyp besteht aus mehreren Elementen mit unterschiedlichen Funktionen. Er besitzt

- dynamische Seiten zur Darstellung der unterschiedlichen Inhaltselemente
- benutzerdefinierte Steuerelemente zur Darstellung der Daten
- Stylesheets (CSS) zur Modifizierung des Aussehens der Steuerelemente und der Seite selbst
- eine umfangreiche Klassenstruktur zur Daten-, Prozess- und Rechteverwaltung.

Die Grundlage bilden der *Laufzeit-Datenbankmanager* (`RuntimeDBManager`) und der *Typ-Datenbankmanager* (`TypeDBManager`), welche mit der Typdatenbank und mit der Laufzeitdatenbank des Prozessmodells kommunizieren und die Daten verwalten. Auf die Typdatenbank wird nur lesend zugegriffen um Typ-Informationen zu einzelnen Instanzen abzurufen. Es folgen der *Prozess-Manager* (`ProcessManager`), der *Formular-Manager* (`FormManager`), der *Rechte-Manager* (`PermissionManager`), der *Aktivitäts-Manager* (`ActivityManager`) und der *Laufzeit-Manager* (`RuntimeManager`). Sie sind für die Verwaltung von Prozess-Instanzen, die Erzeugung von Formularen, die Benutzerintegration und Rechteverwaltung, die Verwaltung von Aktionen auf Instanzen und für die Daten die zur Laufzeit anfallen verantwortlich. Im *Modell-Manager* (`ModelManager`) wird das aktuell geladene Prozessmodell verwaltet (Abbildung 26).

Da der Fokus bei der Implementierung auf der Ausführung von Mikro-Prozess-Instanzen liegt, wird von einem System ausgegangen, dass genau ein Prozessmodell und einen Benutzer verwaltet. In den folgenden Abschnitten wird gezeigt, wie die einzelnen Komponenten aufgebaut sind, wie sie arbeiten und wie sie untereinander kommunizieren.

4.2 Deployment

Ein notwendiger Schritt nach der Modellierung und vor der Ausführung ist das Deployment. Dabei wird anhand der Modellierungsdaten in der Typdatenbank eine Laufzeitdatenbank für das Modell erzeugt, in der die Instanzdaten gespeichert werden. Anders

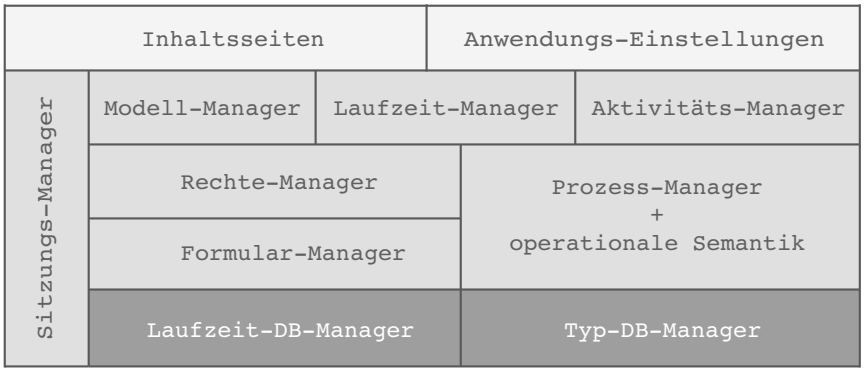

Abbildung 26: Architektur der Ausführungsumgebung von PHILharmonicFlows

als in der Typdatenbank wird für jedes Modell eine eigene Datenbank angelegt, in der alle Instanzen eines Objekt-Typs eine eigene Tabelle besitzen. Ein Vorteil davon ist, dass keine teuren Join-Operationen oder komplizierte Subqueries auf den Tabellen ausgeführt werden müssen und dass zudem die Komplexität bezüglich der Zahl der Instanzen zur Laufzeit auf mehrere Tabellen und Datenbanken verteilt wird. Ein Nachteil ist jedoch, dass sich das Schema der Tabellen (in Anzahl und Namen der Attribute) unterscheidet und grundsätzlich ähnliche Daten nun in unterschiedlichen Tabellen liegen. Diese Denormalisierung der Datenbank führt zudem zu einem erhöhten Zugriff auf die Typdatenbank.

Als erstes wird eine Datenbank erzeugt, die den Namen des Prozessmodells trägt. Der Algorithmus zur Erzeugung durchläuft anschließend die Objekt-Typ-Tabelle und erzeugt für jeden Objekt-Typen eine Tabelle und für jedes dazugehörige Attribut in der Attribut-Tabelle eine Spalte. Die Daten werden als einfache Zeichenketten gespeichert und erst in der Laufzeitkomponente entsprechend ihres Datentyps umgewandelt. Zusätzlich werden Spalten für Daten der zugehörigen Mikro-Prozess-Instanzen angelegt, in denen die aktuelle Laufzeitmarkierung und der aktuell aktivierte Zustand gespeichert werden. Ausserdem wird gespeichert wann und von welchem Benutzer die Instanz erzeugt wurde und wann und von wem sie zuletzt bearbeitet wurde. Die Erzeugung der Tabellen der Mikro-Prozess-Elemente erfolgt immer auf dieselbe Art. Es wird jeweils eine Tabelle für Zustände, Schritte, Werteschritte und Transitionen erzeugt, in der die aktuelle Laufzeitmarkierung und gegebenenfalls Referenzen auf übergeordnete Instanzen in Form des Primärschlüssels gespeichert werden. Das Ganze ist schematisch in Abbildung 27 dargestellt.

Jede erzeugte Tabelle beinhaltet außerdem eine Referenz auf seine Typ-Definition in der Typdatenbank, in Form von dessen Primärschlüssel (ID). Eine Schrittinstanz verweist also nicht nur auf die Zustandsinstanz, der sie zugeordnet ist, sondern auch auf den Schritt in der Typ-Tabelle, von dem sie abgeleitet wurde (Abbildung 28). Somit kann zur Laufzeit z.B. der Name eines Schrittes aus der Typdatenbank abgefragt werden.

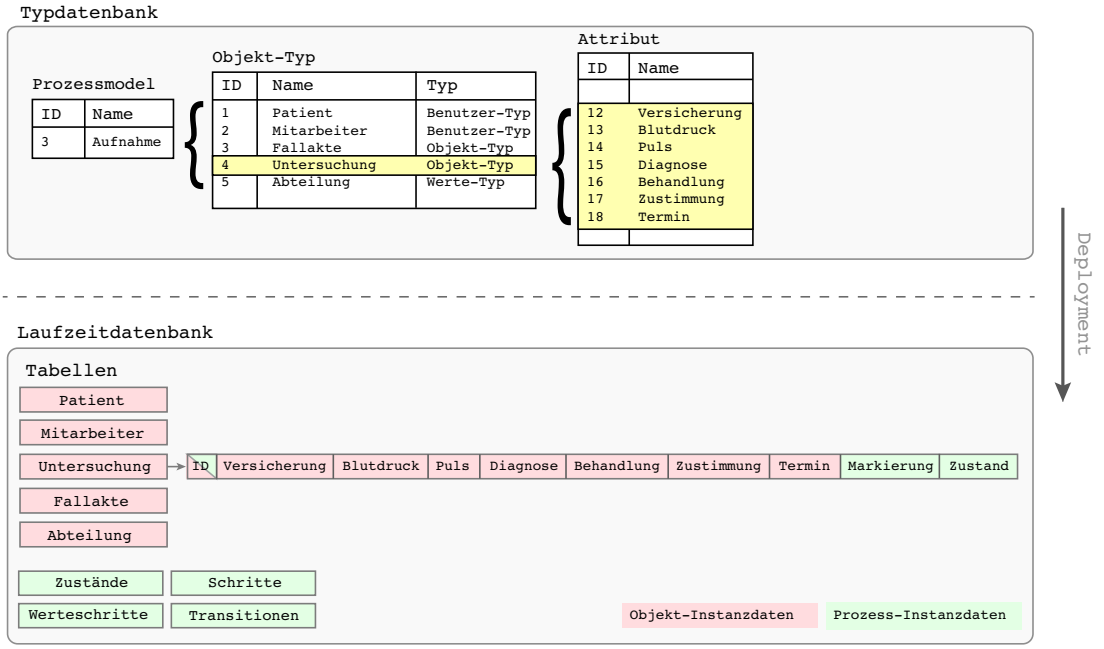

Abbildung 27: Schematischer Ablauf bei der Erzeugung der Laufzeitdatenbank

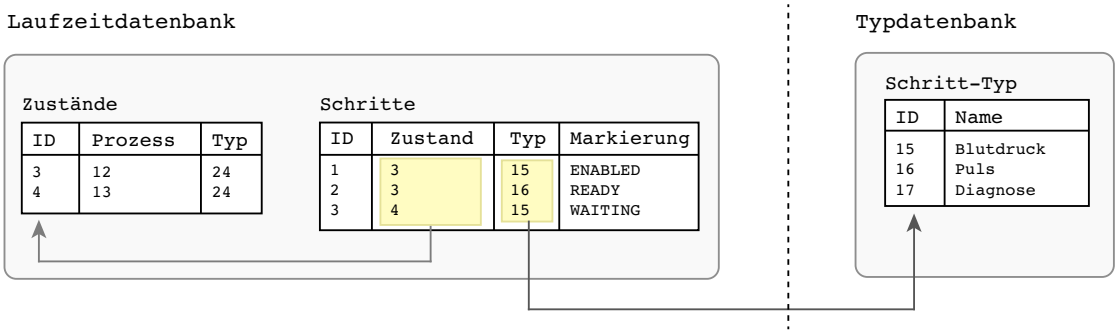

Abbildung 28: Referenzen von Schritten: Innerhalb der Laufzeitdatenbank und zur Typdatenbank

4.3 Anwendungseinstellungen

Für die zentrale Speicherung von Einstellungen und globalen Variablen oder Methoden zur Laufzeit stellt ASP.NET verschiedene Möglichkeiten zur Verfügung. Es gibt eine Konfigurationsdatei für Anwendungseinstellungen sowie eine Datei für globale Methoden oder Variablen auf die zur Laufzeit zugegriffen werden kann.

4.3.1 Konfigurationsdatei

In der Konfigurationsdatei (`web.config`) werden Grundeinstellungen für die Applikation, wie zum Beispiel ihr Name oder das Datumsformat, in Form eines XML-Dokuments gespeichert. Ausserdem kann hier angegeben werden, ob sich Benutzer zum Öffnen von Seiten anmelden müssen und welche Daten in einem Benutzerprofil hinterlegt werden können. In ihr kann auch eine Referenz auf vordefinierte Zeichenketten für die Verbindungen zu einer Datenbank (`ConnectionString`) hinterlegt werden, wenn diese in einer externen Datei gespeichert werden sollen.

4.3.2 Globale Variablen und Methoden

Eine ASP.NET-Webanwendung besitzt eine Datei, in der globale Variablen oder Methoden deklariert werden können (*Globals.asax*). Sie beinhaltet Methoden, die beim Start und Beenden der Anwendung oder bei unerwarteten Fehlern automatisch ausgeführt werden. Hier wurden zusätzlich Konstanten definiert, welche zentral das Ändern von Namen oder Grenzwerten zulassen, denn manche Werte werden neben der Anzeige auch als Identifikator für bestimmte Steuerelemente benutzt. Es ist somit sinnvoller, sie global in einer Konstante zu definieren, um eine durchgehend konsistente Benennung der Elemente zu gewährleisten. Grenzwerte werden in dieser Arbeit verwendet, um lange Zeichenketten innerhalb einer Tabelle auf einen bestimmten Wert zu kürzen. Dies ist erforderlich, um Zeilenumbrüche in einer Zeile zu vermeiden. Sie wurden außerdem verwendet, um die Zahl der angezeigten Attribute in einer Tabelle zu begrenzen. Das ist notwendig, da zu viele Spalten in einer Tabelle fixer Breite keinen Sinn machen, weil diese mit zunehmender Zahl schmaler werden und nur noch wenig Inhalt anzeigen können.

4.3.3 Datenbankverbindung

Um eine Verbindung mit einer SQL-Datenbank herstellen zu können, kann der SQL-Verbindung eine Zeichenkette mit den Verbindungsdaten (`ConnectionString`) übergeben werden. Er enthält den Namen der Datenbank, die Verbindungsart, den Benutzernamen sowie das Passwort des Benutzers. Ein `ConnectionString` können ebenfalls in der Konfigurationsdatei hinterlegt werden. Da sich aber mit jedem Prozessmodell die Datenbank-

verbindung ändert, wurden sie der Übersicht halber in die Datei `connectionStrings.xml` ausgelagert. Auf die Einträge (Listing 3) kann über deren Namen zugegriffen werden. Zusätzlich wurde auch der `ConnectionString` zur Typdatenbank und zur Datenbank der Benutzerverwaltung durch ASP gespeichert.

Listing 3: Eintrag in der connectionStrings.xml

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<connectionStrings>
    ...
    <add name="Recruitment"
      connectionString="server=PHILSERVER;Integrated Security=SSPI;
          database=Recruitment;"
      providerName="System.Data.SqlClient" />
    ...
</connectionStrings>
```

4.4 Inhaltsseiten

Die Grundlage für die Inhaltsseiten bildet ein Seiten-Template (`Page Master`). In ihm werden statische Inhalte wie der Kopfbereich oder eine Fußzeile eingefügt. Bereiche für dynamische Inhalte werden mit einem Platzhalter versehen. Anschließend kann man für einen `Page Master` beliebige Inhaltsseiten erzeugen in denen nur noch die zuvor angegebenen Platzhalter mit Steuerelementen ersetzt werden müssen (Abbildung 29).

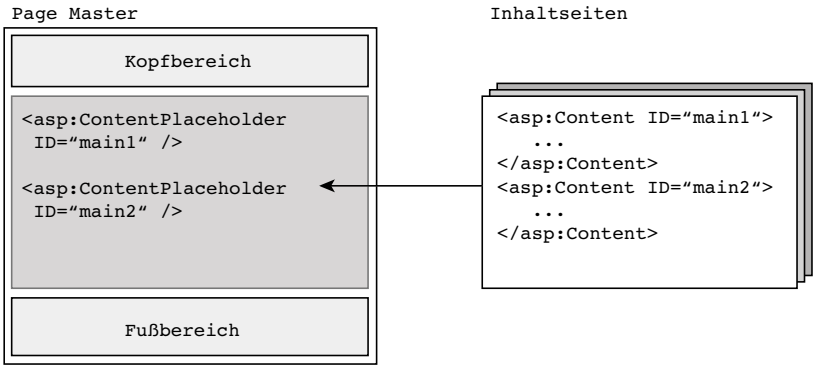

Abbildung 29: Schema eines `Page Master`

Der `Page Master` von PHILharmonicFlows besteht aus der Kopfleiste in der das Logo, der Titel und die Login-Informationen angezeigt werden. Wenn ein Benutzer angemeldet ist, wird eine Begrüßung angezeigt und eine Schaltfläche zum Abmelden. Eine Login-Schaltfläche wird nicht benötigt, da ein unautorisierter Benutzer stets auf die Loginseite umgeleitet wird. Unterhalb der Kopfzeile ist ein Platzhalter für den Inhalt angegeben. Vom Page Master leiten sich vier Inhaltsseiten ab. Die Loginseite, die Registrierungsseite,

die Hauptseite von der man Instanzen und Prozesse verwalten kann (*WorkView*) und eine Formularseite zur Anzeige der Formulare (*FormView*).

4.4.1 Login / Registrierung

Die Loginseite und die Registrierungsseite nutzen für die Registrierung und Autorisierung ein Steuerelement welches von ASP zur Verfügung gestellt wird. Dabei handelt es sich um ein Steuerelement (`Wizard`) das auf einer Klasse zur Mitgliedsverwaltung basiert (`MemberShipProvider`). Diese Klasse sorgt automatisch dafür, dass sich Benutzer für das System registrieren und anmelden können und speichert die Daten dazu in einer von ASP zur Verfügung gestellten Benutzerdatenbank ab. Zusätzlich lassen sich Profile für Nutzer anlegen in denen nutzerspezifische Daten gespeichert werden können. Für die Nutzung dieser Daten werden Methoden angeboten, mit denen sich der Status des Benutzers abfragen lässt, ob er angemeldet ist oder wann er sich zuletzt angemeldet hat. Diese Mechanismen zur Benutzerverwaltung stellt ASP.NET standardgemäß zur Verfügung, eine Benutzung bot sich daher an.

Der `Wizard` stellt nur Datenfelder für Benutzername, E-Mail und Passwort zur Verfügung, welche in der vom System zur Verfügung gestellten Benutzerdatenbank abgelegt werden. Um den neuen Systembenutzer einem bestimmten Benutzer-Typen zuordnen zu können, wurde dem `Wizard` zusätzlich eine Auswahlbox aller verfügbaren Benutzer-Typen des Prozessmodells hinzugefügt. Nach Absendung der Registrierung, wird dann eine Benutzer-Typ-Instanz mit Benutzername und Passwort in der Laufzeitdatenbank angelegt. Ausserdem wird ein Benutzer-Profil für den neuen Benutzer in der Benutzerdatenbank hinterlegt, in dem der Name und die ID des gewählten Prozessmodells gespeichert werden, dem der Benutzer zugeordnet wurde. Nach einem Login kann somit für einen Benutzer das zugehörige Modell geladen werden, ohne dass alle potentiell vorhandenen Laufzeitdatenbanken nach dem passenden Benutzer-Typen durchsucht werden müssen.

Nachdem sich ein Benutzer erfolgreich angemeldet hat, wird eine neue Session für den Nutzer gestartet und alle benötigten Komponenten initialisiert und auf die Hauptseite weitergeleitet. Nach einer erfolgreichen Registrierung wird der Nutzer automatisch angemeldet. Wenn ein Benutzer nicht angemeldet ist, wird er stets auf die Loginseite geleitet und hat keine andere Möglichkeit mit dem System zu interagieren.

4.4.2 Hauptseite

Die Hauptseite (*WorkView*) beinhaltet ein Akkordion-Steuerelement (`AccordionControl`) aus dem AJAX-Toolkit. Die Besonderheit daran ist, dass sich immer nur ein Bereich darin gleichzeitig öffnen lässt (Abbildung 30). Sie zeigt die Hauptbereiche (Aufgaben und Daten) von PHILharmonicFlows an. Welche Bereiche es gibt, wird in der globalen Datei

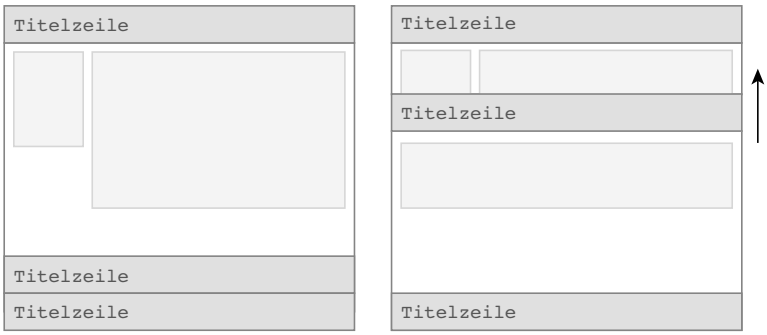

Abbildung 30: Schema der Öffnung eines geschlossenen Bereichs in einem `AccordionControl`. Der zweite Bereich öffnet sich und schließt dabei den ersten geöffneten Bereich darüber.

der Anwendung in einer einfachen Liste gespeichert. Beim Laden der Seite wird für jeden Eintrag in dieser Liste ein `AccordionPanel` für das Akkordion erzeugt, der Kopfbereich mit dem Titel und gegebenenfalls den Zählern bestückt und der Inhalt des jeweiligen Bereichs geladen. Der Inhalt wurde in jeweils ein weiteres Steuerelemente ausgelagert, um bessere Lesbarkeit und Struktur zu gewährleisten. Diese sind jeweils nach ihrem Bereich benannt. Es gibt somit ein Steuerelement für den Aufgabenbereich (*TasksView*) und den Datenbereich (*DataView*).

In der *WorkView* sind außerdem einzelne JavaScripte hinterlegt, um gewisse Funktionen auszuführen die nicht durch das AJAX-Toolkit übernommen werden können, da sie zu speziell sind. Diese Funktionen umfassen:

- die Markierung aller Zeilen über eine einzelne Checkbox in der Kopfzeile der Übersichtsliste in der *TasksView* und der *DataView*.
- das Anzeigen und Verstecken der Aktivitätenliste in der Übersichtsliste der *TasksView* und der *DataView*.
- die Streckung eines `AccordionPanel` auf die gesamte Fenstergröße des Browsers nach dem Laden der Seite.
- das Anzeigen und Verstecken von Untereinträgen in der Arbeitsliste in der *TasksView* durch das Anklicken einer Zeile.

4.4.3 Formularseite

Die Formularseite (*FormView*) kann beliebige Formulare basierend auf einer erzeugten Objekt-Instanz und deren Daten anzeigen. Über sie kann ein Benutzer die Daten für Objekt-Instanzen eingeben und so den Zustand der zugehörigen Prozess-Instanz mani-

pulieren. Sie beinhaltet einige statische Container und Steuerelemente zur Strukturierung sowie ein `UpdatePanel` in das zur Laufzeit die benötigten Formularfelder eingebunden werden. Neben dem Titel beinhaltet die Seite zwei Schaltflächen zum Speichern und Abbrechen, die zur Laufzeit dynamisch ein- oder ausgeblendet werden können (Abbildung 31).

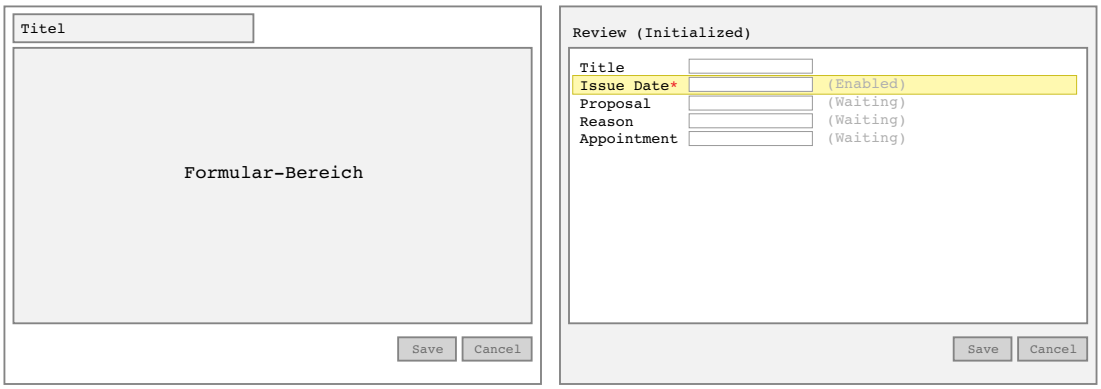

Abbildung 31: Schema der Formularseite mit Beispielformular

Bevor die Formularseite aufgerufen werden kann, muss zuvor der zuständige *Formular-Manager* (`FormManager`) initialisiert werden, damit die korrekten Werte der Instanz aus der Datenbank geladen werden können. In der *FormView* werden dann anhand der Benutzerrechte die zugehörigen Formularfelder erzeugt. Dafür werden die Attribut-Rechte des Benutzers in einer Schleife durchlaufen, für jedes Attribut geprüft ob gelesen oder geschrieben werden darf und dann das entsprechende Eingabefeld samt Titel und aktueller Laufzeitmarkierung aus dem *FormManager* geladen (Abbildung 32). Dieser kommuniziert dafür mit dem Prozessmanager und dem Laufzeit-Datenbankmanager um den Status der zugehörigen Schritte (Markierung, Inkonsistenzen etc.) und eventuell den Attributwert abzufragen. Jede Zeile des Formulars wird in ein eigenes `UpdatePanel` eingebettet um Änderungen in der Markierung anzuzeigen ohne die komplette Seite neu zu laden.

4.5 Benutzerdefinierte Steuerelemente

Benutzerdefinierte Steuerelemente `UserControls` bestehen aus einer Seite in die `WebControls` eingefügt werden können und einer *Code-Behind-Datei* für Programmlogik. Sie ermöglichen es zusammenhängende Elemente in einem Element zu kapseln und an verschiedenen Stellen zur Verfügung zu stellen.

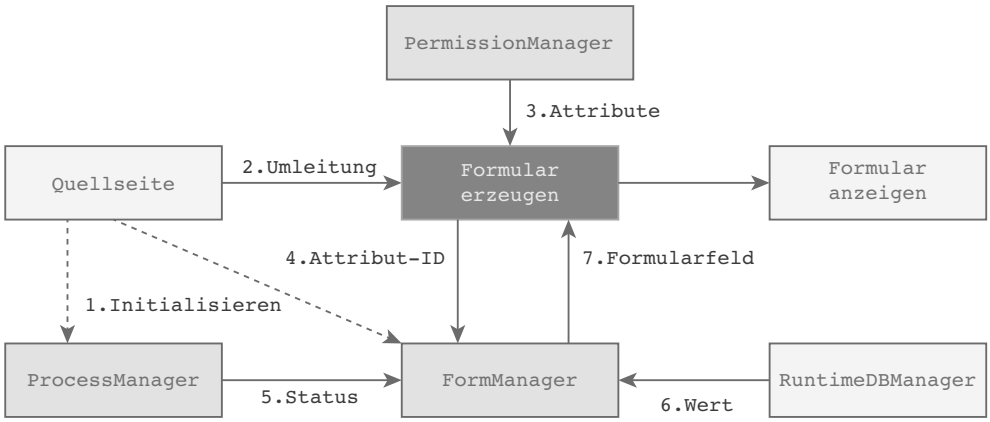

Abbildung 32: Schematischer Ladevorgang der `FormView`

4.5.1 Inhalt des Aufgabenbereichs

Der Inhalt des Aufgabenbereichs (Abbildung 33) wird im *TasksView*-Steuerelement gekapselt und besteht aus zwei Elementen: Einer Arbeitsliste (*HierarchicalGrid*) und einer Übersichtsliste einzelner Objekt-Instanzen (*CustomGrid*). Diese Elemente sind ebenfalls benutzerdefinierte Steuerelemente und werden später genauer erläutert.

Beim Laden der *TasksView* wird das *HierarchicalGrid* erzeugt und dessen Inhalte geladen. Bei einem Klick auf einen der Zähler wird die Übersichtsliste darunter mit den entsprechenden Instanzen angezeigt. Von dieser Liste aus lassen sich Instanzen erzeugen, löschen, bearbeiten oder andere Aktivitäten darauf ausführen.

4.5.2 Inhalt des Datenbereichs

Der Inhalt des Datenbereichs (*DataView*) besteht aus einem Menü und aus einer Übersichtsliste einzelner Instanzen (*CustomGrid*). Beide Elemente sind in ein `UpdatePanel` gekapselt um die Steuerelemente separat aktualisieren zu können (Abbildung 34). Solange noch kein Element aus der Liste gewählt wurde, wird die Übersichtsliste ausgeblendet und ein Hinweis eingeblendet, dass ein Element im Menü ausgewählt werden muss [5].

Das Menü besteht aus einer Liste aller Objekt-Typen des aktuellen Prozessmodells, dabei werden lediglich Objekt-Typen und Benutzer-Typen beachtet. Werte-Typen werden nicht angezeigt. Für jedes Element wird außerdem ein Zähler angezeigt, der angibt, wie viele Instanzen von diesem Typ bereits erzeugt wurden. Nach der Auswahl eines Elements wird die Übersichtsliste aller Instanzen dieses Typs angezeigt über die der Benutzer anschließend einzelne Instanzen bearbeiten kann. Nach dem Löschen oder Hinzufügen einer neuen Instanz, wird der Zähler des entsprechenden Menü-Elements automatisch

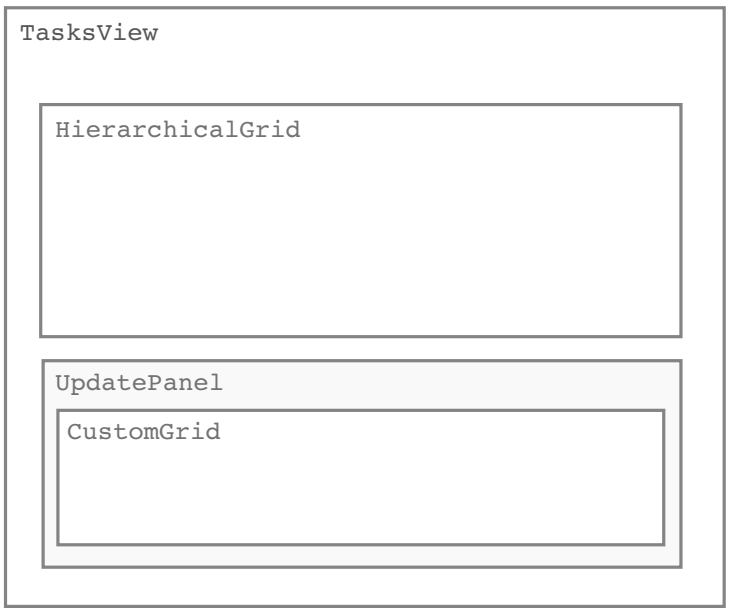

Abbildung 33: Schema des Aufgabenbereichs

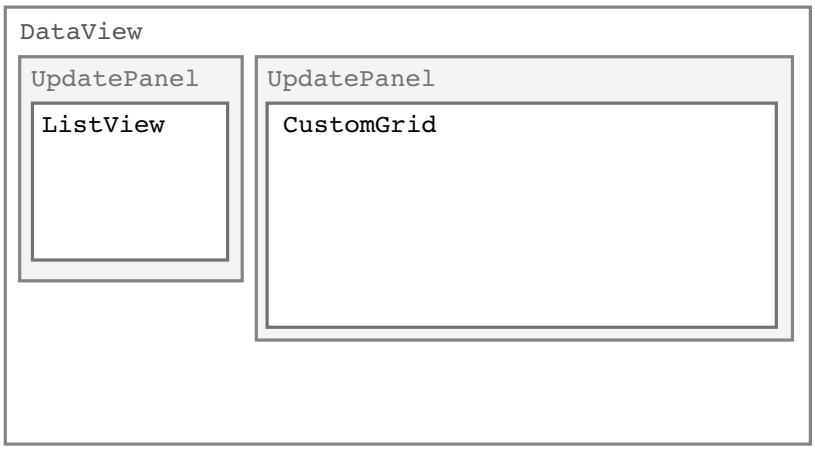

Abbildung 34: Schema des Datenbereichs

aktualisiert [5].

Die Auswahl eines bestimmten Menü-Eintrags löst ein `Event` aus, dessen `EventHandler` die ID des gewählten Objekt-Typs an die Übersichtsliste übergibt. Diese wird mit den Daten der zugehörigen Objekt-Instanzen geladen und angezeigt.

4.5.3 Arbeitsliste

Die Arbeitsliste (*HierarchicalGrid*) ist ein benutzerdefiniertes Steuerelement basierend auf einer `GridView`. Es bietet eine Übersicht über alle Objekt-Typen, den aktuellen Zustand der zugehörigen Prozess-Instanzen sowie die Anzahl erzeugter Instanzen, aufgeteilt nach den drei Kategorien *Todo*, *Responsible* und *Error*.

Um die hierarchische Struktur des Steuerelements zu erreichen, wurden mehrere Elemente vom Typ `GridView` geschachtelt (Abbildung 35). In der übergeordneten `GridView` (das *Eltern-Grid*) werden dafür zunächst die vier Spalten definiert. Die erste Spalte mit dem Titel wird an eine Datenquelle gebunden, um die unterschiedlichen Objekt-Typen aus der Typdatenbank auszulesen. Die anderen drei Spalten sind für die Instanzzähler bestimmt, die erst nach dem Binden der Daten an das *Eltern-Grid* gefüllt werden. Dies ist notwendig, da es sich dabei um aggregierte Instanzdaten aus der Laufzeitdatenbank handelt. Die Angabe unterschiedlicher Datenquellen ist für das Steuerelement jedoch nicht möglich.

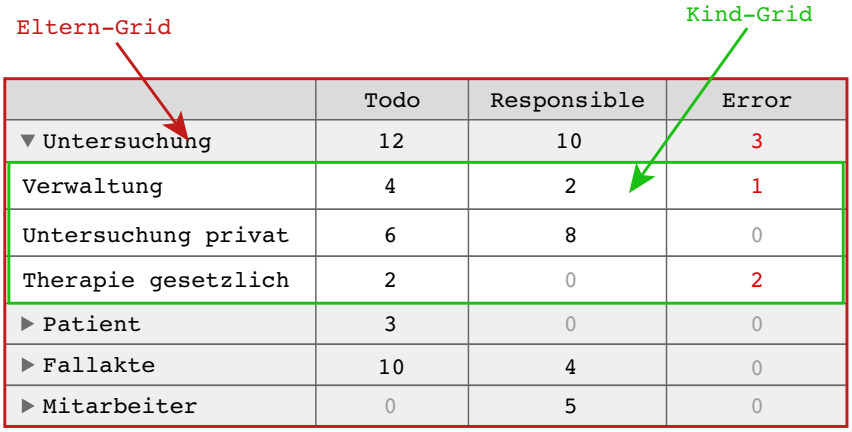

Abbildung 35: Schema der Arbeitsliste

Jeder Zeile des *Eltern-Grids* (Abbildung 35, roter Kasten) wird jeweils eine weitere `GridView` (ein *Kind-Grid*) zugeordnet, die zur Laufzeit die einzelnen Zustände der zugehörigen Prozess-Instanz auflistet (Abbildung 35, grüner Kasten). Es besteht ebenfalls aus vier Spalten. Die erste Spalte listet alle Zustände des zugehörigen Mikro-Prozess-Typs auf und die anderen drei Spalten geben Instanzzähler an, nur dieses Mal zusätzlich nach Zu-

ständen aufgeteilt. Um dies zu erreichen, wird dem *Kind-Grid* ebenfalls eine Datenquelle zugeordnet, die alle vorhandenen Zustände eines Mikro-Prozess-Typs für einen Objekt-Typ aus der Typdatenbank ausliest. Die Zähler werden ebenfalls nach dem Binden der Daten für jede vorhandene Zelle eingefügt.

Die einzelnen Zähler werden bereits nach dem Login des Benutzers geladen und auf dem Server hinterlegt, da sich die Benutzerrechte und Zuständigkeiten während einer Sitzung nicht verändern. Sollte sich die Modellierung der Rechte zur Laufzeit ändern, hat dies erst bei einer erneuten Initialisierung des Rechte-Managers Auswirkungen. Dies geschieht jedoch nur bei einer erneuten Anmeldung des Benutzers am System. Die Instanzzähler werden automatisch aktualisiert, sobald sich eine Instanz ändert.

4.5.4 Übersichtsliste

Die Übersichtsliste (*CustomGrid*) ist ebenfalls ein benutzerdefiniertes Steuerelement. Im Mittelpunkt steht dabei eine `GridView`, in der Instanzdaten eines bestimmten Objekt-Typs angezeigt werden können. Zusätzlich bietet es Aktivitäten an, die auf einzelnen Instanzen ausgeführt werden können. Oberhalb der `GridView` befindet sich eine Auswahlliste mit verfügbaren Massenaktivitäten, die für mehrere Instanzen gleichzeitig ausgeführt werden können und eine Schaltfläche zur Erzeugung neuer Instanzen des angezeigten Objekt-Typs.

Wegen der dynamischen Struktur der Laufzeitdaten, ist es nicht möglich ein fixes Schema anzugeben. Daher müssen die Spalten für die Attribute entsprechend des Objekt-Typs dynamisch angepasst werden, da jeder Objekt-Typ andere Attribute besitzt. Der Anfrage-String über den die Daten aus der Datenbank angefordert werden, muss ebenfalls dynamisch um die Attributnamen erweitert werden. Die restlichen Spalten in der Tabelle sind immer gleich. In der ersten Spalte der `GridView` befindet sich eine `CheckBox`, um die Zeile für eine Massenaufgabe zu markieren. Danach folgen ein Titelfeld, die Attributwerte der Objekt-Instanz, die Laufzeitmarkierung der zugehörigen Prozess-Instanz und der aktuell aktivierte Zustand der Prozess-Instanz. Diese Werte werden über eine Datenquelle geladen und aus der entsprechenden Tabelle der Laufzeitdatenbank ausgelesen. Die letzte Spalte enthält eine Auswahl von Aktivitäten, die für die entsprechende Instanz ausgeführt werden können.

Das *CustomGrid* ist ein `UserControl` und wird sowohl im Aufgabenbereich als auch im Datenbereich verwendet. Deshalb muss es sich dynamisch anpassen können, da nicht immer die gleichen Kombinationen an Instanzen angezeigt werden müssen (Tabelle 14). Es soll in der Lage sein alle Instanzen und Instanzen in einem bestimmten Zustand anzuzeigen. Zusätzlich soll die Menge der Instanzen noch eingeschränkt werden können hinsichtlich der Eigenschaft ob sie Inkonsistent sind, ob der Benutzer Aktionen auf den Instanzen ausführen muss oder ob der Benutzer für die Instanz zuständig ist. Dies erfolgt über unterschiedliche Datenquellen mit unterschiedlichen Anfrage-Strings, welche

anhand verschiedener Merkmale ausgewählt werden. Befindet sich die Übersichtsliste zum Beispiel im Aufgabenbereich und wird ein bestimmter Instanzzähler in der Spalte *Todo* ausgewählt, so wird eine vordefinierte Datenquelle für das *CustomGrid* verwendet, die nur die gewünschten inkonsistenten Instanzen anzeigt.

Bereich auf Hauptseite	Kategorie (Spalte)	Granularitäts-Stufe (Zeile)
Aufgabenbereich	Todo	Objekt-Typ
	Responsible	Objekt-Typ + bestimmter Zustand
	Error	
Datenbereich		Objekt-Typ

Tabelle 14: Kombinationsmöglichkeiten für Instanzen in der Übersichtsliste

Die verfügbaren Aktivitäten lassen sich in zwei Kategorien aufteilen: optional und obligatorisch. Erstere werden in einer Auswahlliste zur Verfügung gestellt, letztere befinden sich in Form von Symbolen außerhalb der Liste. Somit erkennt der Benutzer immer sofort, welche Aktivität(en) er ausführen muss und welche er ausführen kann.

4.5.5 Dialoge

Der Dialog zur Auswahl eines Folgezustands *CommitDialog* wird angezeigt, wenn eine Mikro-Prozess-Instanz explizit weitergeschaltet werden muss. Auch dabei handelt es sich um ein `CustomControl`. In ihm kann der Benutzer auswählen in welchen Folgezustand eine Prozess-Instanz geschaltet werden soll, wenn eine oder mehrere externe explizite Transitionen mit CONFIRMABLE markiert sind. Durch das AJAX-Steuerelement `ModalPopUpExtender` ist es möglich, beim Öffnen des Dialogs die restliche Seite zu deaktivieren, so lange bis der Benutzer einen Folgezustand gewählt hat oder die Aktivität abgebrochen hat. Durch die Bestätigung wird die Instanz weitergeschaltet und alle betroffenen Instanzzähler und Listen aktualisiert.

Vor der Löschung einer oder mehrerer Instanzen wird ein Bestätigungsdialog (*ConfirmDialog*) angezeigt, um sicherzustellen, dass der Benutzer die Instanzen auch tatsächlich löschen möchte. Auch dieses Element ist ein `CustomControl` und verwendet einen *ModalPopUpExtender*. Nach der Bestätigung zum Löschen werden die betroffenen Instanzen aus der Laufzeitdatenbank entfernt und alle betroffenen Ansichten aktualisiert.

Wenn ein Benutzer Inkonsistenzen in der Prozess-Instanz entfernen möchte, kann er in einem Dialog (*InconsistenceDialog*) wählen, ob er den alten konsistenten Wert wiederherstellen möchte oder die Inkonsistenz ignorieren möchte, also den neuen Wert übernehmen möchte. Nach der Bestätigung werden die betroffenen Schritte aktualisiert und der Dialog geschlossen.

4.6 Stylesheets

Das Aussehen und die Platzierung der einzelnen `WebControls` und den Seiten wird in Cascading Style Sheets (CSS) festgelegt. Dazu werden einzelnen Elementen IDs oder Klassen zugeordnet die dann in den CSS-Dateien mit Attributen visuell verändert werden können. Das sorgt dafür, dass keine Gestaltung im Quellcode der Seite selbst vorgenommen wird sondern in ausgelagerten Dateien. Für die unterschiedlichen Bereiche werden für die bessere Übersicht separate Dateien verwendet.

Es muss beachtet werden, dass unterschiedliche Browser diese unterschiedlich darstellen können. Es muss also sichergestellt werden, dass alle Attribute gleichsam unterstützt werden, um Fehler zu vermeiden.

4.7 Managerstruktur

Die Managerstruktur besteht aus verschiedenen Klassen, die jeweils für einen bestimmten Bereich wie die Prozess-Instanz-Verwaltung oder Datenbank-Zugriffe zuständig sind. Sie wurden als statische Klassen implementiert um sicherzustellen, dass nicht mehrere Instanzen von ihnen gestartet werden. Der folgende Abschnitt stellt die einzelnen Manager vor und beschreibt, welche Aufgabe sie besitzen.

4.7.1 Modell-Manager

Der *Modell-Manager* (`ModelManager`) bietet eine Funktion zum Laden eines Prozessmodells aus der Typdatenbank. Dabei wird dem Konstruktor der Name des Modells übergeben, der dann die entsprechende ID aus der Typdatenbank ausliest und speichert. Andere Manager können hier immer das aktuelle Prozessmodell in Form von dessen ID und Name abfragen (Listing 4). Momentan kann nur ein Prozessmodell geladen werden. In weiterführenden Arbeiten können hier noch die Möglichkeiten implementiert werden, mehrere Modelle gleichzeitig zu Laden oder das Modell zur Laufzeit zu wechseln.

Listing 4: Zugriff auf das aktuelle Prozessmodell innerhalb des *Modell-Managers*

```
1 ...
2 # Laden von Zustaenden ueber die ID des aktuellen Modells
3 # im Modellmanager als Parameter der Methode
4 var st = TypeDBManager.GetStatesByModelID(ModelManager.DataModel.Id);
5 ...
```

4.7.2 Sitzungs-Manager

Der *Sitzungs-Manager* (`SessionManager`) hat eine Lese- und Schreib-Methode zum Schreiben und Auslesen von Sitzungs-Variablen. Nach der Anmeldung eines Benutzers wird für ihn automatisch eine neue Sitzung gestartet. Da sämtliche Manager als statische Klasse implementiert wurden, müssen anfallende Daten außerhalb der Klasse gespeichert werden, um sie zwischen Seitenaufrufen zu erhalten. Dafür können andere Manager ihre Daten in die Sitzung des aufrufenden Benutzers schreiben und auslesen (Listing 5). Durch ihn ist es möglich, Daten zwischen unterschiedlichen Seitenaufrufen zu speichern ohne dabei auf eine Datenbank zugreifen zu müssen. Nach einer Abmeldung werden die Daten darin gelöscht und die Sitzung geschlossen.

Listing 5: Speichern einer Variable im *Sitzungs-Manager*

```
1  ...
2  # Get- und Set-Methode fuer eine Variable in einer Klasse
3  # bei dem der Wert  vom SessionManager verwaltet wird
4  public static String ObjectType
5  {
6      #hole den Wert mit dem bestimmten Namen
7      get { return (String) SessionManager.Get("ObjectType"); }
8
9      #setze den Wert unter einem bestimmten Namen
10     set { SessionManager.Set("ObjectType", value); }
11 }
12 ...
```

4.7.3 Laufzeit-Manager

Der *Laufzeit-Manager* (`RuntimeManager`) stellt (indirekt über den Sitzungsmanager) Variablen mit Statusinformationen und andere Laufzeitdaten sowie Methoden zur Verfügung (Abbildung 36). Bei Statusinformationen handelt es sich zum Beispiel um den aktuell geöffneten Bereich auf der Hauptseite und den aktuellen Objekt-Typ (in Form von dessen Typ-ID und Name), der momentan in einer Übersichtsliste im Aufgaben- oder Datenbereich angezeigt wird. Er wird durch die Auswahl eines Objekt-Typs im Datenbereich oder durch die Auswahl eines Zählers im Aufgabenbereich gesetzt. Das ist notwendig, um nach einer Rückkehr von der Formularseite die Übersichtsliste des zuletzt ausgewählten Objekt-Typen anzeigen zu können. Zum aktuellen Objekt-Typ werden außerdem dessen Attribute (in Form von deren Typ-ID, Name und Datentyp) in einer Liste gehalten, um schneller auf Typ-Informationen zugreifen zu können.

Bei den Laufzeitdaten handelt es sich um eine Variable welche die Instanzzähler für die hierarchische Arbeitsliste (*HierarchicalGrid*) im Aufgabenbereich beinhaltet. Sie besteht aus einem mehrdimensionalen Array, das für jeden verfügbaren Objekt-Typen und dessen Zustände einen Zähler für jede der drei Spalten des Steuerelements beinhaltet. Diese

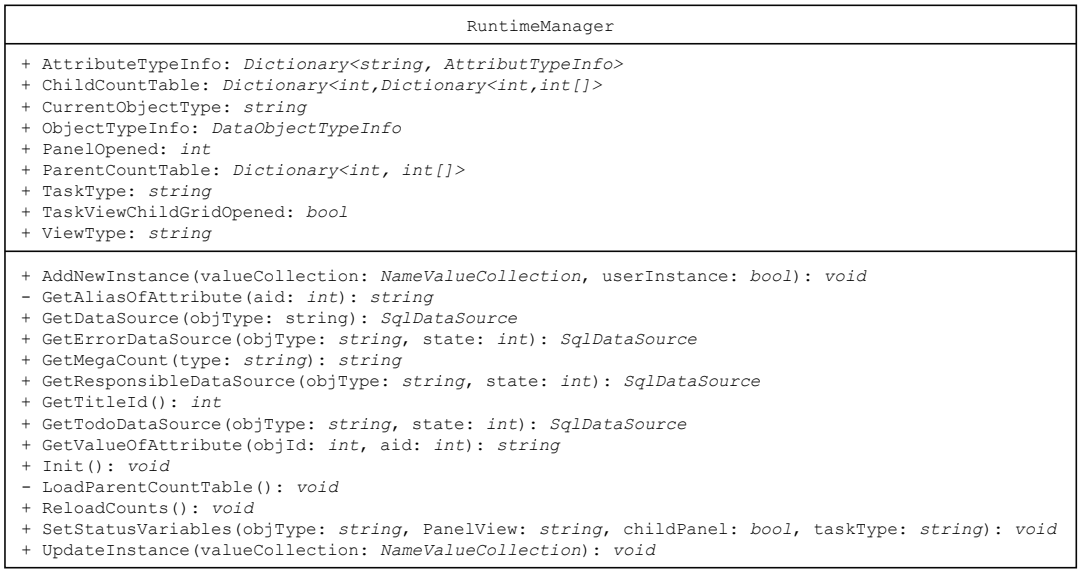

Abbildung 36: Klassendiagramm `RuntimeManager`

Zähler werden aus der Laufzeitdatenbank abgefragt. Die Zähler für einen Objekt-Typen insgesamt werden zur Laufzeit aus den Zählern der einzelnen Zustände des Objekt-Typs berechnet und ebenfalls in einer Variable gespeichert. Es gibt außerdem eine Methode welche die Zählerstände aktualisiert. Sie kommt zum Einsatz wenn eine Aktivität ausgeführt wird, die Einfluss auf die Zähler hat, wie zum Beispiel das Löschen oder Anlegen einer Objekt-Instanz.

Der *Laufzeit-Manager* liefert außerdem die Datenquellen für die Übersichtsliste für Instanzen eines Objekt-Typs (*CustomGrid*), abhängig davon ob sie sich im Aufgaben- oder Datenbereich befindet. Falls sie sich im Aufgabenbereich befindet, muss zusätzlich geprüft werden ob alle oder nur bestimmte Instanzen angezeigt werden sollen, da sich durch die Auswahl eines Instanzzählers im Steuerelement darüber unterschiedliche Kombinationen ergeben. Es muss zwischen den drei Kategorien *Todo*, *Responsible* und *Error* sowie der Granularität unterschieden werden, also ob alle Instanzen des Objekt-Typs oder nur Instanzen in einem bestimmten Zustand angezeigt werden sollen (Tabelle 14).

Über den *Laufzeit-Manager* werden auch Objekt-Instanzen in der Laufzeitdatenbank angelegt oder aktualisiert. Nach der Eingabe von Daten in ein Formular, wird das Array mit den Werten an die entsprechende Methode im *Laufzeit-Manager* übergeben (Listing 6). Dieser entfernt unnötige Systemeinträge aus dem Array, ersetzt den Schlüssel des Arrays mit dem tatsächlichen Spaltennamen in der Datenbanktabelle (*Alias*) und leitet die Erzeugung an den *Laufzeit-Datenbankmanager* weiter. Dieser führt dann die entsprechende Aktion durch. Das *Alias* wird verwendet damit mehrere Attribute in unterschiedlichen Objekt-Typen mit demselben Namen verwendet werden können, ohne unerwünschte Ne-

beneffekte auszulösen.

Listing 6: Erzeugung einer neuen Instanz im *Laufzeit-Manager*

```
...
# Ergebnis-Array
var resultArray = new Array();

#Durchlaufen des unbereinigten Arrays
foreach (string key in oldArray.Keys)
{
    # naechster Eintrag, falls Systemeintrag
    # wie versteckte Felder etc.
    if(systemEntry(oldArray[key])) continue;

    # konvertiere Schluessel zu Integer
    var aid = Convert.ToInt32(key);

    # lade Spaltenname der Tabelle (Alias) ueber Attribut-ID
    var alias = GetAliasOfAttribute(aid);

    #fuege sauberen Eintrag in Ergebnis-Array ein
    resultArray.Add(alias, oldArray[key]);
}

# speichere neuen Eintrag
RuntimeDBManager.CreateNewInstance(resultArray);
...
```

4.7.4 Prozess-Manager

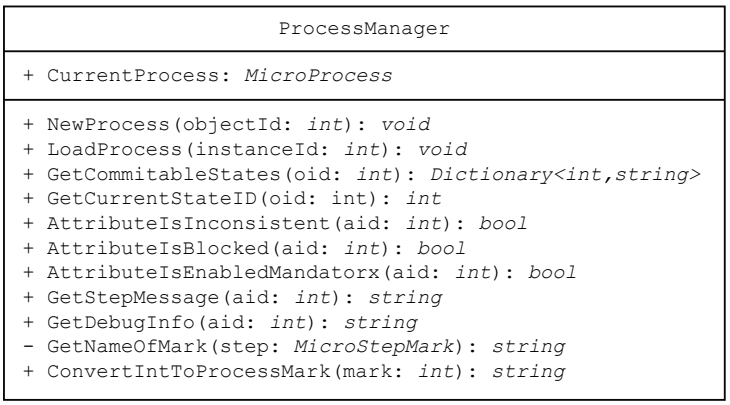

Abbildung 37: Klassendiagramm `ProcessManager`

Die Aufgabe des Prozess-Managers `ProcessManager` ist es, eine einzelne Mikro-Prozess-

Instanz zu verwalten. Diese kann entweder leer erzeugt (wenn eine neue Objekt-Instanz angelegt werden soll) oder aus der Laufzeitdatenbank geladen werden (wenn eine Objekt-Instanz bearbeitet werden soll). Für Letzteres wird zusätzlich die ID der Objekt-Instanz benötigt. Über den `Prozess-Manager` können zudem bestimmte Laufzeit-Eigenschaften abgefragt werden, wie zum Beispiel

- eine Liste aller Zustände, die bei einer expliziten Aktivierung eines Folgezustands aktiviert werden können
- ob ein Schritt (dessen Attribut) konsistent ist oder nicht
- ob ein Schritt (dessen Attribut) die Markierung BLOCKED besitzt
- ob ein Schritt (dessen Attribut) mit ENABLED markiert ist und ob obligatorische Schreibrechte vorliegen
- eine Nachricht in einem werte-spezifischen Mikro-Schritt, die gesetzt wird wenn ein Fehler auftritt
- die Laufzeitmarkierungen eines Attributs

Eine Auflistung aller Methoden ist in Abbildung 37 dargestellt. Diese Eigenschaften sind notwendig um bei der Erzeugung von Formularzeilen im *Formular-Manager* eine entsprechende Markierung anzeigen zu können. Diese wird benötigt, wenn ein Attribut inkonsistent ist, der Wert in keinen Wertebereich innerhalb eines Werteschritts fällt oder ein Attributwert obligatorisch eingegeben werden muss. Für ein Attribut können mehrere Laufzeitmarkierungen abgefragt werden, da es von mehreren Mikro-Schritten in unterschiedlichen Zuständen referenziert werden kann. Die Nachricht enthält einen Beschreibungstext, der auf der Oberfläche benutzt werden kann um dem Benutzer den Fehler zu beschreiben.

4.7.5 Rechte-Manager

Der *Rechte-Manager* (`PermissionManager`) enthält alle Berechtigungen und Zuständigkeiten, die ein Benutzer für alle Instanzen eines Objekt-Typen oder Mikro-Prozess-Typen zur Laufzeit besitzt. Er stellt außerdem Methoden zur Verfügung, um diese zu prüfen (Listing 7). Zusätzlich speichert er die Rolle des angemeldeten Benutzers und dessen Benutzernamen. Eine Übersicht ist in Abbildung 38 dargestellt. Beim Initialisieren wird die Rolle und der Benutzername gesetzt und anschließend alle Berechtigungen und Zuständigkeiten anhand der Rolle aus der Typdatenbank angefordert und gespeichert. Dies findet einmalig nach dem Login statt, da sich Rechte und Zuständigkeiten zur Laufzeit nicht ändern.

Listing 7: Abfragen eines Löschrechts im `PermissionManager` für eine Instanz in einem bestimmten Zustand

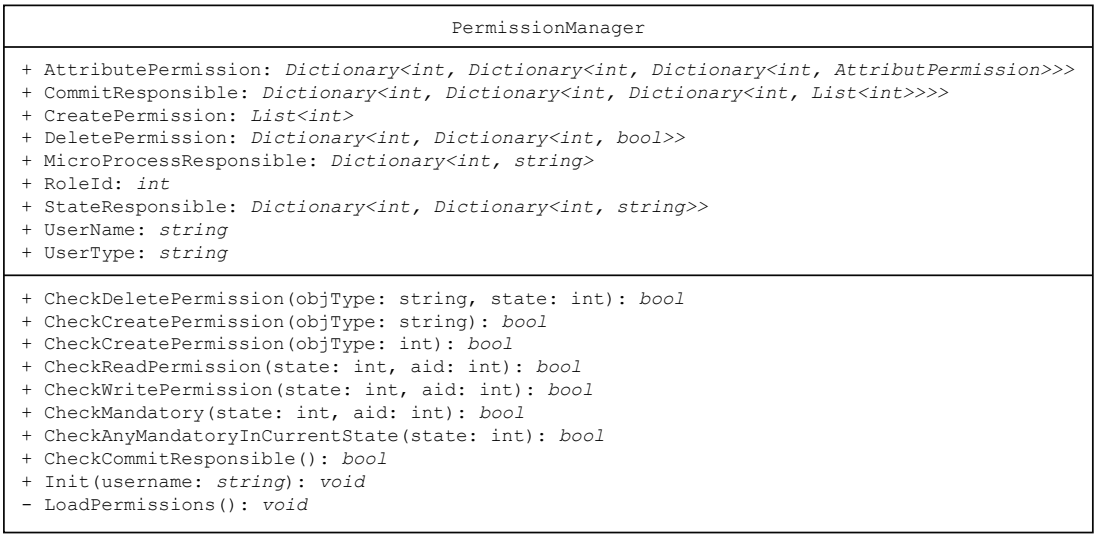

Abbildung 38: Klassendiagramm `PermissionManager`

```
1  # pruefe Loeschberechtigungen fuer eine Instanz
2  # in einem bestimmten Zustand
3  CheckDeletePermission(int oid, int stateId)
4  {
5      #wenn Objekt-Typ-ID und Zustands-ID hinterlegt sind
6      #gebe true zurueck
7      if(DeletePermission[oid][stateId] != null) return true;
8
9      #sonst kein Recht zum loeschen
10     else return false;
11 }
```

Benutzerrechte und Zuständigkeiten werden für Typ-Elemente vergeben und nicht für einzelne Instanzen. Zur Laufzeit muss deshalb vor einer Aktion geprüft werden, ob eine Instanz bestimmte Bedingungen erfüllt um dem Benutzer das Recht auf die Aktion gewähren zu können. Zum Erzeugen einer Instanz eines bestimmten Objekt-Typs, muss für den Benutzer explizit ein Recht für diesen Objekt-Typ zum Erzeugen vorhanden sein. Für das Löschen einer Instanz muss zusätzlich jeder Zustand angegeben werden, in dem der Benutzer eine Instanz löschen darf. Die Bearbeitung einer Instanz erstreckt sich sogar bis hin zu einzelnen Attributen.

Um dies zu erreichen wird für jedes Recht und jede Zuständigkeit ein multidimensionales Array angelegt (Abbildung 39). Jede Dimension des Arrays enthält genau eine Bedingung. Die Existenz aller Bedingungen, wird mit dem Vorhandensein des Rechts oder einer Zuständigkeit gleichgesetzt. In Tabelle 15 werden die unterschiedlichen Bedingungen aufgelistet. Wenn eine Instanz für alle Dimensionen einen Eintrag enthält, darf die Aktion

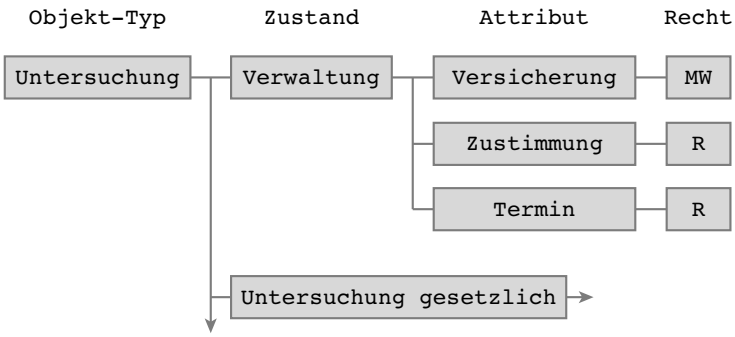

Abbildung 39: Schema eines Array-Eintrags für das Recht zur Bearbeitung von Instanzen. Für die bessere Übersichtlichkeit wurden die tatsächlichen Namen statt der Typ-IDs verwendet.

ausgeführt werden bzw. besitzt der Benutzer die entsprechende Zuständigkeit. Ein Benutzer hat zum Beispiel ein Schreibrecht für ein bestimmtes Attribut einer bestimmten Instanz, wenn im entsprechenden Array folgende Einträge vorhanden sind:

- ID des zugehörigen Objekt-Typs
- Typ-ID des aktuellen Zustands, in der sich die zugehörige Prozess-Instanz befindet
- Typ-ID des betroffenen Attributs selbst
- ein Eintrag für ein Schreibrecht

In Tabelle 15 ist eine Übersicht aller Rechte-Arrays und deren Bedingungen aufgelistet. Bei der Implementierung wurde auf die Verwendung von den Namen der verschiedenen Typen in den Arrays verzichtet um Seiteneffekten vorzubeugen. Stattdessen wird für jeden Eintrag der Primärschlüssel aus der Typdatenbank verwendet. Dadurch ist es möglich, dass unterschiedliche Elemente denselben Namen verwenden können ohne einen Fehler im System zu verursachen.

Wenn die Formularseite zum Beispiel ein Formular anzeigen möchte, kann sie die Liste aller Attribute des betroffenen Objekt-Typs durchlaufen. Für jeden Eintrag kann dann ein entsprechendes Formularfeld angefordert werden wenn Schreib- oder Leserechte vorhanden sind. Das Array enthält dabei nur Einträge für die mindestens ein Benutzerrecht hinterlegt ist.

4.7.6 Formular-Manager

Der *Formular-Manager* (`FormManager`, Abbildung 40) ist dafür verantwortlich, Formularfelder für die unterschiedlichen Datentypen zur Verfügung zu stellen. Er muss vor der

Benutzerrecht / Zuständigkeit	Prüfreihenfolge der Schlüssel im Array
Erzeugen von Instanzen	Objekt-Typ-ID
Löschen von Instanzen	Objekt-Typ-ID, Zustands-ID
Bearbeiten eines Attributs	Objekt-Typ-ID, Zustands-ID, Attribut-ID, Read/Write
Zuständigkeit für Mikro-Prozess	Mikro-Prozess-Typ-ID
Zuständigkeit für Zustand	Objekt-Typ-ID, Zustands-ID
Zuständigkeit für Transition	Objekt-Typ-ID, Zustands-ID, Schritt-ID, Transitions-ID

Tabelle 15: Einträge, die für die Objekt- bzw. Prozess-Instanz vorhanden sein müssen, um dem Benutzer das Recht oder die Zuständigkeit für die Instanz zu gewähren.

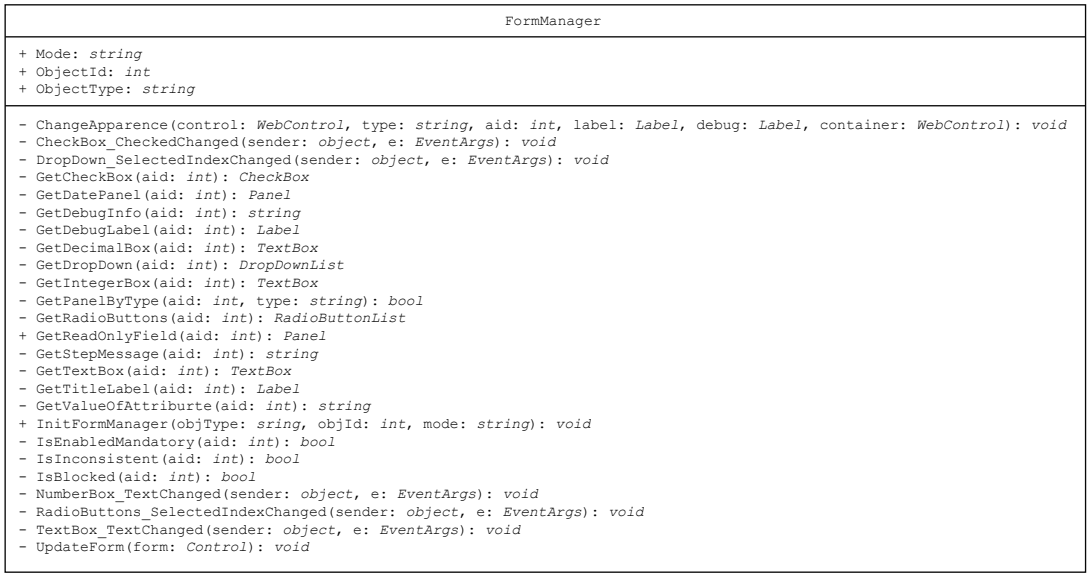

Abbildung 40: Klassendiagramm `FormManager`

Nutzung mit einem Objekt-Typ und optional mit einer Instanz-ID initialisiert werden. Anhand des Objekt-Typ wird bestimmt für welche Attribute ein Eingabefeld zur Verfügung gestellt werden müssen. Mit der Instanz-ID werden die Werte der Attribute aus der Laufzeitdatenbank ausgelesen und in den Feldern angezeigt. Nach der Initialisierung kann eine Inhaltsseite nun fertige Formularelemente für Attribute aus dem *Formular-Manager* laden. Es ist zu beachten, dass im *Formular-Manager* keine Rechte für Attribute überprüft werden. Er liefert lediglich fertige Formularfelder und die Logik zur Aktualisierung der Ansicht entsprechend des momentanen Prozesszustands. Die Rechte müssen zuvor in der entsprechenden Inhaltsseite über die Methoden des Rechte-Managers geprüft werden.

Ein Formularelement besteht aus einem Container der den Titel des Attributs, das Eingabefeld entsprechend dem Datentyp- und die aktuelle Laufzeitmarkierung enthält. Für einen *Text* oder ein *Datum* wird ein Textfeld zurückgegeben, für ein Attribut dessen Werte in einem *Wertebereich* liegen wird eine Auswahlbox oder eine DropDown-Liste zurückgegeben, für *Wahrheitswerte* wird eine Checkbox zurückgegeben und für *Ganzzahlen* und *Fließkommazahlen* wird ein schmales Textfeld zurückgegeben.

Der *Formular-Manager* ist außerdem dafür zuständig, den *Prozess-Manager* nach einer Eingabe eines Wertes in ein Eingabefeld aufzurufen und den eingegebenen Wert zu übergeben, damit die Prozess-Instanz weitergeschaltet werden kann. Anschließend muss der *Formular-Manager* das bestehende Formular verändern, indem er die Laufzeitmarkierungen aller betroffenen Zeilen und deren visuelle Markierungen aktualisiert. Die Prozess-Elemente mit einer neuen Laufzeitmarkierung werden beim Verändern so markiert, dass der *Formular-Manager* lediglich nach dieser Markierung suchen und sie nach der Aktualisierung des zugehörigen Formularfeldes zurücksetzen muss.

4.7.7 Aktivitäts-Manager

Im *Aktivitäts-Manager* (`ActivityManager`) werden zentral alle möglichen Aktivitäten, die in der Übersichtsliste der Instanzen (`CustomGrid`) angezeigt werden als, Schlüssel-Wert-Paar vorgehalten. Der Schlüssel gibt den Anzeigenamen auf der Oberfläche an und der Wert des Paares gibt den `CommandName` an, anhand dessen die Aktivität identifiziert werden soll. Der `CommandHandler` in der *Code-Behind*-Datei des `CustomGrid`, in dem das `Command` ausgeführt werden soll, muss anschließend um den `CommandName` erweitert werden, damit die Aktivität ausgeführt werden kann (Listing 8).

Listing 8: Logik innerhalb des `CommandHandlers` eines `WebControls` für Aktivitäten

```
1 # CommandArgument abfragen
2 var rowId = CommandArgument;
3
4 # Welche Aktivitaet soll ausgefuehrt werden
5 switch (CommandName)
6 {
7     case ("Delete"):
```

```
8           #Loesch-Logik
9           ...
10          DB.Delete(rowId);
11          ...
12          break;
13      case ("Edit"):
14          # Logik fuers Bearbeiten
15          ...
16          break;
17      # anfuegen weiterer Aktivitaeten
18  ...
19  }
20  ...
```

Da sich für jede Instanz eine andere Kombination von optionalen und obligatorischen Aktivitäten ergeben kann, müssen diese vor dem Laden entsprechend in zwei Listen sortiert werden. Eine Liste enthält alle obligatorischen Aktivitäten, die mit einem kleinen Symbol angezeigt werden, und die andere Liste enthält alle optionalen Aktivitäten, die in der Auswahlliste hinter den obligatorischen Aktivitäten abgelegt werden [5]. Wenn die Übersichtsliste dann die Aktivitäten für eine Instanz anzeigen möchte, kann sie auf die Listen zugreifen und sich für jedes Element eine Schaltfläche mit Symbol oder einen einfachen Link für die Auswahlliste generieren lassen.

4.7.8 Laufzeit-Datenbankmanager

Der *Laufzeit-Datenbankmanager* (`RuntimeDBManager`) hält Methoden für den Zugriff auf die Laufzeitdatenbank bereit (Abbildung 41). Die unterschiedlichen Komponenten wie der *Formular-Manager* oder der *Laufzeit-Manager* können von hier auf Attributwerte oder Laufzeitmarkierung der Prozess-Instanzen zugreifen.

Zur Kommunikation mit der Datenbank wird eine Verbindung, eine Anfrage und Parameter benötigt. Da sich die Verbindung mit jedem Prozessmodell ändert, wird diese zur Laufzeit dynamisch erzeugt. Dafür fragt der *Laufzeit-Datenbankmanager* den Namen des aktuellen Modells aus dem Modell-Manager ab und lädt damit den entsprechenden Wert aus der `connectionString.xml` ab. Für den Aufbau des Anfrage-Strings werden parametrisierte Statements verwendet, um unerwünschte Zugriffe auf die Datenbank zu vermeiden. Für die Interaktion mit der Datenbank gibt es in drei Möglichkeiten:

- Eine Anfrage die mehrere Ergebnisse in mehreren Spalten liefert (SELECT). Dieses Ergebnis muss mit einer Schleife durchlaufen werden, da es mehrere Zeilen enthalten kann
- Eine Anfrage die nur das erste Element, der ersten Spalte des Ergebnisses liefert (SELECT)

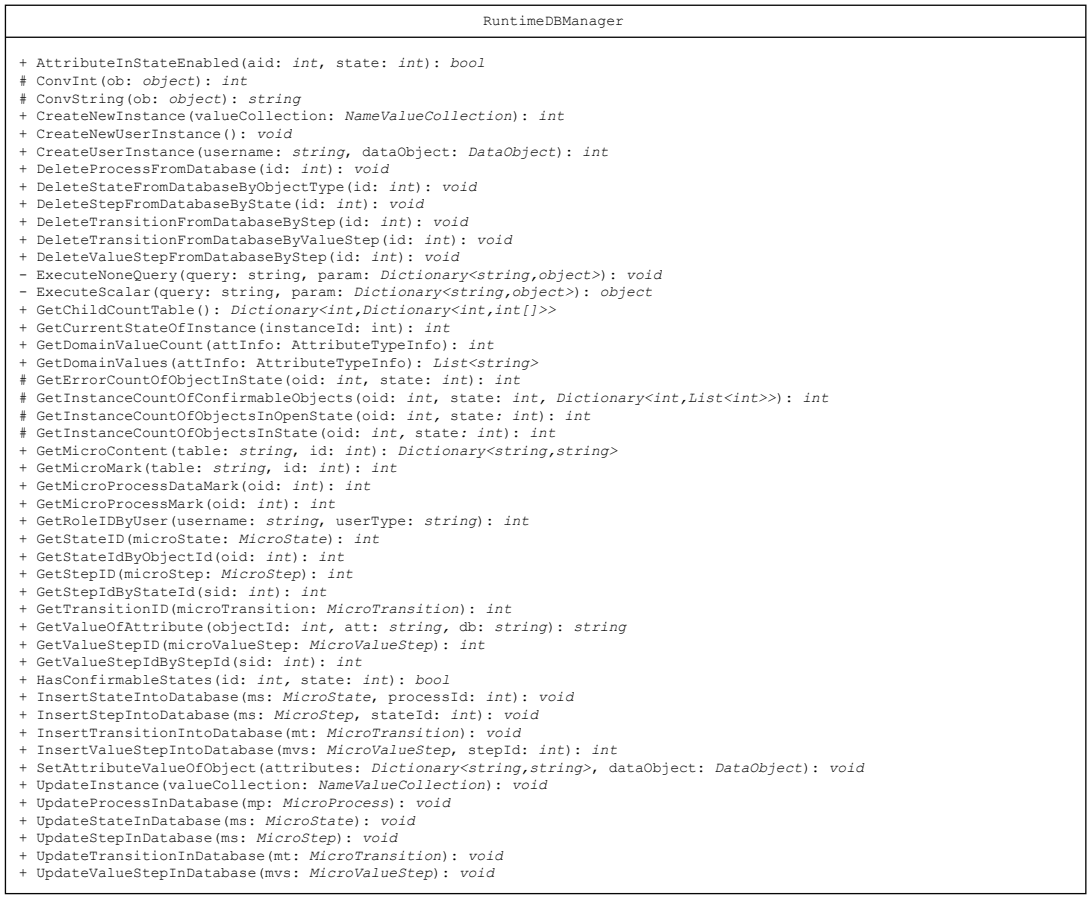

Abbildung 41: Klassendiagramm `RuntimeDBManager`

- Eine Anfrage die eine Änderung an einem Datensatz durchführt (DELETE, UPDATE, INSERT)

Die erste Variante findet vor allem Anwendung, wenn mehr als nur eine ID oder ein Name abgefragt werden sollen. Die ist der Fall wenn große Datenmengen angezeigt werden müssen, wie zum Beispiel in der Übersichtsliste im Aufgaben- und Datenbereich. Die zweite Variante wird benutzt um einzelne IDs, Zähler oder andere aggregierte Werte aus der Laufzeitdatenbank zu lesen. Die letzte Variante wird für das Löschen, Erzeugen und Aktualisieren von Datensätzen verwendet und liefert kein Ergebnis zurück.

Die beiden letzten Varianten finden die häufigste Anwendung. Für sie wurde im *Laufzeit-Datenbankmanager* jeweils eine generische Methode angelegt die lediglich mit der Anfrage und den Parametern aufgerufen werden muss und dann wenn nötig das Ergebnis zurückgeben. Der Auf- und Abbau der Verbindung zur Datenbank und die Ausführung erfolgt zentral und nicht in den einzelnen Methoden die von anderen Managern aufgerufen werden können. Somit wird zum Einen sichergestellt, dass immer nur eine Verbindung aufgebaut wird und zum Anderen dass Fehler bei der Ausführung zentral aufgefangen und verarbeitet werden können.

4.7.9 Typ-Datenbankmanager

Der *Typ-Datenbankmanager* (`TypeDBManager`) besteht ausschließlich aus Methoden zur Abfrage von Typinformationen aus der Typdatenbank (Abbildung 42). Die Veränderung von Typdaten ist zur Laufzeit nicht vorgesehen und auch nur bedingt ratsam, da eine Veränderung der Struktur der Typdaten die komplette Laufzeitkomponente beeinträchtigt.

Einige interne Methoden und Methoden anderer Manager fragen oft einzelne Werte in Form eines Namen oder einer Typ-ID aus der Typdatenbank ab. Da dies nicht sehr effektiv ist gibt es eine Menge von Arrays (Look-Up-Tabellen) mit einfachen Schlüssel-Wert-Paaren für diese Informationen. Es lässt sich so zum Beispiel der Name eines Elements über dessen ID abfragen ohne einen Datenbankaufruf durchführen zu müssen. Dadurch werden viele kleine Anfragen im Vorfeld unterbunden, da nur nach dem Eintrag im entsprechenden Array gesucht werden muss. Die Abfrage erfolgt nach aussen transparent über normale Methoden. Die Arrays selber sind nach außen nicht sichtbar und somit auch nicht veränderbar. Sie werden außerdem erst beim ersten Zugriff darauf geladen. Dabei wird vor jedem Aufruf geprüft, ob das entsprechende Array geladen wurde. Falls es noch nicht initialisiert wurde, wird es aus der Datenbank generiert, gespeichert und der gewünschte Wert zurückgegeben.

Komplexere Anfragen sind zum Beispiel die Abfrage von Listen mit allen Schritten die einem bestimmen Zustand untergeordnet sind oder aller Transitionen eines Mikro-Prozess-Typs, die von einem Benutzer explizit aktiviert werden müssen. Ausserdem lassen sich

Abbildung 42: Klassendiagramm `TypeDBManager`

hier die Rechte und Zuständigkeiten anhand der Rollen-ID des Benutzers abfragen.

4.8 Umsetzung der operationalen Semantik

Für die in der operationalen Semantik festgelegten Markierungsänderungen in einer Prozess-Instanz wurde die Prozessstruktur auf eine Klassenstruktur abgebildet und entsprechende Methoden zur Änderung der Laufzeitmarkierung und zur Benachrichtigung davon betroffener Elemente zur Verfügung gestellt (Abbildung 43). Alle Prozesselemente besitzen eine eigene Klasse und referenzieren sich entsprechend der Prozessstruktur. Ein Schritt referenziert zum Beispiel immer den Zustand, dem er zugeordnet ist, und hält eine Liste aller eingehenden und ausgehenden Transitionen sowie einer Liste aller Werteschritte falls vorhanden. Eine Transition hält dagegen eine Referenz auf seinen Quell- und Zielschritt, und ein Zustand hält eine Liste seiner Schritte.

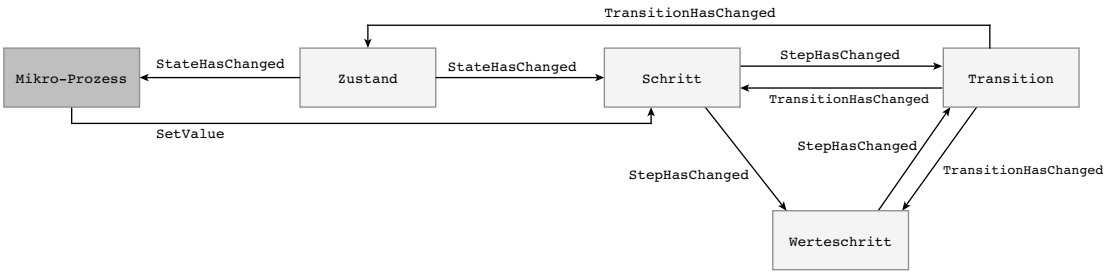

Abbildung 43: Benachrichtigungsmethoden der Elemente bei Markierungsänderungen

Die Veränderung einer Laufzeitmarkierung lässt sich durch diesen Aufbau durch alle Elemente fortpflanzen bis sie auf ein Abbruchkriterium stößt und zum ursprünglichen Aufruf zurückkehrt. Die Besonderheit dabei ist, dass ein Schritt eine Transition nicht direkt ändert, sondern sie lediglich informiert, dass er sich verändert hat und die Transition dann ihre Markierung entsprechend anpasst und ebenfalls alle Nachfolger informiert. Dies geschieht so lange bis keine Änderung mehr möglich ist und der Änderungsfluss gestoppt wird.

Im folgenden Abschnitt werden die einzelnen Klassen erläutert. Die Bezeichnung *Instanz* bezieht sich in diesem Abschnitt auf Instanzen der vorgestellten Klassen und nicht auf das Typ-Instanz-Schema aus den vorangegangenen Abschnitten. Ebenso bezieht sich die Bezeichnung *Schritt* oder *Zustand* auf Instanzen der Klasse Schritt und Zustand.

4.8.1 Mikro-Prozesse

Eine Mikro-Prozess-Instanz besteht aus einer Menge von Variablen und Methoden, zur Steuerung einer Mikro-Prozess-Instanz (Abbildung 44).

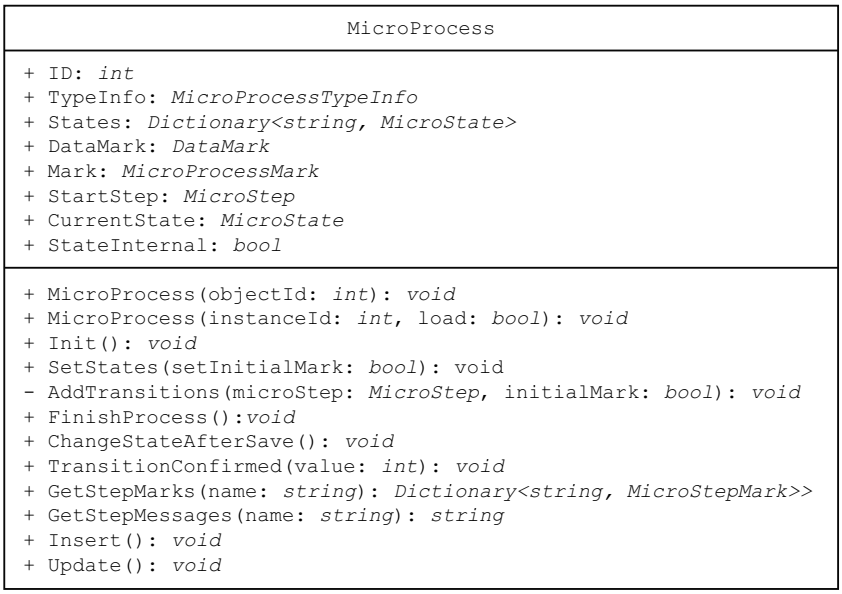

Abbildung 44: Klassendiagramm `MicroProcess`

Der Konstruktor wird entweder mit einer Objekt-Typ-ID oder einer Instanz-ID aufgerufen. Ersteres erzeugt eine neue Mikro-Prozess-Instanz ohne Werte und mit initialen Laufzeitmarkierungen, Letzteres ermöglicht es, eine Prozess-Instanz aus der Laufzeitdatenbank zu laden. Bei einer Erzeugung einer neuen Instanz wird die Laufzeitmarkierung auf RUNNING gesetzt und die Zustände initialisiert. Dafür wird zuerst eine Liste aller Zustände über den *Typ-Datenbankmanager* geladen und anschließend für jeden Zustand eine Instanz erzeugt und in der Mikro-Prozess-Instanz abgelegt. Jeder Zustand wiederum initialisiert dabei alle zugehörigen Schritte, welche wiederum die zugehörigen Werteschritte erzeugen und speichern. Ein Zustand fragt dazu eine Liste der zugehörigen Schritte und ein Schritt eine Liste der zugehörigen Werteschritte aus der Typdatenbank ab, die dann in einer Schleife abgearbeitet und die Elemente instanziiert werden (Listing 9).

Die Verknüpfung der (Werte-)Schritte mit Transitionen erfolgt anschließend rekursiv, dabei wird der Verbindungsmethode der Startschritt der Mikro-Prozess-Instanz als Ausgangspunkt übergeben. Für den Schritt wird dann eine Liste der ausgehenden Transitionen aus der Typdatenbank geladen, für jeden Eintrag ein Transitions-Objekt instanziiert und der aktuelle Schritt als Quelle gesetzt. Anschließend wird die Senke der Transition in den bereits erzeugten Schritt-Objekten gesucht, gespeichert und anschließend der Verbindungsmethode als neuer Startpunkt übergeben. Somit ist sichergestellt, dass alle (Werte-)Schritte über Transitionen verbunden werden. Das Abbruchkriterium ist erfüllt, wenn ein Schritt keine ausgehenden Transitionen besitzt, sich also in einem Endzustand befindet.

Listing 9: Initialisierung eines Mikro-Prozesses im `ProcessManager`

```
var proc = new Process();
...
var stateList = GetStates();
foreach(state in stateList)
{
    var st = new State(state);
    ...
    var stList = GetSteps();
    foreach(step in sList)
    {
        var st = new Step(step);
        ...
        var vsList = GetValueSteps();
        foreach(vstep in vsList)
        {
            var vs = new ValueStep(vstep);
            ...
        }
    }
}
...
SetTransitions(proc.StartStep);
...
```

Die Klasse bietet weiter eine Methode an, um den Wert eines Attributs aus einem Formularfeld in alle betroffenen Schritte zu schreiben. Dafür werden alle betroffenen Schritte (die das Attribut referenzieren) identifiziert und der Wert eingetragen. Dies löst eine Markierungsänderung durch einen Attributwert hervor (Tabelle 4). Über den Mikro-Prozess lässt sich außerdem die Laufzeitmarkierung von Mikro-Schritten abfragen die dasselbe Attribut referenzieren. Dies wird benötigt um einem Formularfeld im *Formular-Manager* die aktuelle Laufzeitumgebung der zugehörigen Schritte hinzuzufügen.

Da ein Zustandswechsel erst explizit nach dem Speichern der Werte im Formular erfolgt, muss dafür eine entsprechende Methode zum Anstoß des Wechsels vorhanden sein. Wird ein Formular gespeichert wird diese aufgerufen und sucht dann in einem unabgeschlossenen Zustand (UNCONFIRMED) nach einer externen Transition mit der Markierung READY (Diese Kombination von Markierungen indiziert einen anstehenden Zustandswechsel) und stößt den Wechsel an.

Für die explizite Schaltung eines Mikro-Prozess in einen Folgezustand gibt es ebenfalls eine Methode, die das übernimmt. Dafür wird ihr lediglich die Instanz-ID der zu aktivierenden Transition übergeben und der Wechsel in der Transitions-Instanz angestoßen.

Ein Mikro-Prozess beinhaltet außerdem jeweils eine Methode zum Erzeugen und zum Aktualisieren seiner Daten in der Laufzeitdatenbank.

4.8.2 Zustände

Eine Zustands-Instanz besteht ebenfalls aus einer Menge von Variablen und Methoden. Durch sie lässt sich ein Zustand steuern und betroffene Elemente benachrichten, falls eine Änderung der Markierung auftritt (Abbildung 45).

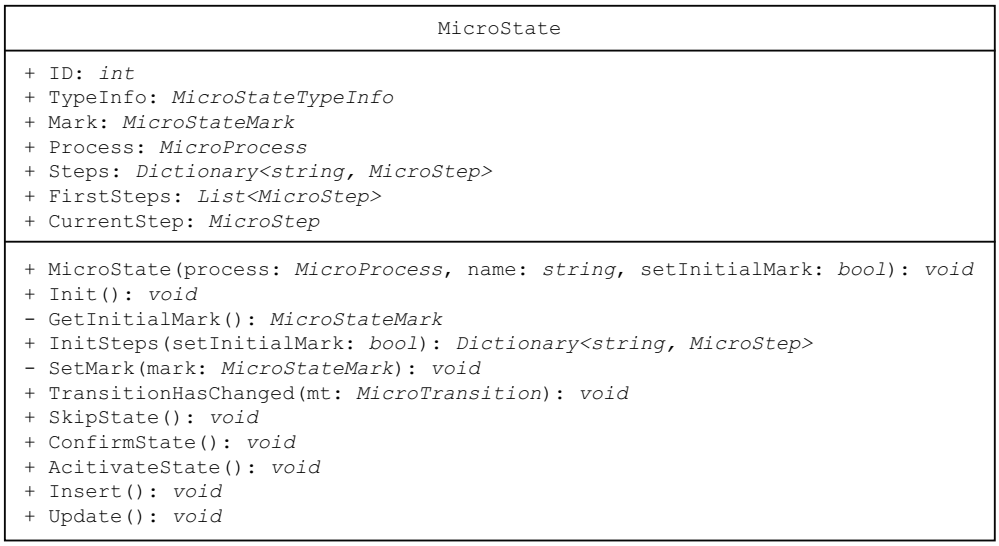

Abbildung 45: Klassendiagramm `MicroState`

Der Konstruktor wird mit der Mikro-Prozess-Instanz, dem Namen des Zustandes und einem Parameter, der angibt ob die Initialmarkierung gesetzt werden soll, aufgerufen. Die Initialmarkierung ist lediglich bei einer neuen Instanz nötig, da vorhandene Instanzen ihre aktuelle Markierung aus der Datenbank bekommen. Nach dem Laden der Typ-Informationen werden die einzelnen Schritte initialisiert. Dafür wird wie schon bei den Zuständen eine Liste aller zugehörigen Schritte geladen, in einer Schleife durchlaufen und dabei jedes Element als Schritt instanziiert und gespeichert.

Ein Zustand muss auf die Veränderungen von externen Transitionen reagieren, da diese einen Zustandswechsel auslösen. Wenn ein Mikro-Prozess über die Methode `TransitionHasChanged` von einer Transition informiert wird, dass sie sich verändert hat, muss er seine Markierung entsprechend anpassen. Dafür verfügt er über eine Methode `SetMark` um seine Laufzeitmarkierung zu verändern. Je nachdem welche Folgemarkierung gesetzt wurde, müssen weitere Elemente benachrichtigt werden. Bei der Aktivierung eines Zustands (ACTIVATED) müssen zunächst alle Schritte im Zustand (die momentan alle noch mit WAITING markiert sind) mit ihrer Ausgangsmarkierung versehen werden (Tabelle 5) und weiterschalten, falls ein Wert vorhanden ist. Falls es sich bei dem aktivierten Zustand um einen Endzustand handelt, muss der Mikro-Prozess abgeschlossen werden, da keine weiteren Daten eingegeben werden müssen. Wenn ein Zustand abgeschlossen wird

(CONFIRMED), muss der erste Schritt des Zustands benachrichtigt werden, da auch alle Schritte abgeschlossen werden müssen.

Ein Zustand beinhaltet außerdem jeweils eine Methode zum Erzeugen und zum Aktualisieren seiner Daten in der Laufzeitdatenbank.

4.8.3 Schritte

Eine Schritt-Instanz lässt sich ebenfalls über bestimmte Methoden steuern (Abbildung 46). Sobald eine Änderung der Laufzeitmarkierung stattfindet, werden betroffene Elemente informiert.

Nach der Initialisierung des Schrittes werden dessen Werteschritte initialisiert, falls es welche gibt. Ein Schritt reagiert neben dem Setzen oder der Veränderung des referenzierten Attributwertes auch auf die Veränderung der Markierung von weiteren Prozesselementen. Über die Methode `SetValue` kann ein Wert für den Schritt gesetzt werden. Dieser wird jedoch nur übernommen, wenn der Schritt in einem aktivierten oder aktivierbaren Zustand (ACTIVATED, WAITING) liegt. Wenn der Schritt bereits abgeschlossen ist (CONFIRMED), wird lediglich die Datenmarkierung auf *Inkonsistent* gesetzt und der ursprüngliche Wert beibehalten.

Nach dem Ändern eines Wertes wird die Methode `ValueHasChanged` aufgerufen, die bestimmt, welche Folgemarkierung der Schritt nach einer Wertänderung, abhängig von seiner aktuellen Markierung, erhalten soll (Tabelle 4). Mit den Methoden `StateHasChanged` und `TransitionHasChanged` wird der Schritt von Änderungen im Zustand und den ein- und ausgehenden Transitionen informiert und legt anhand deren Markierung seine neue Markierung fest (Tabelle 5 und 6).

Ein Schritt muss zudem sicherstellen, dass immer nur die ausgehende Transition aktiviert wird, die die höchste Priorität innerhalb des Schrittes besitzt. Dafür kann er alle ausgehenden Transitionen prüfen, den Gewinner aktivieren und für die Verlierer eine Dead-Path-Elimination anstoßen. Dafür wird die entsprechende Methode in den Transitionen aufgerufen, die dann dafür sorgen, dass sich die Ausnahmebehandlung in alle weiteren Elemente fortsetzt. Auch diese Klasse enthält eine Methode `SetMark`, um mit jeder Änderung der Laufzeitmarkierung alle betroffenen Elemente zu informieren. Dafür wird, wenn nötig, die Liste aller ein- und ausgehenden Transitionen sowie die Liste aller Werteschritte durchlaufen und deren `StepHasChanged`-Methode aufgerufen.

4.8.4 Werteschritte

Eine Werteschritt-Instanz besteht, wie eine Schritt-Instanz, aus einer Menge von Methoden und Variablen. Über sie lässt sich die Laufzeitmarkierung anpassen und davon

Abbildung 46: Klassendiagramm `MicroStep`

$\text{Schritt}'_{Mark}$	Benachrichtigen
READY	Werteschritte
ENABLED	ausgehende Transitionen, Werteschritte
ACTIVATED	eingehende Transitionen, Werteschritte
UNCONFIRMED	ein- und ausgehende Transitionen, Werteschritte
CONFIRMED	ein- und ausgehende Transitionen, Werteschritte
SKIPPED	ausgehende Transitionen, Werteschritte
BYPASSED	ausgehende Transitionen, Werteschritte

Tabelle 16: Angabe, welche Elemente anhand der neuen Markierung eines Schrittes informiert werden müssen

betroffene Elemente benachrichtigen (Abbildung 47).

Der Konstruktor wird vom übergeordneten Schritt aufgerufen und lädt u.a. die Typ-Informationen und das zugehörige Prädikat. Der Werteschritt besitzt die gleichen Laufzeitmarkierungen wie ein Schritt und verändert diese wenn sich die Markierung des übergeordneten Schritts verändert (Tabelle 8). Er muss dabei lediglich die ausgehenden Transitionen informieren und dabei sicherstellen, dass nur die Transition mit der höchsten Priorität aktiviert wird und alle Verlierer deaktiviert werden. Ein Werteschritt verfügt ebenfalls über Methoden zum Einfügen und Aktualisieren von Datenbankeinträgen.

4.8.5 Transitionen

Eine Transitions-Instanz besteht, wie die vorangegangenen Elemente, aus einer Menge von Variablen und Methoden. Durch sie lässt eine Folgemarkierung bestimmen und Änderungen propagieren (Abbildung 48).

Bevor sich eine Transition aktivieren kann, muss sie stets ihre Priorität prüfen lassen, da eine einzelne Transition nicht weiß, ob es alternative Transitionen gibt. Dafür ruft sie eine Methode im Quellschritt auf, die dann wie bereits beschrieben alle vorhandenen Transitionen prüft und die entsprechende Transition aktiviert oder nicht. Die Überprüfung wird ausgesetzt, falls die aufrufende Transition bereits selbst die höchste Priorität (1) besitzt. Dann wird der Schritt lediglich aufgefordert alle anderen Transitionen zu deaktivieren.

Bei der Änderung der Markierung muss beachtet werden, dass sich für die drei Arten von Transitionen unterschiedliche Änderungsabfolgen ergeben (Tabelle 10 und 11). Eine externe explizite Transition besitzt zum Beispiel eine zusätzliche Laufzeitmarkierung CONFIRMABLE. Bei einer Markierungsänderung wird grundsätzlich nur der Zielschritt informiert, da in der operationalen Semantik festgelegt ist, dass eine Transition nur dann aktiviert werden kann, wenn der Vorgänger bereits aktiviert wurde (also keine weitere

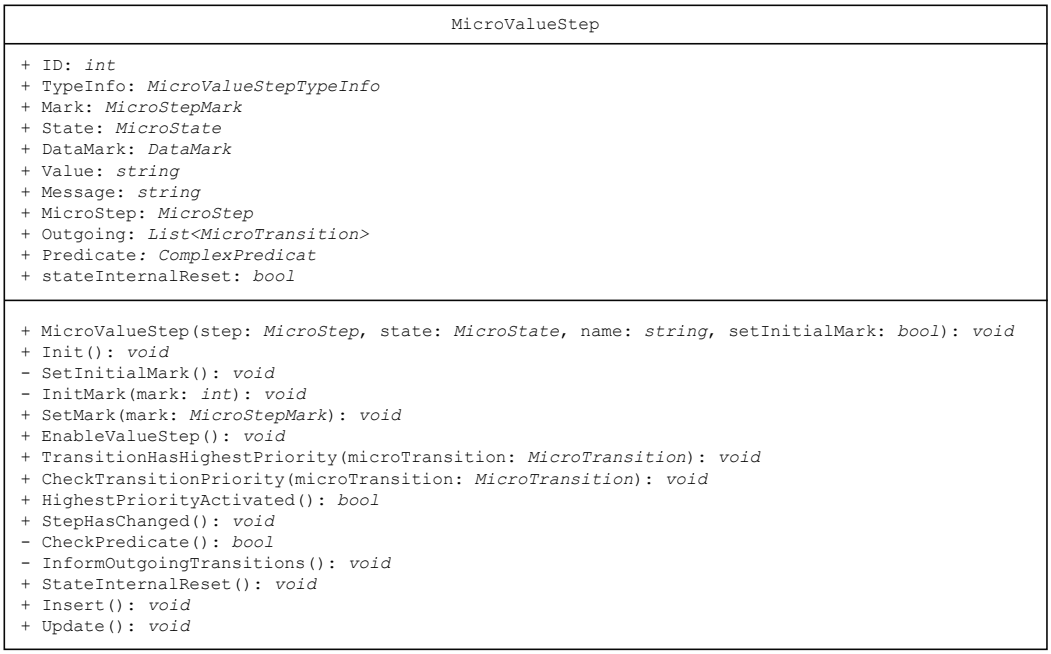

Abbildung 47: Klassendiagramm `MicroValueStep`

Interaktion notwendig ist). Bei externen Transitionen kommt hinzu, dass bei einer anstehenden Aktivierung der Transition (Markierung ist READY) ein Zustandsübergang erfolgen muss, da eine Transition genau dann auf READY gesetzt wird, wenn der Vorgänger mit (UN)CONFIRMED abgeschlossen wurde (Tabelle 9 und 10) oder ein Benutzer die Transition explizit ausgewählt hat. Dabei wird die Methode `TransitionHasChanged` in den beiden betroffenen Zuständen aufgerufen. Bei einer expliziten Weiterschaltung wird die Transition über eine Methode `HasBeenConfirmed` außerhalb des Prozessflusses auf READY gesetzt.

Bei einem Zustandswechsel muss beachtet werden, dass die externe explizite Transition die den Wechsel angestoßen hat nur dann von READY auf ENABLED schalten darf, wenn der Folgezustand bereits aktiviert wurde da diese Änderung den Prozess womöglich weiterschalten lässt, aber dafür der Zustandswechsel komplett abgeschlossen sein muss. Ansonsten kann es passieren, dass der Prozess blockiert ist, da kein Zustand aktiviert ist.

4.8.6 Ausnahmebehandlungen

Ausnahmebehandlungen setzen sich ebenfalls durch alle Prozesselemente fort. Eine interne Dead-Path-Elimination wird von einem Schritt aufgerufen sobald sich eine Transition aktiviert und alternative Transitionen vorläufig deaktiviert werden müssen. Die vorläufig deaktivierten Elemente informieren dann alle direkten Nachfolger, die sich ebenfalls

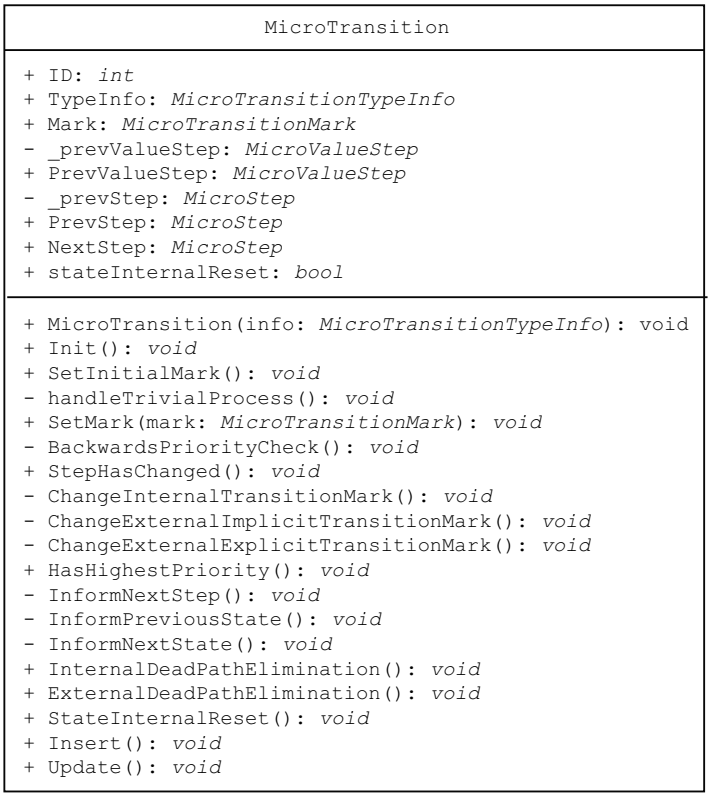

Abbildung 48: Klassendiagramm `MicroTransition`

deaktivieren usw. solange bis auf ein Abbruchkriterium in einem Schritt oder auf eine externe Transition getroffen wird.

Eine externe Dead-Path-Elimination wird aufgerufen, sobald ein Vorgänger-Zustand bei einem Zustandswechsel abgeschlossen wird. Wenn es dabei einen Schritt mit mehreren ausgehenden Transitionen gibt, werden alle vorläufig deaktivierten Pfade endgültig deaktiviert mit dem Unterschied, dass diese Deaktivierung über einen Zustand hinaus läuft bis auf ein Abbruchkriterium in einem Schritt getroffen wird.

Bei einer internen Zurücksetzung (*Reset*) eines Zustands, wird vom betroffenen Schritt der den *Reset* auslöst eine Methode im Startschrittchritt aufgerufen, die dann einen *Reset* der Markierungen in allen Nachfolgern anstößt. Danach ruft der betroffene Schritt eine Methode im Startschrittchritt des Zustands auf, die eine erneute Evaluation der Werte durchführt und den Prozess entsprechend weiterschaltet.

4.9 Aktivitäten auf Instanzen

Die einzelnen Aktivitäten auf der Oberfläche zur Verwaltung von Instanzen und deren Daten benötigen die Manager und deren Daten, um korrekt zu funktionieren. Das Zusammenspiel soll im Folgenden verdeutlicht werden.

4.9.1 Erzeugen / Bearbeiten einer Instanz

Beim Erzeugen einer Instanz wird eine neue Prozessinstanz mit der ID des aktuellen Objekt-Typ erzeugt. Dabei werden die einzelnen Elemente wie beschrieben initialisiert und der Prozess im *Prozess-Manager* hinterlegt. Anschließend wird der *Formular-Manager* mit dem Objekt-Typ initialisiert, für den eine Instanz erzeugt werden soll. Danach folgt eine Weiterleitung auf die Formularseite. Diese lädt basierend auf den Bearbeitungsberechtigungen für Attribute des Objekt-Typs im *Rechte-Manager* die entsprechenden Formularfelder aus dem *Formular-Manager* und fügt sie der Anzeige hinzu. Wenn ein Benutzer einen Wert für ein Attribut eingibt, wird dieser Wert an den *Prozess-Manager* übergeben der ihn allen betroffenen Schritten übergibt die daraufhin ihre Laufzeitmarkierung verändern falls möglich. Diese Änderungen werden anschließend vom *Formular-Manager* abgefragt welcher dann die Anzeige im Formular aktualisiert. Nachdem alle gewünschten Felder ausgefüllt sind und somit die Laufzeitmarkierungen verändert wurden und auf Speichern geklickt wurde, wird erst der neue Mikro-Prozess in der Laufzeitdatenbank angelegt um die entsprechenden Instanz-IDs zu setzen, dann auf einen anstehenden Wechsel in den nächsten Zustand geprüft und ggf. ausgeführt und dann in der Laufzeitdatenbank aktualisiert, um alle Änderungen zu übernehmen. Anschließend wird auf die Hauptseite zurückgeleitet. Beim Laden der Hauptseite werden die einzelnen Zähler aktualisiert. Beim Laden des Aufgaben- und Datenbereichs wird zudem geprüft, welcher Bereich zuletzt geöffnet war und welcher Objekt-Typ in der

Übersichtsliste angezeigt werden soll. Somit ist der Benutzer wieder am Ausgangspunkt angelangt.

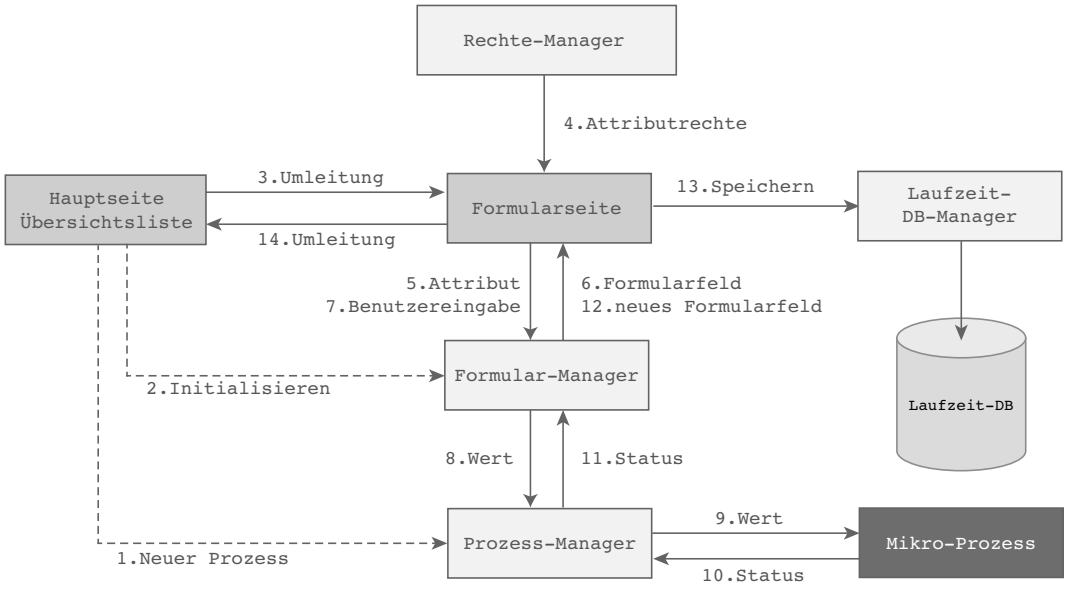

Abbildung 49: Schema des Erzeugungsvorgangs einer Instanz

Das Bearbeiten läuft ähnlich ab wie das Erzeugen einer Instanz. Der *Formular-Manager* wird jedoch zusätzlich mit einer Instanz-ID initialisiert, damit bei der Erzeugung des Formulars bereits vorhandene Werte angezeigt werden können. Beim Speichern wird die Instanz und der aktuelle Zustand des Mikro-Prozess in der Datenbank aktualisiert, nachdem auf einen anstehenden Zustandswechsel geprüft wurde.

4.9.2 Löschen einer Instanz

Beim Löschen aus der Übersichtsliste heraus wird ein Löschdialog aufgerufen und nach einer Bestätigung die entsprechende Löschlogik aufgerufen. Dabei wird der Dialog zunächst mit entsprechendem Inhalt wie dem Titel, einem Warnhinweis und den IDs der zu löschenden Instanzen initialisiert und angezeigt. Nach dem Bestätigen des Dialogs wird die Löschroutine der Übersichtsliste aufgerufen, die dann die betroffenen Zeilen in der Objekt-Tabelle löscht. Anschließend werden noch die Prozesselemente über den *Laufzeit-Datenbankmanager* gelöscht, da durch die Datenbankstruktur keine automatische Löschung durch einen automatischen Mechanismus (*Trigger*) durchgeführt werden kann, da dies über eine Fremdschlüssel-Referenzierung erfolgt und diese bei der generischen Struktur der Laufzeitdatenbank nicht anwendbar ist, weil ein Zustand zwar eine Mikro-Prozess-Instanz anhand seiner Instanz-ID referenzieren kann, aber immer noch zusätzlich eine Zeichenkette gespeichert werden muss, um festzustellen in welcher Instanz-

Tabelle die Prozess-Instanz mit der ID liegt (Abbildung 20). Zuletzt werden alle Zähler und die Übersichtslisten aktualisiert.

4.9.3 Explizite Aktivierung eines Folgezustands

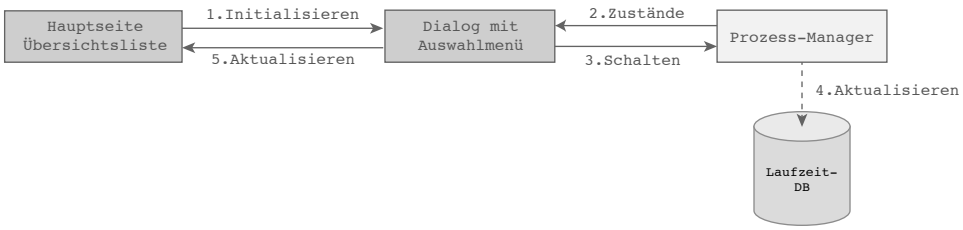

Abbildung 50: Schema der expliziten Aktivierung eines Folgezustands

Bei einer expliziten Aktivierung eines Folgezustands durch den Benutzer (Commit) wird ein Dialog initialisiert und das Auswahlmenü mit den möglichen Folgezuständen gefüllt. Diese werden anhand von externen expliziten Transitionen der Prozess-Instanz bestimmt, welche die Markierung CONFIRMABLE besitzen. Nach der Bestätigung wird der ausgewählte Zustand im *Prozess-Manager* aktiviert und die Alternativen werden durch eine externe Dead-Path-Elimination deaktiviert. Dafür wird die Instanz-ID der Transition einer Methode im Mikro-Prozess übergeben, die dann die Markierung der Transition auf READY setzt und einen Zustandswechsel ausführt. Anschließend wird der Mikro-Prozess in der Datenbank aktualisiert und der Dialog geschlossen.

4.9.4 Inkonsistenzen behandeln

Wenn ein Attribut in einem Mikro-Prozess geändert wird, wenn der dazugehörigen Schritt schon aktiviert wurde (CONFIRMED), dann wird das Feld als Inkonsistent markiert. Um diesen Zustand zu korrigieren, hat der Benutzer drei Möglichkeiten dies zu beheben. Entweder er übernimmt den neuen Wert (ignoriert also die Inkonsistenz), setzt den vorherigen konsistenten Wert oder setzt den Mikro-Prozess an die Stelle in der Ausführung zurück an der sie zuletzt konsistent war. Die letzte Option wurde nur der Vollständigkeit halber angegeben und wurde im Prototyp nicht implementiert. Für die anderen Optionen stellt der *Prozess-Manager* entsprechende Methoden zur Verfügung.

Wenn der Benutzer die Inkonsistenzen ignorieren möchte wird der neue Attributwert behalten und die Inkonsistenzindikatoren in den betroffenen Schritten zurückgesetzt. Wenn er sie beheben möchte wird das Attribut auf seinen ursprünglichen Wert zurückgesetzt.

5 Diskussion

In diesem Kapitel sollen verschiedene Aspekte, wie Abweichungen von zugrundeliegenden Usability-Konzept sowie Erweiterungen des fachlichen Konzepts evaluiert werden.

5.1 Abweichungen vom Usability-Konzept

Die farbliche Hervorhebung von Formularfeldern auf der Formularseite durch eine Modifizierung der Hintergrundfarbe durch Stylesheets (*CSS*), wird in den meisten Browsern nur teilweise unterstützt. Um eine einheitliche Markierung zu gewährleisten, wurde das Konzept so angepasst, dass nun eine gesamte Zeile in einem Formular durch die entsprechende Hintergrundfarbe hervorgehoben wird (Abbildung 51). Um die Ausführung einer Prozess-Instanz besser nachvollziehen zu können, wird zusätzlich hinter jedem Formularfeld die aktuelle Laufzeitmarkierung aller zugehörigen Schritte angezeigt. Um reguläre Benutzer nicht unnötig zu verwirren, sollte diese Ausgabe in einem Produktivsystem nicht weiter angezeigt werden.

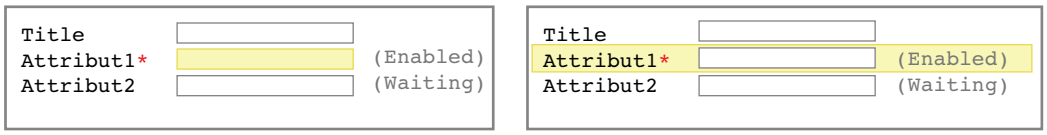

Abbildung 51: Das ursprüngliche und das neue Aussehen eines Formulars.

Im Usability-Konzept wurde die Verwendung von Java-Servlets/JSP (*Java Server Pages*) zur Implementierung vorgeschlagen, da die Modellierungskomponente bei der Erstellung des Konzepts ebenfalls auf Java basierte. Dies war zu Beginn dieser Diplomarbeit jedoch nicht weiter der Fall, da die Komponente nun basierend auf Technologie von Microsoft weiterentwickelt wurde. Die Verwendung von ASP.NET bot sich daher an, um die Vermischung verschiedener Technologien so gering wie möglich zu halten. Da die Laufzeit-Komponente stets auf die Typdatenbank zugreifen muss ist es von Vorteil, dass beide Komponenten auf derselben Datenbank-Technologie (*SQL Server*) basieren. Somit ist ein einheitlicher Zugriff auf die Datenbanken möglich ohne auf zusätzliche Adapter-Klassen für unterschiedliche Datenbank-Technologien zugreifen zu müssen. Das *AJAX-Toolkit* bot zudem die nötige Funktionalität für eine dynamische Benutzeroberfläche.

Eine Einschränkung im Vergleich zum Konzept ist, dass der Prototyp ein System mit einem Prozessmodell und genau einem Benutzer widerspiegelt. Der gleichzeitige Zugriff vieler Benutzer wurde noch nicht evaluiert. Zusätzlich wurde auf bestimmte Oberflächenfunktionen wie die Sortierung oder Suche von Instanzen noch nicht umgesetzt, da der Prototyp seinen Fokus auf die Implementierung der operationalen Semantik für einzelne Mikro-Prozesse legt.

5.2 Erweiterung der Benutzerintegration zur Laufzeit

In PHILharmonicFlows lassen sich beliebig komplexe Datenmodelle erzeugen und Benutzer über Benutzer-Typen integrieren. Diese können eine oder mehrere Rollen besitzen die festlegen, was der Benutzer darf und wofür er zuständig ist. Zur Laufzeit soll es einem Benutzer möglich sein, sich im System anzumelden. Dafür muss eine Benutzer-Instanz für den Benutzer-Typ erstellt werden der er zugeordnet werden soll. Die aktuelle Implementierung sieht dafür vor, dass sich jeder Benutzer registrieren kann. Dafür wählt er einen der verfügbaren Benutzer-Typen, dem genau eine Rolle zugeordnet ist, aus und kann sich dann mit seinem gewählten Benutzernamen und seinem Passwort anmelden. Dabei wird ein Eintrag in der entsprechenden Objekt-Tabelle des Benutzer-Typs und in der Benutzer-Tabelle des .NET Frameworks abgelegt. Dieses Vorgehen wurde gewählt, weil ASP.NET bereits fertige Methoden zur Benutzerverwaltung zur Verfügung stellt.

Prinzipiell lassen sich Benutzer auf zwei Arten im System erzeugen. Entweder sie registrieren sich selbst, oder sie werden durch einen verwaltenden Benutzer (*Administrator*) hinzugefügt. Die erste Variante lässt dem Benutzer mehr Freiheit bei der Wahl seines Benutzernamens und benötigt keinen Administrator zur Erzeugung. Es muss lediglich beachtet werden, dass der Benutzer vorerst eine Rolle mit minimalen Rechten zugeordnet bekommt, um ungewollte Aktionen auf Daten zu vermeiden. Die Zuordnung einer endgültigen Rolle muss dann wieder durch einen Administrator erfolgen. Die zweite Variante erlaubt die Erzeugung von Benutzern nach speziellen Namenskonventionen und bietet zusätzlich die Möglichkeit, dem neuen Benutzer unmittelbar seine endgültigen Rollen zuzuordnen. Durch diese geschlossene Methode ist zusätzlich keine Registrierungsseite mehr nötig. In beiden Fällen muss dem verwaltenden Benutzer ein Bereich innerhalb der Laufzeitumgebung zur Verfügung gestellt werden, in dem er Benutzer und deren Rollen adäquat verwalten kann. Welcher Ansatz verfolgt wird, kann in einer weiterführenden Arbeit evaluiert werden.

5.3 Integration des Deployments

Das Deployment erzeugt eine Laufzeitdatenbank, basierend auf einem Prozessmodell in der Typdatenbank. Dies erfolgt in der aktuellen Implementierung noch sehr rudimentär über eine einfache Klasse, die auf die Typdatenbank lesend zugreift. Für eine bessere Integration kann das Deployment als eigenständiges Paket in die Modellierungsumgebung eingebaut werden. Das ist sinnvoller als das Deployment als externe Komponente oder als Teil der Laufzeitumgebung zu realisieren, da das Deployment unmittelbar auf die Modellierung folgen muss, um die entsprechende Datenbankstruktur aufzubauen. So kann ein Benutzer außerdem die Datenbank per Mausklick erzeugen lassen, ohne dabei die Modellierungskomponente verlassen zu müssen.

6 Zusammenfassung und Ausblick

6.1 Zusammenfassung

Die Aufgabe der Diplomarbeit bestand in der Implementierung einer Laufzeitkomponente zur Ausführung von Mikro-Prozess-Instanzen auf Basis der fachlichen Konzepte von PHILharmonicFlows und dem Usability-Konzept. Dafür wurde das allgemeine Vorgehen zur Erzeugung eines Datenbankschemas für die Datenstruktur evaluiert, das für alle zukünftigen Prozessmodelle verwendet werden kann. Darauf aufbauend wurden Inhaltsseiten und benutzerdefinierte Steuerelemente, entsprechend dem Usability-Konzept, entwickelt, um die Daten auf der Oberfläche anzuzeigen und Aktivitäten darauf ausführen zu können. Für die Umsetzung der Anwendungslogik wurden verschiedene Manager realisiert, die ähnliche Methoden in Klassen zu kapseln. Über sie können Formulare erstellt, Rechte verwaltet oder mit den unterschiedlichen Datenbanken kommuniziert werden. Die Schaltlogik für Mikro-Prozess-Instanzen wurde, basierend auf der umfangreichen operationalen Semantik, realisiert und erlaubt es Benutzern, eine Mikro-Prozess-Instanz durch die Eingabe von Daten zu steuern. Instanzen können dabei unter Berücksichtigung von Benutzerrollen angelegt, gelöscht oder bearbeitet werden, wobei dem Benutzer über Arbeits- und Übersichtslisten jederzeit eine integrierte Sicht auf Prozess-Instanzen und deren Daten gewährt wird.

6.2 Ausblick

Das Konzept PHILharmonicFlows sieht die Implementierung eines kompletten Frameworks zur Modellierung und Ausführung von datenorientierten Prozessen vor. Der vorgestellte Prototyp stellt den Ausgangspunkt weiterer Arbeiten für die Ausführung solcher Prozesse dar und zeigt, dass eine Ausführung von Mikro-Prozessen auf Basis der generischen Datenstruktur performant möglich ist. Es können beliebige Mikro-Prozess-Instanzen ausgeführt und entsprechend verwaltet werden. Durch weitere Arbeiten soll es dann möglich sein, modellierte Makro-Prozesse auszuführen und erzeugte Instanzen in einem eigenen Bereich (*Monitoring*) überwachen zu können. Dafür muss zunächst die Struktur zwischen Objekt-Typen durch Relationen integriert werden und anschließend die Verknüpfung zwischen Mikro-Prozess-Instanzen über Makro-Prozess-Instanzen hinzugefügt werden. Die Oberfläche muss anschließend eine erweiterte Arbeitsliste mit drei Ebenen anzeigen können, die zusätzlich verschiedene Objekt-Typen nach unterschiedlichen Makro-Prozessen gruppieren kann.

Ein im Konzept noch nicht betrachteter Aspekt, ist die Ausführung auf mobilen Endgeräten wie Tablet-PCs oder Smartphones. Diese unterscheiden sich in ihren funktionalen Möglichkeiten nur noch geringfügig von normalen Desktopsystemen und sind auch für normale Endverbraucher erschwinglich. Mit heutigen Webtechnologien ist es möglich, Webanwendungen für unterschiedliche Endgeräte zu erstellen. Dabei muss sich lediglich

die Benutzeroberfläche entsprechend des aufrufenden Browsers anpassen (*Responsive Design*). In manchen Fällen erweist es sich auch als sinnvoll, eine proprietäre Softwarelösung für mobile Geräte zu erzeugen. Dabei wird die Anwendung (*App*) nicht in einem Browser, sondern als eigenständiges Programm auf dem End-Gerät ausgeführt. Der Zugriff auf Daten kann dabei über eine spezielle Schnittstelle (API) oder einen anderen Web-Service erfolgen, der Daten in einem passenden Format zur Verfügung stellt. Die Anwendung stellt die Daten dann in einer passenden Form dar und sendet Änderungen zur Speicherung zurück an den Server.

Durch die genannten Erweiterungen lässt sich die Komponente zu einer umfangreichen Anwendung als Teil des Frameworks PHILharmonicFlows ausbauen, die alle geforderten Eigenschaften eines datenzentrierten Prozess-Management-Systems erfüllt.

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[1] Künzle, V., Reichert, M.: Towards Object-aware Process Management Systems: Issues, Challenges, Benefits. 2009. In: Proc. 10th Int'l Workshop on Business Process Modeling, Development, and Support (BPMDS'09), Springer, LNBIP 29, 197-210.

[2] Künzle, V., Reichert, M.: PHILharmonicFlows: towards a framework for object-aware process management. Journal of Software Maintenance and Evolution: Research and Practice, 23(4): 205-244, Wiley, 2011.

[3] Künzle, V., Reichert, M.: A Modeling Paradigm for Integrating Processes and Data at the Micro Level. In: Proc. 12th Int'l Working Conference on Business Process Modeling, Development and Support (BPMDS'11), London, June 2011, LNBIP 81, Springer, pp. 201-215.

[4] Künzle, V., Reichert, M.: PHILharmonic Flows 3.0 - Execution. Interner Technischer Bericht, Institut für Datenbanken und Informationssysteme, Ulm, Germany, (2010).

[5] Scheb, C.: Entwicklung eines Usability-Konzepts für die Laufzeitumgebung eines datenorientierten Prozess-Management-Systems. Diplomarbeit, Universitat Ulm, (2010).

[6] Pröbstle, A.: Technische Konzeption und Realisierung der Modellierungskomponente fur ein datenorientiertes Prozess-Management-System. Diplomarbeit, Universitat Ulm, (2011).

[7] Wagner, N.: Entwicklung eines Usability-Konzepts fur die Modellierungsumgebung eines datenorientierten Prozess-Management-Systems, Diplomarbeit, Universitat Ulm, (2010).

[8] MacDonald, M., Freeman, A., Szpuszta, M.: Pro ASP.Net 4 in C 2010 (Vierte Edition). 2010. Springer, Berlin.

[9] Evjen, B., Hanselman, S., Rader, D.: Professional ASP.NET 4 in C and VB. 2010. Wiley Publishing, Inc., Indianapolis, Indiana.

[10] RFC 2616: HTTP 1.1, http://www.ietf.org/rfc/rfc2616

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbststandig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Ulm, 31. März 2012 _____

Stefan Schultz