Fakultät für Informatik und Mathematik 07

Masterthesis

Verbesserung des Softwareentwicklungsprozesses
der Landeshauptstadt München
durch modellgetriebene Softwareentwicklung

Improvement of the Software Development Process
of the State Capital of Munich by Model-Driven-Development

März 2016

Autor: B.Sc.-Inf. Martin Kurz
Mail: m.kurz@hm.edu
Matrikelnummer: 02948310

Betreuer: M.Sc.-Wirt.-Inf. Dipl.-Betriebsw. (FH) Claus Straube

Erstprüfer: Prof. Dr. Ulrich Möncke

Zweitprüfer: Prof. Dr. Axel Böttcher

Fakultät für Informatik und Mathematik

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN MÜNCHEN

Die folgende Erklärung ist in jedes Exemplar der Masterarbeit einzubinden und jeweils persönlich zu unterschreiben.

_____ _____
(Familienname, Vorname) (Ort, Datum)

_____ _____ / ___20___
(Geburtsdatum) (Studiengruppe / WS/SS)

Erklärung
Gemäß § 40 Abs. 1 i. V. m. § 31 Abs. 7 RaPO

Hiermit erkläre ich, dass ich die Masterarbeit selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benützt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

_____
(Unterschrift)

Kurzfassung

Die Entwicklung einer Anwendung wurde mit der Einführung des *Prozessmodell IT-Service* bei der Landeshauptstadt München und einer standardisierten Softwareentwicklungsumgebung normiert. Durch die dabei festgelegte Phaseneinteilung entsteht jedoch ein Mehraufwand, wenn zu einem späteren Zeitpunkt eine Änderung an abgenommenen Ergebnissen erforderlich ist. Trotz vorgegebener Werkzeuge ist zudem die Implementierung sehr unterschiedlich. Entsprechend hoch ist der Entwicklungs- und Wartungsaufwand für Anwendungen.

In dieser Ausarbeitung soll, nach eingehender Analyse des Prozessmodells und dem Aufbau zukünftiger Anwendungen, ein modellgetriebener Ansatz zur Prozessoptimierung bei gleichzeitiger Qualitätsverbesserung des Produkts definiert werden. Dazu erfolgt die Spezifizierung einer domänenspezifischen Sprache, die als Machbarkeitsnachweis implementiert und an einem Praxisbeispiel validiert wird.

Abstract

The development of an application has been normalized with the introduction of the *Prozessmodell IT-Service* and a standardized development environment. By devision into separate phases, addition expenses occurs, when a change of previous committed results is required at a later date. In addition the implementation is especially very different despite the prescribed tools. Correspondingly high is the cost of development and maintenance of applications.

After a detailed analysis of the process model and the development of future applications this paper tries to optimize the process while improving the quality of the product using a model-driven approach. For this purpose a specification of a domain-specific language is done. As a proof of concept the language will be implemented and validated on a practical example.

Inhaltsverzeichnis

1 Einleitung

In den vergangenen Jahren hat sich der IT-Bereich der Landeshauptstadt München (LHM) sehr verändert. Grund dafür ist der Beschluss des Stadtrates vom 27. Januar 2010 zur Durchführung des Projektes *Münchner IT - Konkrete Umsetzung und TOP Priorities (MIT-KonkreT)*:

"*Organisiert nach den Grundsätzen eines leistungsfähigen Konzerns wird die IT nach dem Modell der Kernkompetenzfokussierung (KKF) zukunftsweisend für die Anforderungen einer modernen Großstadtverwaltung aufgestellt.*".[1]

Das Konzept der KKF besteht dabei aus drei sogenannten Häusern[2], die eine gemeinsame IT besitzen. Diese Häuser sind

- das dezentrale Informations-, Kommunikations- und Anforderungsmanagement (dIKA),
- die IT-Strategie und IT-Steuerung / IT-Controlling (STRAC) und
- die Informations- und Telekommunikationstechnik der Stadt München (IT@M).[3]

Das dIKA sitzt innerhalb eines jeden Referats, wie zum Beispiel im Kreisverwaltungsreferat (KVR) oder im Personal- und Organisationsreferat (POR).[4] Es agiert dadurch nah an den fachlichen Anforderungen seiner Kunden, den Mitarbeitern oder Bürgern. Ein dIKA besteht aus dem Anforderungsmanagement, fachlich-technischen Diensten und dem lokalen Service-Desk.
STRAC schafft Rahmenbedingungen für eine einheitliche strategische Ausrichtung der IT und unterstützt dabei die dezentrale Informations-, Kommunikations- und Anforderungsmanagements (dIKAs) und IT@M.
Der Eigenbetrieb IT@M wurde im Zuge dieser Neuorganisation[5] als gemeinsame IT gegründet. In diesem zentralen Dienstleister sind die Anwendungen der Referate zusammengeführt und vereinheitlicht worden.

[1] [Münc], Abschnitt: "Neue Ideen einbringen. Neue Wege gehen"
[2] Eine Metapher, die die logische Unterteilung der Aufgabengebiete veranschaulicht.
[3] [Münc], Abschnitt: "Neuorganisation"
[4] Eine vollständige Liste der Referate der Landeshauptstadt München (LHM) steht unter folgendem Link zur Verfügung: http://www.muenchen.de/rathaus/Kontakt/Referate.html
[5] vgl. [Münc], Abschnitt: "Neuorganisation"

Der IT-Dienstleister stellt unter anderem folgende Leistungen für die Referate bereit:[6]

- Technische Lösungsberatung
- Anwendung (Planung, Erstellung und das Betreiben)
- Werkzeuge und Infrastruktur
- Vergabe und Beschaffung

Unterteilt ist IT@M in die Bereiche Anwendung, Betrieb und Infrastruktur. Unterstützt wird diese Aufteilung durch die Definition eines eigenen standardisierten Vorgehens, das sogenannte *Prozessmodell IT-Service*. Es deckt den Lebenszyklus (vgl. Abbildung 1.1) einer Anwendung[7] ab und soll es ermöglichen, den Anforderungen der internen Kunden (Referate) gerecht zu werden (im Detail vgl. Kapitel 2.1).

Abbildung 1.1: Schematische Darstellung (die Länge der Pfeile hat keine Aussage über die Dauer jeder Phase) des Lebenszyklus eines Softwaresystems nach [Bal11], Abb. 1.0-1. Dieser beginnt mit der Spezifikation. Darauf aufbauend wird ein Entwurf des Systems erstellt und anschließend die Implementierung vorgenommen. Über die Installation erfolgt ein Übergang in den Betrieb, bis der Lebenszyklus endet bzw. wieder von vorne beginnt. Das *Prozessmodell IT-Service* der LHM besteht im Kern aus diesen Phasen.

1.1 Motivation

Die Aufteilung in die drei Häuser hat zur Folge, dass am Lebenszyklus eines IT-Service[8] unterschiedliche Bereiche beteiligt sind (vgl. Abbildung 1.2). Zudem wird durch das Modell ein lineares Vorgehen festgelegt. Beides wirkt sich während der Implementierungsphase oftmals durch einen hohen Kommunikationsaufwand zwischen dIKA und IT@M zur Verifizierung der Anforderungen aus. Ist zu diesem Zeitpunkt die Aufnahme weiterer Anforderungen erforderlich, erzeugt dies einen hohen Mehraufwand. Dies liegt der Tatsache zu Grunde, dass die, zur Implementierung verwendeten und in den beiden vorherigen

[6] vgl. [Münb], Abschnitt: "Leistungen von IT@M"
[7] Die Software zur Unterstützung eines Geschäftsprozesses.
[8] "*Ein IT-Service ist die Zusammenfassung von geschäftsprozessunterstützenden IT-Funktionen (bestehend aus Hardware, Software- und Aktivitätselementen), die vom Kunden als eine in sich geschlossene Einheit wahrgenommen werden.*" [Müna], Abschnitt: "IT-Service". Ein IT-Service ist zum Beispiel die Anwendung zur Fundsachenverwaltung.

Phasen (Spezifikation und Entwurf) erstellten, Dokumente nachträglich angepasst werden müssen.[9] Eine längere Laufzeit des Prozesses ist die Folge.

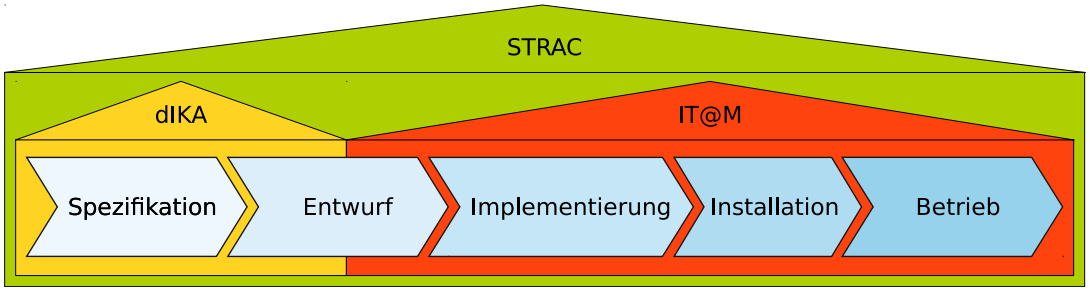


Abbildung 1.2: Erweiterte Darstellung des Lebenszyklus eines Softwaresystems nach [Bal11], Abb. 1.0-1. In Vergleich zu Abbildung 1.1 wird die Aufteilung in die drei Häuser veranschaulicht. In der Phase Entwurf erfolgt eine Zusammenarbeit von dIKA und IT@M mit anschließender Übergabe zur Umsetzung an IT@M.

Ziel dieser Ausarbeitung ist eine Optimierung des Prozesses zum Anbieten eines IT-Service bei gleichzeitiger Erhöhung der Qualität.

1.2 Aufgabenstellung

Mit Hilfe von einem modellgetriebenen Ansatz soll der Prozess zur Entwicklung eines IT-Service optimiert werden, um die Phasen Spezifikation, Entwurf und Implementierung zu vereinfachen und zeitlich zu verkürzen. Dieser Ansatz ist in den bestehenden Prozess einzubinden.

Die Optimierung des Lebenszyklus soll durch eine bessere Standardisierung des Produkts (IT-Service) und Erstellung von Prototypen der Anwendungen zur Verifikation der Anforderungen erfolgen. Die Standardisierung bezieht sich auf die verwendete Softwarearchitektur, die anhand einer vorgegebenen Referenzimplementierung, die den zukünftig gewünschten Aufbau besitzt, abzuleiten ist. Mit Hilfe der daraus gewonnen Informationen ist eine domänenspezifische Sprache (DSL) zu definieren, die von den dIKAs zur Beschreibung ihrer benötigten Anwendungen, im Sinne von Modellen, verwendet werden kann. Diese, auf der DSL basierenden, Modelle sollen eine Erstellung eines Prototyps der Anwendungen durch einen Generator ermöglichen. Anhand der Prototypen ist eine Verifikation der Anforderungen möglich und eine kürzere Rückkopplung gegeben (vgl. Abbildung 1.3). Der Software-Prototyp soll anschließend zur Weiterentwicklung dienen, um

[9] Im Vergleich zu agilen Vorgehensmodellen ist das *Prozessmodell IT-Service* nicht dafür gedacht auf spätere Änderungen reagieren zu können.

diesen Aufwand ebenfalls zu reduzieren und die Standardisierung des Produkts herbeizuführen.

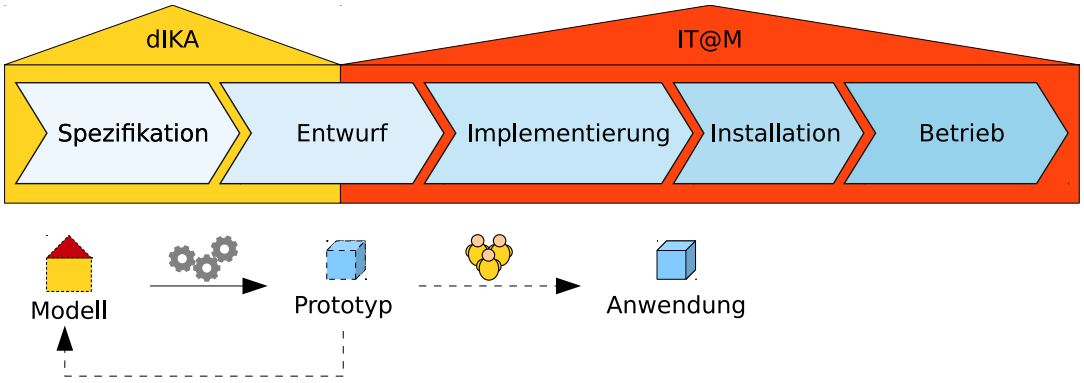


Abbildung 1.3: Darstellung in eigener Notation des modellgetriebenen Ansatzes. Anhand eines Modells der geforderten Anwendung ist die Erstellung von einem Prototyp möglich. Dieser dient entweder als Grundgerüst zur Entwicklung oder kann durch Verändern des Modells nochmals angepasst werden.

Für die Generierung wird auf ein vorgegebenes frei verfügbares Framework[10] zurückgegriffen, wodurch keine Elemente des klassischen Compilerbaus wie Interpreter, Parser, usw. benötigt werden.

Die Zielgruppe dieser Ausarbeitung sind Fachleute[11] der Informatik.

1.3 Überblick

Die Aufteilung der Kapitel ist angelehnt an die Phaseneinteilung von Mernik et al., die den Prozess zur Entwicklung einer DSL beschreiben.[12] Die darin beschriebene Entscheidung zur Entwicklung einer DSL wurde im Zuge dieser Ausarbeitung bereits getroffen (siehe Aufgabenstellung).

Das Hauptkapitel 2 befasst sich mit den Grundlagen. Dabei wird auf das Prozessmodell der LHM eingegangen und die gewünschte Architektur von künftigen Anwendungen anhand der Referenzimplementierung analysiert. Zudem erfolgt eine Vorstellung des Stands der Technik anhand einschlägiger Literatur im Themenbereich der modellgetriebenen Entwicklung. Die beschriebenen Grundlagen werden im Anschluss kritisch betrachtet, um Vor- und Nachteile aufzuzeigen.

[10] Ein Grundgerüst mit gängigen Lösungen und/oder Bausteinen, dass die Entwicklung unterstützt und zugleich beschleunigen kann.
[11] Im Sinne eines Bachelor-Abschlusses bzw. mehrjähriger Berufserfahrung.
[12] vgl. [MHS05], Kapitel "2.1. Pattern classification"

Daraufhin wird das Design in Hauptkapitel 3 erarbeitet. Dieses setzt sich zusammen aus der Integration der modellgetriebenen Entwicklung in den bestehenden Prozess und der Definition der Modellierungssprache. Zur Ermittlung des benötigten Umfangs dieser, dient die Analyse der Referenzimplementierung hinsichtlich generischer Anteile. Darauf aufbauend wird das Meta-Modell definiert und die Verwendung zur Dokumentengenerierung geschildert.

In Hauptkapitel 4, der Umsetzung, wird der Generator anhand des, in Hauptkapitel 3 definierten, Meta-Modells entwickelt. Anschließend erfolgen Testungen des Generators und dessen Verwendung an einem Praxisbeispiel zur Validierung des Vorgehens.

Die Ausarbeitung endet im Fazit mit einer Zusammenfassung und dem Ausblick.

2 Grundlagen

Das Ziel von diesem Hauptkapitel ist die Herstellung von Grundlagen, auf welche in den nachfolgenden Kapiteln zurückgegriffen werden kann. Im Hinblick darauf wird anfangs der Prozess der LHM zum Anbieten eines IT-Service und die, zukünftige Architektur enthaltene, Referenzimplementierung vorgestellt. Anschließend erfolgt der Einstieg in den Themenbereich der modellgetriebenen Softwareentwicklung mit deren Definition und Abgrenzung. Daran anknüpfend wird der Stand der Technik anhand von Literatur zur modellgetriebenen Entwicklung vorgestellt. Am Ende dieses Hauptkapitels erfolgt eine kritische Betrachtung der vorgestellten Grundlagen, woraus Vor- und Nachteile resultieren.

2.1 Prozessmodell IT-Service

Die Entwicklung, Bereitstellung und der Betrieb eines IT-Service wird anhand eines eigens erarbeiteten und eingeführten Prozessmodells durchgeführt. Dieses befindet sich in der aktuellen Version (2.1) in einer kompletten Darstellung im Anhang A.1. Relevante Ausschnitte davon sind gesondert in den jeweiligen Unterkapiteln abgebildet.

Damit die modellgetriebene Softwareentwicklung in Hauptkapitel 3 realisierbar eingebunden werden kann, ist die Betrachtung des Prozessmodells vonnöten.

Das Prozessmodell deckt den bereits in der Einleitung vorgestellten Lebenszyklus eines Softwaresystems (vgl. Abbildung 1.1) ab. Angelehnt ist das *Prozessmodell IT-Service* an die IT Infrastructure Library (ITIL)[13] und an das Vorgehensmodell Rational Unified Process (RUP)[14] zur Softwareentwicklung.

ITIL und **RUP** stellen unter anderem *Best Practices*[15] bereit, wie sich IT-Dienstleistungen anhand der Bedürfnisse des Unternehmens und ihrer Kernprozesse ausrichten können.

Der Lebenszyklus eines Service wird in **ITIL** in folgende fünf Phasen eingeteilt:[16]

- Service Strategy
- Service Design

[13] Für detaillierte Informationen zur ITIL sei an dieser Stelle auf [Off11] verwiesen.
[14] Ausführliche Informationen zur RUP sind [Kru03] zu entnehmen.
[15] deutsch: bewährte Methoden
[16] vgl. [AXE], Abschnitt: "What is ITIL®?"

- Service Transition
- Service Operation
- Continual Service Improvement

Angelehnt daran besitzt das *Prozessmodell IT-Service* folgende Phasen (vgl. Anhang A.1):

- Service Portfolio Management
- Service Design
- Service Transition
- Service Operation

Die Phase *Service Portfolio Management* befasst sich mit der Planung und dem Mehrwert von IT-Dienstleistungen. Sie ist zudem unterteilt in die IT-Vorhabensplanung und das Auftragsmanagement. Das *Service Design* bzw. Anforderungsmanagement ist für die Konzeption und die Entwicklung des Service, mit den dabei zu beachtenden Abhängigkeiten (Kapazitäten, Verfügbarkeit, ...), verantwortlich. Diese Phase ist unterteilt in die Anforderungsqualifizierung, zur Erfassung und Beurteilung, sowie der Anforderungsbearbeitung, zur Beschreibung eins Service.
Anschließend wird der Service in der *Service Transition* bzw. dem Transition Management entwickelt und in den produktiven Betrieb übernommen. Die darin enthaltenen Prozesse sind *Realisierung und Test*, *Abnahme* und *Einführung*.
Die *Service Operation* ist im Anschluss daran für den Betrieb verantwortlich. Dabei erfolgt die Unterteilung in das Incident Management, das Problem Management, Request Fulfilment und dem Betrieb.

Von **RUP** übernommen wurden die, jeden (Teil-)Prozess abschließenden, Meilensteine[17] und die feste Struktur bestehend aus Aktivitäten und Arbeitsabläufen.[18] Der Abschluss mit Meilensteinen sorgt für einen insgesamt sequentiell stattfindenden Prozess, obwohl innerhalb einer Phase die Aktivitäten zum Teil parallel ablaufen können.

Für die Entwicklung sind die Phasen *Service Design* und *Transition Management* aus dem *Prozessmodell IT-Service* von Bedeutung. Diese werden im Anschluss in den Unterkapiteln 2.1.1 und 2.1.2 detailliert beschrieben, da darin der Mehraufwand bei späteren Änderungen entsteht. Die technische Unterstützung der Entwicklung wird im Unterkapitel 2.1.3 erläutert.

[17] Wichtiges definiertes Ereignis in einem Prozess, z.B. Übergang von einer Phase in die Nächste.
[18] vgl. [Sof98], Seite 3: "Process Overview" und [Sof98], Seite 7ff:"Static Structure of the Process"

2.1.1 Phase "Service Design" im Detail

In der Phase Service Design wird zu Beginn (vgl. Abbildung 2.1) der Anforderungsqualifizierung das Fachkonzept (FK) erstellt. Dieses beinhaltet zum Beispiel die zu erfüllenden Ziele, die abzudeckenden Anwendungsfälle (Use Cases) und das benötigte fachliche Datenmodell. Auf dieser Basis wird eine Make-Buy-Use-Compose (MBUC) Entscheidung getroffen. Dabei entscheidet sich, ob der Service entweder innerhalb IT@M entwickelt (Make), gekauft (Buy), ein vorhandener verwendet (Use) oder an einem anderen angeknüpft (Compose) wird. Unabhängig von der Entscheidung wird in der Anforderungsbearbeitung die Systemspezifikation (SysSpec) erarbeitet. In dieser sind unter anderem die IT-Architektur, das Systemdesign und die technischen Schnittstellen im Detail beschrieben. Ebenso findet die Erstellung des Testkonzeptes (TK), in dem die Testfälle definiert und das Testvorgehen geplant wird, statt. In der anschließenden Beschaffung erfolgt entweder die Beauftragung eines externen Dienstleisters (Buy-Entscheidung) und/oder die Beschaffung der benötigten Hardware.

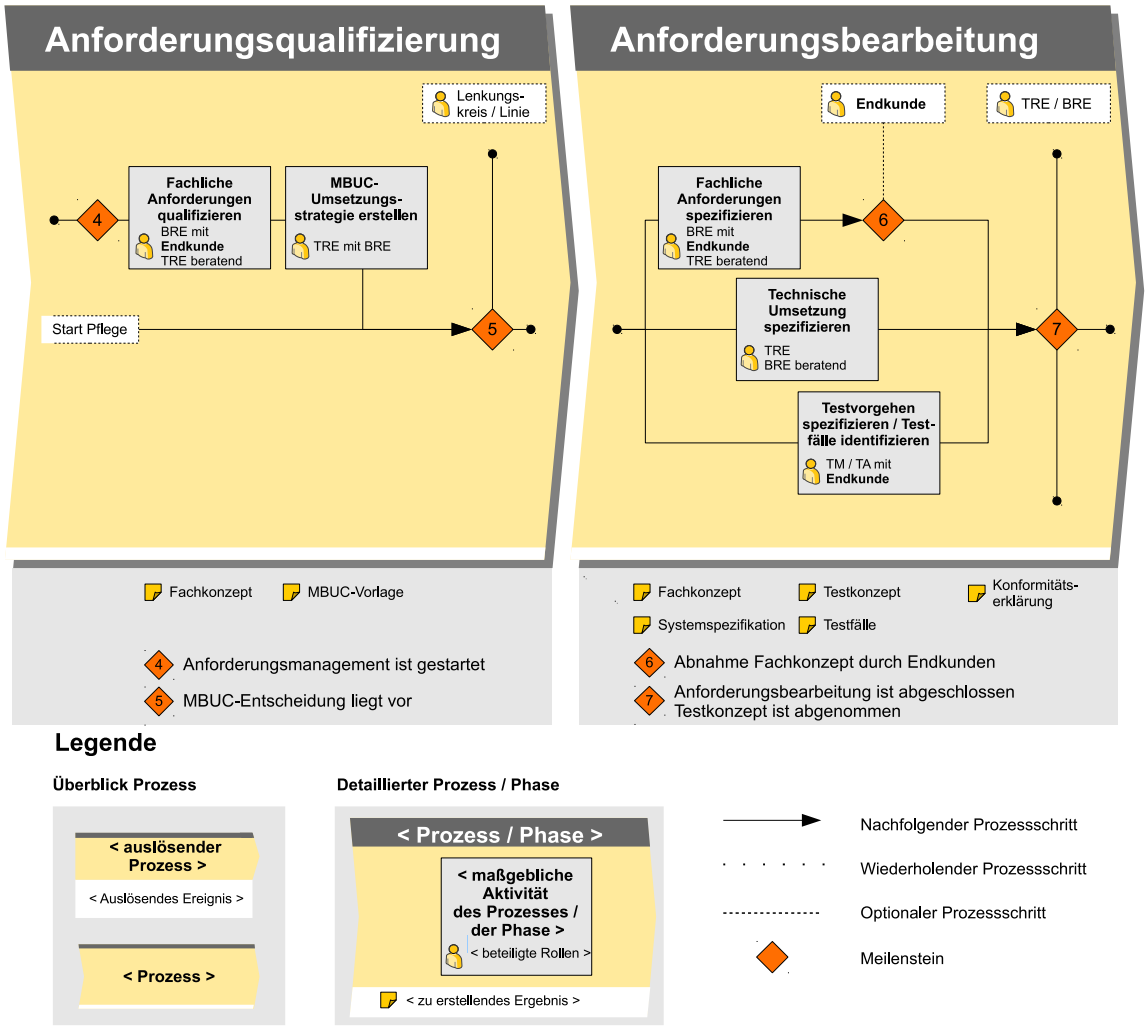


Abbildung 2.1: Darstellung der Phase Service Design inklusive Legende aus dem *Prozessmodell IT-Service* der LHM. Die bisher nicht erläuterten Kürzel sind: Testanalyst (TA), Testmanager (TM) und Technical Requirement Engineer (TRE). Die Phase *Beschaffung* nach der *Anforderungsbearbeitung* wurde in diesem Ausschnitt entfernt, da diese keinen Einfluss auf Eigenentwicklungen hat.

Unterstützt wird die Phase Service Design durch standardisierte Vorlagen für die zu erstellenden Dokumente (FK, SysSpec, TK), sowie durch entsprechende Leitfäden zur gewünschten Befüllung. Beteiligt an dieser Phase sind primär der Endkunde und der Business Requirement Engineer (BRE).

2.1.2 Phase "Transition Management" im Detail

Das *Transition Management* beinhaltet die Prozesse zur Verwirklichung der Lösung. Die Phase startet im Prozess *Realisierung und Test* mit folgenden parallel ausführbaren Aktivitäten: Planung der Einführung und des Betriebs, Umsetzung der Lösung und Erstellung der Testspezifikation (vgl. Abbildung 2.2). Dazu wird auf die SysSpec und das TK aus der

vorangegangenen Phase zurückgegriffen. Im Anschluss werden Integrations- sowie Systemtests durchgeführt, Service-Level-Agreements (SLAs)[19] vorbereitet und das System zum Abschluss des Prozesses *Realisierung und Test* dokumentiert. In der *Abnahme* überprüft der Endkunde den Service, ob dieser die definierten Anforderungen erfüllt und schließt das SLAs mit dem Service-Level-Manager (SLM) ab. Danach kann der Service in der Phase Einführung in Betrieb genommen und Schulungen für die Anwender durchgeführt werden. Gegebenenfalls wird ein Rollout[20] ausgeführt und der Early Life Support[21] beginnt.

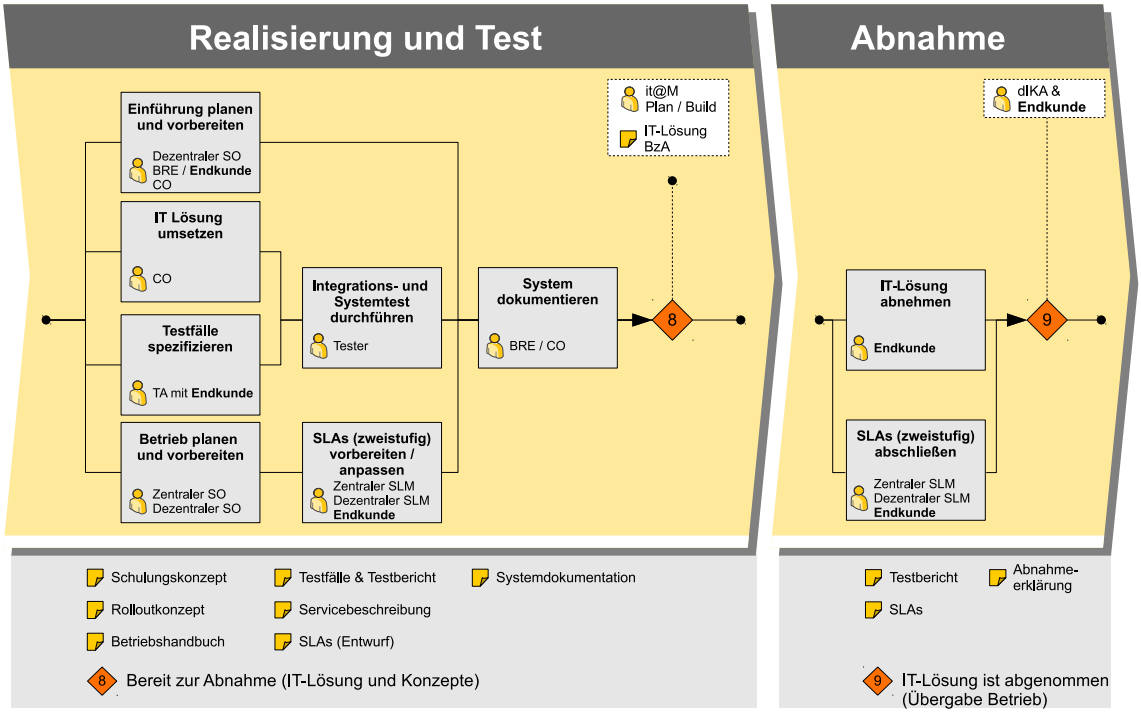


Abbildung 2.2: Darstellung der Phase Transition Management aus dem *Prozessmodell IT-Service* der LHM. Die Kürzel SO und CO stehen für Service Owner und Component Owner. Der Service Owner ist für die Sicherstellung des vereinbarten Betriebs zuständig und der Component Owner für die Erstellung des IT-Service. Die Phase *Einführung* nach der *Abnahme* wurde in diesem Ausschnitt entfernt, da diese keinen Einfluss auf die Entwicklung hat.

Technisch unterstützt wird das Transition Management bei einer Eigenentwicklung durch eine vorgegebene Softwareentwicklungsumgebung (SEU), auf welche im anschließenden

[19] "*Ein Service-Level-Agreement (SLA) ist eine schriftliche Vereinbarung zwischen den Serviceerbringern / Lieferanten und den Serviceabnehmern / Kunden über die Service-Level-Ziele, die Servicequalität und die Leistungsverrechnung für die im Servicekatalog aufgeführten IT-Services.*" [Müna], Abschnitt: "Service-Level-Agreement"

[20] Installation und Konfiguration von Hardware und/oder Software an einem Stichtag.

[21] In den ersten Wochen nach einer Einführung bzw. dem Update eines Service soll mit dieser Unterstützung der Betrieb, durch kurzfristige durchführbare Fehlerbehebungen, sichergestellt werden.

Unterkapitel eingegangen wird. Beteiligt an dieser Phase sind primär der Component Owner (CO) und der Service Owner (SO)[22].

2.1.3 Technische Unterstützung im Detail

Die SEU wurde innerhalb der LHM entwickelt und dient als Vorgabe für Eigenentwicklungen. Primär ist diese unterteilt in die Software-Produktionsstraße (SPS) und dem Entwicklerarbeitsplatz, wobei die jeweils eingesetzten Werkzeuge standardisiert wurden (vgl. Abbildung 2.3). Beispielsweise ist eine der Vorgaben, dass als Integrated Development Environment (IDE) Eclipse[23] zur Entwicklung verwendet wird. Der Aufbau der SEU ermöglicht eine professionelle Anwendungsentwicklung im Team.

Auf eine detaillierte Beschreibung der SEU wird an dieser Stelle verzichtet, da sich durch den modellgetriebenen Ansatz keine Änderungen an dieser ergeben.

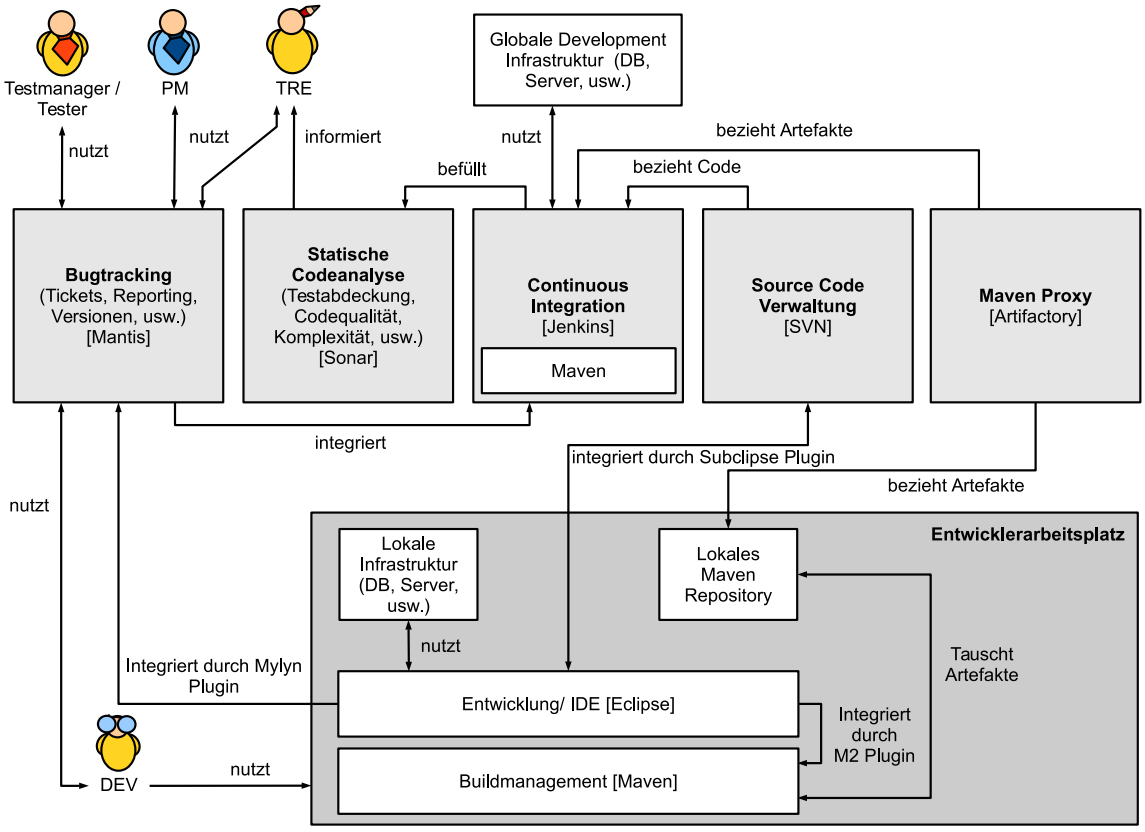


Abbildung 2.3: Darstellung der Softwareentwicklungsumgebung der LHM. In eckigen Klammern steht das verwendete und standardisierte Werkzeug. Die Pfeile stellen die Beziehungen dar. Enthaltene Abkürzungen: Projektmanager (PM), Entwickler (DEV) und Technical Requirement Engineer (TRE). Quelle: [Str13]

[22] Hierbei handelt es sich um Rollen, die von Personen wahrgenommen werden. Entsprechend kann eine Person mehrere Rollen besitzen.

[23] http://www.eclipse.org/

Zur Lösung häufiger Probleme, wie zum Beispiel Zeichensatzkonvertierungen, existieren zudem interne Pakete, die vom Entwickler genutzt werden können. Die Bereitstellung erfolgt über einen *Maven Proxy*, der zugleich Artefakte[24] aus dem Internet zwischenspeichert.

2.2 Referenzimplementierung

Die bereitgestellte Referenzimplementierung der IT-Architektur der LHM hat für diese Ausarbeitung mehrere Bedeutungen hinsichtlich des modellgetriebenen Ansatzes, die in Abbildung 2.4 grafisch veranschaulicht sind. Das übergeordnete Ziel ist die Verifikation des geplanten Vorgehens. Um diese durchzuführen soll anhand eines Referenzmodells eine Generierung stattfinden, deren Ergebnis mit der ursprünglichen Referenzimplementierung zu vergleichen ist. Damit eine Erstellung des Referenzmodells realisierbar ist, ist die Definition eines geeigneten Metamodells vonnöten. Sowohl das Metamodell als auch das Modell sind von der Referenzimplementierung abhängig.

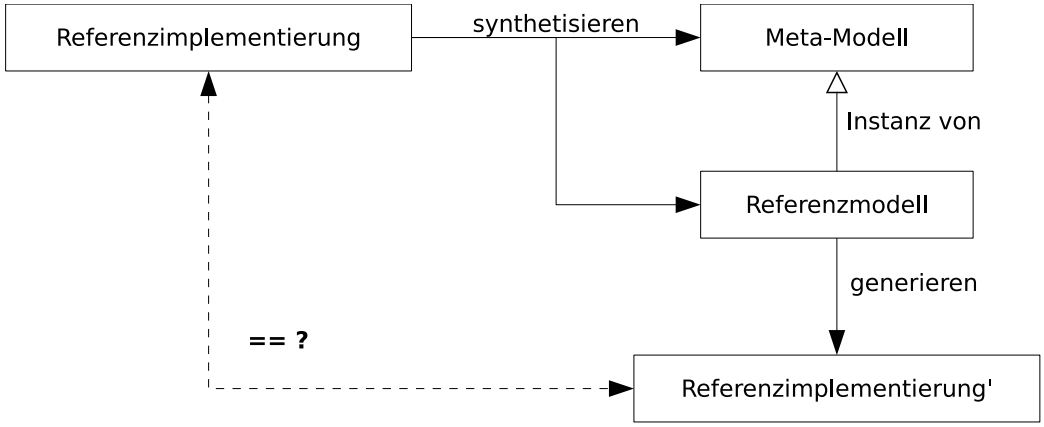


Abbildung 2.4: Darstellung der Bedeutung der Referenzimplementierung. Die Pfeile verdeutlichen die Zusammenhänge bzw. Abhänigigkeiten.

Innerhalb der nachfolgenden Erläuterungen zur Referenzimplementierung werden mehrere Begriffe verwendet, deren Zusammenhang in Abbildung 2.5 grafisch aufbereitet ist. Von zentraler Bedeutung ist der sogenannte Microservice.

Ein Microservice realisiert eine bestimmte Funktion, die dieser für andere Microservices zur Verfügung stellt. Dazu besitzt der Microservice grundlegende Operationen und Geschäftsaktionen. Grundlegende Operationen dienen zur Verwaltung von Daten. Innerhalb von Geschäftsaktionen kann die erforderliche fachliche Logik ausgeführt werden. Anhand

[24] Ein Produkt, dass während der Softwareentwicklung entstanden ist und zur Wiederverwendung und/oder gemeinsamen Nutzung bereitsteht.

dem Beispiel in den Abbildungen 2.5 und 2.7 kann dadurch zu einer Person in der Bürgerverwaltung eine Anschrift aus der Straßenverwaltung zugeordnet werden.[25]

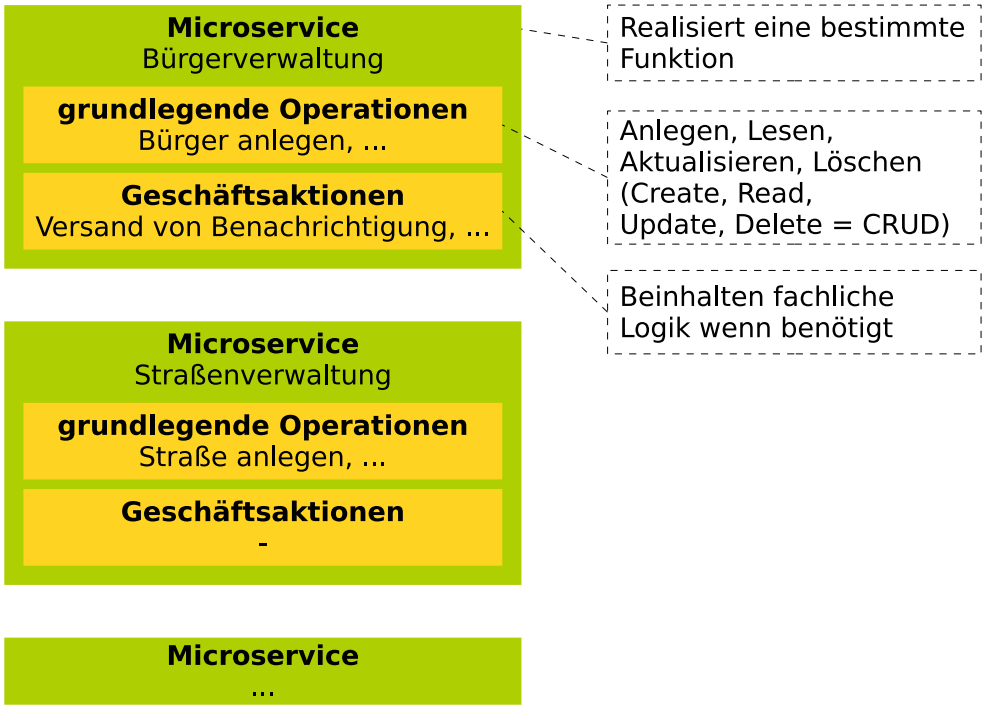


Abbildung 2.5: Zusammenhang der Begriffe der Refrenzimplementierung mit einem dazu passenden Beispiel (auf der linken Seite) und einer kurzen Erläuterung (auf der rechten Seite).

2.2.1 Aufbau der Referenzimplementierung

Die vorhandene Referenzimplementierung besteht aus zwei Microservices, die untereinander kommunizieren. Einer davon, der Graphical User Interface (GUI) Microservice (MS), bietet die Oberfläche für einen Benutzer an. In Abbildung 2.6 wird die **logische Sicht** auf das System dargestellt.[26] Die entwickelte Bürgerverwaltung dient innerhalb dieser Ausarbeitung als Beispiel.

[25] Durch den Aufbau von einzelnen Microservices bleibt die Stabilität des Systems bei einem Ausfall erhalten. Steht beispielsweise die Straßenverwaltung nicht mehr zur Verfügung, funktioniert die Bürgerverwaltung, abgesehen der Einschränkung, dass keine Anschrift zugeordnet werden kann, weiterhin.

[26] Ein Vergleich der Architektur der Referenzimplementierung mit einem einzelnen Pattern, wie MVC ist nicht zweckmäßig, da der Umfang und die Aufteilung keine sinnvolle Herstellung von Parallelen untereinander erlaubt.

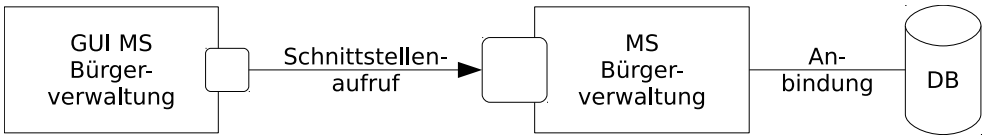


Abbildung 2.6: Logische Sicht auf die Referenzimplementierung. Die GUI schickt Anfragen über eine Schnittstelle an den MS, der wiederum selbst Daten verwaltet und dadruch auf eine Datenbank zugreifen kann.

Ein Microservice stellt über seine Schnittstelle die verwalteten Daten, grundlegende Operationen und Geschäftsaktionen bereit. Geschäftsaktionen können dabei Anforderungen, wie eine zufällige Auswahl an Datensätzen bis hin zu einer komplexen statistischen Auswertung bei gleichzeitiger Abfrage anderer Microservices, umsetzen. Folglich ist es möglich, dass ein Microservice selbst keine Daten verwaltet und ausschließlich Geschäftsaktionen bereitstellt. Die Abbildung 2.7 stellt den Zugriff von der Bürgerverwaltung auf den Microservice zur Straßenverwaltung dar.

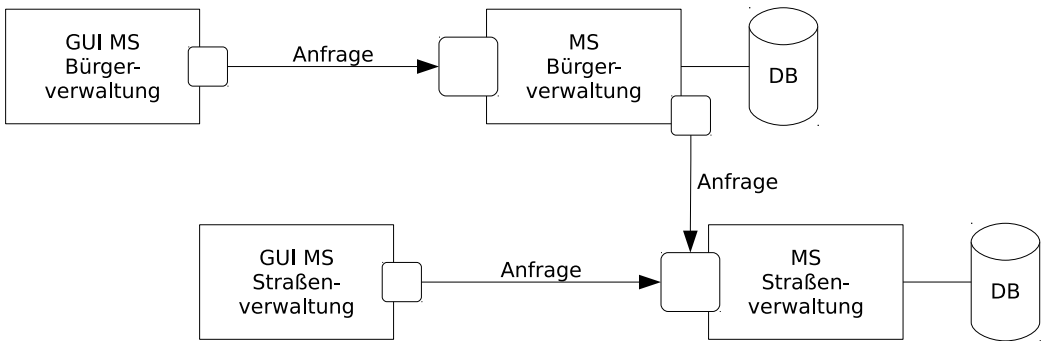


Abbildung 2.7: Logische Sicht auf die Referenzarchitektur. Der Microservice Bürgerverwaltung kann auf den Microservice Straßenverwaltung zugreifen.

Die Kopplung der Microservices ist dynamisch. In den Clients zum Zugriff darauf sind keine festen Uniform Resource Identifiers (URIs) hinterlegt. Lediglich der Name des kompatiblen Microservice und ein zentrales Register ist bekannt. Über dieses bekommt ein Microservice Auskunft, welchen URI der gesuchte Microservice besitzt.[27] Dazu registrieren sich die Microservices beim Start am Register (vgl. Abbildung 2.8).

[27]Das Register bietet weitere, in der Referenzimplementierung nicht genutzte, Funktionen, wie zum Beispiel die Statusüberwachung der registrierten Microservices oder eine Lastverteilung, an.

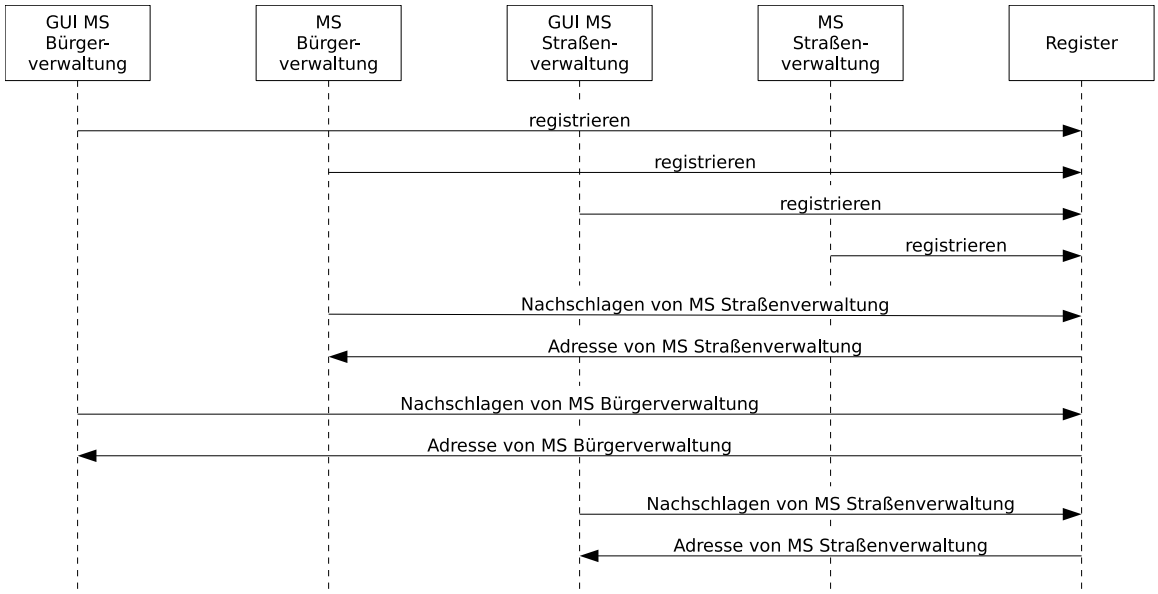


Abbildung 2.8: Sequenzdiagramm zur Darstellung der Interaktionen mit dem Register anhand dem Beispiel aus Abbildung 2.7.

Aus **Prozesssicht** verwendet ein Microservice einen Client zum Zugriff auf das angebotene Application Programming Interface (API). Die Abbildung 2.9 stellt dies am Beispiel der Abfrage von Bürgern dar. Der GUI MS stellt die Oberfläche zur Verfügung und greift mit Hilfe der Client-Bibliothek auf den MS zum Auslesen von Daten zu. Die Aufteilung der Swimlanes orientiert sich anhand der Unterteilung der vorhandenen Projektstruktur.

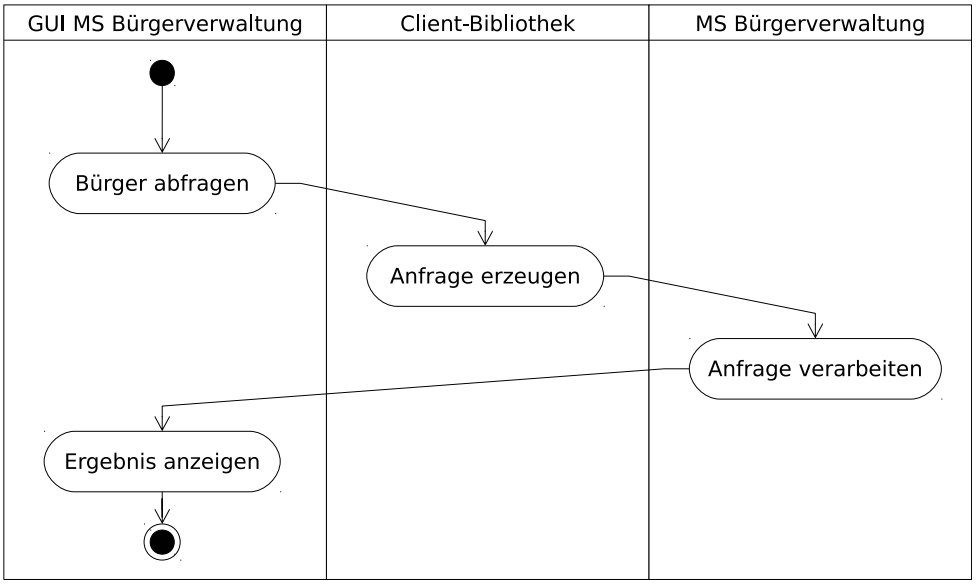


Abbildung 2.9: Aktivitätsdiagramm der Referenzimplementierung am Beispiel zur Abfrage von Bürgern. Der GUI MS nutzt die Client-Bibliothek und greift auf den MS zu.

Aus **Entwicklungssicht** besteht die vollständige Referenzimplementierung aus folgenden Einzelprojekten:

- 🗀 referenzimplementierung-microservice (der Microservice)
- 🗀 referenzimplementierung-client-lib (die Client-Bibliothek für den Microservice)
- 🗀 referenzimplementierung-gui (der GUI Microservice)

Diese Projekte nutzen wiederum mehrere Bibliotheken für übergreifende Funktionalitäten, wie Basisdienste zur Authentifizierung oder zum Datenaustausch. Nachfolgend wird die Grundlage der Referenzimplementierung vorgestellt und daran anschließend im Detail jedes Projekt betrachtet. Auf die allgemein verfügbaren Bibliotheken wird dabei nicht eingegangen.

In der Anwendungsentwicklung ist es Konvention, dass englische Begriffe zur Bezeichnung verwendet werden. Aus diesem Grund und der Einheitlichkeit wird nachfolgend teilweise auf die deutsche Übersetzung verzichtet, wenn es dafür keine allgemeinverständliche Übersetzung gibt. Zum Beispiel *View* als Begriff: eine direkte deutsche Übersetzung wäre Sicht, wobei im Kontext von Webanwendungen Seite geeigneter ist.

2.2.2 Grundlage der Referenzimplementierung

Die Architektur der Referenzimplementierung basiert auf Domain-Driven Design (DDD). Der primäre Fokus von DDD liegt auf der Entwicklung anhand der benötigten Fachlichkeit der Domäne[28]. Dazu wird in DDD eine gemeinsame Sprache, in der Literatur als *Ubiquitous Language* bezeichnet, zwischen dem Programmierer und den Geschäftsprozessen der Domäne geschaffen. Evans beschreibt die Herangehensweise in seinem Buch "*Domain-driven Design: Tackling Complexity in the Heart of Software*" [Eva04] und erläutert die verwendeten Pattern[29]. Nachfolgend werden mehrere, in der Referenzimplementierung verwendete, Pattern von DDD vorgestellt.[30]

- Als **Entity** wird ein Objekt bezeichnet, dass permanent die Identität behält, obwohl sich die Eigenschaften ändern können. Ein Bürger bleibt zum Beispiel trotz Namensänderungen stets der gleiche. Identifiziert werden Entities durch eine eindeutige ID.[31]
- Ein **Value Object** unterscheidet sich zu einer Entity dahingehend, dass die einzelne Instanz keine essentielle Bedeutung für die Domäne besitzt. Eine Identifizierung ist im Gegensatz zu einer Entity nicht erforderlich, dafür sollten diese unveränderbar

[28] Im DDD stellt eine Domäne einen abgeschlossenen Bereich mit ähnlichen fachlichen Anforderungen dar.
[29] Ein bewährter Lösungsansatz.
[30] Für eine vollständige Liste der Pattern sei an dieser Stelle auf [Eva15] verwiesen.
[31] vgl. [Eva04], Hauptkapitel 5: "ENTITIES (A.K.A. REFERENCE OBJECTS)", Seite 89ff

sein. In einer Anwendung zur Bürgerverwaltung kann der Personalausweis beispielsweise ein solches Value Object sein. Eine nachträgliche Änderung ist (fachlich) nicht möglich und der Personalausweis liefert nur zusammen mit einer Bürger-Entity ein Mehrwert. Durch Value Objects soll die Komplexität der Anwendung nicht unnötig erhöht und die Performance gesteigert werden.[32]

- Die, im vorherigen Beispiel beschriebene, Zusammenfassung eines Bürgers mit einem Personalausweis wird als **Aggregate** bezeichnet. Damit soll sichergestellt werden, dass der Personalausweis nicht ohne einen Bürger bestehen kann. Es liegt eine Existenzabhängigkeit bzw. Komposition vor.[33]
- Ein **Service** besitzt keinen internen Status und bietet die Funktionen für die Domäne an.[34] Ein Beispiel hierfür kann ein Dienst zur Auswertung der übermittelten Daten sein.
- **Repositories** kapseln den Zugriff auf persistente Value Objects und Entities. Dabei stellen sie mehr Möglichkeiten zur Abfrage bzw. Suche dieser bereit und verbergen jeglichen Datenbankzugriff.[35]
- **Domain Events** charakterisieren fachliche Ereignisse, die unter Umständen ein oder mehrere Änderungen an den fachlichen Objekten durchführen, die nicht als Aggregate zusammengefasst sind.[36] Zum Beispiel könnte das Ereignis "Umzug" in einer Bürgerverwaltung abgesehen der Adressumstellung weitere Änderungen betreffen.
- Der **Bounded Context** legt den Bereich für ein, mit der gemeinsamen Sprache angefertigtes, Modell fest. Die Referenzimplementierung der Bürgerverwaltung ist ein Beispiel dafür. Eine Anwendung zur Straßenverwaltung besitzt im DDD einen eigenen Bounded Context.[37]

Die technische Umsetzung einiger DDD Pattern mit dem, in der Referenzimplementierung verwendeten, Spring-Framework beschreibt Oliver Gierke auf seiner Website (siehe [Gie15]). Ein Beispiel daraus ist die **Aggregate Root**. Damit werden mehrere Objekte einer Domäne als eines behandelt, wobei ein Zugriff nur über ein Objekt möglich ist.[38] Zum Beispiel kann in einer Anwendung zur Bürgerverwaltung das Anlegen von einem

[32] vgl. [Eva04], Hauptkapitel 5: "VALUE OBJECTS", Seite 97ff
[33] vgl. [Eva04], Hauptkapitel 6: "AGGREGATES", Seite 125ff
[34] vgl. [Eva04], Hauptkapitel 5: "SERVICES", Seite 104ff
[35] vgl. [Eva04], Hauptkapitel 6: "REPOSITORIES", Seite 147ff
[36] vgl. [Eva15], Hauptkapitel 2: "Domain Events *", Seite 13
[37] vgl. [Eva04], Hauptkapitel 14: "BOUNDED CONTEXT", Seite 335ff
[38] vgl. [Fow13], Abschnitt: "DDD_Aggregate"

Personalausweis (Value Object) nur über einen Bürger (Entity) erfolgen.

2.2.3 Aufbau eines Microservice

Jeder Microservice der Referenzimplementierung nutzt das Spring Boot Framework[39].[40] Spring Boot verfolgt das *Convention Over Configuration* Paradigma[41], wodurch mit wenig Anpassungsaufwand Spring Enterprise-Anwendungen (Geschäftsanwendungen) entwickelt werden können. Von Vorteil ist ein bereits enthaltener interner Servlet-Container[42], sodass die standardmäßig generierte jar-Datei direkt ausgeführt werden kann.[43]

Auf Klassenebene[44] besteht der Microservice zur Verarbeitung von grundlegenden Operationen für jede Entity und jedem Value Object aus je einem eignen Repository (vgl. Abbildung 2.10, *BuergerRepository*). Zur Verarbeitung von Geschäftsaktionen dient ein Controller und eine Service-Klasse. Es erfolgt dabei eine interne Übertragung von Entities, Value Objects und Domain Events, wie sie in Unterkapitel 2.2.2 als Pattern von DDD vorgestellt wurden.

Die **Kommunikation im Microservice** beginnt an der Schnittstelle, die entweder direkt das Repository für grundlegende Operationen oder den Controller zur Verarbeitung von Geschäftsaktionen aufruft (vgl. Abbildung 2.10). Das Repository verbirgt den Zugriff auf die Datenbank und die nötigen Abfragen zur Verarbeitung von Anfragen.

Aufrufe von Geschäftsaktionen werden an einen Controller geleitet (vgl. Abbildung 2.10, erster Pfeil von links mit der Beschriftung ”GA”). Dieser überprüft die übergebenen Parameter auf ihre Gültigkeit und ruft den dafür vorgesehenen Service zur Verarbeitung auf. Die fachliche Logik ist in der Service-Klasse umgesetzt. Diese kann sämtliche Funktionalität implementieren, wie den Aufruf von einem anderen Microservice mit Hilfe dessen Representational State Transfer (REST)-Clients (vgl. Abbildung 2.10, gestrichelt dargestellter Pfeil) oder den Zugriff auf die eigene Datenbank über die Repositories.

Die Abbildung 2.10 stellt zusammenfassend die davor beschriebenen Kommunikationswege im Microservice grafisch dar.

[39] http://projects.spring.io/spring-boot/

[40] Von Spring genutzte Bibliotheken und Frameworks sind mit der *Apache License* lizenziert und erlauben eine freie Verwendung. Weitere Informationen unter http://www.apache.org/licenses/LICENSE-2.0.html

[41] Als Paradigma wird die grundsätzliche bzw. übliche Denkweise beschrieben. In diesem Zusammenhang ist die übliche Konfiguration bereits standardmäßig vorgegeben. Eine manuelle Anpassung ist nur für das vom Standard Abweichende erforderlich.

[42] In Java eine Klasse, die speziell das javax.servlet.Servlet-Interface implementiert und damit HTTP-Anfragen annehmen und beantworten kann Der Container ist die dafür benötigte Umgebung, damit das Servlet ausgeführt werden kann.

[43] vgl. [Sofb], Abschnitt: ”Spring Boot makes it easy [...]”

[44] Eine Klasse: Ein Objekttyp von Java mit Eigenschaften und Methoden.

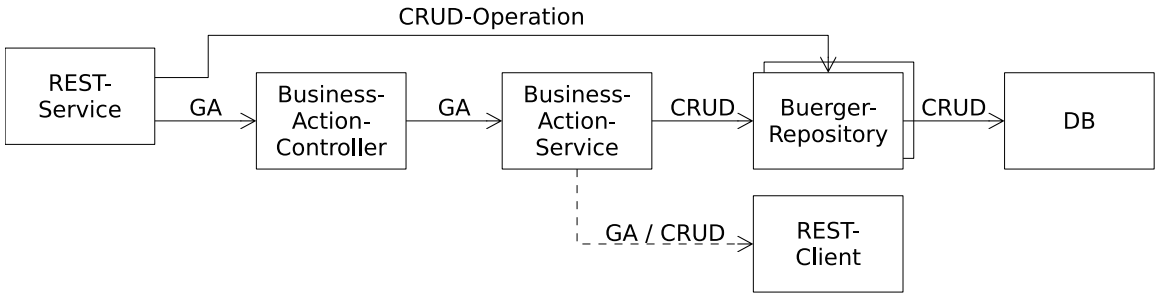


Abbildung 2.10: Beispielhafte Kommunikationswege innerhalb des Microservices mit einer Buerger-Entity. Die Pfeile stellen Aufrufe dar. GA dient als Abkürzung für Geschäftsaktion, welche eine Operation zur Unterstützung eines Geschäftsprozesses ist.

Zur **Kommunikation von außen** bietet der Microservice eine im REST-Architekturstil[45] umgesetzte Schnittstelle an. Dieser kann daher als RESTful bezeichnet werden und unterstützt die gleichzeitige Nutzung ohne expliziter Koordination der Anfragen.[46]

Die Schnittstelle bietet Ressourcen, die mit einem einheitlichen URI identifiziert werden, an.[47] Eine Ressource kann dabei zum Beispiel ein bestimmter Datensatz oder der Aufruf der angebotenen Operation sein. Die Übertragung erfolgt in der Referenzimplementierung über das Hypertext Transfer Protocol (HTTP).

Damit eine Navigation innerhalb der Schnittstelle über die Ressourcen möglich ist, wird Hypermedia As The Engine Of Application State (HATEOAS)[48] verwendet. HATEOAS ist ein Entwurfsmuster und ermöglicht über HTTP-Links in den Antworten eine dynamische Navigation durch das REST-Interface.[49] Ist zum Beispiel zu einer Person eine Anschrift gespeichert, übergibt HATEOAS automatisch die Links, um anschließend wieder direkt auf die Person zurück navigieren zu können.[50] Die Data Transfer Objects (DTOs) bestehen dementsprechend aus den fachlichen Datenstrukturen mit zusätzlichen HATEOAS-Links.

REST nutzt die HTTP 1.1 Methoden (GET, POST, PUT, PATCH, DELETE) und ordnet diese, wenn vorhanden, den CRUD-Operationen zu. Die Operationen werden für eine Entity oder ein Value Object durch Spring Data REST[51] bereitgestellt. Spring Data REST

[45] Ursprünglich in der Dissertation von Fielding erarbeitet, siehe [Fie00], Hauptkapitel 5: "Representational State Transfer (REST)", Seite 76ff
[46] Im Detail wird die parallele Nutzung durch den Webserver ermöglicht, der mehrere Anfragen simultan entgegen nehmen und verarbeiten kann.
[47] vgl. [Fie00], Unterkapitel: "5.1.5 Uniform Interface", Seite 82ff
[48] https://spring.io/understanding/HATEOAS
[49] vgl. [Sofd], Abschnitt: "Understanding HATEOAS"
[50] Im Anhang A.2 befindet sich ein Anwendungsbeispiel von HATEOAS.
[51] http://projects.spring.io/spring-data-rest/

analysiert die definierten Repositories (vgl. DDD in Unterkapitel 2.2.2) und stellt eine entsprechende RESTful-Schnittstelle dafür bereit.[52,53] Diese grundlegenden Operationen benötigen dadurch keine eigenständige bzw. manuell zu implementierende Controller- und Service-Klassen (vgl. Abbildung 2.10, Pfeil mit der Beschriftung "CRUD-Operation").

Zur **Absicherung der Schnittstelle** des Microservice ist die Nutzung nur nach einer Authentifizierung möglich. In der Referenzimplementierung ist das in Abbildung 2.11 dargestellte Rollen- und Rechtekonzept umgesetzt. Der Aufruf einer Operation der Schnittstelle entspricht einem Methodenaufruf. Jeder Methode ist einer, dafür benötigten, Berechtigung zugeordnet.[54] Eine oder mehrere Berechtigungen sind in Rollen zusammengefasst. Diese Rollen werden den entsprechenden Benutzern zugeordnet. Die dadurch ebenso benötigte Benutzerverwaltung wird durch eine verwendete Bibliothek bereitgestellt.

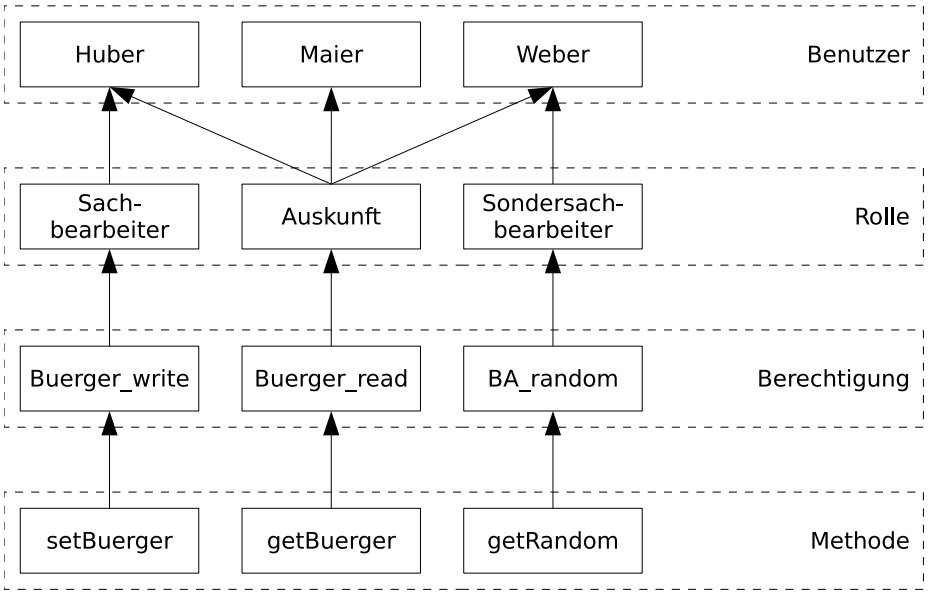


Abbildung 2.11: Rollen und Rechtekonzept der Referenzimplementierung anhand einem Beispiel. In diesem kann der Benutzer *Maier* in seiner Rolle als *Auskunft* lediglich Bürger lesen.

Eine Übersicht über die benötigten Klassen und deren internen Zusammenhänge zum Anbieten eines Bürgers mit entsprechenden Geschäftsaktionen wird durch die nachfolgende Abbildung dargestellt.

[52] vgl. [Sofc], Abschnitt "Features"
[53] Spring Data REST erkennt die bereitzustellenden Repositorys an der anzugebenden *@RepositoryRestResource*-Annotationen.
[54] Implementiert ist dies anhand einer entsprechenden Annotationen an der Methode.

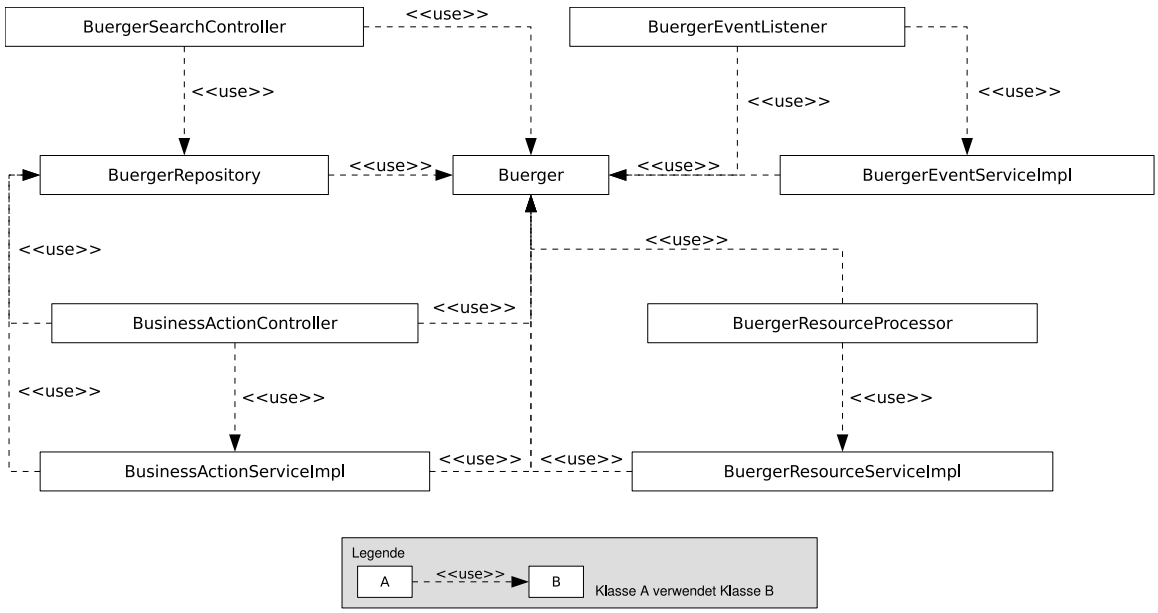


Abbildung 2.12: Klassendiagramm des Microservice der Referenzimplementierung am Beispiel der Buerger-Entity. Für jede weitere Entity oder jedem Value Object sind diese Klassen, abgesehen der BusinessAction[...]-Klassen, ebenso vorhanden. Zur übersichtlicheren Darstellung wurde auf die Abbildung von Interfaces, genutzer Bibliotheken und Package-Struktur verzichtet.

In Abbildung 2.12 ist das zentrale Element die *Buerger*-Klasse. Neben den fachlichen Variablen enthält diese Ergänzende, damit eine direkte Speicherung und Verwaltung in der Datenbank möglich ist. Für die Handhabung in der Datenbank erfolgt eine Kennzeichnung aller Variablen über entsprechende Annotationen[55] aus den Packages *javax.persistence* und *org.hibernate*.

Das *BuergerRepository* ermöglicht durch Spring Data REST die (in Abbildung 2.10 dargestellte) direkte Kommunikation von außen bei Verwendung von Create, Read, Update, Delete (CRUD)-Operationen. Die zusätzliche Klasse *BuergerSearchController* erweitert das Repository und ermöglicht, neben der bereits enthaltenen Suche, eine sogenannte *fuzzy search*[56]. Eine Anfrage mit dem Vornamen ”Lena” liefert dadurch als Suchergebnis ebenfalls die Vornamen ”Lene” und ”Luna”.

Zur Bereitstellung der Geschäftsaktionen (vgl. Abbildung 2.10), dienen die Klassen *BusinessActionController* und *BusinessActionServiceImpl*.

Der Zugriff von außen (mit dem Client) im Kontext des dargestellten Klassendiagramms erfolgt über das *BuergerRepository* und dem *BusinessActionController*. Die eigentlich auf-

[55] Ein Sprachelement von Java, die die Einbindung von Meta-Daten erlaubt (vgl. [Cor14], Abschnitt: ”Description”).
[56] deutsch: unscharfe Suche

gerufene REST-Schnittstelle wird durch Spring bereitgestellt und eine entsprechende Weiterleitung der Anfragen findet statt.

2.2.4 Aufbau der Client-Bibliothek

Das Projekt der *Client-Bibliothek* der Referenzimplementierung beinhaltet das Gegenstück zum Microservice. In beiden ist die anfängliche[57] fachliche Datenstrukturdefinition enthalten. Zusätzlich beinhaltet die Client-Bibliothek einen vorkonfigurierten Client zum Zugriff auf die bereitgestellte Schnittstelle und die DTOs zur Kommunikation.

Für ein Datenelement, nachfolgend am Beispiel der Repräsentation eines Bürgers, existieren die in Abbildung 2.13 enthaltenen Klassen.

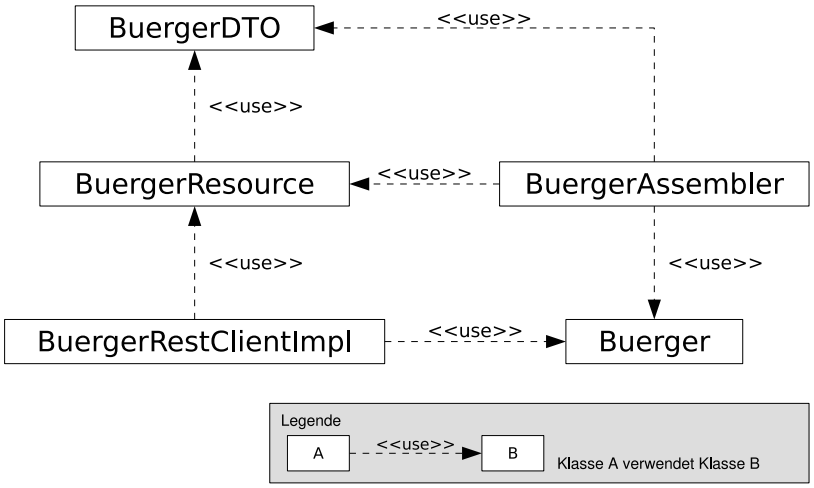


Abbildung 2.13: Klassendiagramm Client-Bibliothek der Referenzimplementierung. Zur übersichtlicheren Darstellung wurde auf die Abbildung von Interfaces, genutzer Bibliotheken und Package-Struktur verzichtet.

Die *BuergerDTO* dient zur Übertragung über die REST-Schnittstelle und beinhaltet die fachlichen und typisierten Felder. Die Klasse *Buerger* (vgl. Listing 2.1) beinhaltet ebenso diese Daten und dient als Bean[58]. Der *BuergerAssembler* stellt Umwandlungsmethoden, sogenannte Mappings, zwischen dem DTO und der Bean bereit.

Zum Zugriff auf einen Bürger über die REST-Schnittstelle wird der *BuergerRestClient* bereitgestellt. Mit diesem können mit geringem Aufwand Anfragen an den Microservice mit Hilfe der *BuergerResource* (vgl. Unterkapitel 3.2.3, Seite 48) gestellt werden.

[57] Während des Lebenszyklus kann es aufgrund der Wartung des Microservices vorkommen, dass dessen angebotene Schnittstelle erweitert wird. Die Kompatibilität bleibt dabei erhalten, wodurch eine gleichzeitige Anpassung des Clients nicht erforderlich ist.

[58] Eine datenhaltende Klasse in Java, die mehrere Konventionen einhält und dadurch eine Persistenz erlaubt.

Die *BuergerResource* kapselt die *BuergerDTO* und enthält zusätzlich die Navigationslinks durch HATEOAS.

Nachfolgend wird im Detail die *Buerger*-Klasse betrachtet, da diese das zentrale Element des Client-Projektes ist.

Die Verwendung von HATEOAS zur Bereitstellung der Navigationslinks erfolgt durch das Ableiten der bereitgestellten ResourceSupport-Klasse (vgl. Listing 2.1: Zeile 3 und 10).

Die fachlich benötigten Variablen (vgl. Listing 2.1: Zeile 15, 20 und 24) besitzen im Vergleich zu den DTOs Validierungen der Daten. Diese werden per Annotationen angegeben (die jeweiligen Zeilen davor) und verhindern das Setzen ungültiger Werte.[59] Zum Beispiel muss das Geburtsdatum (vgl. Listing 2.1: Zeile 22-24) angegeben werden und in der Vergangenheit liegen.

```
1  // [...]
2  import org.hibernate.validator.constraints.NotEmpty;
3  import org.springframework.hateoas.ResourceSupport;
4  import javax.validation.constraints.NotNull;
5  import javax.validation.constraints.Past;
6  import javax.validation.constraints.Pattern;
7  import javax.validation.constraints.Size;
8  import java.util.Date;
9
10 public class Buerger extends ResourceSupport {
11
12   @NotEmpty
13   @Size
14   @Pattern(regexp = "[A-Za-z\\-]+")
15   private String vorname = "";
16
17   @NotEmpty
18   @Size
19   @Pattern(regexp = "[A-Za-z\\-]+")
20   private String nachname = "";
21
22   @NotNull
23   @Past
24   private Date geburtsdatum;
25   // [...]
26 }
```

Listing 2.1: Auszug aus der Klasse zur Representation eines Bürgers aus der Rerefenzimplementierung.

[59] Im Detail wird zur Validierung der "*JSR 303: Bean Validation*" genutzt. Weitere Informationen darüber stehen unter http://beanvalidation.org/1.0/spec/ zur Verfügung.

2.2.5 Aufbau eines GUI Microservice

Die GUI der Referenzimplementierung ist selbst ein Microservice und nutzt zusätzlich das frei verfügbare Vaadin-Framework[60] zur Bereitstellung der Weboberfläche.[61] Vaadin ermöglicht mittels Java die Erstellung von modernen[62] server-seitigen Webanwendungen. Ein Vorteil davon ist, dass sich der Entwickler nicht um die Kommunikation zwischen Browser und Server kümmern muss.[63]

Allgemein orientieren sich Vaadin-Anwendung am Model View Presenter (MVP) Pattern (einer Weiterentwicklung von Model View Controller (MVC)).[64] In Abbildung 2.14 ist die Vaadin-Architektur im Kontext der Referenzarchitektur dargestellt. Der Hauptunterschied besteht im, durch das Pattern vorgegebene, *Model*, welches der Microservice ist und über die verwendete Client-Bibliothek bezogen wird.

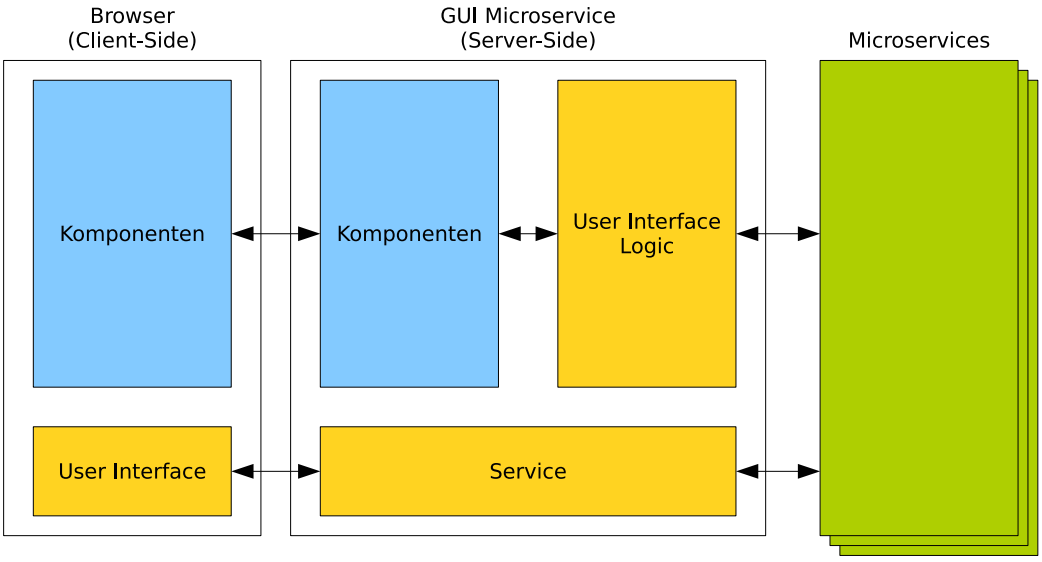


Abbildung 2.14: Auf die Referenzarchitektur angepasste Version der Vaadin-Architektur nach [Ltda], Abbildung: 1.1. Vaadin Application Architecture. In gelber Farbe visualisiert sind Logik beinhaltende Elemente. Die Komponenten als Oberflächenelemente sind in blauer Farbe hinterlegt. Die Pfeile veranschaulichen die Kommunikationswege.

In der Referenzimplementierung wurden Komponenten erstellt, die sich auf Views, den einzelnen Seiten, anzeigen lassen. Eine Komponente ist ein Oberflächenelement und kann

[60] https://vaadin.com/
[61] Lizenziert ist Vaadin mit der *Apache License*. Weitere Informationen unter http://www.apache.org/licenses/LICENSE-2.0.html.
[62] Vaadin erzeugt HTML5 (siehe http://www.w3.org/TR/html5/) konforme Anwendungen.
[63] vgl. [Ltdc], Abschnitt: "Vaadin Framework lets you [...]"
[64] Weitere Informationen unter https://vaadin.com/book/vaadin7/-/page/advanced.architecture.html.

ihrerseits wiederum mehrere weitere Komponenten beinhalten (vgl. Abbildung 2.15). Neben den selbst Definierten stehen die Basiselemente, wie Schaltflächen, Textfelder oder Auswahlmenüs zur Verfügung.[65]

Die Einbindung von Views und Komponenten erfolgt über die sogenannte *Annotation-Driven Configuration*. Das Spring-Framework sucht dazu beim Start der Anwendung Klassen mit entsprechenden Annotationen und führt ein sogenanntes *Autowiring* durch. Dabei werden die erkannten Abhängigkeiten automatisch untereinander aufgelöst und benötigte Instanzen des jeweiligen Typs selbstständig zugewiesen. Benötigt zum Beispiel eine View die Instanz des vorhandenen Controller's, ist die Bereitstellung einer entsprechend typisierten Variable, die die Annotation *@Autowired* besitzt, ausreichend.[66]

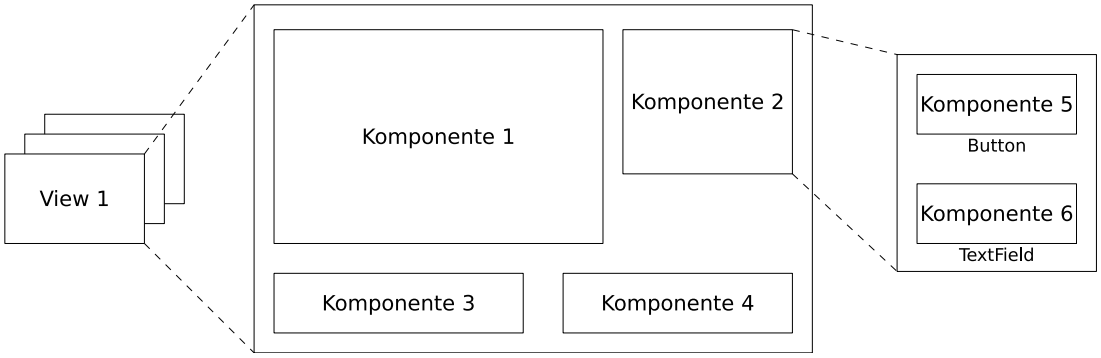


Abbildung 2.15: Aufbau einer GUI mit Hilfe von Vaadin. Diese besteht aus mehreren Views, die jeweils Komponenten beinhalten.

Kommuniziert wird mittels allgemein definierter Events, die über die Komponenten an einen zentralen ViewController innerhalb der GUI geleitet werden. Dieser arbeitet als Dispatcher, welcher die Anfragen an die dafür vorhandenen Service-Klassen weiterleitet.[67] Darin verwendet wird der REST-Client aus der jeweiligen Client-Bibliothek, welcher die Anfragen zur Verarbeitung an den Microservice weiterleitet. Die nachfolgende Abbildung 2.16 stellt diesen Kommunikationsweg grafisch dar.

[65] Eine vollständige Übersicht der, im Vaadin-Framework vorhanden, Komponenten ist im Anhang A.3 enthalten.
[66] Weitere Informationen zur Umsetzung siehe [Sofa].
[67] Es ist folglich nicht erforderlich eigene Events zu definieren.

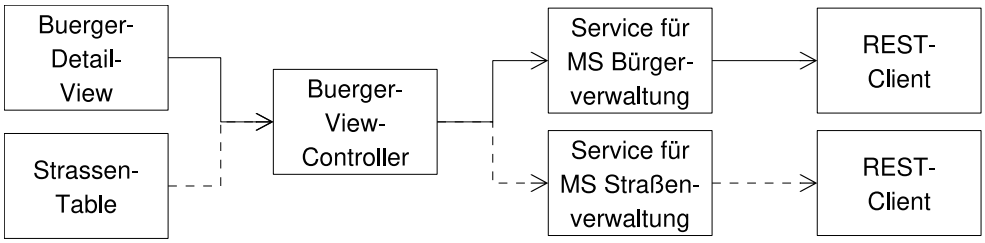


Abbildung 2.16: Beispielhafte Kommunikationswege in der GUI an den MS Bürgerverwaltung oder in gestrichelt dargestellten Pfeilen an den MS Straßenverwaltung. Die Pfeile stellen die Operationen dar, die gleichzeitig die Events übertragen.

Die in Abbildung 2.14 grün dargestellten Microservices sind die genutzten Schnittstellen und befinden sich nicht innerhalb der Vaadin Anwendung. Die GUI auf der Server-Seite leitet Anfragen, wie zum Beispiel eine Authentifizierungsanfrage eines Benutzers, an einen Microservice weiter. Dieser kann jene überprüfen und gleichzeitig die Autorisierung erteilen. Aktionen, die die Darstellung oder die Navigation betreffen, werden in der GUI selbst verarbeitet, wodurch keine Kommunikation mit einem Microservice stattfindet.

Im Detail wurden zur Verwaltung eines Bürgers in der GUI folgende Klassen entwickelt:

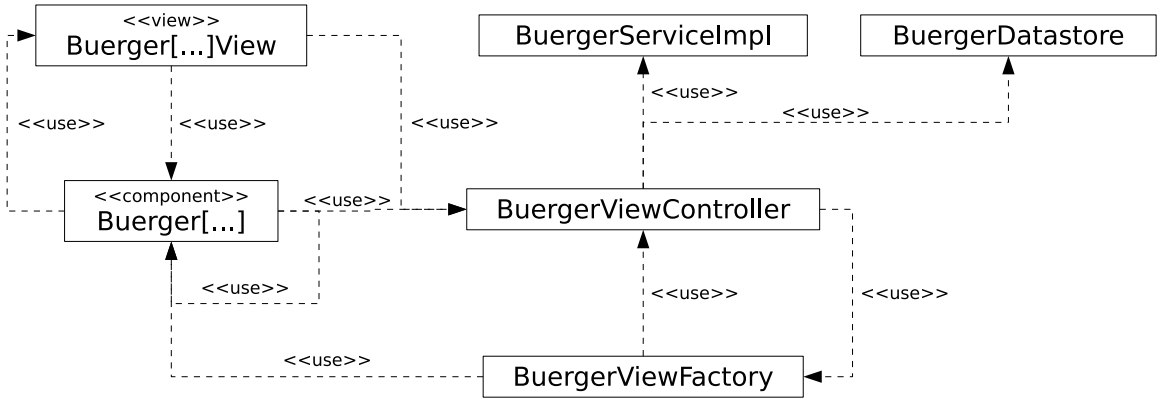


Abbildung 2.17: Klassendiagramm des GUI MS der Referenzimplementierung. Die Stereotypen *view* und *component* werden verwendet, um zu verdeutlichen, dass es sich dabei um mehrere, dem Zweck entsprechend implementierte, Klassen handelt. Zum Beispiel BuergerTableView oder BuergerReadForm. Zur übersichtlicheren Darstellung wurde auf die Abbildung von Interfaces, genutzer Bibliotheken und Package-Struktur verzichtet.

Um die Schnittstelle vom Microservice komplett nutzen zu können sind mehrere *Buerger[...]View*-Klassen und *Buerger[...]*-Komponenten (vgl. Abbildung 2.15) entwickelt worden. Diese ermöglichen unter anderem das Auflisten, Anlegen, Bearbeiten, Löschen und Zuordnen von Bürgern. Die Tabelle zum Auflisten beinhaltet aufgrund der Übersichtlichkeit nicht alle Felder der Buerger-Klasse (siehe Unterkapitel 2.2.4), sondern eine fachlich

benötigte Auswahl davon (vgl. Abbildung 2.18).

Für jede Sitzung[68] werden über die *BuergerViewFactory* die Komponenten und Views instantiiert.

Die auftretenden Events auf der Oberfläche, wie beispielsweise ein Seitenwechsel oder das Anzeigen der Hilfe (vgl. Abbildung 2.16, ein Ereignis tritt zum Beispiel in der BuergerDetailView auf), verarbeitet der *BuergerViewController* und greift entweder auf den internen *BuergerDatastore* für bereits abgefragte Daten zurück oder schickt eine Anfrage an den Microservice über die *BuergerServiceImpl*. Zum Beispiel wird in einer Tabelle mit vielen Datensätzen (Zeilen) nur der gerade sichtbare Teil an den Browser übergeben und nicht alle. Auf Server-Seite sind dennoch alle Daten bereits vorhanden und werden auf Anfrage übergeben. Der Datastore ermöglicht dies und dient als Zwischenspeicher[69], um die Abfragen an den Microservice zu reduzieren.

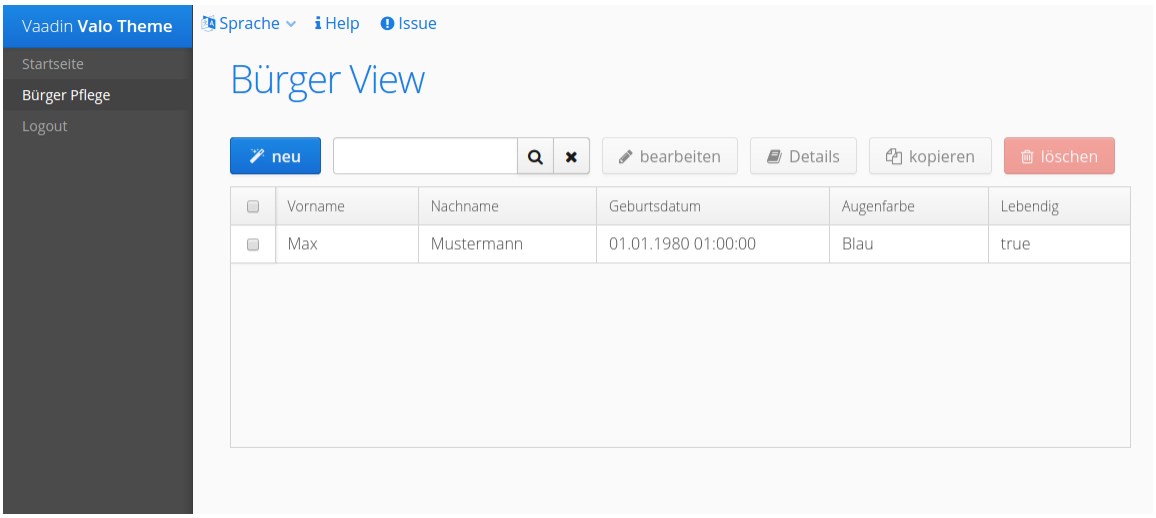


Abbildung 2.18: Screenshot einer View des GUI MS der Referenzimplementierung zur Auflistung von Bürgern. Die Tabelle beinhaltet in diesem Beispiel aufgrund der fachlichen Anforderung unter anderem die Spalte *Augenfarbe*.

Mit der GUI ist die Betrachtung der Referenzimplementierung abgeschlossen. Auf die Referenzimplementierung wird im Kapitel 3.2 zurückgegriffen.

[68] Eine Sitzung kommt nach einer erfolgreichen Anmeldung zustande. Die Instanz der Anmeldeseite wird durch das Vaadin-Framework erstellt.
[69] Eine Art Cache.

2.3 Begriffsdefinitionen und Abgrenzung

Die wesentlichen Begriffe dieser Ausarbeitung werden nachfolgend definiert, um für ein einheitliches Verständnis zu sorgen und eine Diskussionsgrundlage zu schaffen.

2.3.1 Model Driven Software Development

Ein **Modell** bezeichnet allgemein eine abstrahierte Abbildung der Wirklichkeit. Nach Stachowiak ist es gekennzeichnet durch die folgenden drei Hauptmerkmale:[70]

- Abbildung: "*Modelle sind stets Modelle von etwas, nämlich Abbildungen, Repräsentationen natürlicher oder künstlicher Originale, die selbst wieder Modelle sein können.*"
- Verkürzung: "*Modelle erfassen im allgemeinen nicht alle Attribute des durch sie repräsentierten Originals, sondern nur solche, die den jeweiligen Modellerschaffern und/oder Modellbenutzern relevant scheinen.*"
- Pragmatismus: "*Eine pragmatisch vollständige Bestimmung des Modellbegriffs hat nicht nur die Frage zu berücksichtigen, wovon etwas Modell ist, sondern auch, für wen, wann und wozu bezüglich seiner je spezifischen Funktionen es Modell ist.*"

Stahl et al. definieren Model Driven Software Development (MDSD) in ihrem gleichnamigen Buch wie folgt:

> "*Modellgetriebene Softwareentwicklung (MDSD) ist ein Oberbegriff für Techniken, die aus formalen Modellen automatisiert lauffähige Software erzeugen.*"[71]

Diese Definition besteht aus drei Teilen:[72]

- **Formal** bedeutet, dass durch das Modell zumindest ein Aspekt der Software vollständig beschrieben werden kann. Die Vollständigkeit ist anhand klarer Regeln abzugrenzen. Es muss definiert werden, worüber das Modell Aussagen macht und worüber nicht, da damit nicht alles beschreibbar ist.
- Die **Automatisierung** besagt, dass das Modell direkt verarbeitet (wie Quelltext) und nicht ausschließlich zur Spezifikation bzw. Dokumentation verwendet wird.
- Der Übergang vom Modell zu **lauffähiger Software**, ohne dass eine manuelle Implementierung (anhand dessen) bzw. manuelle Umwandlung erfolgen muss.

Unter Berücksichtigung der Definitionen von Stachowiak und Stahl et al. ergibt sich folgender grafischer Zusammenhang:

[70] [Sta73], Überschrift: "Die drei Hauptmerkmale des allgemeinen Modellbegriffs", Seite 131ff
[71] [SVE+07], Überschrift: "Einführung in MDSD", Seite 11
[72] vgl. [SVE+07], Überschrift: "Einführung in MDSD", Seite 11ff

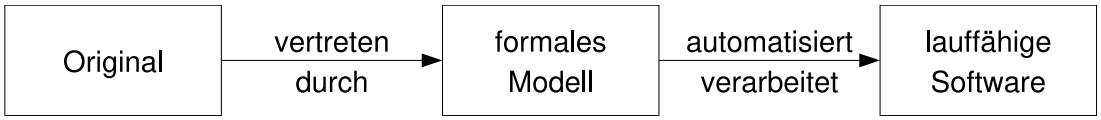


Abbildung 2.19: Darstellung der Definition des Modellbegriffs nach Stachowiak und von MDSD nach Stahl et al. Die Pfeile veranschaulichen dabei die Ablaufrichtung und den Zusammenhang.

In der Literatur findet sich in Zusammenhang mit MDSD der Begriff Model Driven Development (MDD). Dabei handelt es sich um ein Markenzeichen der Object Management Group (OMG), die in der modellgetriebenen Entwicklung den Model Driven Architecture (MDA) Standard definiert und rechtlich geschützt hat. MDA besitzt unter anderem die Einschränkung, dass ausschließlich Unified Modeling Language (UML) basierte Modellierungssprachen verwendet werden dürfen.[73]

In dieser Ausarbeitung wird primär auf MDSD eingegangen, da der Fokus auf der Erzeugung von lauffähiger Software liegt und dabei die Modellierungssprache nicht zwingend UML ist. Im nachfolgendem Kapitel 2.4 wird dennoch auf Literatur aus dem Bereich der MDD eingegangen, da teilweise die Ansätze und Konzepte auch im MDSD verwendet werden können.

2.3.2 Domänenspezifische Sprache

Im Kontext der Informatik bedeutet domänenspezifisch für einen Bereich. Entsprechend handelt es sich bei einer DSL um eine (Programmier-)Sprache, die für ein bestimmtes Anwendungsgebiet eingesetzt werden kann.[74] Beispiele hierfür sind die Structured Query Language (SQL) in Zusammenhang mit Datenbanken und im Behaviour-Driven Development (BDD) die Syntax mit *given*, *when* und *then*[75].

Eine DSL besteht aus einem Metamodell, welches die Domäne formal beschreibt, und einer konkreten Syntax (vgl. Abbildung 2.20). Das Metamodell umfasst eine abstrakte Syntax und eine statische Semantik.[76] Die abstrakte Syntax legt die Sprachelemente und deren Beziehung fest. Formale Bedingungen (Constraints), wie zum Beispiel die Bedingung eines eindeutigen Namens, werden von der statischen Semantik vorgegeben. Die konkrete Syntax ist die Darstellung bzw. die lesbare Repräsentation der Sprache.

[73] vgl. [Gro15], Abschnitt: "MDA Specification Support"

[74] Die Definition einer Domäne ist im DDD spezifischer: ein abgeschlossener Bereich mit ähnlichen fachlichen Anforderungen. Beispielsweise ist im DDD die Bürgerverwaltung und die Straßenverwaltung jeweils eine eigene Domäne. Eine DSL hingegen kann sich für beide Fachbereiche und weitere eignen.

[75] Implementierung davon sind beispielsweise `http://jgiven.org/` und `https://cucumber.io/` zu entnehmen.

[76] vgl. [Voe13], Überschrift "Introduction to DSLs", Seite 26 und [SVE+07], Kapitel "3.1 Definitionen", Seite 27ff

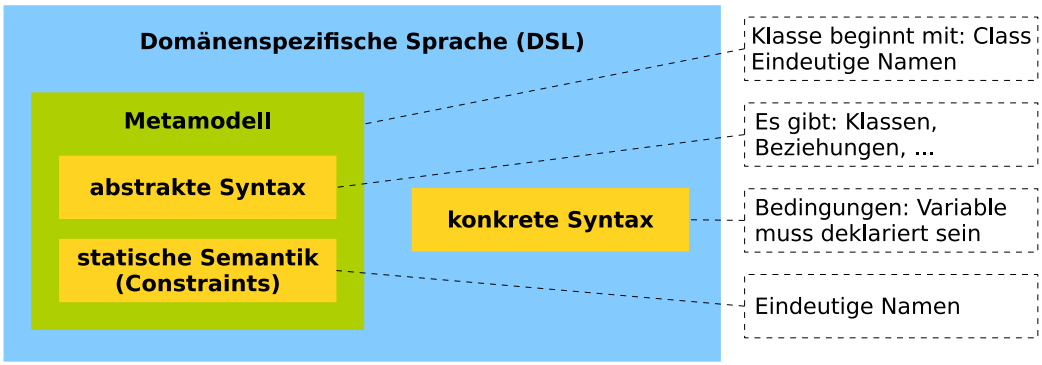


Abbildung 2.20: Darstellung der Bestandteile einer DSL farbig hervorgehoben auf der linken Seite mit jeweiliger Erläuterung bzw. einem Beispiel auf der rechten Seite.

2.4 Einschlägige Literatur

Nachfolgend wird auf den Stand der Technik anhand von Literatur zum Themenbereich MDSD, MDD und Entwicklung einer DSL eingegangen.

Stahl et al. schildern im Buch mit dem Titel "*Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*" [SVE+07] ausführlich die modellgetriebene Entwicklung mit praxisnahen Beispielen. Dabei beschreiben die Autoren anfangs die Grundlagen und befassen sich im Anschluss mit Modellierungssprachen. Über die dazugehörigen Editoren und Generatoren erläutern Stahl et al. zudem die Konstruktion von domänenspezifischen Architekturen. Das Buch enthält viele *Best Practices* für die Anwendung der beschriebenen Theorie und stellt Werkzeuge zur Unterstützung vor.

In der Lektüre von **Trompeter et al.** mit dem Titel "*Modellgetriebene Softwareentwicklung: MDA und MDSD in der Praxis*" [TB07] befassen sich die Autoren, abgesehen von den Grundlagen, mit der Kompatibilität von MDSD und den Vorgehensmodellen. Anschließend wird auf die Modellierung eingegangen und zwei DSLs in der Praxis beschrieben. Im eigenen Kapitel "Best Practices" beschreiben die Autoren ihre Empfehlungen anhand der unterschiedlichen Themenbereiche der MDSD. Im Anschluss wird die beschriebene Theorie anhand eines Anwendungsbeispiels angewendet.

In "*Model Driven Architecture*" von **Petrasch und Meimberg** [PM06] erläutern die Autoren verschiedene Herangehensweisen und Notationen der modellgetriebenen Architektur nach der OMG und verweisen in diesem Umfeld teilweise zusätzlich noch auf [SVE+07]. Sie gehen dabei auf Probleme ein und geben Lösungsansätze, die oftmals mit Beispielen veranschaulicht werden. Anhand von einem konkreten Projekt wird das vermittelte Wissen in der Praxis angewendet und technisch sehr detailliert beschrieben.

Im Paper "*Model-driven development: The good, the bad, and the ugly*" von **Hailpern und Tarr** [HT06] erfolgt nach der Beschreibung der Grundlagen die Nennung der Vor- und Nachteile von MDD. Als Vorteile schildern die Autoren, dass das Abstraktionslevel steigt, die Software weniger komplex ist und resultierend daraus der Entwicklungsaufwand geringer wird. Sind die Modelle zu weiteren Systemen verbunden, die nicht als Modell vorliegen, wirkt sich das jedoch negativ aus, da Redundanzen erzeugt werden (die Schnittstellen sind doppelt vorhanden). Umso mehr Beziehungen dabei bestehen, desto komplexer wird die Software. Die Autoren sind daher der Meinung, dass lediglich eine Verschiebung anstelle einer Reduzierung der Komplexität stattfindet. Als weiteren Nachteil führen Hailpern und Tarr die Expertise auf, die bei der Benutzung der Modelle benötigt wird. Ebenfalls sind Sie der Meinung, dass eine höhere Abstraktion zu einer Übervereinfachung führen kann. Demnach besitzt das Modell zu wenig Details, um einen sinnvollen Zweck zu erfüllen, so die Autoren.

France und Rumpe befassen sich in "*Model-driven development of complex software: A research roadmap*" [FR07] mit Model Driven Engineering (MDE) und wie mit diesem Ansatz die Softwareentwickelung möglich ist. Sie verweisen dabei auf die Schwierigkeit, dass Laufzeitprobleme versteckt bzw. Gegebenheiten, wie Server, Systeme oder die Persistenz nicht mit einbezogen werden. Sie betrachten daher zwei Klassen von Modellen: die Entwicklungsmodelle (für Anforderungen, Architektur, ...) und die Laufzeitmodelle. Die Autoren sind der Meinung, dass durch die Abstraktion bestimmte Probleme nicht bzw. nur mit einer höheren Komplexität gelöst werden können. Als Lösungsansatz empfehlen Sie unter anderem die progressive Entwicklung von Domain Specific Application Development Environments (DSAEs). Diese Softwareentwicklungswerkzeuge sollen bei der Entwicklung helfen, indem sie das sogenannte Round-Trip-Engineering[77] unterstützen. Ebenso sollen diese mit Mechanismen ausgestattet sein, um Modelle mit einem unterschiedlichen Abstraktionsgrad zu synchronisieren und die generierte Software in Bestandssysteme zu integrieren.

Mernik et al. beschreiben in ihrem Artikel "*When and How to Develop Domain-specific Languages*" [MHS05] die Vorteile von DSLs, wie zum Beispiel die Ermöglichung der Wiederverwendbarkeit, und unterteilen deren Entwicklung in die Phasen Entscheidung, Analyse, Design, Implementation und Einführung. Für jede diese Phasen werden anschließend unterstützende Patterns aufgelistet und näher beschrieben (vgl. Tabelle 1). Die angegebenen Pattern zur Unterstützung stammen wiederum aus weiterführender Literatur, auf die Mernik et al. verweisen.

[77] Dabei erfolgt ein automatischer Abgleich zwischen dem Entwurf (z.B. Modell) und der Umsetzung (z.B. Quelltext) oder umgekehrt (siehe Forward Engineering und Reverse Engineering).

Development Phase	Pattern
Decision	Notation
(Section 2.2)	Task automation
	Data structure representation
	GUI construction
Analysis	Informal
(Section 2.3)	Formal
	Extract from code
Design	Language exploitation
(Section 2.4)	Language invention
	Informal
	Formal
Implementation	Interpreter
(Section 2.5)	Compiler/application generator
	Extensible compiler/interpreter
	Hybrid

Tabelle 1: Auszug der Zusammenfassung der Patterns zu jeder Entwicklungsphase einer DSL nach [MHS05], Tabelle 20: "Summary of DSL Development Phases and Corresponding Patterns".

In "*Development of Internal Domain-specific Languages: Design Principles and Design Patterns*", der Zusammenfassung seiner eigenen Doktorarbeit, stellt **Günther** [Gün11] die Entwurfsprinzipien von DSLs vor: Abstraktion, Generalisierung, Optimierung, Notation, Eingliederung, Komprimierung. Anhand von Quelltextauszügen werden dazu die Interpretersprachen Ruby und Python sowie der Compilersprache Scala näher betrachtet.[78]

Strembeck und Zdun erläutern im Artikel "*An Approach for the Systematic Development of Domain-Specific Languages*" [SZ09] die notwendigen Schritte, um systematisch eine DSL zu erzeugen. Diese sind:

1. Definition der abstrakten Syntax
2. Definition der statistischen Semantik
3. Definition der konkreten Syntax
4. Integration der DSL Ergebnisse auf die Zielplattform

[78] Interpretersprachen verarbeiten den Quelltext zur Laufzeit zeilenweise. Im Gegensatz dazu wird bei Compilersprachen der komplette Quelltext übersetzt, bevor dieser ausgeführt werden kann.

Die Autoren beziehen sich dabei unter anderem auf [SVE+07]. Weiterhin werden die einzelnen Schritte im Detail mit ihren Aktivitäten betrachtet und mit Hilfe von Aktivitätsdiagrammen veranschaulicht. Sie geben zudem an, dass es sich (in den meisten Fällen) um einen sich entwickelnden und iterativen Prozess handelt. Eine Anpassung dessen ist durch sogenanntes Tailoring[79] möglich. Zusätzlich wenden Strembeck und Zdun die beschriebene Theorie im jeweiligen Kapitel anhand von einem Beispiel an. Dieses wird Schrittweise weiterentwickelt, wodurch am Ende des Artikels eine vollständige DSL vorliegt.

Im Buch "*DSL Engineering: Designing, Implementing and Using Domain-specific Languages*" von **Voelter** [Voe13] beschreibt der Autor das Vorgehen für den Entwurf, die Implementierung und die Verwendung von DSLs. Zu Beginn wird dazu der Aufbau von Sprachen beschrieben und anhand von Beispielen erläutert. Anschließend geht Voelter auf mehrere typische Fragen zur Entwicklung einer eignen DSL ein. Beispielsweise verweist der Autor darauf, dass auch der Entwicklungsprozess diese entsprechend unterstützen muss. Die Entwicklung einer DSL erfolgt in mehreren Iterationen, wodurch Ansätze auf Basis des Wasserfallmodells scheitern. Im Kapitel der Implementierung wird auf mehrere vorhandene Frameworks, wie zum Beispiel ANTLR[80] und Xtext[81], eingegangen und korrespondierende Beispiele vorgestellt. Danach schildert Voelter anhand verschiedener Sichtweisen den Zusammenhang von DSLs mit dem Software Engineering.

Das Buch von **Fowler** mit dem Titel "*Domain Specific Languages*" [Fow10] ist aufgeteilt in den Grundlagen- und den Referenzteil. In beiden Teilen erfolgt zu jedem Themenbereich ein Beschreibung der Theorie, ein implementiertes Praxisbeispiel (oftmals in der Programmiersprache Java) und die Empfehlung des Autors. In den Grundlagen befasst sich Fowler mit Grundsatzfragen, wie zum Beispiel, ob eine *internal* oder *external* DSL entwickelt werden soll. Als *internal* bezeichnet der Autor Sprachen, die auf anderen aufbauen. Folglich besitzen *external* DSLs eine eigene Syntax und benötigen selbst alle Elemente aus dem klassischen Compilerbau.
Der Referenzteil beinhaltet viele Herausforderungen und deren Lösungsmöglichkeiten während der Entwicklung einer DSL. Beispielsweise den Aufbau von Entscheidungstabellen oder die Manipulation des geparsten Syntaxbaums. Ebenso setzt sich Fowler mit der Einbindung von "fremden" Quelltext, der nicht Teil der DSL ist, oder anonymen Methoden auseinander.

[79] Damit wird der "Zuschnitt", entsprechend das Weglassen von Aktivitäten, an den eigentlichen Bedarf ausgedrückt.
[80] http://www.antlr.org/
[81] https://eclipse.org/Xtext/

2.5 Kritische Betrachtung

Das Prozessmodell und die vorgestellte Literatur wird im Folgenden kritisch hinterfragt um Vor- bzw. Nachteile aufzuzeigen. Unbeantwortete Fragen, sowie die Praxistauglichkeit des Beschriebenen stehen dabei im Vordergrund. Erfolgt keine weitere Angabe, ist die Meinung des Autors dieser Ausarbeitung wiedergegeben.

2.5.1 Prozessmodell IT-Service

Das *Prozessmodell IT-Service* gibt die Reihenfolge anhand von Phasen mit Meilensteinen vor. Dadurch ist es an mehreren Stellen nicht möglich, Dokumente aus vorherigen Phasen zu ändern, da diese bereits durch einen Meilenstein mit einer entsprechenden Abnahme finalisiert wurden. Speziell während der Phase *Realisierung und Test* zur *Anforderungsbearbeitung* (vgl. Abbildung 2.21 bzw. Anhang A.1) ist es nicht angedacht, dass das Fachkonzept (FK) nochmals bearbeitet wird, da dieses bereits durch den Endkunden (vgl. Meilenstein 6 in Abbildung 2.21) abgenommen wurde. Oftmals ist es jedoch der Fall, dass in der Aktivität *IT Lösung umsetzen* nochmals das FK bzw. die Systemspezifikation (SysSpec) angepasst werden müssen. Wenn beispielsweise die Umsetzung von Anforderungen nicht möglich ist, neue hinzukommen oder diese nicht testbar sind. Problematisch ist dabei, dass die beiden vorherigen Prozessphasen zu diesem Zeitpunkt bereits komplett abgeschlossen wurden, sodass die Durchführung von Änderungen nur mit einem hohen Mehraufwand möglich ist.

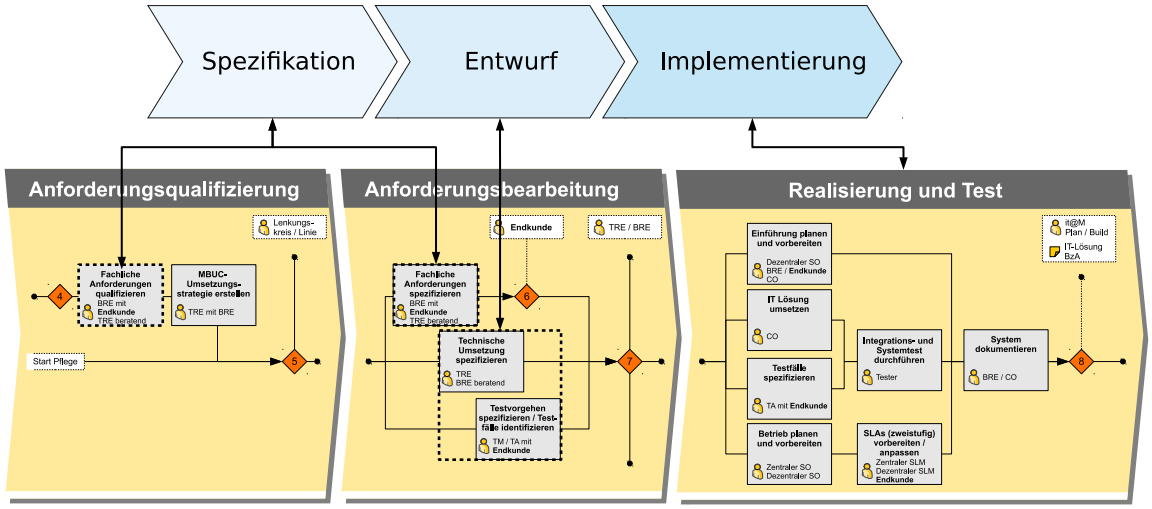


Abbildung 2.21: Darstellung der Zuordnung des Lebenszyklus zu den Phasen *Anforderungsqualifizierung*, *Anforderungsbearbeitung* und *Realisierung und Test* aus dem *Prozessmodell IT-Service* der LHM. Die Spezifikation beginnt in der *Anforderungsqualifizierung* und wird in der *Anforderungsbearbeitung* abgeschlossen. Zur Anfertigung des FKs und der SysSpec ist die Erstellung von einem Entwurf gebräuchlich. Die Phase *Beschaffung* wurde in diesem Ausschnitt entfernt, da diese keinen Einfluss bei einer Eigenentwicklung hat.

Das Prozessmodell beschreibt die Aktivitäten, die zur Entwicklung eines IT-Service stattfinden müssen. Da jedoch keine Vorgaben hinsichtlich der Ziel-Architekturen und einzusetzenden Frameworks existieren unterscheiden sich Projekte oftmals dahingehend sehr stark. Die technische Unterstützung (vgl. Unterkapitel 2.1.3) sorgt dabei ebenfalls nicht für eine Standardisierung[82]. Entsprechend ist der Wartungsaufwand hoch, da die jeweils geeigneten Kompetenzen vorhanden sein müssen. In Zeiten, in denen auf externe Mitarbeiter zur Unterstützung von Projekten zurückgegriffen wird, ergibt sich eine zusätzliche Herausforderung: Die Weitergabe des Wissens aus den Projekten an die festen Mitarbeiter, damit dieses nach Projektende noch vorhanden ist. Unterscheiden sich die Produkte der Projekte sehr stark wird dieser Vorgang der Wissensweitergabe zusätzlich komplizierter.

2.5.2 Einschlägige Literatur

Zu Beginn der kritischen Betrachtung der Literatur sei nochmals erwähnt, dass ein Ziel des MDSD die **Generierung** von Software ist (vgl. Unterkapitel 2.3.1). Ansätze des MDD verwenden nicht immer das Modell, um automatisiert eine Software zu erzeugen. Nach der Meinung des Autors verursacht dieses Vorgehen bei MDD, wenn dennoch eine Software entwickelt werden soll, einen wiederholt anfallenden Mehraufwand während der manuellen

[82] Trotz der Lösungen für häufige Probleme, wie beispielsweise Zeichensatzkonvertierungen, ist unter Umständen eine Verwendung aufgrund der gewählten Architektur nicht möglich.

Transformation (je nachdem, wie oft das Modell zu einem späteren Zeitpunkt zu verändern ist). Ebenso kann die manuelle Überführung in Quelltext zu einem Informationsverlust führen, da der Entwickler frei entscheiden kann, was übernommen wird. Auf der anderen Seite ist im MDD keine Lösung für Probleme, die mit der wiederholten Generierung auftreten (Umgang mit manuell implementierten Inhalten) nötig.

In Kapitel 1.3 wurde bereits beschrieben, dass die Kapiteleinteilung dieser Ausarbeitung an die Phasenaufteilung aus "*When and How to Develop Domain-specific Languages*"[83] anlehnt ist. Die Autoren beschreiben in Ihrem Paper auf nachvollziehbare weise die logische Abfolge der Schritte. Die aufgezeigten Patterns sind jedoch nur bedingt zu gebrauchen. Einerseits werden diese nur sehr kurz vorgestellt und andererseits wird nicht auf die praktische Anwendung eingegangen. Aufgeführt sind lediglich Beispiele für DSLs, die anhand der entsprechenden Patterns implementiert wurden. Das Paper ist zudem, wie auch andere vorgestellte Literatur, speziell in Hinblick auf die Werkzeugunterstützung, um eine Sprache zu definieren und anschließend zu verwenden, sehr veraltet. Enthalten sind zum Beispiel die Language Implementation System on Attribute Grammars (LISA) und die Jakarta Tool Suite (JTS). Aktuellere Werkzeuge wie beispielsweise *Xtext*, *MPS* und *ANTLR* werden nur in wenigen Schriftwerken, wie zum Beispiel in [Völ09], beschrieben. Dieser Umstand ist dahingehend auffällig, da speziell in den letzten Jahren die Verwendung von modellgetriebenen Ansätzen zugenommen hat. Nachvollzogen werden kann das an der vermehrten Entwicklung von unterstützenden Werkzeugen.

Im Themenbereich der modellgetriebenen Softwareentwicklung ist das Buch von **Stahl et al.** [SVE+07] aufgrund der detailliert beschriebenen Theorie mit den regelmäßig geschilderten praktischen Anwendungen sehr empfehlenswert. Speziell die zu betrachtenden Themen zur Entwicklung einer geeigneten DSL sind jedoch im Buch von **Trompeter et al.** besser auf den Punkt gebracht:[84]

- Einstellen des Abstraktionsniveaus
- Orientierung an Fachdomäne
- Formale Definition der DSL
- Convention over Configuration
- Modularität
- Kompakte vs. explizite Notation
- Nicht nur UML (-Profile)

[83] vgl. [MHS05], Überschrift: "2.1. Pattern classification"
[84] vgl. [TB07], Unterkapitel: "3.6.4 Anhaltspunkte auf dem Weg zu einer geeigneten DSL", Seite 62ff

- Nicht nur grafische Modelle (bei hoher Komplexität)
- Nicht nur grafische Modelle (bei vielen "einfachen" Modellinstanzen)
- Fachliche vs. technische DSLs

Das, in diesem Buch beschriebene, Praxisbeispiel ist allerdings in der Umsetzung weniger zur Nachbildung geeignet (nachfolgend erläutert). Ist die Zielarchitektur und das Metamodell festgelegt muss der Generator entwickelt werden. Die dazu nötige Transformation von Modell zu (Quell-)Text erfolgt anhand dafür definierter Vorlagen, die Muster für das jeweilige Modellelement beinhalten. Dabei muss entschieden werden, wie diese aufgebaut sind. Auf der einen Seite sollen die Vorlagen lesbar und nachvollziehbar sein. Auf der anderen Seite die Anforderungen, wie zur Qualität (z.B. das Vermeiden von Wiederholungen[85]) oder den Wechsel der verwendeten Technologie ohne großen Aufwand, genügen. Die Technologie bezieht sich auf das Ergebnis des Generators, das entweder als die zu verwendende Zielsprache oder die verwendeten Frameworks bzw. Bibliotheken aufgefasst werden kann. Der im Buch beschriebene Ansatz teilt die Vorlagen in die unterschiedlichen Technologien auf.[86] Wird die Technologie bzw. das verwendete Framework oft gewechselt, sind nur wenige Vorlagen anzupassen. Die Aufteilung anhand der Technologie hat jedoch zwei Auswirkungen. Einerseits entstehen aus einer Vorlage mehrere Dateien, die im Projekt an unterschiedlichen Stellen abgelegt sind. Andererseits ist nach einem Generierungsvorgang nur noch schwer nachvollziehbar, welche Teile aus welcher Vorlage verwendet wurden. Oftmals werden mehrere Technologien im Sinne von Frameworks eingesetzt, die untereinander kommunizieren. Innerhalb einer Datei sind dadurch mehrere Vorlagen enthalten.

Das Paper "*Model-driven development: The good, the bad, and the ugly*" beschreibt als ein Ziel von MDD, dass die Entwickler auf einen höheren Abstraktionslevel arbeiten. Dadurch soll der Entwicklungsaufwand und die Komplexität der Software verringert werden.[87] Nachvollziehbar ist dies bei MDSD, wenn automatisiert aus dem Modell eine direkt ausführbare Anwendung entsteht, da der Generator bzw. Transformator die Komplexität verbergen kann, nicht jedoch bei Ansätzen des MDD. Daraus ergibt sich die Frage, ob das Modell nicht sinnvollerweise von Fachexperten erstellt und verwendet werden sollte, da der Entwickler erst nach der Generierung einen Mehrwert liefern kann. Bei MDD hingegen wird zu Beginn das plattformunabhängige Modell, als Platform Independent Model (PIM) bezeichnet, erstellt. Mit Hilfe von Werkzeugen lässt sich dieses zu einem oder mehreren plattformabhängigen Modellen, den Platform Specific Models (PSMs), umwandeln.[88] Der

[85] auch DRY (Don't Repeat Yourself) genannt
[86] vgl. [TB07], Kapitel "6.4 Entwicklung der Generator Cartridge", Seite 201ff
[87] vgl. [HT06], Erster Absatz von: "WHAT PROBLEM IS MDD INTENDED TO SOLVE? (THE GOOD)"
[88] vgl. [Gro15], Überschrift: "MDA Overview"

Entwicklungsaufwand ist aufgeteilt, eventuell sogar leicht erhöht, nachdem mindestens immer zwei Modelle zu betrachten sind. Martin Fowler hat darüber auf seiner Webseite[89] bereits 2003 einen Kommentar verfasst, in welchem er dieses, unter Umständen überflüssige, Vorgehen anspricht. Er hinterfragt darin kritisch, warum ein Modell (das PIM) gleichzeitig für unterschiedliche Implementierungen (im Sinne der PSMs) benötigt wird, nachdem es bereits andere Lösungen zur Herstellung einer Plattformunabhängigkeit gibt. Eine Plattform ist in diesem Fall die Basis für die Anwendung im Sinne der Hardware und dem Betriebssystem. Ein Beispiel für eine plattformunabhängige Lösung ist die Programmiersprache ist Java.[90]

Eine gute Beschreibung für die Vorgehensweise zur Entwicklung einer DSL liefert der Artikel "*An Approach for the Systematic Development of Domain-Specific Languages*". Die zwei Jahre, die zwischen dieser Literatur und der Veröffentlichung von [SVE+07] liegen machen sich durch die, in der Praxis erprobten und zugleich beschriebenen, Ansätze bemerkbar. Ebenso hilfreich ist die jeweilige direkte Umsetzung der Theorie zur Entwicklung einer DSL. Diese Ausarbeitung lehnt sich an die vorgestellten Ansätze an. Im nachfolgenden Hauptkapitel werden dazu die abstrakte Syntax und die statische Semantik definiert.

[89] http://martinfowler.com
[90] vgl. [Fow03], Überschrift: "PlatformIndependentMalapropism"

3 Design

Die im vorangegangenen Hauptkapitel vorgestellten Grundlagen werden in diesem Hauptkapitel verwendet, um eine modellgetriebene Softwareentwicklung bei der Landeshauptstadt München (LHM) geeignet zu definieren. Zum einen muss dabei die Modellierung in das Prozessmodell eingebettet und zum anderen die Zielarchitektur hinsichtlich der Generierung betrachtet werden. Wie bereits in Unterkapitel 2.3.2 auf Seite 29, erläutert, besteht eine DSL aus einem Metamodell und einer konkreten Syntax. In diesem Hauptkapitel erfolgt die Definition des Metamodells. Es handelt sich dabei um das **was** in der Sprache enthalten ist. Das **wie** wird in Hauptkapitel 4 mit der Umsetzung des Metamodells beschrieben.

Eine optimale Lösung erfordert die Einhaltung der Konformität zum *Prozessmodell IT-Service* (vgl. Kapitel 2.1) bei gleichzeitiger Umsetzung der vorgegebenen Softwarearchitektur, wie zukünftige Anwendungen aufgebaut sein sollen. Im Ablauf des Prozesses zur Entwicklung eines IT-Service werden mehrere Dokumente erstellt: das Fachkonzept (FK), die Systemspezifikation (SysSpec) und das Testkonzept (TK). Die beste Integration in den Prozess ist erreicht, wenn gleichzeitig Teile besagter Dokumente automatisch erzeugt werden können.

Das primäre Ziel der Entwicklung der DSL soll im Gegensatz dazu die Generierung der Softwarearchitektur sein, da diese mehr Aufwand als die Erstellung der Dokumente in Anspruch nimmt. Aus diesem Grund wird auf den generierbaren Anteil der Dokumente nach der Metamodell-Definition eingegangen.

Der komplette Prozess der Entwicklung und Verwendung der DSL ist in Abbildung 3.1 dargestellt. Die, innerhalb dieser Ausarbeitung einmalig durchzuführenden, Schritte zur DSL-Entwicklung sind durch den oberen (gelben) Pfeil hervorgehoben. Der darunterliegende (blaue) Pfeil umfasst die wiederholt ausführbaren Schritte, deren Durchführung an einem Praxisbeispiel im Kapitel 4.3 erfolgt.

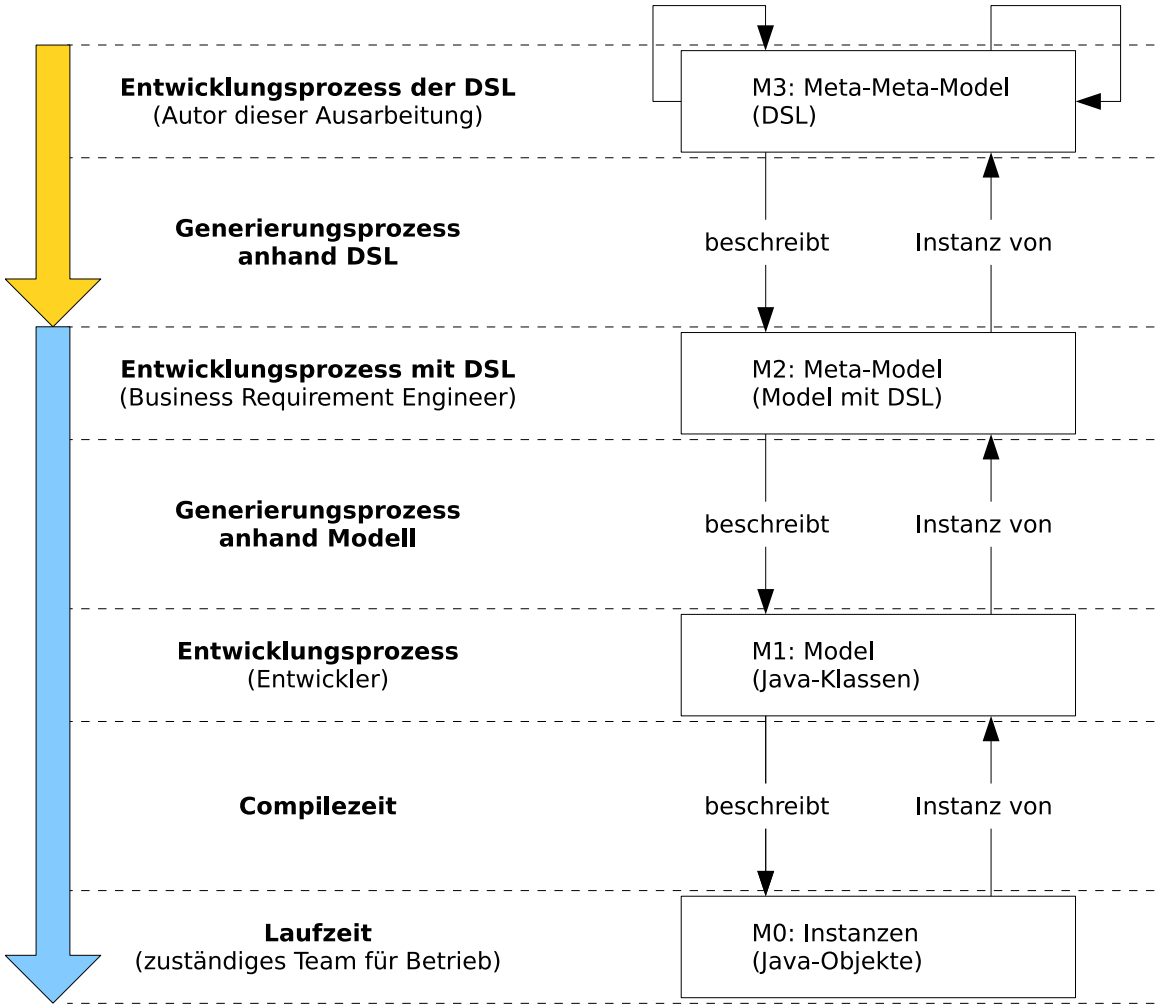


Abbildung 3.1: Darstellung der Prozessschritte der modellgetriebenen Entwicklung (auf der linken Seite) mit den korrespondierenden Modellschichten anhand UML 2.4.1 (auf der rechten Seite).

Nachfolgend ist die Integration der modellgetriebenen Entwicklung in das *Prozessmodell IT-Service* beschrieben. Die generierbaren Teile der Referenzimplementierung zur gleichzeitigen Bestimmung der erforderlichen Metamodellelemente erfolgt im Anschluss daran. Darauf aufbauend wird das Metamodell spezifiziert und auf die daraus erstellbaren Teile der Dokumente Bezug genommen.

3.1 Prozessintegration

In Abbildung 2.21 auf Seite 35 wurde bereits dargestellt, dass die Spezifikation und der Entwurf im *Prozessmodell IT-Service* in zwei voneinander getrennten Phasen stattfinden. Die Modellierung muss dementsprechend übergeordnet erfolgen (vgl. Abbildung 3.2), sodass mehrere Iterationen innerhalb dieser ermöglicht werden (vgl. Abbildung 3.3).

Für das *Prozessmodell IT-Service* bedeutet das, dass während der Anforderungsqualifizierung die Aktivitäten aus der Anforderungsbearbeitung zu einem gewissen Grad erfolgen. Ein dadurch bedingter vorgezogener Aufwand hält sich jedoch in Grenzen, da das Modell als Grundlage für sämtliche Ergebnisse (Dokumente, Entwürfe der Anwendung) der Aktivitäten aus beiden Phasen dient. Die Erstellung dieser wurde bisher komplett manuell (abgesehen der Vorlagen für die Dokumente) durchgeführt, was den Aufwand durch die Generierung insgesamt verringert, da lediglich eine Ergänzung bzw. Vervollständigung im Anschluss erforderlich ist.

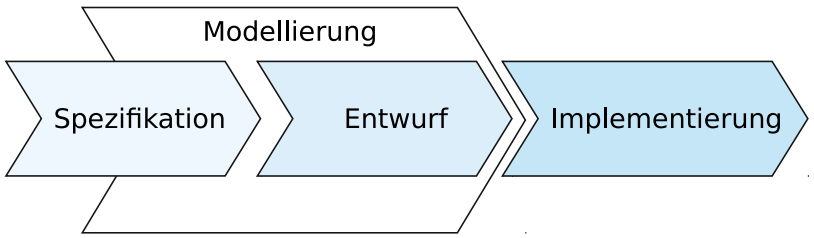


Abbildung 3.2: Darstellung des, um die Modellierung erweiterten, Lebenszykluses eines Softwaresystems.

Im Detail besteht die Modellierung aus folgenden Schritten (vgl. Abbildung 3.3), die sich wiederholen können:

1. Ein Business Requirement Engineer (BRE) erstellt bzw. bearbeitet das Modell der gewünschten Anwendung.
2. Der Generator erzeugt aus dem Modell unter anderem einen lauffähigen Prototyp.
3. Die Fachabteilung überprüft den Prototyp und gibt Rückmeldung an den BRE.

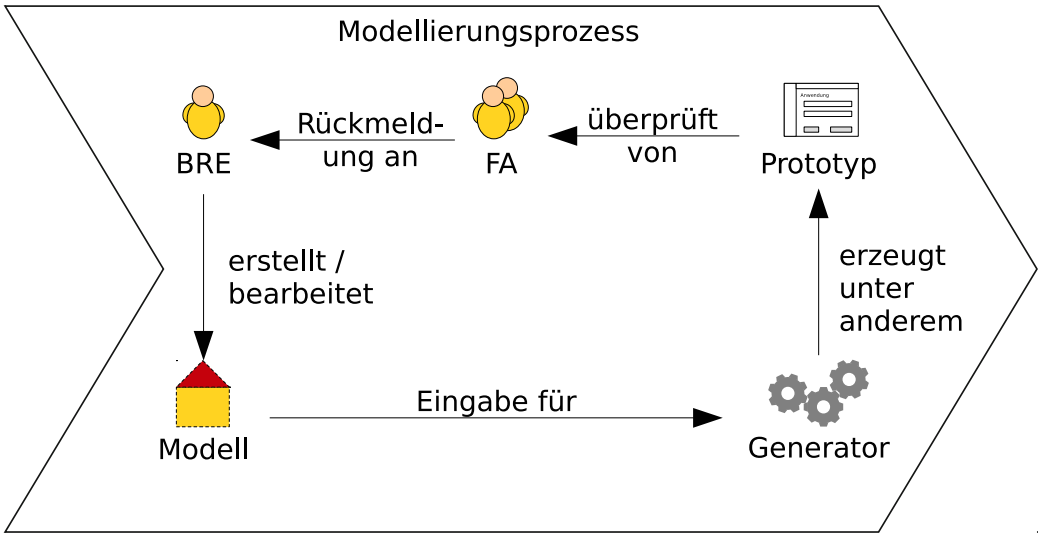


Abbildung 3.3: Detaildarstellung des Modellierungsprozesses aus Abbildung 3.2. Die Pfeile stellen die Ablaufrichtung der Schritte dar. Innerhalb der Modellierung können mehrere Durchläufe stattfinden. FA dient als Abkürzung für Fachabteilung.

Die Einbettung der beschriebenen Modellierung in das *Prozessmodell IT-Service* erfordert zudem, dass das Metamodell die Implementierung des vorgegebenen Softwarearchitekturmusters ermöglicht. Ebenso ist es eine Vorgabe, dass ein BRE, der keine Entwicklerkenntnisse besitzt, das Modell erstellen kann. Dieses sollte daher eine einfache bis selbsterklärende Syntax besitzen.

3.2 Generierbarkeit

Der Modellierungsprozess (vgl. Abbildung 3.3) beinhaltet die Schritte Modellierung und Generierung. Wie bereits in Abbildung 3.2 dargestellt erfolgt im Anschluss daran in der Implementierung die Ergänzung bzw. Erweiterung der Inhalte, die der Generator nicht erzeugt.

Um mit dem Generator einen Großteil zu automatisieren ist das Metamodell entsprechend aufzubauen. Die Referenzimplementierung dient dabei als Grundlage für das Metamodell. Eine Verkürzung, wie es nach der Definition von einem Modell (vgl. Unterkapitel 2.3.1, Seite 28) stattfindet, trifft speziell auf die zu erzeugende Struktur zu: es ist nicht vonnöten, dass die Zielarchitektur im Modell enthalten ist. In Kapitel 2.2 wurde die Referenzimplementierung bereits vorgestellt, wobei nachfolgend die darin enthaltenen Projekte erneut einzelnen reflektiert werden. Ziel ist in diesem Schritt die Beantwortung folgender Fragen:

- Was ist immer zu generieren und wird dementsprechend überschrieben?
- Was wird einmalig generiert und erfordert eine anschließende Vervollständigung?

- Wie sind diese Teile miteinander verknüpft?
- Was muss, um die vorherigen Fragen beantworten zu können, im Metamodell vorhanden sein?

Fowler bezeichnet die Aufteilung in einmalig und wiederholt generierte Teile in seinem Buch "*Domain-Specific Languages*" als *Generation Gap*. Er[91] und Stahl et al.[92] stellen zur Verquickung unterschiedliche Möglichkeiten vor, auf die an den entsprechenden Stellen verwiesen wird.

Nachfolgend beginnt die Betrachtung mit dem Microservice, als Anbieter einer Schnittstelle. Zur Verwendung dieser wird im Anschluss die Client-Bibliothek hinsichtlich der Fragen untersucht. Der GUI Microservice schließt die Analyse ab.

3.2.1 Betrachtung des Microservice

Der Microservice stellt die Funktionalität zur Unterstützung der Geschäftsprozesse bereit (vgl. Unterkapitel 2.2.3, Seite 18). Dazu immaniert dieser die benötigten Datenstrukturdefinition, bietet eine REST-Schnittstelle an und besitzt, wenn erforderlich, eine Anbindung an eine Datenbank (vgl. Abbildung 3.4).

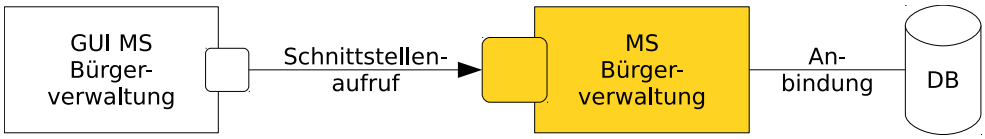


Abbildung 3.4: Übersicht über die Architektur der Referenzimplementierung mit hervorgehobenen MS.

Hinsichtlich der Datenstrukturdefinition ist es sinnvoll, wenn diese vollständig generiert wird. Hierfür ist die Eingabe von Datenmodellen mit entsprechenden Validierungen für die jeweiligen Datentypen zu ermöglichen (vgl. Listing 2.1, Seite 23). Eine Unterstützung der Vererbung und die Herstellung von Beziehungen untereinander ist zudem erforderlich. Die Vererbung wird häufig für die Definition von Grunddaten verwendet. Eine Beziehung besteht zum Beispiel im Einwohnermeldewesen zwischen zwei Buerger-Objekten, die verheiratet sind.

Damit im Modell eine Unterscheidung zwischen selbst zur Verfügung gestellter Schnittstelle und genutzten möglich ist, soll eine Markierung von "externen" Datenschemata möglich

[91] vgl. [Fow10], Chapter 57: "Generation Gap", Seite 571ff
[92] vgl. [SVE+07], Unterkapitel: "8.5.2 Verquickung von generiertem und nicht-generiertem Code", Seite 159ff

sein. Dies erlaubt die Verwendung anderer Schnittstellen, deren Daten selbst nicht zu verwalten sind.

Zur Modellierung von Verhalten, der fachlichen Logik im Microservice, existieren im MDSD folgende Alternativen:[93]

1. Angeboten wird lediglich eine Auswahl einfacher Funktionen.
2. Es wird auf andere Ansätze wie Zustandsdiagramme, Prädikatenlogik oder Regel-Engines zur Modellierung zurückgegriffen.
3. Die Definition einer eigenen Ausdruckssprache.
4. Das Verhalten wird mit der Sprache der Zielplattform im Modell beschrieben.

In Anbetracht von Alternative 1 kann die fachliche Logik unter anderem das Versenden von E-Mails, der zufälligen Auswahl von Datensätzen aus einer Datenbank oder die Aufbereitung von Daten unterstützen. Von Vorteil ist die schnelle Umsetzbarkeit zur Entwicklung dieser Auswahl. Es entsteht jedoch das Problem, dass in der Regel nicht alle möglichen Fälle abgedeckt sind. Durch eine Erweiterung dessen entstehen generierte Teile und Mischformen aus automatisch erzeugten und manuell implementierten Quelltexten.

Zur Verwendung von Alternative 2 sind entweder zusätzliche Werkzeuge einzusetzen oder die Abbildung im Metamodell erforderlich. Durch weitere Werkzeuge entstehen jedoch zusätzliche Abhängigkeiten, wie zum Beispiel deren Beschaffung (mit der benötigten Anzahl an Lizenzen), Installation und Einbindung in den Arbeitsablauf der Modellierung. Von Nutzen ist dafür die Möglichkeit der sehr detaillierten Modellierung des Verhaltens. Dadurch bedingt können zwei Schwierigkeiten eintreten. Zum einen muss der Modellierer das entsprechende Wissen zur Verwendung bzw. Anwendung besitzen. Die Einhaltung der Vorgabe von *einfach bis selbsterklärend* (vgl. Kapitel 3.1) ist dadurch nahezu nicht möglich. Zum anderen ist das modellierte Verhalten in die Zielsprache zu transformieren. Insgesamt ist durch diese Alternative die Komplexität erhöht und die Wartbarkeit verringert, obwohl der erhaltende Mehrwert im Vergleich gering bleibt.

Die Alternative 3 besitzt den Vorteil, dass die Ausdruckssprache speziell auf die benötigten Funktionen anpassungsfähig ist. Es bestehen jedoch die gleichen Problematiken, wie in Alternative 2.

Die aufgelistete Alternative 4 ist einerseits in der Umsetzung sehr praktikabel, da keine weitere Transformation stattfinden muss. Andererseits wird das Modell im Modellierungsprozess (vgl. Abbildung 3.3) von einem BRE erstellt, welcher keine Kenntnisse über die

[93] vgl. [SVE+07], Unterkapitel "6.3.2 Modellierung von Verhalten", Seite 114f

Sprache der Zielplattform besitzt. Infolgedessen ist die Anwendung von diesem Ansatz nicht möglich.

Zusammengefasst eignet sich keine der vorgestellten Alternativen aufgrund der überwiegenden Nachteile. Auf die Modellierung von Verhalten soll daher im Sinne der Verkürzung (vgl. Unterkapitel 2.3.1, Seite 28), als eines der Hauptmerkmale von einem Modell, verzichtet werden. Sinnvoll ist dennoch die Möglichkeit ein Grundgerüst[94] zur späteren Vervollständigung modellieren zu können. Die dafür benötigten Informationen sind vergleichbar mit den grundlegenden Angaben von Anwendungsfällen bzw. Ansätzen aus dem BDD: Given, Then, When (vgl. Anhang A.4). Die Modellierung von "Geschäftsaktionen", die folgende Daten umfassen, ist zu unterstützen:

- einen Namen,
- das Ziel,
- die Eingabe und
- die Ausgabe.

Eine Aufteilung des Microservice in wiederholt und einmalig generierte Teile ist in Folge der nachträglich zu vervollständigen Geschäftsaktionen sinnvoll. Dazu ist dieser im weiteren Verlauf in Hinblick auf die interne Kommunikation zwischen den Klassen (vgl. Abbildung 2.12, Seite 21) detaillierter betrachtet, damit die von einem Entwickler erweiterbaren Bereiche bestimmt werden können.

Die Bereitstellung der Geschäftsaktionen durch den Microservice (vgl. Unterkapitel 2.2.3, Seite 18) erfolgt durch die *BusinessAction*-Klassen. Der *BusinessActionController* übergibt die jeweiligen Anfragen zur Verarbeitung an die *BusinessActionServiceImpl* (vgl. Abbildung 3.5). Anhand der eingeführten Geschäftsaktionen eignet sich das Vorgehen *entwurfsmuster-basierte Integration* aus [SVE+07][95]. Umgesetzt auf die Referenzimplementierung wird im vollständig generierten Controller eine Methode der nicht-generierten Service-Klassen aufgerufen (vgl. Kapitel 4.1). Der Controller kann anhand der bereits im Metamodell geforderten Geschäftsaktionen vollständig erzeugt werden. Die Service-Klasse ist einmalig zu erstellen und anschließend durch einen Entwickler um die benötigte fachliche Logik zu vervollständigen. Angewendet ist in diesem Fall das Pattern der *dreistufigen Vererbung*[96], in dem die Implementierung von einem Interface im einmalig zu generierenden Teil abgelegt wird.

[94] In der Programmierung abgebildet durch Interfaces.
[95] Überschrift: "Entwurfsmuster-basierte Integration" in Unterkapitel 8.5.2, Seite 162
[96] vgl. [SVE+07], Unterkapitel: "8.5.2 Verquickung von generiertem und nicht-generiertem Code", Seite 161

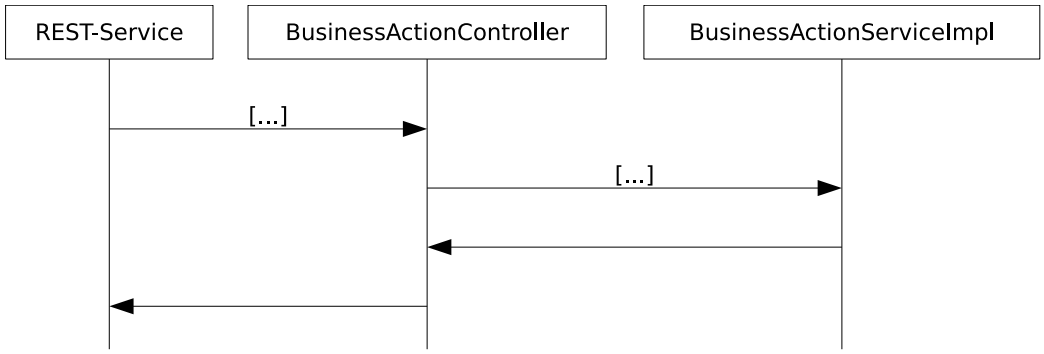


Abbildung 3.5: Sequenzdiagramm zur Verarbeitung von Geschäftsaktionen im Microservice. Ausgelöst werden die Geschäftsaktionen über die REST-Schnittstelle.

Neben den Geschäftsaktionen (GA) bietet der Microservice die Create, Read, Update, Delete (CRUD)-Operationen an. Wie in Abbildung 2.10 (Seite 19) ersichtlich, erfolgt die Bereitstellung dieser direkt vom Repository. Damit die CRUD-Operationen um eigene fachliche Logik erweiterbar sind, fungieren die Klassen *BuergerResourceServieImpl* und *BuergerEventServiceImpl* (vgl. Abbildung 2.12, Seite 21).
Im Detail übergibt der *BuergerResourceProcessor*, bevor dieser die Antwort an den Client schickt, die Daten an die *BuergerResourceServieImpl*. Dadurch können diese noch verändert werden. Der *BuergerEventListener* dient als Erweiterungsklasse, um Einstiegspunkte an jeder CRUD-Operation sowohl vorher als auch nachher bereitstellen zu können. Entsprechende Aufrufe werden an die *BuergerEventServiceImpl* mit der Möglichkeit zur Manipulation der Daten weitergegeben (vgl. Abbildung 3.6). Umgesetzt ist auf diese Weise die *entwurfsmuster-basierte Integration*[97], in welcher über Annotationen der manuell zu implementierende Quelltext aufgerufen wird (vgl. Abbildung 3.6).

[97] vgl. [SVE+07], Unterkapitel: "8.5.2 Verquickung von generiertem und nicht-generiertem Code", Seite 162

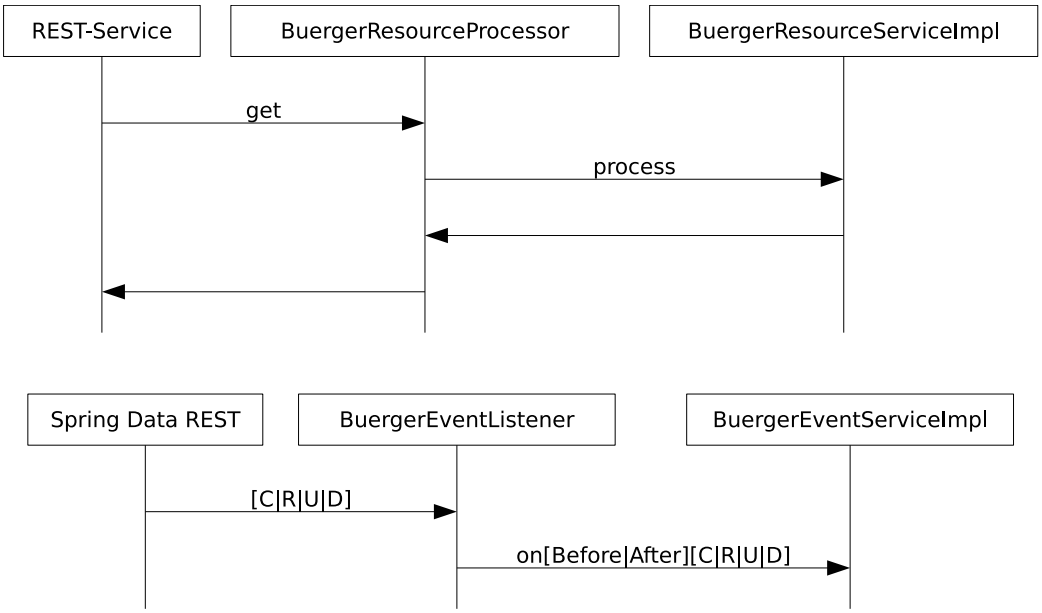


Abbildung 3.6: Sequenzdiagramme zur Verarbeitung von den Ereignissen bevor oder nachdem eine CRUD-Operation im Microservice aufgerufen wird. Auslöser des jeweilige Ereignisses ist Spring Data REST.

Die Service-Klassen beinhalten folglich die erforderliche fachliche Logik und sind, wie die davor beschriebene *BusinessActionServiceImpl*, durch einen Entwickler erweiterbar.

Die Abbildung 3.7 stellt eine Zusammenfassung anhand der, im Microservice enthaltenen, fachlichen Klassen am Beispiel eines Bürgers aus der Referenzimplementierung dar. Farbig hervorgehoben sind darin die einmalig zu generierenden Klassen, die von einem Entwickler um die benötigte (fachliche) Logik erweiterbar sind und bei erneuter Modellierung erhalten bleiben sollen. Die weiß hinterlegten Klassen können vollständig erzeugt werden.

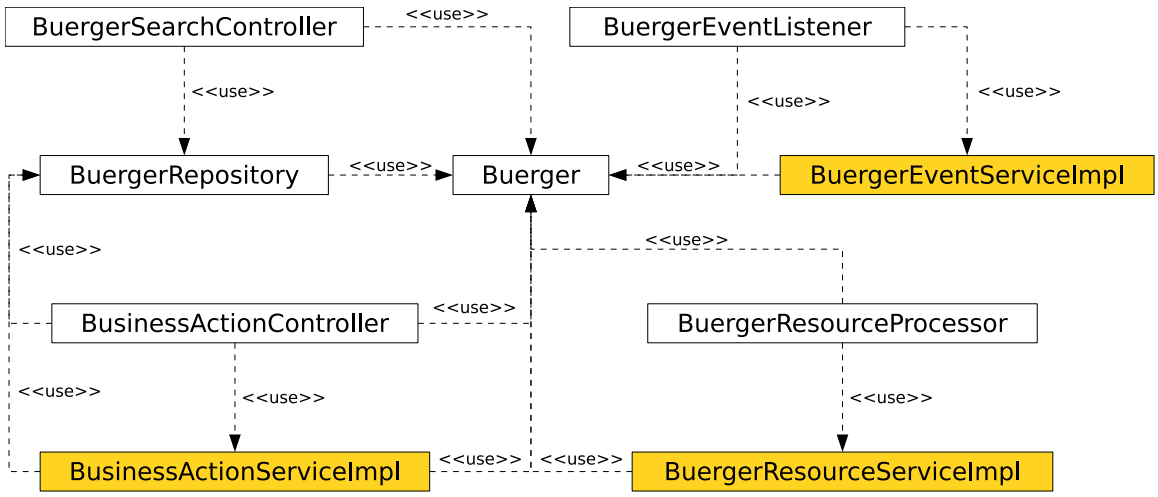


Abbildung 3.7: Klassendiagramm des MS der Referenzimplementierung. Eine vollständige Generierung ist bei Klassen mit weißem Hintergrund möglich. Farblich hervorgehoben sind Klassen, die als Grundgerüst einmalig generiert und anschließend von einem Entwickler vervollständigt werden.

3.2.2 Betrachtung der Client-Bibliothek

Die Client-Bibliothek stellt das komplette fachliche Datenmodell und den REST-Client zur Microservice-Anbindung bereit (vgl. Unterkapitel 2.2.4, Seite 22). Dazu wird diese von anderen Projekten, wie der GUI oder einem anderen Microservice eingebunden und verwendet. Das Projekt ist unabhängig zu den Microservices und frei von fachlicher Logik. Die Abbildung 3.8 stellt die Client-Bibliothek, die innerhalb der *GUI MS Bürgerverwaltung* verwendet wird, hervorgehoben dar.

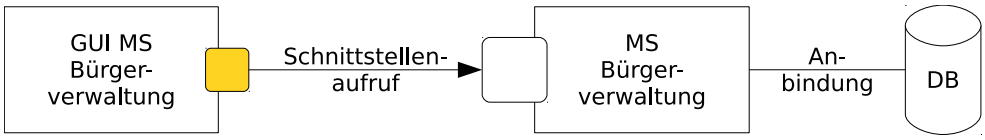


Abbildung 3.8: Übersicht über die Architektur der Referenzimplementierung mit hervorgehobener Client-Bibliothek zum Zugriff auf die REST-Schnittstelle.

Auf Basis der im Microservice identifizierten erforderlichen Elemente ist es möglich und sinnvoll die Client-Bibliothek komplett zu generieren. Somit werden REST-Schnittstelle und -Client gemeinsam vollständig generiert, wodurch die Kompatibilität erhalten bleibt.

3.2.3 Betrachtung der GUI

Die GUI nutzt die Client-Bibliothek zum Zugriff auf eine Schnittstelle und befüllt damit die Oberfläche mit den fachlichen Daten. Die Abbildung 3.9 stellt die Architektur mit

der hervorgehobenen *GUI MS Bürgerverwaltung* dar. In diesem Beispiel nutzt *GUI MS Bürgerverwaltung* die Schnittstelle von *MS Bürgerverwaltung*.

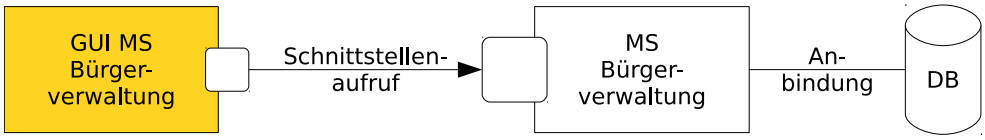


Abbildung 3.9: Übersicht über die Architektur der Referenzimplementierung mit hervorgehobenen GUI MS.

Die Erstellung standardmäßig benötigter Views und Komponenten (vgl. Unterkapitel 2.2.5, Seite 24), unter anderem für die CRUD-Operationen, ist anhand der bisher geforderten Elemente für ein Datenschema realisierbar. Geschäftsaktionen können in die Navigation aufgenommen und die Oberflächenelemente als Grundgerüst generiert werden. Durch den verwendeten *Autowiring*-Mechanismus des Spring-Frameworks kann ein Entwickler nachträglich weitere Views und Komponenten erstellen. Diese können dabei zudem auf Vorhandene zurückgreifen (vgl. Abbildung 2.15, Seite 25).
Standardmäßig dient eine Tabelle-Komponente zur Darstellung von mehreren Datensätzen (vgl. Abbildung 2.18, Seite 27). Zur Optimierung der darin möglichen Suche (zur Filterung) und der anzuzeigenden Spalten der Tabelle muss eine zusätzliche Markierung von Attributen, sozusagen als Hauptattribut, verfügbar sein.
Zur Vereinfachung von Formulareingaben ist es sinnvoll, dass jedes Feld ein Beispiel beinhaltet, welches in der Oberfläche als Standardwert oder als Unterstützungsvorschlag angezeigt wird. Ein Start des, im Modellierungsprozess erstellten, Prototyps zu Demonstrationszwecken kann auf diese Weise mit bereits enthaltenen Daten erfolgen.

In Abbildung 3.10 ist eine Zusammenfassung anhand der im GUI Microservice enthaltenen fachlichen Klassen dargestellt. Farbig hervorgehoben sind darin die einmalig zu generierenden Klassen. Die Erweiterung durch einen Entwickler erfolgt, im Vergleich zum Microservice (vgl. Unterkapitel 3.2.1), nach dem Pattern der *dreistufigen Vererbung*[98]. Dabei erfolgt eine wiederholte Generierung der Grundlage und ein Entwickler implementiert die nötigen Anpassungen in der abgeleiteten, einmalig durch den Generator erstellten, Klasse. Beispielsweise ist dadurch die Vervollständigung der Views für die Geschäftsaktionen durchführbar.
Die gewünschte Oberfläche mit den entsprechenden Elementen ist damit implementierbar und bleibt bei erneuter Modellierung erhalten. Die weiß hinterlegten Klassen können vollständig erzeugt werden.

[98] vgl. [SVE+07], Unterkapitel: ”8.5.2 Verquickung von generiertem und nicht-generiertem Code”, Seite 161

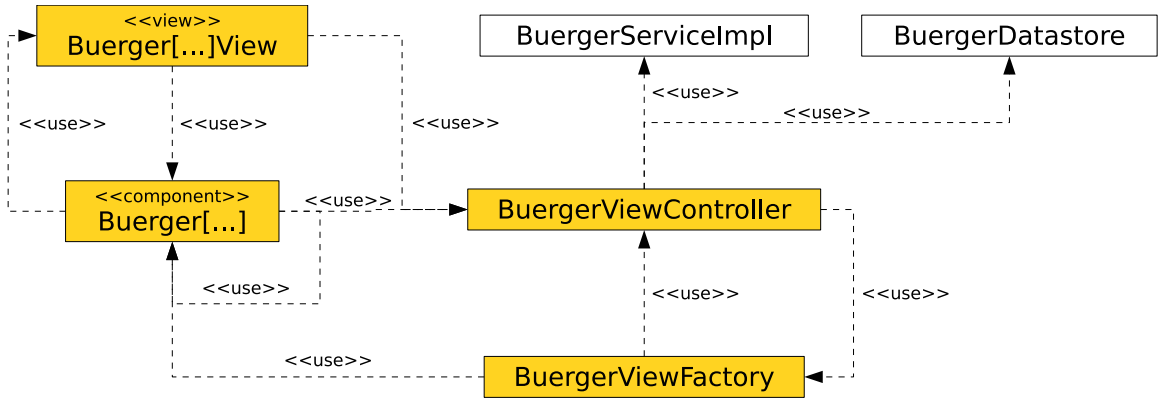


Abbildung 3.10: Klassendiagramm des GUI MS der Referenzimplementierung. Eine vollständige Generierung ist bei Klassen mit weißem Hintergrund möglich. Farblich hervorgehoben sind Klassen, die als Grundgerüst einmalig generiert und anschließend von einem Entwickler vervollständigt werden.

Mit der GUI ist die Betrachtung der Referenzimplementierung abgeschlossen und die erforderlichen Elemente für das Metamodell identifiziert. Dieses wird im folgenden Kapitel detailliert beschrieben.

3.3 Metamodell

Das Metamodell umfasst die abstrakte Syntax und statische Semantik (vgl. Unterkapitel 2.3.2, Seite 29). Die abstrakte Syntax wird im Folgenden anhand der benötigten Sprachelemente und deren Beziehung definiert. Die dafür erforderlichen formalen Bedingungen sind die Eindeutigkeit der Namen und die vorherige Definition von referenzierten Elementen.

Die Basis einer Anwendung besteht aus den übergeordneten Informationen, den Meta-Daten. Diese umfassen: Anwendungsname, organisatorische Zuordnung und die Version. Damit ist eine eindeutige Zuordnung jeder Anwendung gewährleistet und spätere Änderungen bleiben nachvollziehbar. Die Abbildung 3.11 stellt die eingangs beschriebenen Grunddaten grafisch dar und wird schrittweise um die weiteren Metamodellelemente ergänzt.[99]

[99] Angelehnt an die iterative Entwicklung der DSL, wie es von Voelter empfohlen wird (vgl. [Voe13], Unterkapitel: ”6.1.2 Iterative Development”, Seite 163ff).

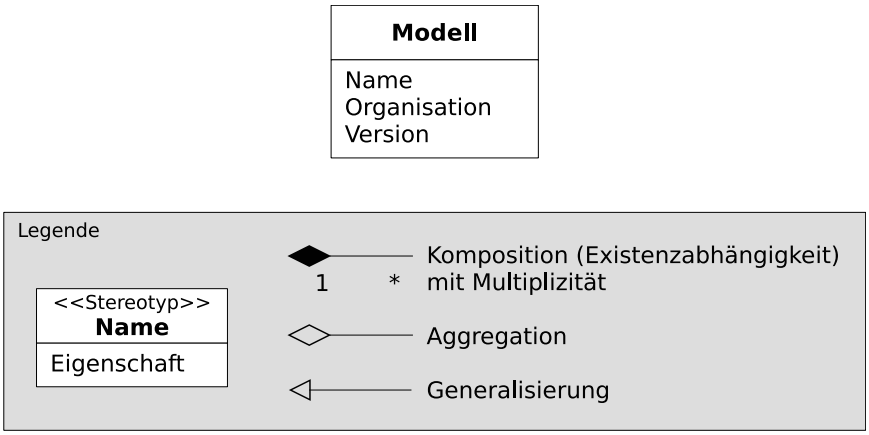


Abbildung 3.11: Das grundlegende Metamodell erlaubt die Definition eines *Modell*s, dargestellt als UML-Klassendiagramm mit Legende.

Die fachliche Grundlage wird durch das erforderliche Datenmodell festgelegt (vgl. Unterkapitel 3.2.1). Das Metamodell muss die namentliche Definition von Entitäten, damit unter anderem eine Vererbung realisierbar ist, einfordern. Da eine Anwendung darüber hinaus auf Daten anderer Anwendungen bzw. Microservices zugreifen kann (vgl. Unterkapitel 2.2.3, Seite 18), ist eine entsprechende Markierung für diese Entitäten nötig.

Eine Entität enthält mindestens ein Attribut. Das Attribut wiederum besteht aus dem Namen, den Angaben, ob es optional und/oder ein Hauptattribut ist und einem Beispiel. Die Spezifikation von einem optionalen Attribut ist Teil der benötigten Validierung und wurde anhand dem *Convention Over Configuration*-Paradigma[100] gewählt. Hintergrund war, dass der Großteil der Attribute in der Referenzimplementierung standardmäßig benötigt wird und eine Speicherung ohne diese nicht gestattet ist. Die Markierung als Hauptattribut hat zwei Ziele: einerseits die Festlegung der zu durchsuchenden Felder zur Optimierung der Suche auf Seiten des Microservice (vgl. Unterkapitel 2.2.3, Seite 18) und andererseits die Vorgabe der Spalten von Tabellen auf Seiten des GUI Microservice (vgl. Abbildung 2.18, Seite 27).
Im Modellierungsprozess (vgl. Abbildung 3.3) wird ein Prototyp erzeugt, der zur Überprüfung der Modellierung dient. Dieser kann optional mit den Beispieldaten von den modellierten Attributen gestartet werden.
Ebenso kann ein Attribut wahlweise eine Zuordnung zu einer Entität erhalten. Diese dient zur Erstellung von Beziehungen untereinander. Beispielsweise können dadurch zu einem Bürger die Kinder, die wiederum ebenfalls Bürger sind, angegeben werden.

Die Abbildung 3.12 stellt das, um die Entität und dem Attribut ergänzte, Metamodell dar.

[100] vgl. [SVE+07], Unterkapitel "6.3.6 Convention over Configuration", Seite 116ff

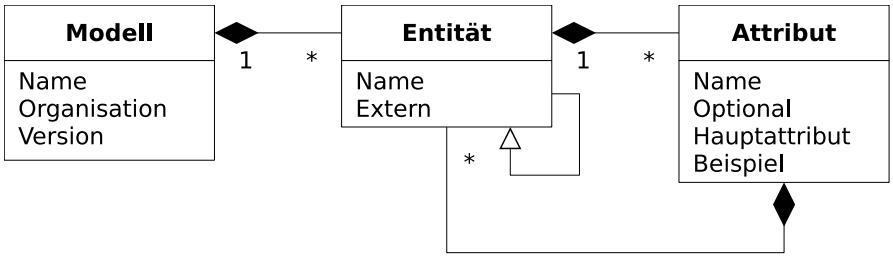


Abbildung 3.12: UML-Klassendiagramm des um Entitäten und Attribute erweiterten Metamodells. Ein Attribut kann selbst eine Entität sein.

Die Grundlage für ein Attribut ist der Datentyp. Dieser wird technisch zur Erstellung der Java-Klassen benötigt und enthält weitere Information zur Validierung. Entsprechend des Datentyps sollen die dafür sinnvollen Validierungsparameter angegeben werden. Die Standarddatentypen mit den zusätzlich erlaubten Optionen sind:

- Ein **Text** unterstützt die Angabe der minimalen und der maximalen Länge, sowie die erlaubten Muster[101]. Zum Beispiel ist damit die Validierung einer E-Mail-Adresse durchführbar. In Kombination mit der, im Attribut vorhandenen, optional-Markierung ist es realisierbar, dass diese Überprüfung im Anschluss an die Eingabe ausgeführt wird.
- Eine **Zahl** kann eindeutig und in der angegebenen Größe begrenzt sein. Beispielsweise können damit fortlaufende fachliche Nummern oder eine Postleitzahl validiert werden.[102]
- Eine **Liste** entspricht einer vordefinierten festen Auswahl möglicher Elemente.[103] Diese können entweder direkt im Modell angegeben oder von einer anderen Quelle bezogen werden. Ein Beispiel hierfür wäre die Angabe einer vordefinierten Liste von Augenfarben.
- Ein **Wahrheitswert** dient zur Speicherung von ja/nein Feldern. In Kombination mit der optional-Angabe im Attribut kann dabei zusätzlich gespeichert werden, ob bereits eine Eingabe erfolgt ist, sodass kein Standardwert angenommen wird.[104]
- Ein **Datum** kann in der Vergangenheit oder in der Zukunft liegen. Somit ist zum Beispiel die Angabe und Überprüfung eines Geburtstages realisierbar.

[101] Die Muster werden über reguläre Ausdrücke umgesetzt.
[102] Die fachliche Validierung einer Postleitzahl (Existenz) ist durch diesen Validierungsparameter nicht möglich. Ein Entwickler kann diese Überprüfung nach der Generierung (vgl. Unterkapitel 3.2.1) hinzufügen.
[103] Vergleichbar mit einem Aufzählungstyp.
[104] In Java existieren die Typen *boolean* und *Boolean*. Der zuletzt genannte, sogenannte Referenztyp, unterscheidet sich dahingehend, dass ebenso der Wert ”null”, für noch nicht angegeben, eine valide Belegung ist.

Die Kombination von einem Datentyp mit erzwungener Angabe eines Beispiels bietet hinsichtlich des Modellierungsprozesses einen weiteren Vorteil: während der Definition eines Modells ist eine Überprüfung möglich, ob das angegebene Beispiel zur festgelegten Validierung gültig ist.

Das bisher beschriebene Metamodell mit den hinzugefügten Datentypen zeigt Abbildung 3.13. Die grundlegenden Operationen (CRUD) einschließlich der dafür benötigten Schnittstelle und der Oberfläche können damit bereits komplett erstellt werden.

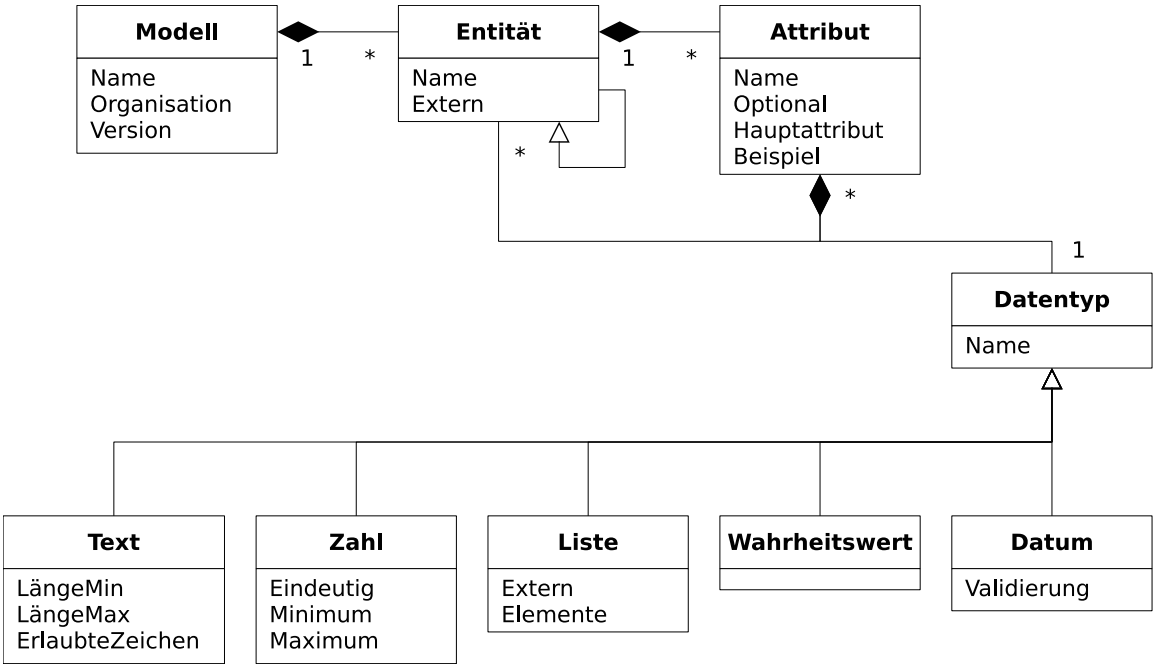


Abbildung 3.13: UML-Klassendiagramm des Metamodells im Vergleich zu Abbildung 3.12 ergänzt um die Zuordnung eines Datentyps zu einem Attribut.

Die Elemente zur Unterstützung von Geschäftsaktionen vervollständigen das Metamodell. Sie beinhalten, wie bereits in Unterkapitel 3.2.1, der Generierbarkeit des Microservice festgelegt, zur Identifizierung einen Namen, eine Zielbeschreibung und die zur Eingabe bzw. Ausgabe verwendeten Entitäten oder Datentypen. Dabei ist es möglich, dass eine Geschäftsaktion weder Ein- noch Ausgabeparameter besitzt. Das kann zum Beispiel für einen asynchronen Methodenaufruf zu einem anderen Service nötig sein.

In Abbildung 3.14 wird das vollständige Metamodell mit den erforderlichen Elementen abgebildet.

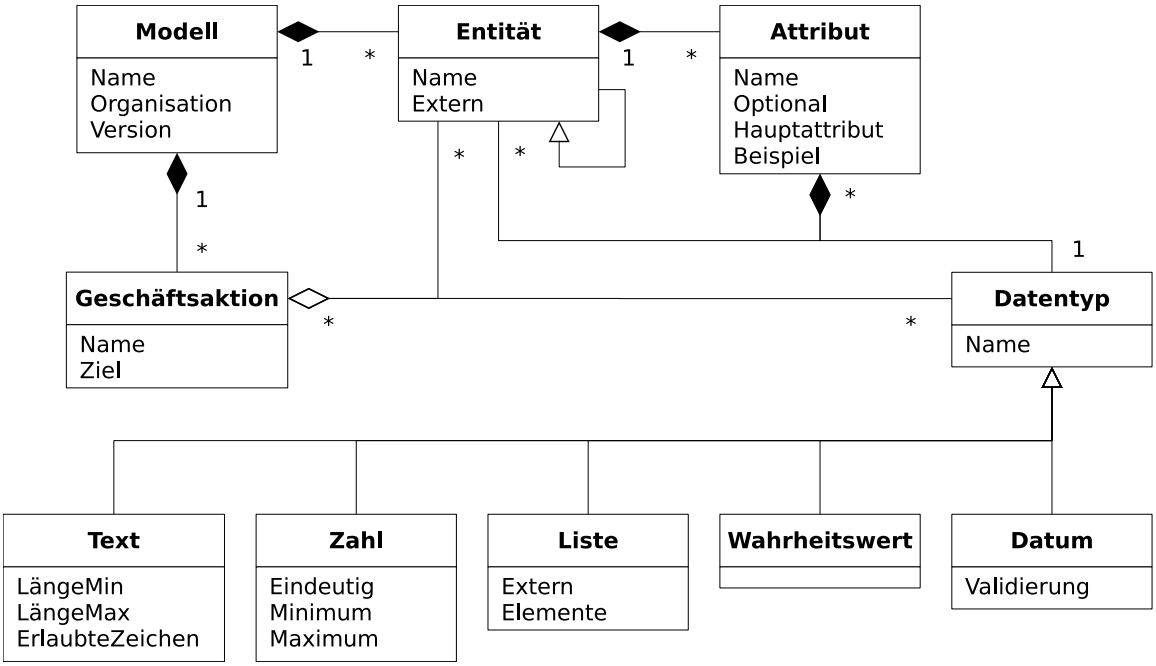


Abbildung 3.14: Das komplette Metamodell dargestellt als UML-Klassendiagramm. Die Aggregation einer Geschäftsaktion kann entweder mit einer Entität oder mit einem Datentyp erfolgen.

3.4 Dokumentenvorlagen

Für die innerhalb des *Prozessmodell IT-Service* zu erzeugenden Dokumente existieren zur Verwendung festgelegte Vorlagen. Um den Prozess weitestmöglich durch die Modellierung zu unterstützen, werden die Dokumente, die bisher in den Phasen Spezifikation und Entwurf entstehen, näher untersucht. Als Ergebnis sind die generierbaren (Unter-)Kapitel anhand des definierten Metamodells aufgelistet.[105] Angegeben ist zudem, welches Metamodellelement zur Befüllung geeignet ist. Des Weiteren kann oftmals auf die Inhalte des jeweiligen Dokuments der Referenzimplementierung zurückgegriffen werden, wenn die Modellierung darauf keinen Einfluss hat. Dies ist zum Beispiel hinsichtlich der Architektur der Fall, welche nicht im Modell enthalten ist (vgl. Kapitel 3.2).

3.4.1 Fachkonzept-Vorlage

Das FK beschreibt allgemein **was** fachlich umgesetzt werden soll. Es ist vergleichbar mit einem Lastenheft und gibt unter anderem Auskunft über folgende Fragen:

- Was ist zu implementieren?
- Welche Anwendungsfälle sind vorhanden?

[105] Die Kapitel-Gliederung wurde direkt übernommen.

- Welches Schnittstellen werden benötigt bzw. angeboten?

Anhand des definierten Metamodells ist eine teilweise bis vollständige Generierung der folgenden (Unter-)Kapitel realisierbar:

- **8.1 Anwendungsfall-Liste**: Beinhaltet eine Tabelle mit den Spalten für Nummer, Namen und der Kurzbeschreibung pro Anwendungsfall. Es ist mit Hilfe einer fortlaufender Nummer und anhand der modellierten Geschäftsaktionen möglich diese vollständig zu generieren.
- **8.3 Beschreibung der Anwendungsfälle**: Jeder Anwendungsfall besitzt eine eigene Tabelle mit allen dafür notwendigen Daten, wie Vorbedingung, Schritte, Nachbedingung. Diese Tabellen können mit der erstellten fortlaufenden Nummer und der jeweiligen Geschäftsaktion teilweise befüllt werden.
- **9 Fachliches Datenmodell**: Das fachliche Datenmodell wird als UML-Klassendiagramm ohne Stereotypen, Methoden und Variablen dargestellt. Die vollständige Erstellung aus dem modellierten Datenmodell ist möglich.
- **11.1 Von der Anwendung angebotene Schnittstellen**: Eine komplette technische Beschreibung der Schnittstelle ist anhand des Datenmodells generierbar. Die fachliche Beschreibung, wie oft zum Beispiel die Schnittstelle genutzt wird, ist nachzutragen.
- **11.2 Von der Anwendung benötigte Schnittstellen**: Im Datenmodell sind die entsprechenden Entitäten mit *extern* markiert, wodurch diese in diesem Unterkapitel technisch beschrieben werden können. Die fachliche Erläuterung ist im Anschluss zu ergänzen.
- **12 Rollen- und Berechtigungskonzept**: Das Konzept kann komplett von der Referenzimplementierung übernommen werden. Die benötigten Rollen sind nachträglich zu ergänzen.

3.4.2 Systemspezifikation-Vorlage

Die SysSpec beschreibt im Detail das **wie** der Umsetzung. Es ist vergleichbar mit dem Pflichtenheft.

Eine teilweise bis vollständige Generierung der folgenden (Haupt-)Kapitel ist anhand des definierten Metamodells umsetzbar:

- **2 Systemüberblick**: Besteht aus einer Übersicht mit allen wesentlichen Aspekten und Bestandteilen. Dieser kann teilweise aus den als *extern* markierten Entitäten und den Geschäftsaktionen generiert werden.
- **3 IT-Architektur**: Ein Befüllung ist nicht erforderlich. Zum besseren Verständnis

der nachfolgenden Unterkapitel hier aufgeführt.

- **3.1.4 Entwurfsentscheidungen**: Kann ohne spezielle Elemente aus dem Modell anhand der Referenzimplementierung komplett automatisch erstellt werden.
- **3.2 Logische Sicht**: Die Beschreibung der Aufteilung in Client (Browser) und Server (Anwendung) kann aus der Referenzimplementierung übernommen werden. Die von der Anwendung genutzten Schnittstellen aus den als *extern* markierten Entitäten. Entsprechend ist dieses Unterkapitel komplett generierbar.
- **3.3 Entwicklungssicht**: Im Vergleich zum FK ist darin ein detaillierteres UML-Klassendiagramm und eine Erläuterung der Package-Struktur enthalten. Beides kann aus dem modellierten Datenmodell und der Angaben aus der Referenzimplementierung komplett erstellt werden.
- **3.4 Laufzeitsicht**: Sequenzdiagramme, die die Kommunikation darstellen, können als Vorlage aus der Referenzimplementierung verwendet und um die modellierten Geschäftsaktionen automatisiert vervollständigt werden.
- **3.5 Verteilungssicht**: Eine komplette Übernahme dieses Kapitels aus der Referenzimplementierung ist möglich.
- **3.6 Systemumgebung**: Die erforderlichen Komponenten, wie zum Beispiel die Datenbank, optional der Webserver und die Java-Version können vorgegeben aber nur teilweise generiert werden.
- **4.1 Statisches Modell**: Die technische Umsetzung des fachlichen Datenmodells einschließlich dem Mapping[106] und die Persistenz können komplett aus dem modellierten Datenmodell und der Referenzimplementierung generiert werden.
- **4.2 Dynamisches Modell**: Beschrieben wird das, für den Kontrollfluss vorhandene, Modell (die DTOss) und veranschaulicht mit Aktivitäts- und Sequenzdiagrammen. Eine Generierung aller Inhalte aus dem modellierten Datenmodell ist realisierbar.
- **4.3 Datenhaltungskonzept**: Die Grundlage bezüglich der Datenbank kann aus der Referenzimplementierung verwendet werden. Der Aufbau und die Schnittstellen anhand des modellierten Datenmodells. Entsprechend eignet sich dieses Unterkapitel zur automatisierten vollständigen Erstellung.
- **4.4 Namenskonventionen**: Diese Erläuterungen können komplett aus der Referenzimplementierung übernommen werden.
- **5 Schnittstellen (zu Nachbarsystemen)**: Die angebotenen und verwendeten Schnittstellen, sowie deren Mapping und Verwendung kann komplett aus dem modellierten Datenmodell generiert werden. Zu vervollständigen sind Angaben, wie zum Beispiel die Nutzungshäufigkeit der Schnittstelle.

[106] Abbildung der Daten von einem Schema A auf ein Schema B.

- **6.3 Validierungen und Plausibilitäten**: Die Validierungen können komplett aus den modellierten Datentypen abgeleitet und generiert werden. Nachträglich hinzugefügte fachliche Validierungen sind zu ergänzen (vgl. Fußnote 102 auf Seite 52).
- **6.6 Konfigurierbarkeit**: Die Möglichkeiten zur Konfiguration ist aus der Referenzimplementierung zu adoptieren.
- **7 Rollen- und Berechtigungskonzept**: Das Konzept kann aus der Referenzimplementierung übernommen werden. Die benötigten Rollen sind zu ergänzen.
- **8 Weitere Sicherheitsaspekte**: Wie das System geschützt ist und welche Frameworks in Verwendung sind kann komplett aus der Referenzimplementierung übernommen werden.

3.4.3 Testkonzept-Vorlage

Das Testkonzept (TK) beschreibt **wie was** zu testen ist. Die Sicherstellung der gewünschten Funktion der Anwendung ist das Ziel zur Durchführung der Tests anhand dem TK. Die Standards zum Testvorgehen in der LHM sind im internen *Testhandbuch* festgelegt. Auf Basis von diesem, sowie anhand des FKs und der SysSpec der zu testenden Anwendung wird das TK erstellt.

Im Detail werden zu beginn dieses Dokuments die Testziele anhand der Aufteilung in die Testarten Integrations-, System- und Abnahmetests definiert. Über die Beschreibung der Testschwerpunkte wird eine Planung pro Testart festgelegt. Ebenfalls enthalten ist die Darstellung der Testdurchführung, der Testumgebung und der Abnahmekriterien.

Das definierte Metamodell beinhaltet keine Informationen über die Durchführung bzw. das Ziel der Tests. Entsprechend kann dieses Dokument nicht weiter automatisiert befüllt werden.

4 Umsetzung

Aufbauend auf dem Design erfolgt in diesem Hauptkapitel die technische Umsetzung. In Unterkapitel 2.1.3 auf Seite 11 wurde bereits über die Vorgabe von Eclipse als IDE informiert. Zur Modellierung besitzt die IDE ein eigenes Modellierungs-Framework, das sogenannte Eclipse Modeling Framework (EMF). Damit können Anwendungen anhand eines strukturierten Datenmodells erstellt werden.[107] Ein Teil des EMF-Projekts und darauf basierend ist das Xtext-Framework.[108] Mit Xtext ist die Entwicklung von Programmiersprachen und DSLs umsetzbar. Es deckt alle Aspekte einer kompletten Sprach-Infrastruktur (Parser, Linker, Compiler oder Interpreter) ab.[109] Dadurch wird auf einen Vergleich mit dem klassischen Compilerbau an dieser Stelle verzichtet.
Das Xtext-Framework unterstützt die Eingabe vom Metamodell und Modell in Textform und kann diese anschließend grafisch aufbereiten. Die Integration in die Eclipse IDE erlaubt es, die definierte Sprache direkt nutzen zu können.
Im Vergleich zu EMF wird Xtext noch aktiv weiterentwickelt und erlaubt einen schnelleren Einstieg in die Entwicklung eigener DSLs aufgrund einer leicht verständlichen Eingabe. In Xtext sind bereits Grammatiken enthalten, deren Inanspruchnahme eine, in der Praxis erprobten, Grundlage bereitstellt. Ein weiterer Vorteil ist, dass die erzeugten Modelle als Textdatei vorliegen und somit das Öffnen und Bearbeiten in einem beliebigen Texteditor möglich ist. Ebenso ist dadurch die Versionsverwaltung und ein textueller Vergleich von Modellen ohne einer speziellen Aufbereitung realisierbar.
Zur Umsetzung des, in Hauptkapitel 3 definierten, Metamodells (vgl. Kapitel 3.3) wird aus diesen Gründen und aufgrund der Vorgabe von Eclipse als IDE (vgl. Unterkapitel 2.1.3, Seite 11) das Xtext-Framework verwendet.

Als Standard in Xtext wird zur Generierung aus Modellen, die in den selbst definierten Metamodellen erstellt wurden, Xtend[110] eingesetzt. Xtend ist ein flexibler und aussagekräftiger Dialekt von Java, welcher zu lesbarem Java 5.0 kompatiblen Quelltext kompiliert.[111] Lesbar bezieht sich in diesem Fall auf den Java-Compiler ebenso wie für einen Java-Entwickler. Xtend unterstützt zudem weitere nützliche Funktionalitäten, wie die an-

[107] vgl. [Foua], Überschrift: "Eclipse Modeling Framework (EMF)"
[108] http://www.eclipse.org/Xtext/
[109] vgl. [Fouc], Überschrift: "What is Xtext?"
[110] https://www.eclipse.org/xtend/
[111] vgl. [Foub], Überschrift: "Java 10, today!"

onymen Funktionen (sogenannte Lambdas)[112] und die Erweiterungsmethoden[113].[114]

Der Generierungsvorgang von Xtext verwendet die in Xtend implementierten Generatoren. Diese beinhalten, abgesehen von der erforderlichen Logik, Bereiche mit Vorlagen für die zu erstellenden Dateien. Nachfolgend werden diese Vorlagen als Templates bezeichnet, um einer Verwechslung mit den Vorlagen für die Dokumente aus dem *Prozessmodell IT-Service* vorzubeugen. In einem solchen Template ist neben der Verwendung der Metamodellelemente der entsprechende generische Quelltext aus der Referenzimplementierung, der sowohl vor, als auch nach der Generierung eine praxistaugliche Einrückung besitzt, anzugeben.

Die beschriebene technische Umsetzung wird in Abbildung 4.1 grafisch veranschaulicht. Das Metamodell ist mit Hilfe von Xtext definiert. Das Modell kann in Textform erstellt werden und dient als Eingabe für den Generator, der durch das Xtext-Framework bereitgestellt und in Java implementiert ist. Zur Transformation der Modellelemente verwendet ein Generator die in Xtend verfassten Templates. Erzeugt wird daraus ein in Java implementierter Prototyp und gegebenenfalls die Dokumente, die entweder in Textform oder Grafiken ausgegeben werden.

[112] Beispielsweise ist damit die Implementierung einer komplexen Filterung in wenigen Ausdrücken realisierbar. Für weitere Informationen siehe https://eclipse.org/xtend/documentation/203_xtend_expressions.html#lambdas.

[113] Eine Ergänzung von eigenen Methoden an vorhandene Typen. Ein Aufruf diese Methoden ist analog zu den bereits enthaltenen möglich.

[114] Für eine nachvollziehbare und für Java-Entwickler geeignete technische (Umsetzungs-) Beschreibung der Vorteile von Xtend sei an dieser Stelle auf [Ben12] verwiesen.

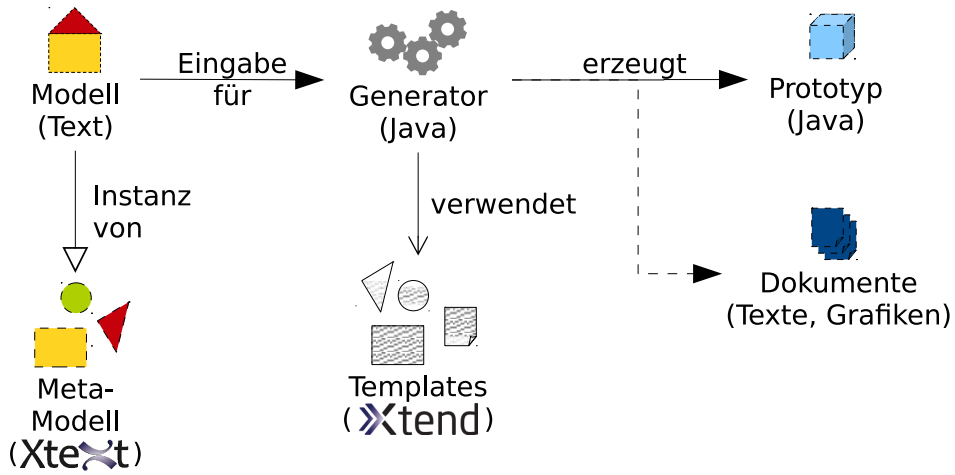


Abbildung 4.1: Darstellung der technischen Umsetzung der Modellierung. Die Pfeile stellen die Ablaufrichtungen und Zusammenhänge dar. In den Klammern steht die Umsetzung des jeweiligen Elements. Zur Wiedererkennung sind die Logos von Xtext und Xtend eingebunden. Der gestichelte Pfeil signalisiert, dass die Generierung der Dokumente innerhalb dieser Ausarbeitung theoretisch betrachtet werden.

Zum Zeitpunkt der Ausarbeitung waren folgende Versionen[115] aktuell und wurden daher verwendet:

- Eclipse: Mars
- Xtext: 2.8.4
- Xtend: 2.1.0

Verwendung der DSL

Im Zuge dieser Ausarbeitung soll zur Modellierung der bereits durch Eclipse bereitgestellten Editor genutzt werden. Dieser bietet anhand der definierten Sprache unter anderem folgende Funktionen, welche einem BRE während der Modellierung helfen:[116]

- Syntax-Highlighting
- inhaltsbasierte Unterstützung und Vervollständigung
- Validierung und Lösungsvorschläge

Über die IDE ist zudem die Erzeugung eines grafischen Syntaxbaumes ausführbar. Im Normalfall wird dieser für das Modell erzeugt. Aus der Sicht von Xtext ist das eingegebene Metamodell ein Modell (vgl. Abbildung 4.5).

[115] Als Komplettpaket zum Download unter http://www.eclipse.org/Xtext/download.html.
[116] vgl. [Fouc], Überschrift: "A Selection Of Supported Features"

Nachfolgend wird auf die Entwicklung und den Test eingegangen. Dabei erfolgt die Umsetzung des in Kapitel 3.3 definierten Metamodells und die Entwicklung einer konkreten Syntax. Im Anschluss daran dient ein Praxisbeispiel als *Proof of Concept*[117] des modellgetriebenen Ansatzes.

Der erstellten DSL wurde im Zuge der Ausarbeitung der Name *Barrakuda* gegeben.

4.1 Entwicklung

Die Grundlage besteht aus der Erstellung eines Xtext-Projekts. Diese kann über einen bereitgestellten Assistenten in Eclipse durchgeführt werden. In Abbildung 4.2 erfolgt die Darstellung der grundlegenden Angaben.

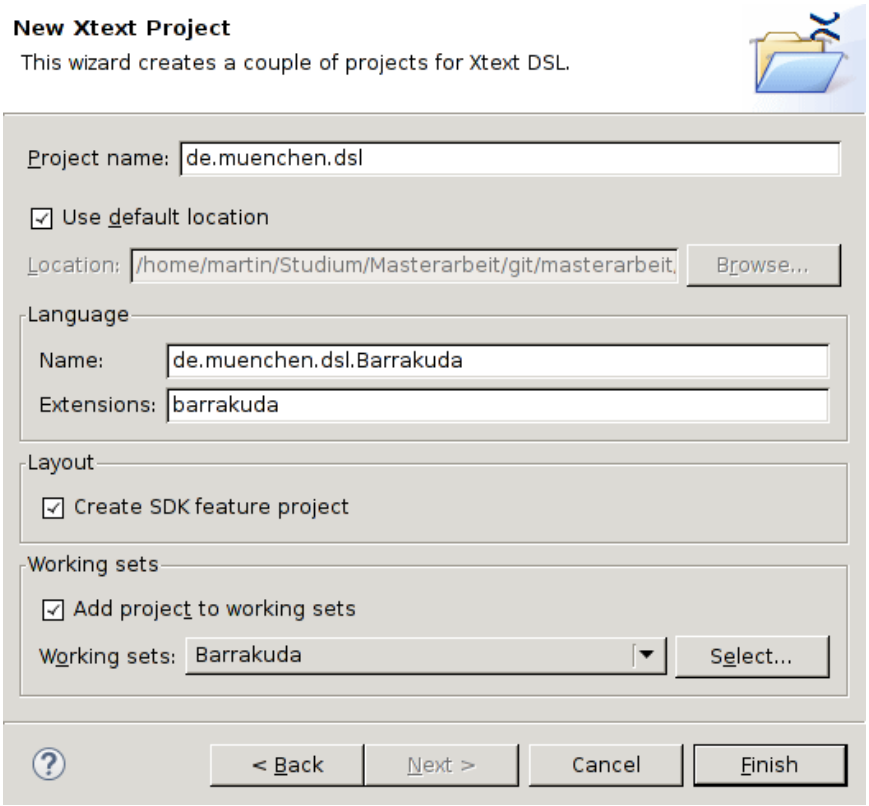


Abbildung 4.2: Screenshot des Assistenten zum Anlegen des Xtext-Projekts in Eclipse. Die Sprache besitzt den Namen *Barrakuda*, ist in einem entsprechenden Package angesiedelt und wird für Dateien mit der Endung *.barrakuda* angewendet.

[117] deutsch: Machbarkeitsnachweis

Der Assistent erstellt daraufhin vier Xtext-Projekte:

- Das Hauptprojekt:
  de.muenchen.dsl.barrakuda
- Das Projekt zum Testen von Barrakuda; u.a. mit dem JUnit-Framework[118]:
  de.muenchen.dsl.barrakuda.tests
- Das Projekt zur Anpassung der angebotenen Eclipse IDE:
  de.muenchen.dsl.barrakuda.sdk
- Das Projekt zur Anpassung der bereitgestellten Eclipse Oberfläche bzw. des Editors:
  de.muenchen.dsl.barrakuda.ui

In den folgenden Unterkapiteln wird auf das Hauptprojekt und das Test-Projekt eingegangen. Zur Umsetzung der DSL war keine Anpassung der zur Verfügung gestellten Eclipse IDE mit Oberfläche und Editor notwendig. Hinsichtlich der Verwendung der Sprache erfolgt eine Vorstellung von möglichen (Modellierungs-) Verbesserungen in Kapitel 5.2.

Zur eigenen DSL-Entwicklung mit Xtext existieren sehr viele Anleitungen.[119] In Folge dessen werden primär die Besonderheiten von Barrakuda thematisiert und auf die entsprechende weiterführende Literatur verwiesen.

4.1.1 Das Hauptprojekt

Das Hauptprojekt von Barrakuda ist durch das Xtext-Framework standardmäßig aufgeteilt in folgende Packages[120] (vgl. Abbildung 4.3):

de.muenchen.dsl.barrakuda beinhaltet die Grammatikdatei (Barrakuda.xtext), die Workflow-Datei[121] (GenerateBarrakuda.mwe2) und zwei Dateien zur Anpassung der bereitgestellten Eclipse IDE.

de.muenchen.dsl.barrakuda.formatting besteht aus einem Formatter[122], der die gewünschte Formatierung des Modells im Editor automatisch durchführt.

[118] http://www.junit.org

[119] Empfehlenswert (durch den Autor dieser Ausarbeitung) sind unter anderem die Dokumentation von [Fouc] und [Foub] (frei verfügbar), das Buch von Bettini [Bet13], ein Paper von Efftinge et al. [EV06] (frei verfügbar) und ein Artikel von Eysholdt [Eys11]. Die Autoren sind Entwickler und Unterstützer der genanten Frameworks, referieren darüber auf Konferenzen und bieten eine entsprechende Dienstleistung zur Unterstützung an.

[120] In Java die Technik zur Anordnung zusammengehöriger Klassen.

[121] In Bezug auf das Xtext-Framework eine Steuerungsdatei, die die Generierung der Sprache aus der Grammatikdatei mit den nötigen Schritten definiert.

[122] In Bezug auf das Xtext-Framework die Komponenten zur Formatierung der Ausgabe.

de.muenchen.dsl.barrakuda.generator enthält den noch leeren Generator, der die Elemente des Metamodells anhand von Templates transformieren kann.

de.muenchen.dsl.barrakuda.scoping umfasst einen ScopeProvider, der die Gültigkeitsbereiche von Metamodellelementen, das sogenannte Scoping[123], festlegt.

de.muenchen.dsl.barrakuda.validation enthält einen Validator[124], der um eigene Regeln erweitern werden kann.[125]

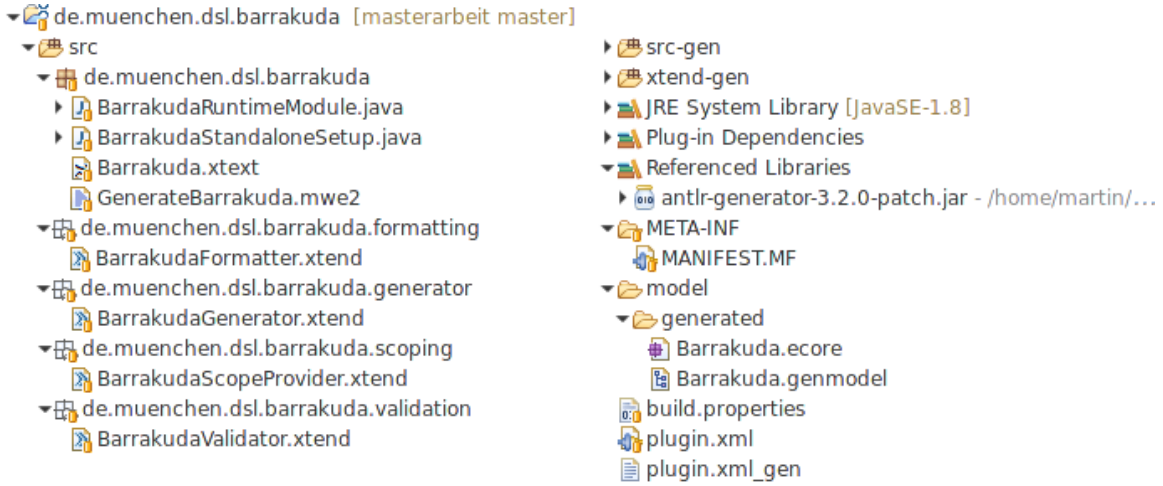


Abbildung 4.3: Zweispaltige Darstellung der Projektstruktur des Hauptprojektes aus der Entwicklungsumgebung nachdem der Workflow ausgeführt wurde. In der linken Spalte (dem src-Ordner) findet die Entwicklung und Anpassung der Sprache statt. Die daraus generierten Hilfsklasse und verwendeten Bibliotheken[126] sind in der rechten Spalte dargestellt.

Im Zuge der Ausarbeitung war es nicht erforderlich die bereitgestellte Klasse für den Formatter, das Scoping und den Validator anzupassen (*Convention Over Configuration*). Auf diese wird infolgedessen nicht im Detail eingegangen. In Abbildung 4.4 sind die Zusammenhänge zwischen den, aus Abbildung 4.3 relevanten, Dateien grafisch veranschaulicht.

[123] Im Kontext des Xtext-Frameworks die Komponente, die die Bereiche festlegt, in der Modellelemente (Variablen und Methoden) gültig sind.

[124] In Zusammenhang mit dem Xtext-Framework eine Komponente, um zusätzliche Constraints für die Sprache festlegen zu können.

[125] Damit bietet Xtext die Möglichkeit, weitere Constraints zur Grammatik hinzuzufügen. Sinnvoll ist das für, durch das Metamodell, nicht überprüfbare oder optionale Angaben. Ein Validator kann diese evaluieren und entsprechende Warnungen oder Fehler ausgeben. Zum Beispiel kann eine Meldung erfolgen, wenn ein fakultatives Element nicht angegeben ist, es aber sinnvoll wäre.

[126] Zur Erstellung der Xtext-Artefakte wird der ANTLR 3 Parser verwendet: "[...]*It is recommended to use the ANTLR 3 parser generator*[...]" (Ausgabe in der Konsole der IDE). Aufgrund anderen Lizenzbedingungen ist dieser nicht in der IDE enthalten, wodurch bei jeder Generierung ein Download erforderlich wäre. Durch einen manuellen Download der benötigten *antlr-generator-3.2.0-patch.jar* und deren Einbindung als externe Library/jar in den *build path* kann diese Abhängigkeit aufgelöst bzw. der Aufwand verringert werden. Entsprechend wurde es im Zuge dieser Ausarbeitung umgesetzt.

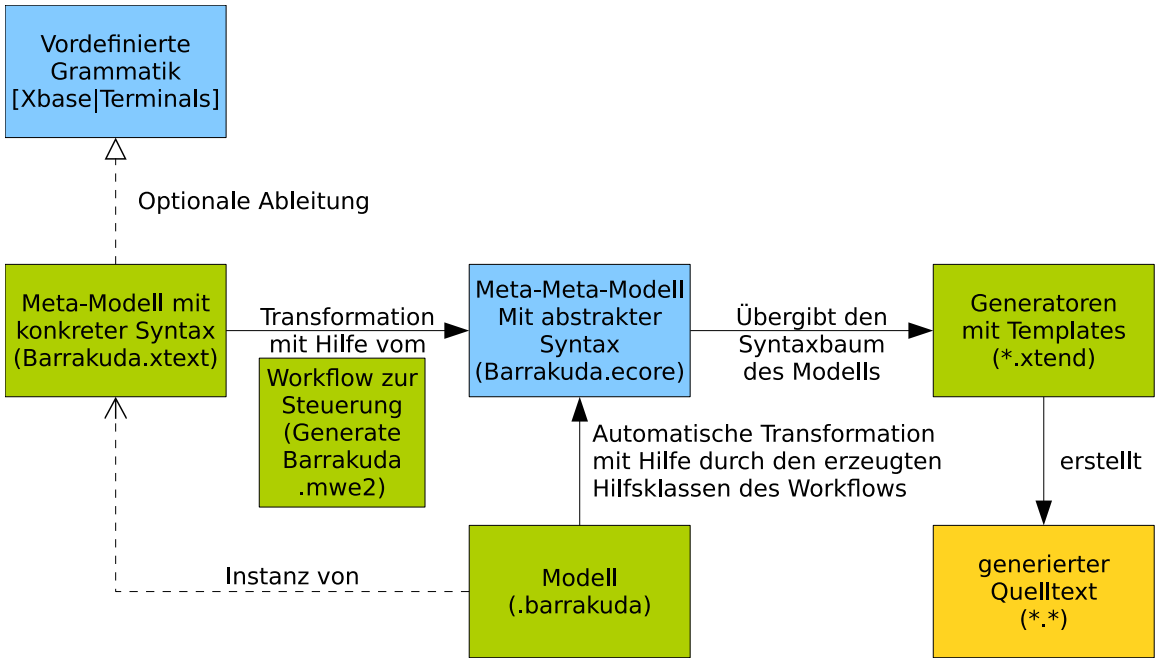


Abbildung 4.4: Darstellung des Zusammenhangs der in dieser Ausarbeitung angepassten Dateien (mit grünem Hintergrund), der durch die Frameworks bereitgestellten, sowie intern verarbeiteten Daten (in blauer Farbe hinterlegt) und der in gelb gestalteten Ausgabe.

4.1.2 Die Sprache

Die Grammatikdatei setzt das in Kapitel 3.3 definierte Metamodell im Sinne eines Regelsets in einer konkreten Syntax um. Xtext unterstützt dabei das Erben von einer anderen Grammatik. Die Auswahl des Vorgehens, ob eine eigene definiert oder eine bereits vorhandene verwendet wird, ist die Basis für die DSL. Daher erfolgt eine Betrachtung der drei grundlegend unterschiedlichen Möglichkeiten.[127]

Eigene Grammatik Eine eigene Grammatik ist frei definierbar. Beispielsweise ist die Beschreibung einer aus ausschließlich drei Zeichen bestehender Sprache möglich.
Der Vorteil ist die umsetzbare Spezialisierung der Sprache auf die benötigten Anforderungen. Im Gegenzug erfordert dies mehr Einarbeitungszeit und Wissen zur Verwendung, da sich diese an keine Standards halten muss.

XBase beinhaltet bereits einen großen Umfang an vordefinierten Elementen. Unter anderem können zusätzlich Bedingungen und arithmetische Operationen definiert werden. Angelehnt ist die Verwendung dieser Elemente an Java, sodass ebenso ein direkter Import vorhandener Klassen realisierbar ist, die anschließend zur Verwendung

[127] Zum Zeitpunkt der Ausarbeitung standen acht vordefinierte Grammatiken, die sich im Wesentlichen auf einer der zwei Grundlagen beziehen, zur Verfügung. Eine dieser acht Grammatiken ist Xtext selbst, wodurch dessen Grundlage, das EMF, nachvollzogen werden kann.

verfügbar sind.

Ein Vorteil von XBase ist, dass im Modell direkt Logik und damit Verhalten abbildbar ist. Ermöglicht wird dies durch die Formulierung von Ausdrücken. Ebenso ist die Ableitung oder Verwendung von bestehenden Klassen bei gleichzeitiger Validierung im Java-Umfeld realisierbar.

Zur Generierung von standardmäßig benötigten Elementen in den zu erstellenden Klassen, wie zum Beispiel die Setter- und Getter-Methoden, bietet XBase bereits vordefinierte Methoden an.

Von Nachteil ist die starke Bindung an Java, welches entsprechendes Wissen zur Verwendung im Modellierungsprozess erfordert. Ein weiterer Nachteil entsteht durch die Fähigkeit der Modellierung von Ausdrücke und Operationen, die nicht in ihrer Schachtelung begrenzt sind und dadurch die Komplexität erhöhen, wenn in den Templates daraus ausführbarer Quelltext zu erzeugen ist. Es müssen Konstellationen bedacht und unter Umständen ausgeschlossen werden, die zu Laufzeitproblemen führen können. Zum Beispiel die Angabe einer endlosen Rekursion.

Terminal-Grammatik Diese, auch nur als Terminals[128] bezeichnet, beinhaltet eine Basis mit wenigen grundlegenden Regeln, die für viele Sprachen ausreichend ist. Enthalten ist beispielsweise ein Identifikator (darf nicht mit einer Zahl beginnen), eine positive ganze Zahl und ein- sowie mehrzeilige Kommentare.

Von Vorteil sind dabei die leichtgewichtige Basis und ein Rahmen aus dem einfache, übersichtliche Modelle erstellt werden können. Die Kehrseite ist ein einmaliger Mehraufwand, wenn Mengen, Listen oder Ausdrücke erforderlich sind.

In Anbetracht des definierten Metamodells (vgl. Abbildung 3.14, Seite 54) wurde für Barrakuda als Basis die Terminal-Grammatik gewählt.
Im Vergleich zur komplett eigenen Grammatik bleibt dadurch ein Wiedererkennungswert, zum Beispiel durch die Möglichkeit von Kommentaren im Modell, erhalten.
Im Vergleich zu XBase ist die Terminal-Grammatik zur Umsetzung des definierten Metamodells aus zwei Gründen besser geeignet. Einerseits kann der Funktionsumfang von XBase nicht ausgenutzt werden, wodurch ein Mehraufwand zur Verhinderung ungewollter Konstellationen entsteht. Andererseits stellte sich bei Tests heraus, dass auf der Terminal-Grammatik basierende Modelle weniger Leistung und Zeit zur Generierung (vgl. Abbildung 3.1, *Generierungsprozess anhand DSL*) in Anspruch nehmen.

[128] Der Begriff Terminals ist in Zusammenhang mit dem Xtext-Framework eine Bezeichnung für die Grammatik. Im Gegensatz zum klassischen Compilerbau besteht eine Grammatik aus zwei Typen von Zeichen, die Terminals und die Nonterminals. Die Terminals sind darin die Grundsymbole der Sprache.

In Listing 4.1 ist die komplette Terminal-Grammatik enthalten, die als Grundlage für Barrakuda verwendet wird.

```
/*******************************************************************************
* Copyright (c) 2008 itemis AG and others.
* All rights reserved. This program and the accompanying materials
* are made available under the terms of the Eclipse Public License v1.0
* which accompanies this distribution, and is available at
* http://www.eclipse.org/legal/epl-v10.html
*******************************************************************************/
grammar org.eclipse.xtext.common.Terminals hidden(WS, ML_COMMENT, SL_COMMENT)

import "http://www.eclipse.org/emf/2002/Ecore" as ecore

terminal ID         : '^'?('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;
terminal INT returns ecore::EInt: ('0'..'9')+;
terminal STRING :
'"' ( '\\' . /* 'b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\' */ | !('\\'|'"') )* '"' |
"'" ( '\\' . /* 'b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\' */ | !('\\'|"'") )* "'"
;
terminal ML_COMMENT : '/*' -> '*/';
terminal SL_COMMENT     : '//' !('\n'|'\r')* ('\r'? '\n')?;

terminal WS         : (' '|'\t'|'\r'|'\n')+;

terminal ANY_OTHER: .;
```

Listing 4.1: Listing der Terminal-Grammatik, die durch das Xtext-Framework bereitgestellt wird. Diese greift ihrerseits auf die Ecore-Grundlage aus dem EMF zurück.

Durch die Nutzung der Terminal-Grammatik war eine Optimierung des Workflows (vgl. Abbildung 4.4) zur Erstellung der internen Repräsentation der Sprache möglich. Diese findet im *Generierungsprozess anhand DSL* (vgl. Abbildung 3.1) statt.

In Kapitel 3.3 wurden zwei Bedingungen festgelegt: die Eindeutigkeit der Namen und die vorherige Definition von referenzierten Elementen. Die Eindeutigkeit wurde durch die Erweiterung des Workflows erreicht. Der bereits enthaltene, im Standard deaktivierte, *NamesAreUniqueValidator* stellt die Eindeutigkeit über das komplette Modell sicher.
Die Bedingung der vorherigen Definition von referenzierten Elementen ist standardmäßig enthalten. Von großem Vorteil ist hierbei, dass ebenfalls eine Referenzierung auf Elemente aus anderen Modellen realisierbar ist. Das vordefinierte Scoping stellt die Elemente aus anderen Modellen automatisch zur Verfügung. Damit ist die Verbindung von mehreren Microservices über die Modelle möglich. In Abbildung 2.7 auf Seite 14 ist diese Konstellation dargestellt.

Das festgelegte Metamodell (vgl. Abbildung 3.14, Seite 54) wurde als Grundlage zur Definition des Regelsets der Sprache verwendet. Die konkrete Syntax bzw. die Keywords von Barrakuda sind dabei aus folgenden Gründen in Englisch angeben:

- Es handelt sich um den Standard in der Programmierung.
- Die Bildung der Mehrzahl ist regulierter und einfacher im Vergleich zu deutschen Ausdrücken.
- Die Begriffe besitzen keine Umlaute.
- Erfolgt die Modellierung auf deutsch, ist eine Unterscheidung zwischen den Keywords und den fachlichen Elementen auch ohne Syntax-Hervorhebung möglich.

Die konkrete Syntax ist angelehnt an bestehende Programmiersprachen. Eine Definition, wie zum Beispiel einer Entität mit Attributen befindet sich in geschweiften Klammern. Der Abschluss einzeiliger Regeln erfolgt durch ein Semikolon.

Das Listing 4.2 beinhaltet die ersten Regeln von Barrakuda. Der Ausgangspunkt ist die Regel *Domainmodel*. Im Detail besteht dieses aus einem Package, dem Namen, der Version (Zeile 2) und mindestens einem TopElement (Zeile 4). Ein TopElement ist wiederum entweder ein DataType, eine Entity oder eine BusinessAction (Zeile 12).

```
1  Domainmodel:
2      'model' package = QualifiedName 'artifact' name = ID 'version' version = Version
3      '{'
4          topElements += TopElement+
5      '}'
6  ;
7
8  QualifiedName: ID ('.' ID)*;
9
10 Version: INT ('.' INT)*;
11
12 TopElement: (DataType | Entity | BusinessAction);
13
14 Entity: 'entity' name = ID
15     (external ?= 'external')?
16     ('extends' superType = [Entity | QualifiedName])?
17     '{'
18         attributes += Attribute+
19     '}'
20 ;
21
22 Attribute: name = ID
23     (mapping = Mapping)?
24     type = [ElementType | QualifiedName]
25     (optional ?= 'optional')?
26     (mainFeature ?= 'mainFeature')?
27     (example = STRING)?
28     ';'
29 ;
```

Listing 4.2: Auszug aus der Grammatikdatei mit den ersten Regeln von Barrakuda. In einfachen Anführungszeichen angegeben sind feststehende Begriffe (Keywords), die im Modell enthalten sein müssen. Eine Variable wird durch = belegt (z.B. Zeile 14, der Name der Entität). In Zeile 4 erfolgt mit Hilfe von += die Erstellung einer Liste. Diese muss dabei mindestens ein Element, dazu das + am Ende der Zeile, enthalten. Anhand von ?= werden optionale Elemente definiert, die wiederum innerhalb von ( )? im Modell auch komplett entfallen können (Zeile 15). Die Angaben zwischen [ ] dienen der internen Weitergabe von referenzierten Elementen (Zeile 16). Die Variable *name* ist ein erforderliches Feld von Xtext mit welchem in den Generatoren die Elemente angesprochen werden können.

In Abbildung 4.5 ist der, auf Basis von Listing 4.2 erstellte, Syntaxbaum des Metamodells wiedergegeben.[129] Die Bedingung aus Zeile 4 "mindestens ein TopElement muss angegeben werden", demzufolge eine Liste mit minimal einem Element, ist jedoch in der Darstellung nicht unmittelbar erkennbar.

[129] Im Normalfall wird der Syntaxbaum für das Modell erzeugt. Aus der Sicht von Xtext ist das Metamodell (Barrakuda) ein Modell. Folglich handelt es sich in dieser Konstellation bei Xtext um das Metametamodell.

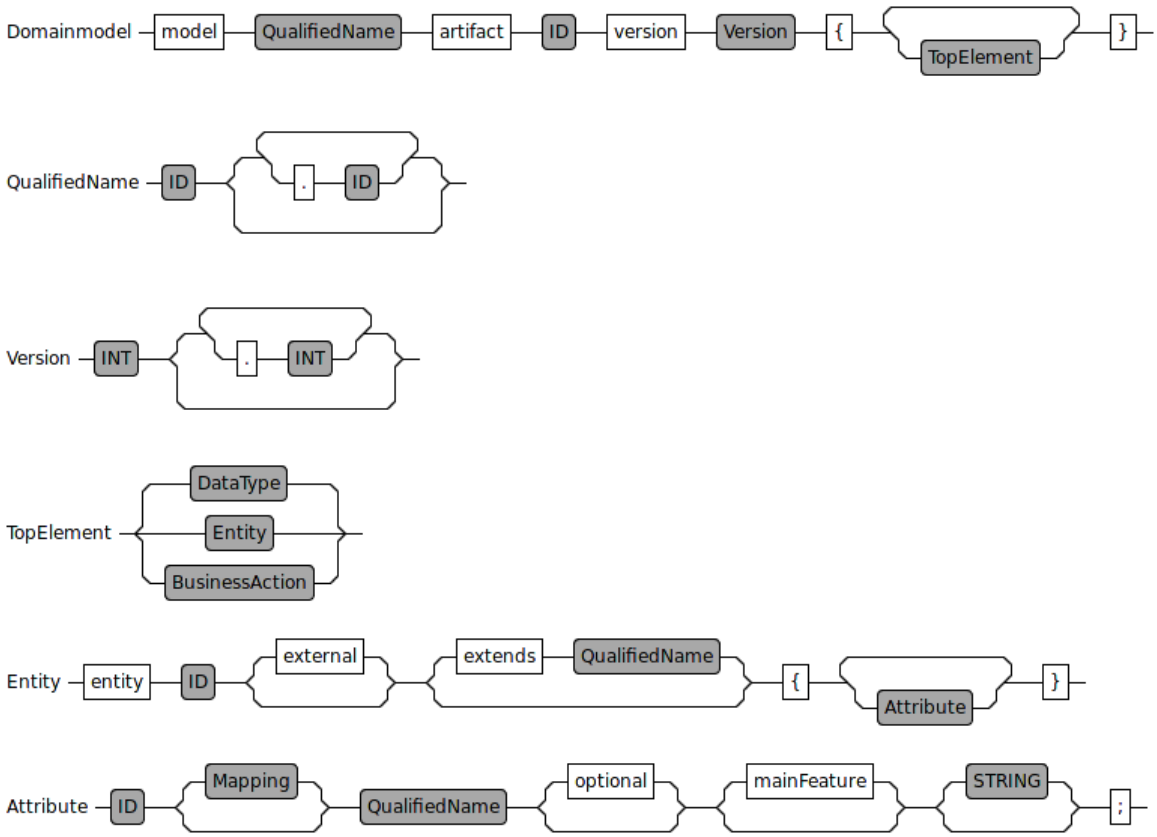


Abbildung 4.5: Darstellung eines Ausschnitts des, in Eclipse generierten, Syntaxbaums des Metamodells von Barrakuda. Jede Regel beginnt mit einem Namen. In weiß hinterlegt sind feststehende Zeichen. Verweise auf andere Regeln sind grau hinterlegt.

Treten Fehler mit den erstellten Regeln in Xtext, wie *multiple alternatives* bzw. *ambiguities*[130] auf, kann einerseits in der Eclipse IDE direkt die Fehlersuche durchgeführt oder ein Werkzeug des auslösenden Frameworks *ANTLRWorks*[131] verwendet werden. Unter anderem ist damit eine Visualisierung des Syntaxbaums einschließlich der auftretenden Mehrdeutigkeit möglich.[132]

Die, bis zu diesem Schritt durchgeführten, Anpassungen sind ausreichend, damit eine Modellierung anhand der Sprache erfolgen kann. Das Xtext-Framework beinhaltet die im klassischem Compilerbau erforderlichen Elemente, wie Interpreter, Parser, usw., welche daher nicht manuell zu entwickeln sind.

[130] deutsch: Mehrdeutigkeiten
[131] http://www.antlr3.org/works/
[132] vgl. [Sto11], Überschrift "Xtext Grammar Visualization"

4.1.3 Die Generatoren

Zur Umwandlung bzw. Transformation des Modells in den Prototyp bzw. die Dokumente (vgl. Abbildung 4.4, *Generatoren mit Templates*) dienen die Templates, die sich innerhalb der Xtend-Generatoren befinden. Ausgeführt werden die Generatoren im *Generierungsprozess anhand Modell* (vgl. Abbildung 3.1, Seite 40).
Eine der ersten Anpassungen ist die Aufteilung in wiederholt und einmalig zu erstellende Dateien. Im Anschluss daran erfolgt die Betrachtung der Quelltext- und Dokumentgenerierung.

Der standardmäßig erstellte Generator (siehe Unterkapitel 4.1.2) kann zur Generierung nicht unterscheiden zwischen den zu überschreibenden und einmalig zu erstellenden Dateien (anpassbar durch den Entwickler). Dadurch wird das Zielverzeichnis vor jedem Generierungsvorgang geleert.
Über eine eigene Ausgabe-Konfiguration[133] kann einerseits dieses Verhalten angepasst oder andererseits die Dateien in unterschiedlichen Ordnern abgelegt werden. Bewährt und oftmals in IDEs, sowie Build-Management-Tools umgesetzt, ist die Aufteilung in die Ordner *src* und *src-gen*. Standardmäßig erfolgt die Generierung, der in den Templates angegebenen Dateien, in den Ordner *src-gen*. Die hinzugefügte Ausgabe-Konfiguration "GEN_ONCE" kann dem Zugriffsobjekt, bei einmalig zu generierende Dateien, übergeben werden.

Die Aufteilung in mehrere **Generatoren für den Quelltext** erfolgte anhand der Hierarchie der Projekte (Micorservice, Client-Bibliothek und GUI Microservice). Mehrfach benötigte Methoden konnten mit Hilfe von Erweiterungsmethoden direkt an die benötigten Metamodellelemente angeknüpft werden. Damit sind diese von allen Generatoren verwendbar.

Nachfolgend ist am Beispiel der Erstellung der DTO-Klassen für die Client-Bibliothek der Aufbau der entwickelten Generatoren mit ihren Templates beschrieben. Einstiegspunkt ist der in Listing 4.3 dargestellte Ausschnitt des Generators. Zur Erstellung der DTOs werden innerhalb einer Schleife (Zeilen 4-6) die Entitäten des Modells (Filter in Zeile 4) abgearbeitet. Für jede Entität erfolgt die Erstellung einer Java-Klasse über das Zugriffsobjekt auf das Dateisystem, dem *IFileSystemAccess* (Zeile 5, blaue Schriftfarbe). Als Dateiinhalt dient die Rückgabe der Methode *makeEntity*. Generell handelt es sich um alle Zeichen innerhalb von drei einfachen Anführungszeichen um als Text interpretierte Inhalte. Sind innerhalb dessen französische Anführungszeichen wird deren Inhalt interpretiert und das Ergebnis eingefügt.

[133] Für eine detaillierte Anleitung der Implementierung sei an dieser Stelle auf die Quelle [Bro12] verwiesen.

```
1 def compile(Domainmodel domainmodel, IFileSystemAccess fsa) {
2     val basePath = TARGET_DIR + domainmodel.nameInLowerCase + '-client-lib/' +
          SRC_MAIN_JAVA + domainmodel.FQDN.replace('.', '/') + '/'
3
4     domainmodel.topElements.filter(typeof(Entity)).forEach [
5         fsa.generateFile('''«basePath»«name.toFirstUpper».java''', makeEntity(it,
              domainmodel.FQDN));
6     ]
7 }
```

Listing 4.3: Ausschnitt aus dem Generator zur Erzeugung der Client-Bibliothek. Die *compile*-Methode wird von Xtext aufgerufen und übergibt das Model und ein Zugriffsobjekt auf das Dateisystem.

Die, in Listing 4.3 aufgerufene, *makeEntity*-Methode besteht aus dem in Listing 4.4 aufgeführten Quelltext. Zum Großteil ist darin das Template für das Grundgerüst einer Entität enthalten (Zeilen 4-16). Dieses beinhaltet Absätze und Einrückungen, die in die Ausgabedatei übernommen werden und die Lesbarkeit des Generats erhalten. Hervorzuheben ist dabei die Erstellung der Attribute, welche über eine manuell implementierte Erweiterungsmethode (eingebunden durch Zeile 1) der automatisiert erstellten Attribut-Klasse aus dem Metamodell erfolgt (Zeile 13).

```
1  @Inject extension AttributeUtils
2
3  def makeEntity(Entity entity, String fqdn) '''
4      package «fqdn»;
5
6      import java.io.Serializable;
7      import java.util.Date;
8      import java.util.List;
9
10     class «entity.name.toFirstUpper»Dto extends AbstractDto {
11
12         «FOR attribute : entity.attributes»
13             «attribute.makeAttribute»
14         «ENDFOR»
15
16     }
17 '''
```

Listing 4.4: Auszug aus dem Template zur Erzeugung einer Entity. Der Injektor in Zeile 1 ermöglicht die Methodenaufrufe der Klasse *AttributeUtils* direkt auf den Attributen (Zeile 13).

Das Listing 4.5 beinhaltet einen Auszug der Erweiterungsklasse des Attribut-Metamodellelements. Der Vorteil von Xtend mit der dynamischen Typzuordnung ist darin erkennbar. Unabhängig vom Attributtyp wird in Zeile 5 die Methode *makeCompleteApi* aufgerufen. In den Zeilen 8-16 befindet sich als Beispiel das Template für den Text-Typ mit der

Definition der Variable, dem Setter und dem Getter.

```
1  // [...]
2  class AttributeUtils {
3
4      def makeAttribute(Attribute attribute) {
5          makeCompleteApi(attribute.name, attribute.type)
6      }
7
8      def dispatch makeCompleteApi(String name, Text text) '''
9          String «name.toFirstLower»;
10         public String get«name.toFirstUpper»() {
11             return «name.toFirstLower»;
12         }
13         public void set«name.toFirstUpper»(String «name.toFirstLower») {
14             this.«name.toFirstLower» = «name.toFirstLower»;
15         }
16     '''
17
18     // [...]
19 }
```

Listing 4.5: Auszug aus der Erweiterungsklasse für Attribute. Aufgrund der Übersichtlichkeit und des Umfangs wurde auf die Angabe der weiteren Typen, wie Zahl, Liste, Wahrheitswert und Datum verzichtet. Deren Templates sind ähnlich aufgebaut.

Die **Generatoren für die Dokumente** lassen sich anhand von zwei Vorgehensweisen entwickeln, die nachfolgend theoretisch betrachtet werden. In beiden gleichermaßen realisierbar ist die Erstellung der Abbildungen für die Architektur, das Datenmodell und den Anwendungsfällen mit einer entsprechenden Grafik-Bibliothek.

Einerseits ist die Erzeugung der Dokumente im Zieldateiformat realisierbar. Zum Zeitpunkt der Erstellung dieser Ausarbeitung wurden Vorlagen im OpenDocument-Format bereitgestellt. Um diese zu nutzen und zudem als Ausgabedateien zu erstellen, ist zum Beispiel die Verwendung der Bibliothek jOpenDocument[134] möglich. Eine Festlegung, wie bei einer erneuten Generierung umgegangen wird, wenn bereits eine manuelle Erweiterung stattgefunden hat, ist in diesem Fall erforderlich. Eine Lösung ist die Definition von geschützten Bereichen, die gelöscht und erneut generiert werden können. Der Generator muss dafür das bereits bearbeitete Dokument als Eingabe erhalten, dieses entsprechend einlesen und überarbeiten.

Andererseits ist die Ausgabe der generierbaren Abschnitte direkt als Textfragmente bzw. Grafiken umsetzbar. Damit bleibt die Kompatibilität zu den bereits bestehenden und zukünftigen Vorlagen gewährleistet. Ebenso wird die Versionsverwaltung mit einem direkten

[134] http://www.jopendocument.org/

Vergleich früherer Versionen realisierbar.[135] Jedoch muss die Überführung in die eigentlichen Dokumente manuell stattfinden, wodurch unter Umständen ein erhöhter Aufwand entsteht.

4.2 Test

Die Tests der Generatoren sollen sicherstellen, dass das Metamodell wie beabsichtigt eingelesen wird, die Templates den gewünschten ausführbaren Quelltext generieren und die wiederholte Generierung (nachdem Änderungen am Modell durchgeführt wurden) funktioniert.[136]

Die Definition von Tests ist im, standardmäßig von Xtext erstellten, *tests*-Projekt (vgl. Einstieg in Kapitel 4.1) realisierbar. Testbar ist zum einen die Verarbeitung der Modelle (Ein-/Ausgaben, Zwischenschritte, Validatoren, ...) und zum anderen der bereitgestellte Editor (Verhalten, Vervollständigungsvorschläge, ...). Realisiert sind diese Einstiegsmöglichkeiten im Xtext-Framework über sogenannte Injektoren. Für einen aktuellen und praxisnahen Vortrag zum Testen von Xtext Sprachen siehe [Bet15].

4.2.1 Testen des Metamodells

Das Metamodell muss frei von Syntaxfehlern sein und nach dem Einlesen den gewünschten internen Aufbau besitzen. Diese Tests dienen zur Überprüfung des *Generierungsprozesses anhand DSL* (vgl. Abbildung 3.1, Seite 40).

Mit Hilfe von sogenannten Parser-Tests[137] ist eine Überprüfung der zuvor beschriebenen Anforderungen anhand der erstellten internen Darstellung, dem Syntaxbaum, möglich. Die Tests sind, wie die Templates, mit Hilfe von Xtend verfasst und besitzen daher einen großen Funktionsumfang. In Listing 4.6 ist ein Auszug der implementierten Klasse *BarrakudaParserTest* dargestellt. Das Eingabemodell (Zeilen 2-8) wird geparst und anschließend angenommen, dass dabei keine Fehler auftreten (Zeile 18).[138]

[135] Oftmals handelt es sich bei Dateien von Textverarbeitungsdokumenten um Archive, die für einen Vergleich zu entpacken sind.

[136] Die Generierung von Tests, um den ebenfalls generierten Quelltext zu Testen, der von einem Entwickler nicht veränderbar ist, liefert keinen Mehrwert. Diese Tests wurden bereits in der Referenzimplementierung durchgeführt.

[137] Xtext fasst darin folgende Phasen zusammen: Lexing, Parsing, Linking, Validation (vgl. [Foud], Abschnitt: "Terminal Rules").

[138] Implementierte Validatoren sind ebenfalls über diesen Mechanismus testbar.

```
1  val smallModel = '''
2          model de.muenchen.test artifact Testing version 1.0 {
3              customTextType text;
4
5              entity Beispiel {
6                  Beschreibung text;
7              }
8          }
9  '''
10
11 // add methods to parse a modell
12 @Inject extension ParseHelper<Domainmodel>
13 // add methods to validate the parser
14 @Inject extension ValidationTestHelper
15
16 @Test
17 def void testSmallEntity() {
18     smallModel.parse.assertNoErrors
19 }
```

Listing 4.6: Auszug aus der Klasse zum Testen des Barrakuda-Parsers. Über die Injektoren kann das Modell direkt geparst und zugesichert werden, dass keine Fehler dabei auftreten (Zeile 18).

4.2.2 Testen der Templates

Die Templates sollen einen Prototypen anhand eines Modells erstellen, welcher wie die Referenzimplementierung aufgebaut ist. Diese Tests dienen zur Überprüfung des *Generierungsprozesses anhand Model* (vgl. Abbildung 3.1, Seite 40).

Xtend bietet mit dem *CompilationTestHelper* die Möglichkeit ein Modell einlesen und generieren zu lassen. Das Listing 4.7 zeigt einen Ausschnitt der Funktionsweise dieser Tests. Die Erweiterungsmethode (Zeile 7) parst das Modell, validiert und kompiliert es. Anschließend erfolgt ein Vergleich mit dem gewünschten Template. Die Erstellung dieser stellte sich, aufgrund der großen Anzahl an generierten Dateien, als Nachteil dieser Testmethode heraus. Zum einen kann die Ausgabe des Generators nur komplett überprüft werden. Soll ein Stichproblem-Vergleich, anhand einer zentralen Datei erfolgen, ist dieser manuell zu implementieren. Zum anderen ist das Vergleichs-Template zu erstellen, welches entsprechenden Aufwand erzeugt. In der Praxis eignet sich das Vorgehen, dass dazu die Ausgabe einer Generierung manuell überprüft und anschließend als Referenz eingebunden wird.

```
1 // add a temp folder to collect the generators output
2 @Rule @Inject public TemporaryFolder temporaryFolder
3 // add methods to compile the input model
4 @Inject extension CompilationTestHelper
5
6 @Test def void testGeneratedJavaCode() {
7     smallModel.assertCompilesTo('''MULTIPLE FILES WERE GENERATED[...]''')
8 }
```

Listing 4.7: Ausschnitt aus der Klasse zum Testen der Barrakuda-Templates. Eingabemodell ist das *smallModel* aus Listing 4.6. Das erwartete Ergebnis wurde aufgrund der Vielzahl an generierten Dateien in diesem Auszug abgeschnitten. Der Platzhalter [...] ist stellvertretend für jede generierte Datei mit ihrem Pfad und dem Inhalt angegeben.

Anstelle des in Listing 4.7 in Zeile 7 festgelegten Tests werden über die alternative *compile*-Erweiterungsmethode die generierten Java-Klassen zur Ausführung übersetzt und im temporären Verzeichnis vorgehalten. Mit Hilfe von Reflection[139] ist die Erstellung von Instanzen dieser und deren Verwendung möglich. Folglich sind Testungen des gewünschten Laufzeitverhaltens und des Zusammenspiels über mehrere Klassen hinweg realisierbar.

4.2.3 Testen der wiederholten Generierung

Die Tests der wiederholten Generierung sollen die Praxistauglichkeit sicherstellen. Nach der Modellierung (vgl. Abbildung 3.2, Seite 41) wird das erstellte Projekt durch einen Entwickler in seiner IDE eingebunden und weiterentwickelt.

Implementiert sind diese Tests identisch zu denen der Templates mit dem Unterschied, dass zweimal die Generierung stattfindet und dazwischen eine Änderung am Modell erfolgt.

4.3 Verifikation

Zur Verifikation des Modellierungsprozesses (vgl. Abbildung 3.3, Seite 42) und der Ein- sowie Ausgabe (vgl. Abbildung 4.1) fungiert das Praxisbeispiel zur Auswahl und Verwaltung von Schöffen, nachfolgend als *Schöffenverfahren* bezeichnet. Als Basis zur Modellierung diente das vorhandene Fachkonzept (FK).

[139] Eine Möglichkeit unbekannte Java-Klassen zu instantiieren, analysieren und die angebotenen Methoden aufzurufen.

4.3.1 Modellierung

Die Voraussetzung zur Modellierung ist eine Eclipse-Instanz, die mit eingebundener Barrakuda-Sprache als Plugin gestartet ist. In der Praxis wird dazu das Hauptprojekt (vgl. Kapitel 4.1) als Eclipse-Anwendung ausgeführt.[140] Während der Modellierung sind dementsprechend zwei Eclipse-Instanzen, eine mit der Definition der Sprache und eine mit deren Verwendung, aktiv.[141] Für den Modellierer ist in diesem Fall nur die zweite Instanz von Bedeutung.

Zu Beginn der Modellierung wurde in der bereitgestellten IDE ein eigenes Projekt erstellt. Durch das Anlegen der Datei *Schoeffen.barrakuda* innerhalb dessen erfolgte die automatische Erkennung der Sprachzugehörigkeit zu Barrakuda (vgl. Abbildung 4.6), welche den Editor entsprechend konfiguriert.

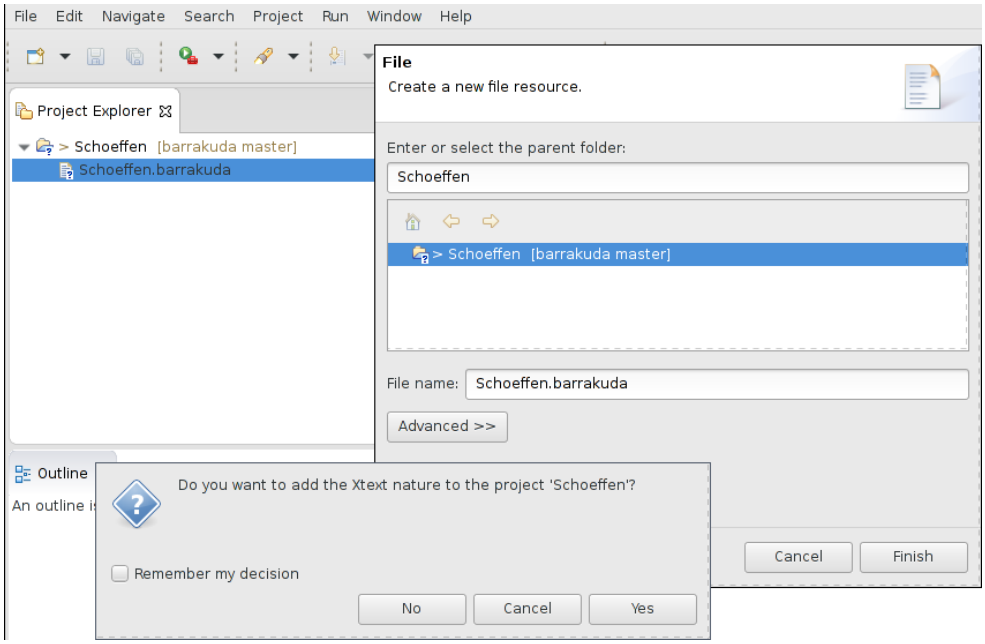


Abbildung 4.6: Screenshot der IDE im Projekt *Schoeffen* zum Zeitpunkt der automatischen Erkennung der Sprachzugehörigkeit zu Barrakuda.

Im Editor erfolgte anschließend die Erstellung des Modells. Der Modellierer wird dabei durch die automatische Vervollständigung (vgl. Abbildung 4.7) bei gleichzeitiger Validierung unterstützt.

[140] In der IDE über *Run As -> Eclipse Application*.

[141] Eine eigenständige Bereitstellung der Sprache als Eclipse-Plugin ist ebenso möglich, sodass nur eine Eclipse-Instanz benötigt wird (siehe https://eclipse.org/Xtext/documentation/350_continuous_integration.html).

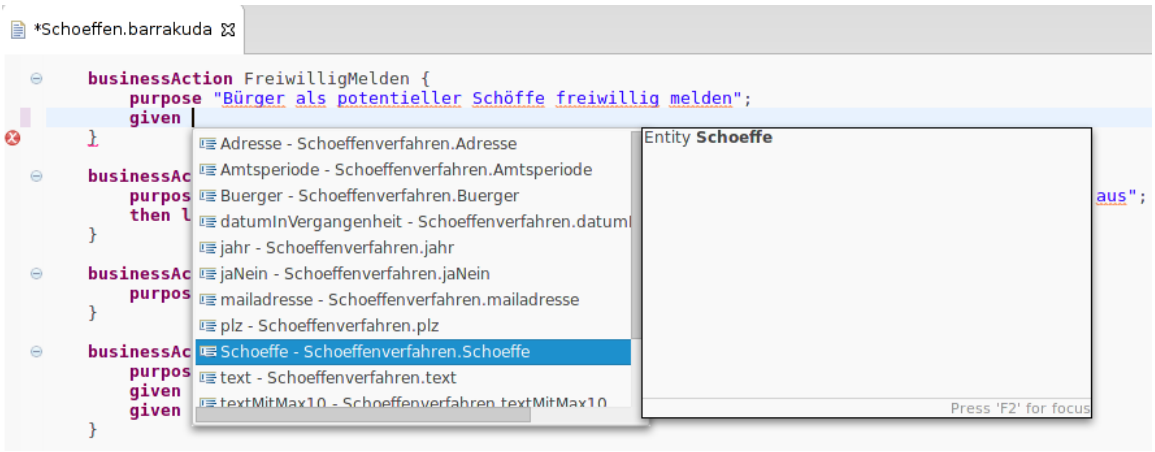


Abbildung 4.7: Screenshot des Editors mit Vorschlägen zur automatischen Vervollständigung. Die Entwicklungsumgebung erkennt die bereits angelegten Datentypen und bietet diese an. Der angegebene Fehler auf der linken Seite der ersten Geschäftsaktion (der rot hinterlegte Kreis mit dem weißen x) wird durch die sofortige Validierung angezeigt und weist auf die noch unvollständige Definition hin.

Anhand dem vorliegenden FK wurde das in Listing A.4 enthaltene Modell erstellt.

Sobald das Modell abgespeichert ist und keine Validierungsfehler vorliegen startet automatisch der Generator. Im Ausgabeordner sind anschließend die definierten Unterordner (vgl. Unterkapitel 4.1.3) *src* und *src-gen* mit den generierten Projekten enthalten. Diese können wiederum zur Implementierung der fachlichen Logik als eigenständige Projekte in Eclipse oder anderen IDEs eingebunden werden. Ein direkter Start derer ist ebenfalls möglich, was die Abbildung 4.8 veranschaulicht.

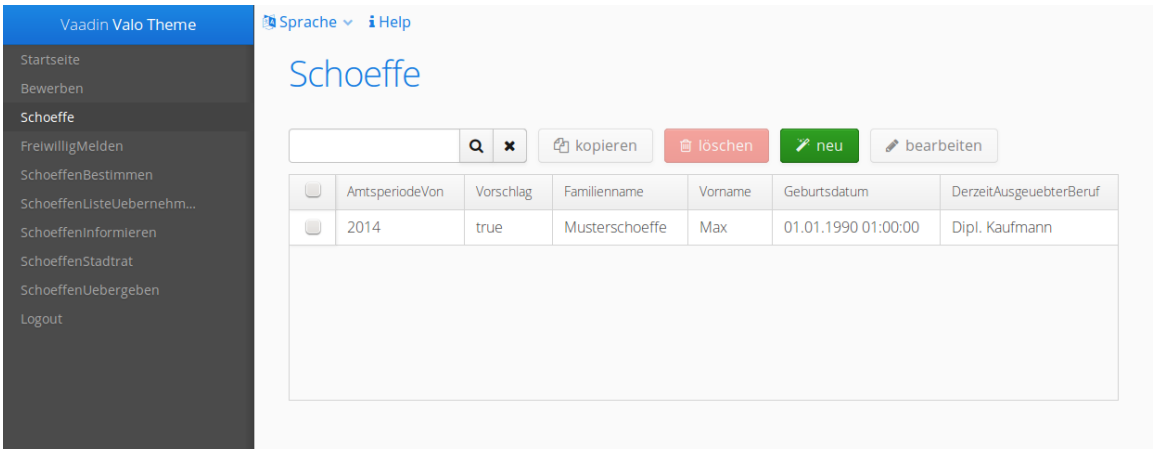


Abbildung 4.8: Screenshot des gestarteten Schöffenverfahrens, welches aus dem angefertigten Modell erstellt wurde. Die Anwendung, noch ohne fachlicher Logik, funktioniert. Die jeweiligen Bezeichnungen und Beschreibungen sind standardmäßig aus dem Modell. Eine Anpassung, sodass zum Beispiel anstelle von ”Schoeffe” die schreibweise mit Umlaut angezeigt wird, ist möglich.

4.3.2 Evaluation

Wie beschrieben, wurde das Schöffenverfahren anhand dem vorhandenen FK modelliert. Eine Beurteilung des geplanten Vorgehens (vgl. Abbildung 1.3, Seite 4), in welchem das FK aus dem Modell hervorgeht (vgl. Abbildung 4.1), kommt dadurch nicht in Betracht.

In Hinblick auf den Prozess zur Entwicklung konnten die Phasen Entwurf und Implementierung (vgl. Abbildung 3.2, Seite 41) anhand der Angaben im Fachkonzept zu einem Großteil über das erstellte Modell durchgeführt werden. In kurzer Zeit war es mit Hilfe der Angaben über das Datenmodell und der Anwendungsfälle im FK möglich, diese in das entsprechende Schema in Barrakuda zu überführen und Geschäftsaktionen zu erstellen. Während der Modellierung ist aufgefallen, dass trotz der strikten Definition der Sprache die Benutzerfreundlichkeit erhalten bleibt. Durch die Variationsmöglichkeit bezüglich der Reihenfolge der Modellelemente und nachträglicher Ergänzungen kann das Modell dynamisch wachsen. Es ist machbar, dass zu Beginn nur der kleiner Teil des Datenschemas und darauf aufbauend direkt die benötigte Geschäftsaktion angegeben wird.

Als praxistauglich hat sich die Unterstützung des Modellierenden durch die IDE herausgestellt. Mit Hilfe der automatischen Vervollständigung und der Vorschläge durch die Sprache (vgl. Abbildung 4.7) können in kurzer Zeit Modelle ohne Syntax- und Semantikfehler erstellt werden. Treten diese dennoch auf, ist die Erkennung und Behebung, durch die sofortige Validierung und den aussagekräftigen Meldungen, durchführbar.

Negativ aufgefallen ist der direkte Start des Generators, wenn das Modell frei von Feh-

lern ist. Wird das Modell zum ersten mal abgespeichert, kann der Generator zu diesem Zeitpunkt nicht unterscheiden, ob es aus der Sicht des Modellierers bereits vollständig ist oder nicht. Das führt dazu, dass die einmalig generierten Teile erstellt werden. Sollte das Modell zu diesem Zeitpunkt noch nicht komplett sein, überschreibt der Generator die einmalig zu generierenden Teile nicht wieder. Es ist daher erforderlich, dass der Zielordner gelöscht und die Generierung erneut angestoßen wird, sobald das Modell vollständig ist. Der Generierungsvorgang an sich ist unter Anbetracht der Menge und dem Umfang der zu erstellenden Dateien sehr schnell. Entsprechend ergibt sich dadurch kein Zeitverlust bzw. Mehraufwand bei einer erneuten Generierung.

Der Mehrgewinn durch kürzere Iterationen zwischen den Tests des Prototypen kann bestätigt werden. Ebenso wie die erleichterte Wartbarkeit. Mit dem Hintergrundwissen über die Referenzimplementierung und deren Architektur ist die Erweiterung des Prototypen während der Implementierungsphase (vgl. Abbildung 3.1, *Entwicklungsprozess*) ohne großen Einarbeitungsaufwand möglich. Durch eine zukünftig vermehrte Verwendung von Barrakuda in Projekten erlangen dieses Wissen die entsprechenden Entwickler und der Wiedererkennungswert steigt.

5 Fazit

In diesem abschließenden Hauptkapitel werden die Ausführungen resümiert und mit der einleitenden Aufgabenstellung verglichen. Anschließend gibt der Ausblick Auskunft über Weiterentwicklungsmöglichkeiten.

5.1 Zusammenfassung

In Hauptkapitel 2 wurde im Zuge der kritischen Betrachtung das Prozessmodell analysiert und die zu optimierenden Stellen hervorgehoben. Durch die modellgetriebene Entwicklung sollte eine Vereinfachung und Beschleunigung der Pasen Spezifikation, Entwurf und Implementierung stattfinden. Dazu wurde in Hauptkapitel 3 die Prozessintegration vorgestellt und die vorhandene Referenzimplementierung im Detail betrachtet. Aus Letzterer entstand die Definition eines angemessenen Metamodells (vgl. Kapitel 3.3). Auf dessen Mehrwert für die, im *Prozessmodell IT-Service* zu erstellenden, Dokumente eingegangen wurde.
Im anschließenden Hauptkapitel 4 erfolgte die technische Umsetzung des Metamodells auf Basis des Xtext-Frameworks. Die Durchführung von Tests des, in diesem Schritt entstandenen, Generators erfolgte bereits während der Entwicklung. Zusätzlich fand die Modellierung als *Proof of Concept* an einem Praxisbeispiel, dem *Schöffenverfahren*, statt. Die Machbarkeit und Anwendbarkeit der entwickelten DSL wurde dabei bestätigt und weitere Erkenntnisse gewonnen. Einerseits ist aufgefallen, dass in kurzer Zeit das automatisierte erzeugen von Prototypen möglich ist und andererseits das Metamodell noch Freiheiten bezüglich der Reihenfolge der Modellierung erlaubt (vgl. Unterkapitel 4.3.2, Seite 78). Sowohl die Optimierung des Prozessmodells, als auch die Standardisierung des Produkts konnten damit erreicht werden, wie es in der Aufgabenstellung (vgl. Kapitel 1.2) als Ziel definiert wurde. Mit wenig Aufwand ist folglich die Erstellung eines Modells einer Anwendung von einer Person ohne Entwicklerkenntnisse realisierbar. Das hat eine hohe Zeit- und Kostenersparnis zur Folge. Kürzere Iterationszeiten (vgl. Abbildung 3.3, Seite 42) erlauben zudem kurzfristige Änderungen. Befindet sich die Anwendung bereits in der eigentlichen Implementierungsphase (vgl. Abbildung 3.2, Seite 41) sind, im Gegensatz zum bisherigen *Prozessmodell IT-Service*, noch Änderungen am Modell ohne zusätzliche Mehrarbeit realisierbar.

Die verwendeten Werkzeuge sind einerseits bei der LHM freigegeben und andererseits

frei verfügbar, wodurch keine zusätzlichen Kosten entstanden sind. Aus technischer Sicht ergeben sich durch die bedingten Abhängigkeiten zu den Bibliotheken keine Nachteile, da diese im internen Artifactory (vgl. Abbildung 2.3, Seite 11) vorgehalten werden.

5.2 Ausblick

Zwei bereits angesprochene Ergänzungen erscheinen zur Umsetzung in der Praxis sinnvoll:

Einerseits die Generierung der Dokumente des *Prozessmodell IT-Service*. In Kapitel 3.4 wurde auf diese theoretisch eingegangen und die potentiell möglichen (Unter-)Kapitel beschrieben.

Demgegenüber steht die Bereitstellung einer GUI zur Modellierung anstelle der textuellen IDE. Die Verwendung der DSL wurde zu Beginn von Hauptkapitel 4 auf den bereitgestellten Editor festgelegt. Zur Steigerung der Akzeptanz der Modellierer kann eine grafische Oberfläche dienen. Ist diese zudem webbasiert wird simultan eine Vereinfachung des Vorgehens erreicht, da keine Installation der Entwicklungsumgebung an den entsprechenden Arbeitsplätzen erforderlich ist. Ebenso erfolgt damit eine Lösung des Problems mit den einmalig zu generierenden Teilen, da durch die Oberfläche die Generierung aktiv vom Modellierer anzustoßen ist.
Die Entwicklung der GUI für die DSL ist wiederum mit Barrakuda modellierbar, da die Sprache und das Ergebnis die dafür erforderlichen Inhalte besitzt.

Literaturverzeichnis

[AXE] AXELOS: *What is ITIL?* https://www.axelos.com/best-practice-solutions/itil/what-is-itil. – Zugriff: 01.04.2015 [16, 5.2]

[Bal11] Balzert, Helmut: *Lehrbuch der Softwaretechnik: Entwurf, Implementierung, Installation und Betrieb*. Spektrum Akademischer Verlag, 2011. – ISBN 978–3–8274–1706–0 [1.1, 1.2]

[Ben12] Benz, Sebastian: *5 Things that make Xtend a great Language for Java Developers*. http://www.sebastianbenz.de/5-Things-that-make-Xtend-a-great-Language-for-Java-developers. Version: 2012. – Zugriff: 04.01.2016 [114]

[Bet13] Bettini, Lorenzo: *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing, 2013. – ISBN 1782160302, 9781782160304 [119]

[Bet15] Bettini, Lorenzo: *Testing Xtext Languages*. https://www.eclipsecon.org/france2015/sites/default/files/slides/presentation.pdf. Version: 2015. – Zugriff: 09.01.2016 [4.2]

[Bro12] Brodski, Boris: *#15 Output Configurations - XtextCasts*. http://xtextcasts.org/episodes/15-output-configurations. Version: 2012. – Zugriff: 07.01.2016 [133]

[Cor14] Corporation, Oracle: *JSRs: Java Specification Requests*. https://jcp.org/en/jsr/detail?id=175. Version: 2014. – Zugriff: 12.12.2015 [55, 5.2]

[EV06] Efftinge, Sven ; Völter, Markus: oAW xText: A framework for textual DSLs. (2006), September. http://www.voelter.de/data/workshops/EfftingeVoelterEclipseSummit.pdf. – Zugriff: 31.03.2015 [119]

[Eva04] Evans, E.: *Domain-driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 2004. – ISBN 9780321125217 [2.2.2, 31, 32, 33, 34, 35, 37]

[Eva15] Evans, Eric: *Domain-Driven Design Reference*. http://www.domainlanguage.

com/ddd/reference/DDD_Reference_2015-03.pdf. Version: 2015. – Zugriff: 25.12.2015 [30, 36]

[Eys11] Eysholdt, Moritz: *Dr. Dobb's | Generating Code from DSLs | September 27, 2011.* http://www.drdobbs.com/architecture-and-design/generating-code-from-dsls/231602237. Version: 2011. – Zugriff: 23.01.2016 [119]

[Fie00] Fielding, Roy T.: *Architectural Styles and the Design of Network-based Software Architectures*, Diss., 2000. – AAI9980887 [45, 47]

[Foua] Foundation, The E.: *Eclipse Modeling Framework.* http://www.eclipse.org/modeling/emf/. – Zugriff: 27.12.2015 [107]

[Foub] Foundation, The E.: *Xtend - Modernized Java.* http://eclipse.org/xtend/. – Zugriff: 20.11.2015 [111, 119, 5.2]

[Fouc] Foundation, The E.: *Xtext - Language Development Made Easy!* http://eclipse.org/Xtext/. – Zugriff: 20.11.2015 [109, 116, 119, 5.2]

[Foud] Foundation, The E.: *Xtext - The Grammar Language.* https://eclipse.org/Xtext/documentation/301_grammarlanguage.html. – Zugriff: 09.02.2016 [137]

[Fow03] Fowler, Martin: *PlatformIndependentMalapropism.* http://martinfowler.com/bliki/PlatformIndependentMalapropism.html. Version: 2003. – Zugriff: 05.12.2015 [90]

[Fow10] Fowler, Martin: *Domain Specific Languages.* 1st. Addison-Wesley Professional, 2010. – ISBN 0321712943, 9780321712943 [2.4, 91]

[Fow13] Fowler, Martin: *DDD Aggregate.* http://martinfowler.com/bliki/DDD_Aggregate.html. Version: 2013. – Zugriff: 07.12.2015 [38]

[FR07] France, Robert ; Rumpe, Bernhard: Model-driven development of complex software: A research roadmap. In: *2007 Future of Software Engineering* IEEE Computer Society, 2007, S. 37–54 [2.4]

[Gie15] Gierke, Oliver: *Domain-Driven Design and Spring.* http://static.olivergierke.de/lectures/ddd-and-spring/. Version: 2015. – Zugriff: 07.12.2015 [2.2.2]

[Gro15] Group, Object M.: *MDA Specifications.* http://www.omg.org/mda/specs.htm. Version: 2015. – Zugriff: 11.10.2015 [73, 88]

[Gün11] Günther, Sebastian: Development of Internal Domain-specific Languages: Design Principles and Design Patterns. In: *Proceedings of the 18th Conference on Pattern Languages of Programs*. New York, NY, USA : ACM, 2011 (PLoP '11). – ISBN 978–1–4503–1283–7, 1:1–1:25 [2.4]

[HT06] Hailpern, B. ; Tarr, P.: Model-driven Development: The Good, the Bad, and the Ugly. In: *IBM Syst. J.* 45 (2006), Juli, Nr. 3, 451–461. http://dx.doi.org/10.1147/sj.453.0451. – DOI 10.1147/sj.453.0451. – ISSN 0018–8670 [2.4, 87]

[Kru03] Kruchten, Philippe: *The Rational Unified Process: An Introduction*. 3. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2003. – ISBN 0321197704 [14]

[LC13] Le Coent, Christophe: *How to link User stories, DDD and BDD | christophelecoent*. https://christophelecoent.wordpress.com/2013/06/17/how-to-link-user-stories-ddd-and-bdd/. Version: 2013. – Zugriff: 12.07.2015 [A.2]

[Ltda] Ltd., Vaadin: *Book of Vaadin - vaadin.com*. https://vaadin.com/book/-/page/intro.html. – Zugriff: 26.11.2015 [2.14]

[Ltdb] Ltd., Vaadin: *Docs - vaadin.com*. https://vaadin.com/docs/-/part/framework/components/components-overview.html. – Zugriff: 17.02.2016 [A.1]

[Ltdc] Ltd., Vaadin: *Framework - vaadin.com*. https://vaadin.com/framework. – Zugriff: 25.11.2015 [63]

[MHS05] Mernik, Marjan ; Heering, Jan ; Sloane, Anthony M.: When and How to Develop Domain-specific Languages. In: *ACM Comput. Surv.* 37 (2005), Dezember, Nr. 4, 316–344. http://dx.doi.org/10.1145/1118890.1118892. – DOI 10.1145/1118890.1118892. – ISSN 0360–0300 [12, 2.4, 1, 83]

[Müna] München, Landeshauptstadt: *01_Informationen und Definitionen - LHM Intranet*. http://intranet.muenchen.de/basis/it/mitkonkret/00_uebung_sebastian/info_und_definitionen/index.html. – Zugriff: 25.03.2015 [8, 19, 5.2]

[Münb] München, Landeshauptstadt: *Landeshauptstadt München - it@M*. http://www.muenchen.de/rathaus/Stadtverwaltung/Direktorium/IT-Beauftragte/Wir-ueber-Uns/ITatM.html. – Zugriff: 14.03.2015 [6]

[Münc] München, Landeshauptstadt: *Landeshauptstadt München - MIT-KonkreT*. http://www.muenchen.de/rathaus/Stadtverwaltung/Direktorium/IT-Beauftragte/MIT-KonkreT.html. – Zugriff: 14.03.2015 [1, 3, 5]

[Off11] Office, Cabinet: *ITIL Lifecycle Suite 2011 Edition*. Norwich : The Stationery Office, 2011. – ISBN 0113313233, 9780113313235 [13]

[PM06] Petrasch, Roland ; Meimberg, Oliver: *Model-Driven Architecture: Eine praxisorientierte Einführung in die MDA*. dpunkt, 2006. – ISBN 978–3–89864–343–6 [2.4]

[Sofa] Software, Pivotal: *6.2. JavaConfig and Annotation-Driven Configuration*. http://docs.spring.io/spring-javaconfig/docs/1.0.0.M4/reference/html/ch06s02.html. – Zugriff: 02.01.2016 [66]

[Sofb] Software, Pivotal: *Spring Boot*. http://projects.spring.io/spring-boot/. – Zugriff: 27.11.2015 [43]

[Sofc] Software, Pivotal: *Spring Data REST*. http://projects.spring.io/spring-data-rest/. – Zugriff: 07.12.2015 [52]

[Sofd] Software, Pivotal: *Understanding HATEOAS*. https://spring.io/understanding/HATEOAS. – Zugriff: 07.12.2015 [49, 142]

[Sof98] Software, Rational: *Rational Unified Process*. https://www.ibm.com/developerworks/rational/library/content/03July/1000/1251/1251_bestpractices_TP026B.pdf. Version: 1998. – Zugriff: 22.05.2015 [18]

[Sta73] Stachowiak, H.: *Allgemeine Modelltheorie*. Springer-Verlag, 1973. – ISBN 9783211811061 [70, 5.2]

[Sto11] Stoll, Dietmar: *Dietmar Stoll's Blog: Xtext Grammar Visualization*. http://blog.dietmar-stoll.de/2011/08/xtext-grammar-visualization.html. Version: 2011. – Zugriff: 08.01.2016 [132]

[Str13] Straube, Claus: *Software-Produktionsstraße-Java*. 2013. – Zugriff: 26.10.2015 [2.3]

[SVE+07] Stahl, Thomas ; Völter, Markus ; Efftinge, Sven ; Haase, Arno ; Bettin, Jorn ; Helsen, Simon ; Kunz, Michael: *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*. dpunkt, 2007. – ISBN 978–3–89864–448–8 [71, 72, 76, 2.4, 2.4, 2.5.2, 92, 93, 3.2.1, 96, 97, 98, 100, 5.2]

[SZ09] Strembeck, Mark ; Zdun, Uwe: An Approach for the Systematic Develop-

ment of Domain-Specific Languages. In: *Softw. Pract. Exper.* 39 (2009), Oktober, Nr. 15, 1253–1292. http://dx.doi.org/10.1002/spe.v39:15. – DOI 10.1002/spe.v39:15. – ISSN 0038–0644 [2.4]

[TB07] Trompeter, J. ; Beltran, J.C.F.: *Modellgetriebene Softwareentwicklung: MDA und MDSD in der Praxis*. Entwickler.Press, 2007. – ISBN 9783939084112 [2.4, 84, 86]

[Voe13] Voelter, Markus: *DSL Engineering: Designing, Implementing and Using Domain-specific Languages*. CreateSpace Independent Publishing Platform, 2013 http://voelter.de/data/books/markusvoelter-dslengineering-1.0.pdf. – ISBN 9781481218580 [76, 2.4, 99]

[Völ09] Völter, Markus: Werkzeuge zur Erstellung und Verarbeitung von DSLs. (2009). http://www.voelter.de/data/articles/ToolsFuerDSLs.pdf. – Zugriff: 11.10.2015 [2.5.2]

Abkürzungsverzeichnis

API	Application Programming Interface
BDD	Behaviour-Driven Development
BRE	Business Requirement Engineer
CO	Component Owner
CRUD	Create, Read, Update, Delete
DDD	Domain-Driven Design
DTO	DataTransfer Object
dIKA	dezentrale Informations-, Kommunikations- und Anforderungsmanagement
DSL	domänenspezifische Sprache
EMF	Eclipse Modeling Framework
FK	Fachkonzept
GA	Geschäftsaktion
GUI	Graphical User Interface
HATEOAS	Hypermedia As The Engine Of Application State
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
IT@M	Informations- und Telekommunikationstechnik der Stadt München
ITIL	IT Infrastructure Library
KKF	Kernkompetenzfokussierung
KVR	Kreisverwaltungsreferat
LHM	Landeshauptstadt München
MBUC	Make-Buy-Use-Compose
MDA	Model Driven Architecture

MDD	Model Driven Development
MDE	Model Driven Engineering
MDSD	Model Driven Software Development
MVC	Model View Controller
MVP	Model View Presenter
MIT-KonkreT	Münchner IT - Konkrete Umsetzung und TOP Priorities
MS	Microservice
OMG	Object Management Group
PIM	Platform Independent Model
PSM	Platform Specific Model
POR	Personal- und Organisationsreferat
REST	Representational State Transfer
RUP	Rational Unified Process
SEU	Softwareentwicklungsumgebung
SLA	Service-Level-Agreement
SLM	Service-Level-Manager
SO	Service Owner
SPS	Software-Produktionsstraße
SQL	Structured Query Language
STRAC	IT-Strategie und IT-Steuerung / IT-Controlling
SysSpec	Systemspezifikation
TA	Testanalyst
TM	Testmanager
TK	Testkonzept
TRE	Technical Requirement Engineer
UML	Unified Modeling Language
URI	Uniform Resource Identifier

Glossar

Annotation Ein Sprachelement von Java, die die Einbindung von Meta-Daten erlaubt (vgl. [Cor14], Abschnitt: ”Description”).

Anwendung Die Software zur Unterstützung eines Geschäftsprozesses.

Anwendungsfall Ein Anwendungsfall beschreibt ein Systemverhalten bzw. die Interaktion zur Erreichung eines bestimmten Ziels.

Artefakt Ein Produkt, dass während der Softwareentwicklung entstanden ist und zur Wiederverwendung und/oder gemeinsamen Nutzung bereitsteht.

Bean Eine datenhaltende Klasse in Java, die mehrere Konventionen einhält und dadurch eine Persistenz erlaubt.

Constraint Formale Bedingung einer Sprache.

DSL Eine Programmiersprache für eine Domäne (begrenztes Interessens-/Wissensgebiet), die aus einem Metamodell mit Syntax und statischer Semantik besteht (vgl. [SVE+07], Kapitel: ”3.1.1 Modellierung”, Abschnitt: ”Domänenspezifische Sprache”, Seite 30).

Early Life Support In den ersten Wochen nach einer Einführung bzw. dem Update eines Service soll mit dieser Unterstützung der Betrieb, durch kurzfristige durchführbare Fehlerbehebungen, sichergestellt werden.

Formatter In Bezug auf das Xtext-Framework die Komponenten zur Formatierung der Ausgabe.

Forward Engineering Modelländerungen werden automatisch auf den entsprechenden Quelltext übertragen.

Framework Ein Grundgerüst mit gängigen Lösungen und/oder Bausteinen, dass die Entwicklung unterstützt und zugleich beschleunigen kann.

Geschäftsaktion Operation, die zur Unterstützung eines Geschäftsprozesses dient.

Injektor Ein Werkzeug, um an definierten Stellen innerhalb einer Anwendung eingreifen zu können.

Interface Beschreibung einer Schnittstelle in Java.

IT-Service "*Ein IT-Service ist die Zusammenfassung von geschäftsprozessunterstützenden IT-Funktionen (bestehend aus Hardware, Software- und Aktivitätselementen), die vom Kunden als eine in sich geschlossene Einheit wahrgenommen werden.*" [Müna], Abschnitt: "IT-Service".

IT@M Eigenbetrieb der Landeshauptstadt München, der als professioneller zentraler IT-Dienstleister für die Referate zur Verfügung steht.

ITIL ITIL plädiert für die Ausrichtung der IT-Dienstleistungen an die Bedürfnisse des Unternehmens und ihrer Kernprozesse. Dabei stellt sie Anleitungen und "*Best Practices*" bereit, wie diese Anpassung erfolgen kann (vgl. [AXE], Abschnitt: "What is ITIL®?").

Java Objektorientierte Programmiersprache, deren Programme durch die Bereitstellung von Laufzeitumgebungen eine plattformunabhängige Ausführung gewährleisten.

Klasse Ein Objekttyp von Java mit Eigenschaften und Methoden.

Mapping Abbildung der Daten von einem Schema A auf ein Schema B.

Meilenstein Wichtiges definiertes Ereignis in einem Prozess, z.B. Übergang von einer Phase in die Nächste.

Metamodell Die abstrakte bzw. übergeordnete Beschreibung eines Modells.

Microservice Ein einzelner Dienst für einen bestimmten Zweck, der unabhängig von anderen ausgeliefert und betrieben werden kann.

MIT-KonkreT Das referatsübergreifende Projekt zur strategischen Neuausrichtung der IT der Landeshauptstadt München.

Modell Ein Abbildung oder ein Vorbild mit nur relevanten Eigenschaften von bzw. für etwas (vgl. [Sta73], Kapitel "2.1.1 Die drei Hauptmerkmale des allgemeinen Modellbegriffs", Seite 131ff).

Package In Java die Technik zur Anordnung zusammengehöriger Klassen.

Pattern Ein bewährter Lösungsansatz.

Reflection Eine Möglichkeit unbekannte Java-Klassen zu instantiieren, analysieren und die angebotenen Methoden aufzurufen.

Reverse Engineering Quelltextänderungen werden automatisch in die entsprechenden Modelle übernommen.

Rollout Installation und Konfiguration von Hardware und/oder Software an einem Stichtag.

Round-Trip-Engineering Dabei erfolgt ein automatischer Abgleich zwischen dem Entwurf (z.B. Modell) und der Umsetzung (z.B. Quelltext) oder umgekehrt (siehe Forward Engineering und Reverse Engineering).

RUP Ein Vorgehensmodell zur Softwareentwicklung.

Scoping Im Kontext des Xtext-Frameworks die Komponente, die die Bereiche festlegt, in der Modellelemente (Variablen und Methoden) gültig sind.

Servlet In Java eine Klasse, die speziell das javax.servlet.Servlet-Interface implementiert und damit HTTP-Anfragen annehmen und beantworten kann.

SLA "*Ein Service-Level-Agreement (SLA) ist eine schriftliche Vereinbarung zwischen den Serviceerbringern / Lieferanten und den Serviceabnehmern / Kunden über die Service-Level-Ziele, die Servicequalität und die Leistungsverrechnung für die im Servicekatalog aufgeführten IT-Services.*" [Müna], Abschnitt: "Service-Level-Agreement".

Template In Verbindung mit dieser Ausarbeitung ein Bereich im Generator, der eine Vorlage bzw. ein Muster für die zu erstellenden Dateien beinhaltet.

Validator In Zusammenhang mit dem Xtext-Framework eine Komponente, um zusätzliche Constraints für die Sprache festlegen zu können.

Workflow In Bezug auf das Xtext-Framework eine Steuerungsdatei, die die Generierung der Sprache aus der Grammatikdatei mit den nötigen Schritten definiert.

Xtend Eine statisch getypte Programmiersprache, die in verständlichen Java-Quelltext übersetzt wird (vgl. [Foub], Abschnitt: "Java 10, today!").

Xtext Ein Framework zur Entwicklung von Programmier- und domainenspezifischen Sprachen (vgl. [Fouc], Abschnitt: "What is Xtext?").

A Anhang

A.1 Prozessmodell "IT-Service" der Landeshauptstadt München

Das Prozessmodell in der, zum Zeitpunkt der Ausarbeitung vorliegenden, aktuellen Version 2.1 befindet sich auf der nachfolgenden aufklappbaren Seite.

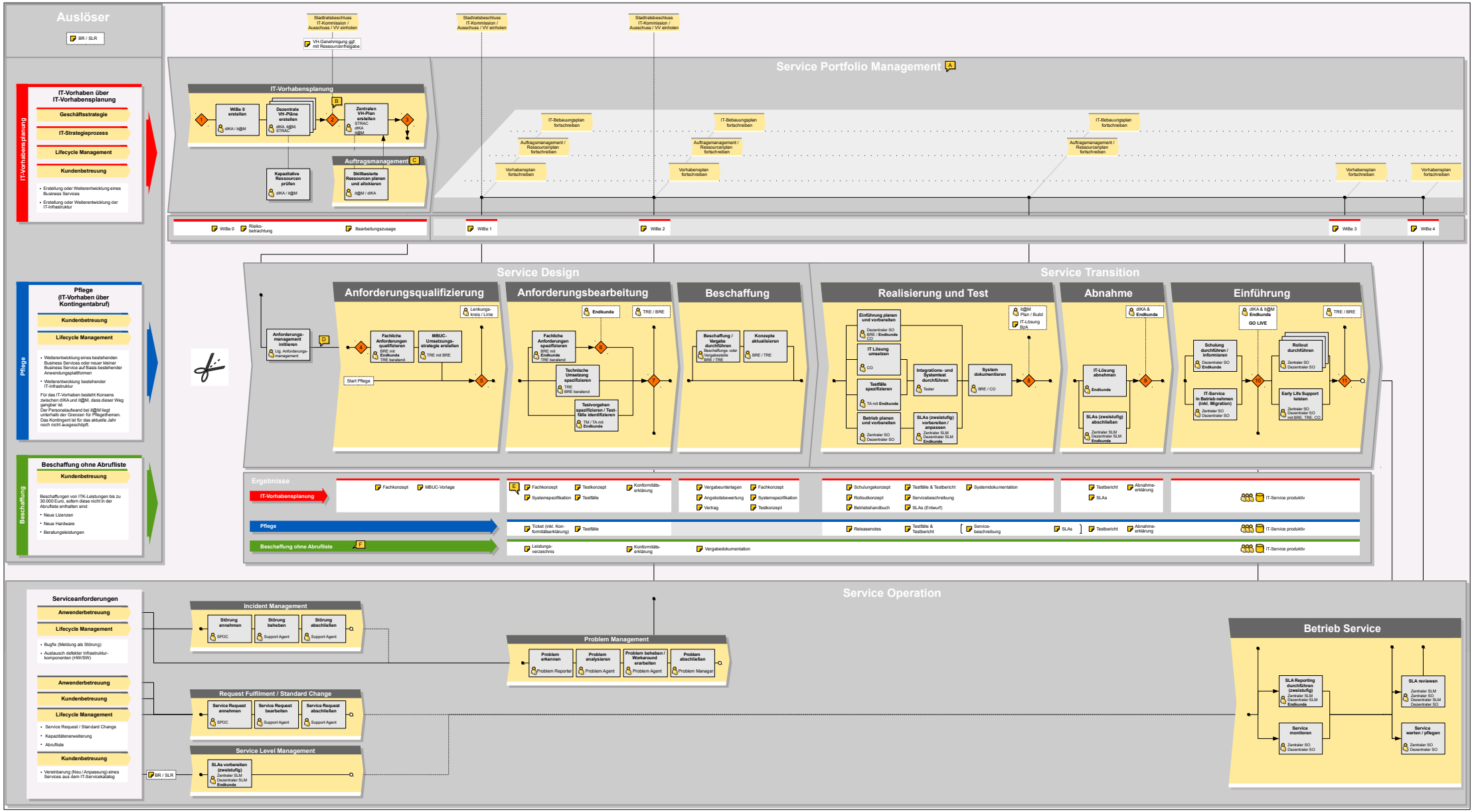
Prozessmodell IT-Service 2.1

Das Prozessmodell IT-Service bietet einen vereinfachten Überblick zu den wesentlichen Aktivitäten der Prozesse. Ausführliche Informationen und weitere Details finden Sie in den Prozessbeschreibungen im Intranet unter:

LHM Intranet > IT + Kommunikation > übergreifende Prozesse und Themen > Prozessmodell IT-Service

Beachten Sie in der Umsetzung der Prozesse die Möglichkeit und Notwendigkeit der Anpassung auf das jeweilige IT-Vorhaben (Tailoring), siehe unten

Landeshauptstadt München



Tailoring

Bedeutung: Die einzelnen im Prozessmodell IT-Service aufgeführten Prozesse können bei Bedarf an die Bedingungen und Bedürfnisse des jeweiligen IT-Vorhabens, IT-Services und der jeweiligen Organisationseinheit angepasst werden.

Die Entscheidung hierüber wird zwischen den unmittelbar Beteiligten abgestimmt.

Solche **Anpassungen** wirken sich beispielhaft aus auf die Dimensionen

- **Ablauf** *(iteratives oder inkrementelles Vorgehen, geänderte Reihenfolge)*
- **Aktivitäten** *(geänderte Inhalte, Ergänzen, Vereinfachen)*
- **Rolle** *(eine Person – mehrere Rollen, eine Rolle – mehrere Personen)*
- **Rollenverortung** *(BRE aus it@M, TRE aus dIKA)*
- **Dokumentstruktur** *(Fortführung Fachkonzept als Leistungsbeschreibung)*
- **Dokumententiefe** *(Kapitel unzutreffend, Kapitel anpassen und ergänzen)*

Weiterführende Informationen

Beschreibungen der Prozesse, Aktivitäten und Rollen, Zugang zum Adonis Webportal, Dokumentenvorlagen, Designvorgaben, Checklisten, etc. finden Sie unter:

LHM Intranet Basisangebot
> IT + Kommunikation
> Vorlagen („Gelbe Seite")
> übergreifende Prozesse und Themen
> IT-Vorhabensplanung
> Anforderungsmanagement
> Transition Management
> Service Operation
> etc.

Ebenso Hinweise zur Anwendung des Tailorings.

Meilensteine

Service Portfoliomanagement

- Initialisierung IT-Vorhaben
- Finanzierung des IT-Vorhabens ist geklärt
- IT-Vorhaben wird gestartet

Service Design / Service Transition

- Anforderungsmanagement ist gestartet
- MBUC-Entscheidung liegt vor
- Abnahme Fachkonzept durch Endkunden
- Anforderungsbearbeitung ist abgeschlossen
- Testkonzept ist abgenommen
- Bereit zur Abnahme (IT-Lösung und Konzepte)
- IT-Lösung ist abgenommen (Übergabe Betrieb)
- IT-Service ist produktiv
- IT-Service ist abgenommen

Kommentare und Anmerkungen

A Service Portfoliomanagement / IT-Vorhabensplan
Der IT-Vorhabensplan umfasst neben der Erstellung eines neuen IT-Services bzw. der Erweiterung bestehender IT-Services noch weitere IT-Vorhaben, die im Rahmen des IT-Strategieprozesses durchgeführt werden (beispielsweise: IT-Organisation, IT-Prozesse, IT-Personalmanagement etc.).

B Prüfung Linie / IT-Projekt
An dieser Stelle wird überprüft, ob das IT-Vorhaben als ein IT-Projekt oder in der Linie umgesetzt wird.

C Auftragsmanagement
Im Auftragsmanagement werden die über den zentralen IT-Vorhabensplan angeforderten IT-Vorhabensphasen mit den verfügbaren Ressourcen und deren Skills abgeglichen.

D Verantwortung IT-Vorhaben
Der Task Responsible plant, steuert und kontrolliert das IT-Vorhaben, insbesondere die Ressourcen, Kosten, Termine und Ergebnisse. Falls notwendig stößt er einen Stadtratsbeschluss an. Wird das IT-Vorhaben als IT-Projekt durchgeführt, übernimmt der IT-Projekt Manager die Aufgaben des Task Responsible.

E Detailtiefe der Ergebnisdokumente
Dokumentenvorlage und Detailtiefe der Ergebnisdokumente sind abhängig von der MBUC-Entscheidung.

F Nicht zu erstellende Dokumente bei der Beschaffung von Beratungsleistungen
Risikobetrachtung und Konformitätserklärung

Legende

Überblick Prozess

< auslösender Prozess >
< Auslösendes Ereignis >
< Prozess >

Detaillierter Prozess / Phase

< Prozess / Phase >
< maßgebliche Aktivität des Prozesses / der Phase >
< beteiligte Rollen >
< zu erstellendes Ergebnis >

- Nachfolgender Prozessschritt
- Wiederholender Prozessschritt
- Optionaler Prozessschritt
- Meilenstein
- A Kommentar (siehe Kommentarliste)

Anregungen an strac.dir@muenchen.de

A.2 HATEOAS-Beispiel

HATEOAS ermöglicht eine Navigation innerhalb von Datenstrukturen bei REST-Interfaces.[142] Das hier aufgeführte Beispiel soll die Änderung der übertragenen Daten durch die Verwendung aufzeigen.

Über die REST-Schnittstelle sollen Instanzen der Beispiel-Beans aus Listing A.1 übertragen werden.

```
1 class Person {
2   String name;
3   Anschrift anschrift;
4 }
5
6 class Anschrift {
7   String strasse;
8   String hausnummer;
9 }
```

Listing A.1: Beispiel für zwei Java-Beans. Zur Person ist die Angabe einer Anschrift möglich.

Standardmäßig beinhaltet der übermittelte Datensatz die in Listing A.2 notierten Daten.

```
1 {
2   "name" : "Tester",
3   "anschrift" : {
4     "strasse" : "Muster"
5     "hausnummer" : "42"
6   }
7 }
```

Listing A.2: Beispiel für die als JSON übertragenen Daten der in Listing A.1 enthaltenen Java-Beans.

Erfolgt die Speicherung der Anschrift nicht direkt während der Erstellung der Person, muss mindestens eine URI zwischengespeichert werden, um anschließend wieder zurück navigieren zu können. Mit der Verwendung von HATEOAS besitzen die Datensätze zusätzlich Links, wie es in Listing A.3 veranschaulicht ist.

[142] vgl. [Sofd], Abschnitt: "Understanding HATEOAS"

```
{
  "name" : "Tester",
  "links": [
    {
      "rel": "self",
      "href": "http://localhost:8080/person/1"
    },
    {
      "rel": "anschrift",
      "href": "http://localhost:8080/anschrift/1"
    }
  ]
}
```

Listing A.3: Beispiel für die als JSON übertragenen Daten der in Listing A.1 enthaltenen Java-Beans mit Verwendung von HATEOAS.

Die Links wurden von HATEOAS automatisch erzeugt und angegeben.

A.3 Vaadin Komponenten

Das Vaadin-Framework stellt Komponenten zur Verwendung zur Verfügung. In Abbildung A.1 sind die Bereitgestellten anhand ihrer Vererbungshierarchie dargestellt.

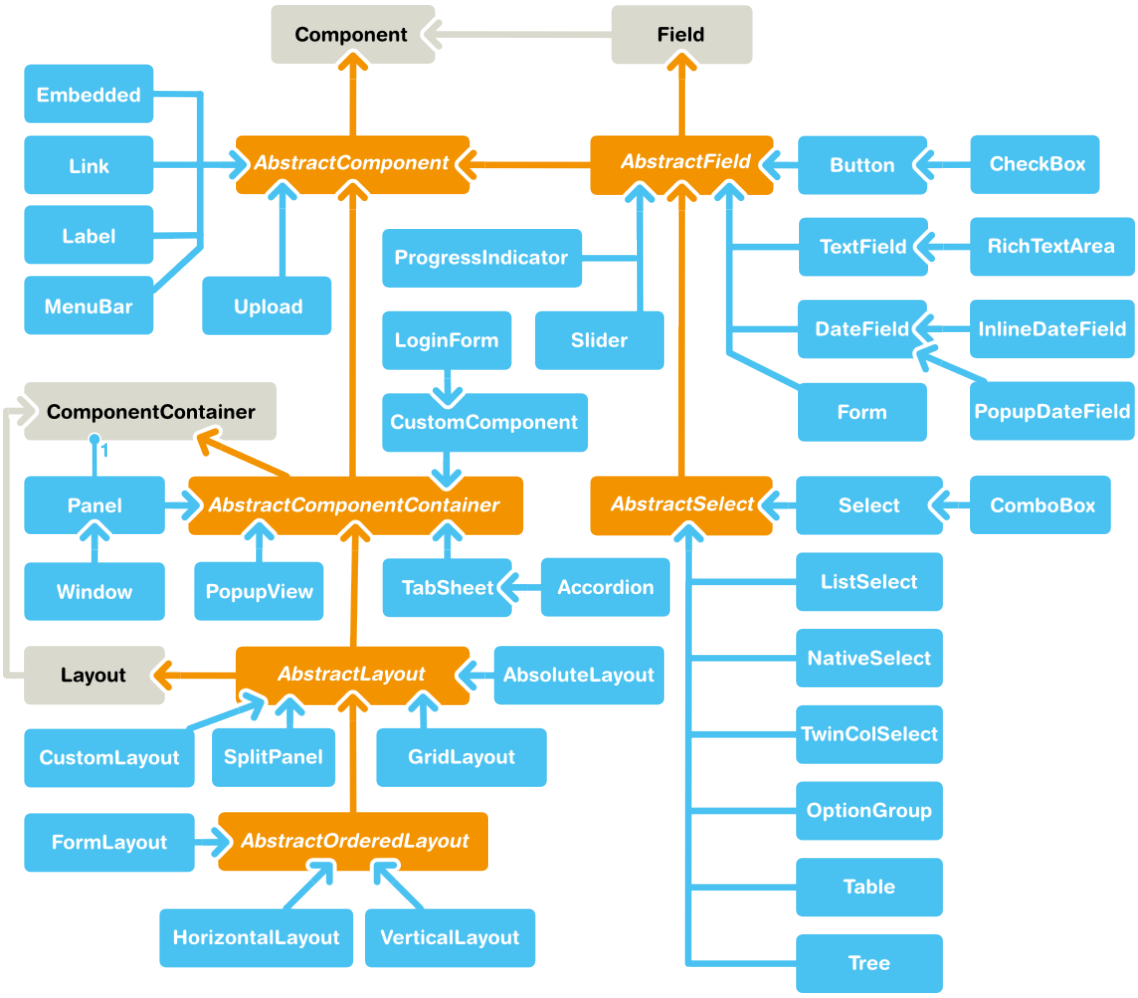


Abbildung A.1: Darstellung möglicher Komponenten im Vaadin-Framework nach [Ltdb], Abbildung: "Figure 1. User Interface Component Class Hierarchy".

A.4 Zusammenhang zwischen Domain-Driven Design und Behaviour-Driven Development

Die vorgestellte Literatur (vgl. Kapitel 2.4) beinhaltet Beispiele für den Aufbau von DSLs, jedoch keine Ansätzen anhand Domain-Driven Design (DDD) oder Behaviour-Driven Development (BDD). In Abbildung A.2 werden Interaktionen und der Zusammenhang zwischen diesen Vorgehensweisen dargestellt. DDD fokussiert sich mehr auf die gemeinsame Sprache innerhalb einer Domäne, wobei der Fokus von BDD primär auf Akzeptanzkriterien liegt.

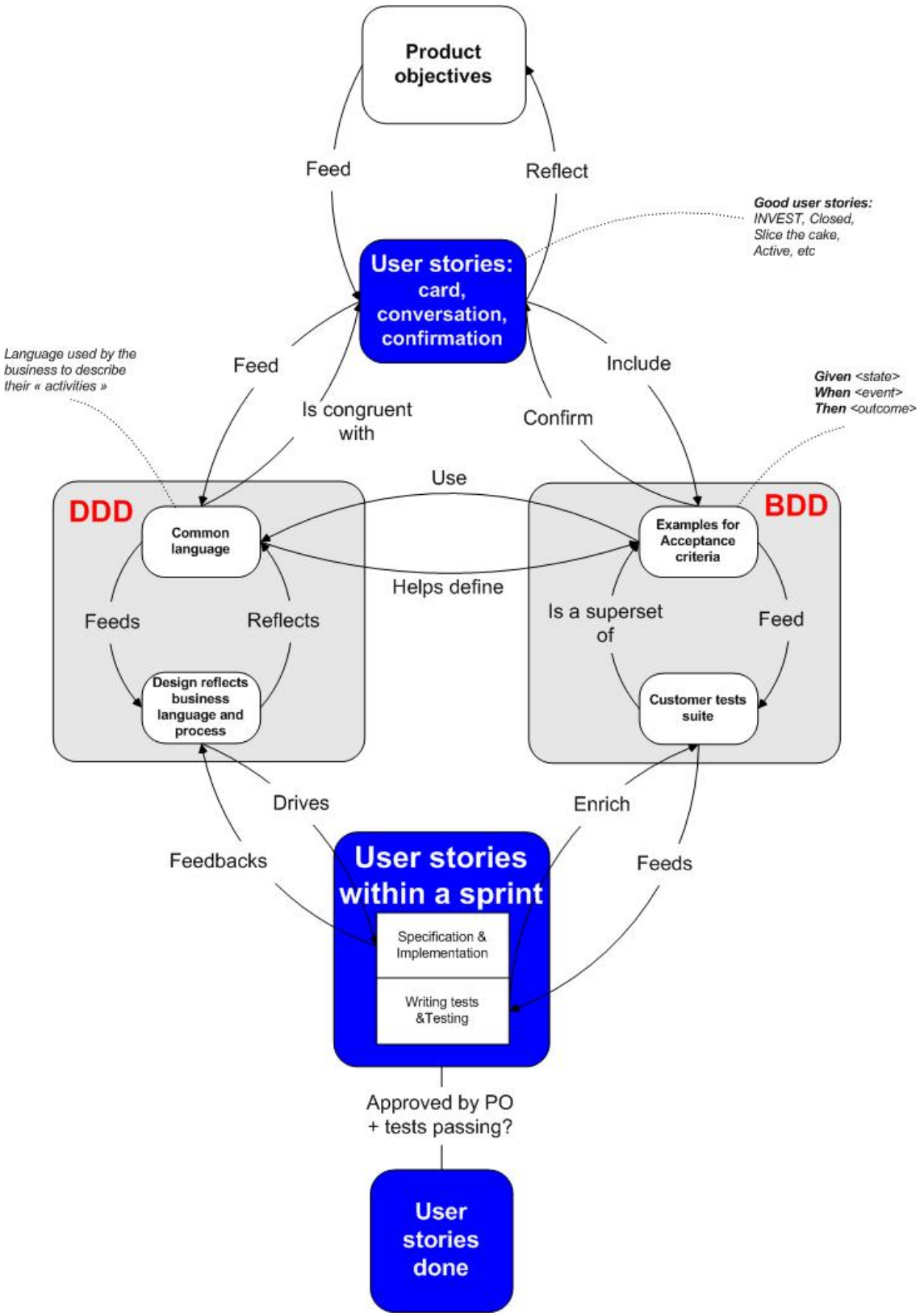


Abbildung A.2: Darstellung der Interaktionen und dem Zusammenhang zwischen Domain-Driven Design und Behaviour-Driven Development nach [LC13].

A.5 Modell des Praxisbeispiels

Während der Verifikation (in Kapitel 4.3) wurde ein Modell des Verfahrens zur Auswahl und Verwaltung von Schöffen erstellt. Dieses ist nachfolgend angegeben.

```
model de.muenchen.kvr.ewo.schoeffen artifact Schoeffenverfahren version 1 {
    customTextType text;
    customTextType textMitMax10 maxLength=10;
    customTextType textMitMax50 maxLength=50;
    customTextType jahr allowedRegex="20[0-9][0-9]";
    customTextType mailadresse allowedRegex=".+@.+\\..+";
    customDateType datumInVergangenheit inThePast;
    customNumberType zahl;
    customNumberType plz minValue=10000 maxValue=99999;
    customLogicType jaNein;

    entity Buerger {
        Ordnungsmerkmal zahl;
        AkademischerGrad textMitMax50 optional "Dr.";
        Familienname textMitMax50 mainFeature "Musterschoeffe";
        Vorname textMitMax50 mainFeature "Max";
        Geburtsdatum datumInVergangenheit mainFeature "01.01.1990";
        Geburtsort textMitMax50;
        Familienstand textMitMax10;
        Staatsangehoerigkeit textMitMax50;
        Hauptwohnung Adresse;
        Selbststaendig jaNein "true";
        Arbeitnehmer jaNein "false";
        DerzeitAusgeuebterBeruf textMitMax50 mainFeature "Dipl. Kaufmann";
        InMuenchenSeit datumInVergangenheit;
        Telefonnummer textMitMax50 "098233";
        Mailadresse mailadresse "m.kurz@muenchen.de";
    }

    entity Adresse {
        Strasse textMitMax50;
        Hausnummer textMitMax10;
        Postleitzahl plz;
        Ort textMitMax50;
    }

    entity Amtsperiode {
        AmtsperiodeVon jahr mainFeature "2014";
        AmtsperiodeBis jahr "2018";
    }

    entity Schoeffe {
        Amtsperiode Amtsperiode;
        Vorschlag jaNein mainFeature "ja";
        AufAntragAufgenommenSeit datumInVergangenheit "01.09.2015";
        EWODaten Buerger;
    }

    businessAction FreiwilligMelden {
```

```
50          purpose "Bürger als potentieller Schöffe freiwillig melden";
51          given Schoeffe;
52      }
53
54      businessAction SchoeffenBestimmen {
55          purpose "Wählt per Zufall eine repräsentative Liste an Schöffen aus dem
                Einwohnermeldesystem aus";
56          then listOf Schoeffe;
57      }
58
59      businessAction SchoeffenListeUebernehmen {
60          purpose "Überträgt die Vorschläge in die feste Liste";
61      }
62
63      businessAction SchoeffenInformieren {
64          purpose "Informiert Schöffen";
65          given text; // Mitteilung
66          given listOf Schoeffe;
67      }
68
69      businessAction SchoeffenStadtrat {
70          purpose "Schöffenliste durch Stadtrat bestätigen lassen";
71          then listOf Schoeffe;
72      }
73
74      businessAction SchoeffenUebergeben {
75          purpose "Schöffenliste an Verwaltungsgericht übergeben";
76          given listOf Schoeffe;
77      }
78  }
```

Listing A.4: Das Schöffenverfahren modelliert mit Barrakuda.