Otto-von-Guericke-Universität Magdeburg

Fakultät für Informatik

Bachelorarbeit

Machbarkeitsanalyse einer Workflow-Automatisierung am Beispiel eines spezifischen Kundenauftrages

Autoren:

Christian Steusloff
Jens Wiemann

05. Juni 2015

Betreuer:

Prof. Dr. rer. nat. habil. Gunter Saake,
M.Sc. David Broneske

Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Postfach 4120, D–39016 Magdeburg

Jens Wiemann, Christian Steusloff:
Machbarkeitsanalyse einer Workflow-Automatisierung am Beispiel eines spezifischen Kundenauftrages
Bachelorarbeit, Otto-von-Guericke-Universität Magdeburg, 2015.

Danksagung

An dieser Stelle möchten wir uns bei allen Personen bedanken, die uns bei der Anfertigung der Arbeit unterstützt haben.

Ein besonderer Dank geht an Prof. Dr. rer. nat. habil. Gunter Saake und M.Sc. David Broneske für die umfangreiche Betreuung, die zahlreichen Anregungen und Hilfestellungen.

Außerdem geht ein spezieller Dank an die SAB Ingenieurgesellschaft mbH für die produktive Arbeitsatmosphäre und die Bereitstellung der benötigten Technik.

Vielen Dank auch an die zahlreichen Korrekturleser, die in diversen Zwischenständen Fehler gefunden und berichtigt haben.

Inhaltsverzeichnis

Abbildungsverzeichnis **ix**

Tabellenverzeichnis **xi**

Quelltextverzeichnis **xiii**

1 Einleitung **1**
1.1 Motivation und Problemstellung . . . 1
1.2 Aufgabenstellung und Zielsetzung . . . 2
1.3 Gliederung der Arbeit . . . 3

2 Grundlagen **5**
2.1 Entwurfsmuster . . . 5
2.2 Modellierung . . . 8
2.3 Schnittstelle . . . 10
2.3.1 Objektschnittstelle . . . 10
2.3.2 Anwendungsschnittstelle . . . 12
2.3.3 Workflow . . . 16
2.4 Zusammenfassung . . . 16

3 Anforderungsanalyse **17**
3.1 Ist Analyse . . . 17
3.1.1 Fulfillment-Prozess . . . 17
3.1.2 Problemsituation . . . 21
3.2 Anforderungen . . . 22
3.2.1 Funktionale Anforderungen . . . 22
3.2.1.1 Anbindungsmöglichkeit an eine Middleware . . . 22
3.2.1.2 Definition von Abarbeitungsschritten . . . 22
3.2.1.3 Recovery . . . 23
3.2.1.4 Analysemöglichkeit . . . 23
3.2.1.5 Erweiterbarkeit . . . 23
3.2.2 Nichtfunktionale Anforderungen . . . 24
3.2.2.1 Benutzerfreundlichkeit . . . 24
3.2.2.2 Leistung und Effizienz . . . 24
3.3 Zusammenfassung . . . 24

4 Konzept **25**
4.1 Konzeptstruktur . . . 25

4.1.1 Webservice . . . 27
4.1.2 Workflow . . . 27
4.1.3 Workflow-Manager . . . 29
4.2 Recovery . . . 29
4.3 Analyse . . . 30
4.4 Statusüberwachung . . . 31
4.5 Zusammenfassung . . . 33

5 Implementierung 35
5.1 Struktureller Aufbau . . . 35
5.2 Webservices . . . 38
5.3 Workflow . . . 40
5.4 Activities . . . 42
5.5 Workflow-Manager . . . 44
5.6 Execution-Manager . . . 46
5.7 Persistenter Speicher . . . 47
5.8 Zusammenfassung . . . 50

6 Evaluierung 53
6.1 Validierung . . . 53
6.1.1 Anbindungsmöglichkeiten an eine Middleware . . . 54
6.1.2 Definition von Abarbeitungsschritten . . . 54
6.1.3 Recovery . . . 54
6.1.4 Analysemöglichkeiten . . . 55
6.1.5 Erweiterbarkeit . . . 55
6.1.6 Benutzerfreundlichkeit . . . 56
6.1.7 Leistung und Effizienz . . . 56
6.2 Probleme und grundlegende Erkenntnisse . . . 58
6.3 Zusammenfassung und Fazit . . . 60

7 Zusammenfassung und Ausblick 61
7.1 Zusammenfassung . . . 61
7.2 Ausblick . . . 62

A Anhang 63
A.1 Bestellmodell . . . 64
A.2 Dependency Injection Muster . . . 65
A.3 Delegation Muster . . . 66
A.4 Bestellungsfluss Messungen . . . 67

Literaturverzeichnis 69

Abbildungsverzeichnis

2.1 Klassendiagramm Singleton 6
2.2 Diagrammerstellung 9
2.3 Zusammenhang von Objektschnittstelle, Klasse und Objekt 11
2.4 Beispiel einer Bestellung in XML(links) und JSON(rechts) 14

3.1 Anwendungsfalldiagramm IST-Situation 18
3.2 Sequenzdiagramm 19
3.3 Auftragsstatus 20

4.1 Überblick der Konzeptstruktur 26
4.2 Workflow 28

5.1 Code Map 37
5.2 WCF Test-Client 39
5.3 Workflowbeispiel 41
5.4 ActivityTreeNode 42
5.5 Activity Klassen 43
5.6 Workflowmanager 45
5.7 ExecutionManager 46
5.8 Persistenten Speicher des Prototyps 47
5.9 Datenbankschema 49

6.1 Benchmark 57

A.1 Bestellmodell 64
A.2 Dependency Injection 65
A.3 Delegation 66

Tabellenverzeichnis

3.1 Parameter einer Kontaktlinse 21

6.1 Vorteile und Probleme des Prototyps 59

A.1 Bestellungsfluss Messungen 67

Quelltextverzeichnis

2.1 Singleton Entwurfsmuster in C# 7
2.2 XHTML Aufbau 13
2.3 Webservice in WSDL 15
5.1 Fulfillment Interface 38
5.2 Fulfillment Implementierung 40
5.3 BaseEntity 50

1. Einleitung

Der Begriff der systematischen Wiederverwendung von Software (engl. software reuse) wurde auf der NATO Software Engineering Konferenz im Jahre 1968 eingeführt. [SS07] Das Ziel der Produktivitätssteigerung und Kostenreduzierung hat stätig an Bedeutung gewonnen, sodass versucht werden soll möglichst viele Komponenten einer neuen Software wiederverwenden zu können. [Nar09] Die vorliegende Arbeit beschäftigt sich mit einem konkreten Kundenauftrag zur Optimierung des Arbeitsablaufs des Fulfillments. Vor der Projektrealisierung soll eine Machbarkeitsanalyse (engl. feasibility study) durchgeführt werden, um zu prüfen, ob die Anforderungen es erlauben die Softwareumsetzung nach dem Ansatz der serviceorientierten Architektur umzusetzen. Mit Hilfe der progressiven Abstraktion soll eine automatische Abbildung von beliebigen Geschäftsprozessen realisiert werden.

1.1 Motivation und Problemstellung

Für einen Kunden der SAB[1] soll eine Projektstudie durchgeführt werden. Schwerpunkt der vorliegenden Arbeit ist die technische Realisierung. Es soll geprüft werden, ob eine Implementierung gefunden werden kann, die es erlaubt die entwickelten Komponenten für zukünftige Kunden wiederzuverwenden. Die SAB ist als zukunftsorientiertes Unternehmen bestrebt die Effizienz und Transparenz von geschäftlichen Arbeitsschritten mit gewonnenen Technologien zu verbessern.

Bei dem Kunden handelt es sich um ein Unternehmen aus dem Bereich der Augenoptik. Ziel des Kunden ist die Steigung der Kundenzufriedenheit durch einen neuen Vertriebszweig. Dem Endkunden des Augenoptikers soll es möglich sein, Kontaktlinsen über eine Website beziehungsweise mobiler App beziehen zu können. Zusätzlich soll der Kunde ein Kontaktlinsenabonnement erhalten, wodurch Kontaktlinsen und Zubehör automatisch, in einem selbst gewählten Zyklus, nachbestellt werden.

Zu diesem Zweck wurde vom Augenoptiker eine interne Anwendung entwickelt, welche aktuell in einem Testland zum produktiven Einsatz kommt. Der Kunde erhält

[1] http://www.sab-team.com

dort bereits heute über die App oder Website den zukünftigen Service. Nach erfolgreicher Testphase soll der Service nach und nach in allen vertretenen Ländern des Augenoptikers für den Endkunden bereitgestellt werden.

Eine geringe Planungszeit und fehlende Fachkenntnis bei der Umsetzung der Entwicklung der internen Anwendung führte zu etlichen inkrementellen Optimierungen um die anfänglichen Fehler zu korrigieren. Dadurch wurde die Wartbarkeit der Anwendung in Mitleidenschaft gezogen, was die Zeitintensität der Fehlerkorrektur erhöht hat. Mit den zunehmenden Datenmengen ist die Anwendung für den Mitarbeiter des Auftraggebers sehr träge geworden. Zusätzlich fehlt die Möglichkeit der Skalierung bezogen auf die Parallelität, asynchrones Nachladen oder Caching. In der aktuellen Form der Anwendung ist die Abwicklung der nachgelagerten Bestellprozesse nicht vorhanden, wodurch die Abwicklung der Bestellungen mit Excel-Dateien abgebildet wird, weshalb ein hoher manueller Aufwand, einhergehend mit hohem Zeitaufwand und hoher Fehlerrate entsteht. Es ist notwendig, dass die nachgelagerten Bestellprozesse, genannt Fulfillment, durch definierte Abläufe automatisiert werden, da die täglichen Bestellungen im Testland stetig zunehmen.

> „[Das Fulfillment beschreibt den] gesamten Prozess der Auftragsabwicklung im E-Commerce. Er umfasst Lagerung, Kommissionierung, Transport, Auslieferung und teilweise die Bezahlung der Ware sowie den After-Sales-Service bis hin zur Retourenbearbeitung. Er setzt unmittelbar nach der Bestellung ein.“[2]

Die Arbeitsschritte des Geschäftsprozess sollen austauschbar sein und die Möglichkeit besitzen, bereits existierende Services, beispielsweise aus SAP[3], zu verwenden. Auf Grund der beschriebenen Mängel innerhalb der aktuellen Middleware ist eine Neuimplementierung geplant, jedoch ist aktuell nicht bekannt, wie diese umgesetzt werden soll. Es wird sich aber weiterhin um eine Webanwendung im Intranet handeln.

In der vorliegenden Arbeit soll daher eine kritische Analyse der Möglichkeiten durchgeführt werden, um zu prüfen, wie das gewünschte Fulfillment sowohl mit aktuellen als auch der zukünftigen Middleware betrieben werden kann. Die vollständige Implementierung der einzelnen Arbeitsschritte ist nicht Ziel der Arbeit. Die Machbarkeit einer Implementierung zur Definition und automatischen Abarbeitung allgemeiner Prozesse steht im Fokus. Hier ist die Entwicklung eines Konzepts notwendig, um Standardkomponenten zu ermitteln, damit die Ansätze für zukünftige Kunden wiederverwendet werden können.

1.2 Aufgabenstellung und Zielsetzung

Schwerpunkt der Arbeit ist die Evaluierung und Überprüfung des Kundenauftrags, um die Frage zu beantworten, ob anhand der beauftragten Fulfillment-Automatisierung eine Software entwickelt werden kann, die zugleich für andere Kunden genutzt werden kann.

[2] http://wirtschaftslexikon.gabler.de/Definition/fulfillment.html
[3] http://www.sap.com

Hierzu soll ein Prototyp entwickelt werden, um Probleme bei der Realisierung der Workflow-Automatisierung zu erkennen. Der Workflow stellt einen speziellen Arbeitsablauf dar, bestehend aus einer Vielzahl von Arbeitsschritten, genannt Aktivitäten. Die Funktionalität der Aktivitäten ist nicht Teil dieser Arbeit und soll später an die Anwendungsentwicklung des Augenoptikers übergeben werden. Im Fokus steht die Übergabe der Ergebnisse einer Aktivität an deren Nachfolger, um damit dem automatisiert ablaufenden Workflow zu realisieren. Das zu entwickelnde Workflow-Management-System stellt ein Anwendungssystem zum Definieren, Verwalten und Ausführen des Workflows dar, mit einer definierten Ausführungsreihenfolge.

Die Arbeit soll das Thema der Zusammenfassung unterschiedlicher und unbekannter Objekte in einem Objekttyp aufgreifen und Lösungsansätze aufzeigen. Zusätzlich wird in der Arbeit die Frage behandelt, inwiefern eine Anbindung an eine bestehende Middleware und zugleich an eine zukünftige unbekannte Middleware erfolgen kann. Es soll eine möglichst universelle Schnittstelle gefunden werden.

Ziel der Arbeit ist es, zu überprüfen ob ein Lösungsansatz gefunden werden kann, der zum einen die aktuelle Fulfillment-Situation im Testland optimiert und zum anderen als wiederverwendbare Workflow-Management-System Lösung etabliert werden kann. Es soll zusätzlich evaluiert werden, ob der Prototyp zur Machbarkeitsanalyse bereits als Alphaversion für die Projektrealisierung dienen kann.

1.3 Gliederung der Arbeit

Die Arbeit ist in sieben Kapitel unterteilt; auf die Einleitung folgen die Grundlagen in Kapitel 2. Da es sich bei dieser Arbeit um eine technische Machbarkeitsanalyse handelt, bezieht sich der Grundlagenteil hauptsächlich auf den technischen Hintergrund der Wiederverwendbarkeit. Kapitel 3 erläutert die spezifische Kundensituation und extrahiert daraus die notwendigen Anforderungen für eine geplante Umsetzung. Die ermittelten Anforderungen sollen zum Konzeptentwurf des Prototyps beitragen. Das Konzept unseres Workflow-Management-Systems wird in Kapitel 4 ausführlich vorgestellt. Die notwendigen Komponenten stützen sich auf die eruierten Anforderungen. Mit der Implementierung in Kapitel 5 soll validiert werden, ob eine Realisierung möglich ist und ob das vorgestellt Konzept Fehler aufweist. Der Prototyp soll bereits so konzipiert werden, dass theoretisch beliebige Geschäftsprozesse abgebildet werden können. Darauf folgen in Kapitel 6 die Validierungen, ob unsere Workflowlösung die Anforderungen des spezifischen Kundenproblems lösen, und die Diskussion der aufgetretenen Probleme. Hierbei wird kritisch untersucht, ob die Machbarkeit bestätigt werden kann. Abschließend wird im Kapitel 7 die Arbeit zusammengefasst und ein Ausblick auf weiterführende Arbeiten gegeben.

2. Grundlagen

In diesem Kapitel wird ein Einblick in die Umsetzung einer wiederverwendbaren Software gegeben. Hierzu soll der Begriff des Entwurfsmusters erläutert werden, um Entwurfsprobleme zu vermeiden und eine wiederverwendbare Softwareentwicklung gewährleistet. Modellierungskonzepte sollen diesen Entwicklungsprozess unterstützen. Um die Flexibilität zu gewährleisten und um Kommunikationsmöglichkeiten mit externen Systemen zu gewährleisten sollen Schnittstellen erläutert werden. Eine kurze Definition des Workflow-Begriffes für die vorliegende Arbeit soll dieses Kapitel abschließen.

2.1 Entwurfsmuster

Um die Wiederverwendbarkeit und gewünschte Anbindung an ein zukünftiges System (siehe Abschnitt 1.2) zu gewährleisten empfiehlt sich die Verwendung von bewährten Mustern. Für die Softwarearchitektur existieren bestimmte Entwurfsmuster, welche verwendet werden sollen um einen Entwurf schneller zum Erfolg zu führen und fehlerhafte Entscheidungen zu minimieren. [GHJV00]

Entwurfsmuster (engl. design patterns) sind zuverlässige Lösungsschablonen zur Schaffung von Flexibilität. Entwurfsmuster abstrahieren, identifizieren und benennen relevante objektorientierte Entwurfsprobleme in der Softwarearchitektur und Softwareentwicklung. Die Autoren Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides, genannt Gang of Four, haben die Standardreferenz zur Thematik Entwurfsmuster erarbeitet und definieren es wie folgt: „Entwurfsmuster sind Beschreibungen zusammenarbeitender Objekte und Klassen, die maßgeschneidert sind, um ein allgemeines Entwurfsproblem in einem bestimmten Kontext zu lösen“. [GHJV00]

Die Autoren beschreiben ein konkretes Entwurfsmuster durch den *Musternamen*, *Problemabschnitt*, *Lösungsabschnitt* und *Konsequenzabschnitt*. Die einheitliche Grundstruktur erlaubt einen Vergleich untereinander und unterstützt den Entwickler bei der Auswahl. Der *Musternamen* definiert den Namen des Entwurfsmusters für eine erhöhte Abstraktion und Schaffung eines eineindeutigen Verständigungsvokabulars

unter Entwicklern. Im *Problemabschnitt* wird die Problemsituation des Musters genannt, in der es angewendet werden soll. Darin werden spezifische Probleme beschrieben und oft Klassen- oder Objektstrukturen vorgegeben, um das Muster sinnvoll anzuwenden. Der *Lösungsabschnitt* zeigt die abstrakte Lösung des Entwurfsproblems und gibt Beziehungen, Zuständigkeiten und Interaktionen von Klassen und Objekten vor. Im *Konsequenzabschnitt* werden die Vorteile und Nachteile des Musters aufgeschlüsselt. Hierin wird hauptsächlich auf die Ausführungszeit, Flexibilität und die Erweiterbarkeit eingegangen.

Am Beispiel eines Singleton wird im folgenden die Struktur eines Entwurfsmusters verdeutlicht.

Mustername: Singleton

Problemabschnitt: Es darf nur ein Objekt zu einer Klasse existieren.

Lösungsabschnitt: Die Klasse hält ein privates statisches Attribut der Instanz. Über eine Instanzoperation `getInstance()` wird das Objekt erstmalig erzeugt oder das Bestehende verwendet und zurückgegeben. Die Abbildung 2.1 zeigt ein Klassendiagramm einer Singleton-Klasse nach der UML Notation[1]. Der Name der Klasse befindet sich fett gedruckt im oberen Drittel und heißt zur Veranschaulichung `Singleton`. Darunter folgt die Angabe der Attribute: ein Singleton benötigt eine Variable des eigenen Typs, die einzige Instanz dieser Klasse. Im unteren Abschnitt werden die notwendigen Operationen dargestellt, der Standardkonstruktor `Singleton()` welcher überschrieben wird, sowie die Methode zum Erhalt der einzigen Klasseninstanz.

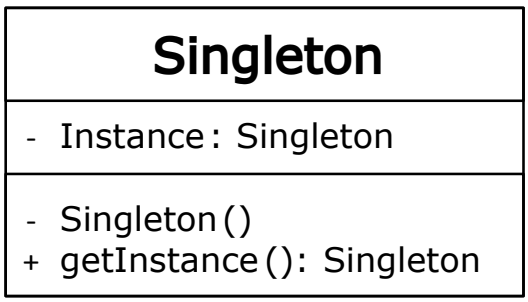

Abbildung 2.1: Klassendiagramm Singleton

Konsequenzabschnitt:
Vorteile:

- Es kann eine Zugriffskontrolle erfolgen.
- Die Einzelinstanz wird nur erzeugt, wenn sie benötigt wird.
- Spezialisierung durch Unterklassenbildung möglich.
- Ein Singleton in mehrere Instanzen zu überführen ist einfacher, als bei einer globalen Variable.

[1] http://www.uml.org

Nachteile:

- Inflationäre Verwendung kann zu einer prozeduralen statt objektorientierten Programmierung führen.
- Aus dem Interface geht nicht hervor, dass es sich um einen Singleton handelt.
- Technisch kann ein Singleton in Bezug auf verteilte Systeme oder parallelen Ausführungen fehlerhaft von einer Programmiersprache umgesetzt worden sein.

Eine mögliche Realisierung des Entwurfsmusters anhand des Klassendiagramms aus Abbildung 2.1 auf der vorherigen Seite stellt die Abbildung Quelltext 2.1 aus der Microsoft MSDN Bibliothek dar. [Mic15b] Es zeigt die Klasse `Singleton` mit einer internen privaten Variablen `_instance` vom Typ der Klasse, welche die einzige Instanz dieser Klasse repräsentiert. Der Standardkonstruktor `Singleton()` muss daher als privat deklariert werden, da sonst beliebig viele Instanzen erzeugt werden können. Der Konstruktor kann dadurch nur noch von einer Methode der Klasse aufgerufen werden um die erste und einzige Instanz zu erstellen. Diese Funktionalität ist in `GetInstance` enthalten. Die Methode prüft, ob `_instance` eine Instanz zugewiesen wurde, ist dies der Fall wird die Instanz zurückgegeben, andernfalls wird `_instance` eine Instanz der Klasse zugewiesen. Dadurch ist sichergestellt, dass nur eine einzige Instanz der Klasse existiert.

```
1  /// <summary>
2  /// Sample implementation of the singleton design pattern.
3  /// </summary>
4  public class Singleton
5  {
6      /// <summary>
7      /// The one and only instance of the singleton class.
8      /// </summary>
9      private static Singleton _instance;
10
11     /// <summary>
12     /// Private constructor, create new instance is not available.
13     /// </summary>
14     private Singleton(){}
15
16     /// <summary>
17     /// Instance as static property.
18     /// </summary>
19     public static Singleton GetInstance
20     {
21         get{ return _instance ?? (_instance = new Singleton()); }
22     }
23 }
```

Quelltext 2.1: Singleton Entwurfsmuster in C#

Das Entwurfsmuster soll dem Entwickler zu Beginn eines Projekts helfen, die Aufgaben der zu entwerfenden Software in kleinere Funktionalitätsabschnitte aufzuteilen.

Die einzelnen Bestandteile können danach in Klassen und Objekte umgewandelt werden. Entwickler neigen dazu, zu früh in die Programmierung einzusteigen, wodurch die Planungsphase und Auseinandersetzung mit bestehenden Entwurfsmustern oft zu kurz kommt. Dadurch bildet die entstehende Software nur eine Lösung zur aktuellen Problemsituation ab. Erst durch die Abstraktion von Klassen und Objekten ist es möglich, dass die entwickelte Software zukünftigen Problemen gewachsen ist und die notwendigen Anpassungen gering bleiben. [GHJV00]

2.2 Modellierung

Da die geplante Software das Potenzial haben soll unzählige Erweiterungen und Weiterentwicklungen zu unterstützen, ohne bei exzessiver Nutzung selber zum Flaschenhals zu werden, muss gerade der strukturelle Unterbau und die für zusätzliche Erweiterungen definierten Schnittstellen sehr gut geplant und Dokumentiert sein.

Die Erstellung verschiedener Diagramme versetzt den Designer immer wieder in unterschiedliche Perspektiven und erweitert mit jedem Perspektivwechsel das Gesamtbild des Projektes. Oft werden Diagrammtypen falsch oder zu spät eingesetzt, sodass bestimmte Diagramme erst während des Projekts vervollständigt werden.

Bei der Erstellung vieler Diagramme fällt schnell auf, dass es Abhängigkeiten gibt, sodass die Erstellung in einer bestimmten Reihenfolge den Prozess optimieren könnte. In dem Buch *Model-Driven Software Development*, unter anderen von Sami Beydeda, wird eine Modellierungsreihenfolge empfohlen die inkrementell Diagramme auf vorher erstellte Diagramme aufbauen lässt. Das Datenflussdiagramm Abbildung 2.2 auf der nächsten Seite aus genanntem Werk, visualisiert die logische Abfolge der Diagrammerstellung. Beim Befolgen dieser Abfolge bauen alle Diagramme aufeinander auf und erleichtern somit den Design-Prozess. Im Folgenden wird der Ablauf mit den entstehenden Diagrammen erläutert.

Nach dem *Schritt a*, der Erfassung der Anforderungen in Abschnitt 3.2 ist das erste zu erstellende Diagramm in *Schritt b* ein UML Anwendungsfalldiagramm (engl. use case diagram). Es zeigt die verschiedenen Systeme und Akteure und wie diese miteinander interagieren aber auch welche Abhängigkeiten untereinander bestehen. Es definiert das erwartete Verhalten und spezifiziert die Anforderungen an das System. Letztendlich kann überprüft werden, ob alle benötigten Features in dem fertigen Produkt umgesetzt worden sind.

Aufbauend auf das Use-Case Diagramm kann die Struktur entworfen werden. Ausgehend von den Systemen und Akteuren wird im *Schritt d* ein erstes Objekt Diagramm nach der UML erstellt. Es beinhaltet alle Objekte, die für den Ablauf notwendig sind in einer ähnlichen Darstellung wie das Klassendiagramm, jedoch können bei Objektdiagrammen einzelne Instanzen einer Klasse oder eines Typs dargestellt werden, inklusive dessen Attribute. Die Objektbeziehungen in diesem Diagrammtyp folgen keinem vorgegebenen Format und können deshalb so gestaltet werden wie es benötigt wird. Dieses Objektdiagramm dient als erste Übersicht über alle nötigen Objekte und kann noch Dopplungen oder Beispielwerte enthalten um alle Funktionalitäten abzubilden. In den folgenden Schritten wird dieses Diagramm weiter geschärft und generalisiert, damit schlussendlich ein präzises und optimales Objektdiagramm und

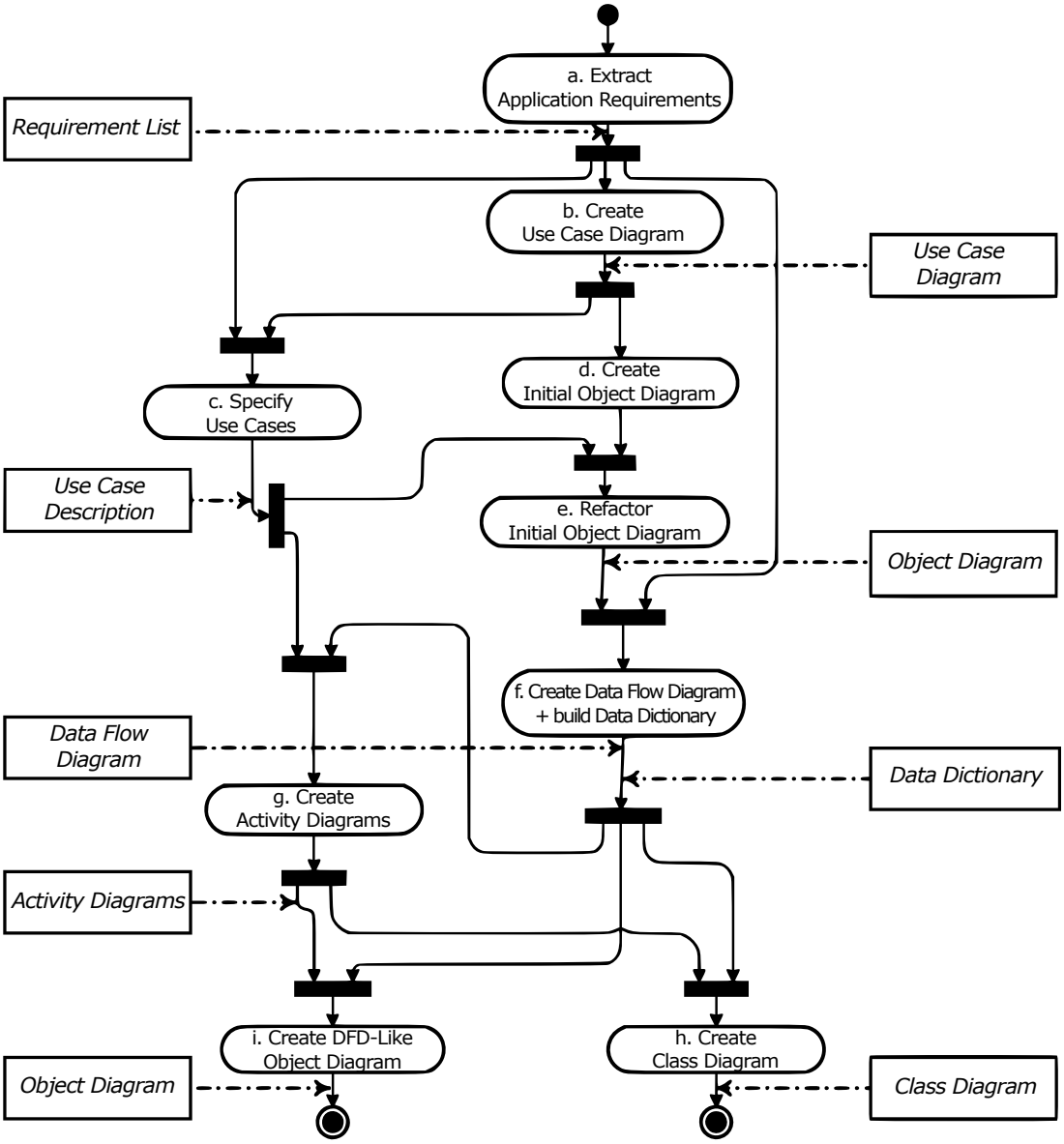

Abbildung 2.2: Diagrammerstellung

zusätzlich noch ein Klassendiagramm entstehen können. Der Vorteil der beiden Diagramme ist, dass sie bei der Implementierung nur einen minimalen Spielraum bieten und das Konzept des Projektes veranschaulichen.

Nach dem ersten Entwurf des Objektdiagramms wird es im *Schritt e* überarbeitet, in dem Objekte gruppiert, aufgeteilt oder gelöscht werden. Das heißt, dass alle unnötigen Objekte entfernt werden, ähnliche zusammengefasst oder falls eine Differenzierung nötig ist, aufgeteilt werden. Außerdem wird es weitestgehend generalisiert, falls es noch nicht in dem initialen Objektdiagramm geschehen ist. Im Laufe der Modellierung wird es weiter optimiert und schlussendlich um die Informationen aus dem Datenflussdiagramm ergänzt.

Ausgehend von dem Objektdiagramm wird in *Schritt f* ein Datenflussdiagramm (engl. Data Flow Diagramm) erstellt. Dieses gehört im Gegensatz zu den vorherigen Diagrammen weder zur UML noch ist es standardisiert, weshalb verschiedene No-

tationen möglich sind. Das Datenflussdiagramm zeigt den Datenfluss zwischen Datenspeichern, externen Schnittstellen und Prozessen. Hieraus ergeben sich erstmals grundlegende Datenflüsse und Beziehungen zwischen den Systemen oder Objekten. Wie auch das Objektdiagramm dient das Datenflussdiagramm als erste Übersicht und wird in den folgenden Prozessen weiter präzisiert.

Ergänzend wird ein Datenkatalog (engl. data-dictionary) erstellt, welcher alle Objekte bis auf die Datentypebene definiert. Es fängt üblicherweise mit dem Entitätsbezeichner an, gefolgt von einer Beschreibung. Die Eigenschaften werden aufgelistet und, falls zu diesem Zeitpunkt möglich, mit Datentypen versehen.

Nach dem sowohl die Objekte in dem Objektdiagramm, der Datenfluss in dem Datenflussdiagramm und der Anwendungsfall bekannt sind, lässt sich in *Schritt g* ein Aktivitätsdiagramm erstellen. Es gehört zu den Verhaltensdiagrammen des UML und eignet sich zur Modellierung von Aktivitäten innerhalb eines Systems. Hierbei können alle Aktionen und Daten- oder Objektflüsse dargestellt werden. Selbst Objektzustände, Verzweigungen mit Bedingungen und Parallelitäten können mit einem Aktivitätsdiagramm dargestellt werden. Das Gesamtbild wird durch die Abfolge der einzelnen Aktivitäten und die dabei verwendeten Objekte erweitert.

Ausgehend vom Objektdiagramm und den gewonnen Informationen durch das Aktivitätsdiagramm kann nun das Klassendiagramm in *Schritt h* erstellt werden. Es ist für die Umsetzung das Wichtigste und Nützlichste, da es alle wichtigen Strukturelemente enthält. Dazu gehören neben den Klassen auch die Schnittstellen und deren Beziehungen. Die Klassen sind in objektorientierten Sprachen der Oberbergriff für Strukturen und Objekte und sind damit der Grundstein der zukünftigen Software. Das erstellte Klassendiagramm kann in Visual Studio bei eingehaltener Benennungskonventionen des Codes beim Kompilieren überprüfen, ob der Code dem Klassendiagramm entspricht, oder an welchen Stellen er davon abweicht. Die somit erkannten Abweichungen regen einerseits dazu an sich nach dem geplanten Diagramm zu richten, andererseits zeigt es auch mögliche Fehler in dem Diagramm auf. Im Kapitel 5 wird zu jeder entwickelten Komponente das dazugehörige Klassendiagramm gezeigt. [BBG05, GHR+14]

2.3 Schnittstelle

Im Folgenden soll auf die Schnittstellen eingegangen werden. Hierbei soll sowohl auf die Objektschnittstellen als auch auf die Anwendungsschnittstelle eingegangen werden. Beide Schnittstellen definieren einen Vertrag und geben damit Services zur Interaktion vor. Die Objektschnittstelle definiert Typen innerhalb des Framework-Prototyps und die Anwendungsschnittstelle definiert die Kommunikation nach außen.

2.3.1 Objektschnittstelle

Anhand erstellter Anwendungsfälle (engl. use cases) und der Kenntnis bewährter Entwurfsmuster können Objektschnittstellen (engl. interfaces) definiert werden. Eine Objektschnittstelle bezeichnet die Gesamtheit aller Signaturen, bestehend aus einer Bezeichnung, einem Rückgabewert und beliebig vielen Parametern, wobei auch kein

Parameter zulässig ist. Es ist keine Implementierung enthalten, sondern lediglich die Signatur, die Vorgabe für den Aufbau einer Methode in einer Klasse. Somit wird der Typ eines zur Laufzeit erstellten Objekts definiert. Ein Objekt ist eine Instanz einer Klasse und kann somit nicht direkt über die Objektschnittstelle erzeugt werden, sondern immer nur über eine Klasse, welche von dieser Schnittstelle erbt. Die Klasse kann beliebig viele Objektschnittstellen (Typen) implementieren und erfüllt den vorgeschriebenen Vertrag in Form von Signaturen für die zu implementierenden Methoden.

Die Abbildung 2.3 visualisiert den Zusammenhang zwischen Typ, Klasse und Objekt. Ein Objekt erhält die Implementierung von der Klasse und den Typ von der Objektschnittstelle. Dadurch ist gewährleistet, dass ein Objekt eines vorgegebenen Typs die gewünschten Funktionalitäten anbietet. Es wird keinerlei Logik von der Objektschnittstelle übergeben, wie es bei der Vererbung der Klassen untereinander der Fall ist. Somit können zwei Objekte des gleichen Typs komplett unterschiedliche Ergebnisse liefern. Dies Bedeutet auch, dass Objekte einfach austauschbar sind, solange die erforderliche Objektschnittstelle identisch ist. Dadurch kann das Konzept der Polymorphie vorteilhaft angewendet werden. Durch diese Techniken wird die Wiederverwendbarkeit des Quellcodes, sowie dessen Flexibilität erhöht, da die Implementierung eines Objekts einfach ausgetauscht werden und die Anwendung an veränderte Anforderungen leicht angepasst werden kann. Die Abhängigkeiten können auch erst zur Laufzeit bestimmt werden, zum Beispiel durch die Verwendung des Dependency Injection Musters. [GHJV00] Die Erklärung dieses Entwurfsmusters befindet sich im Abschnitt A.2.

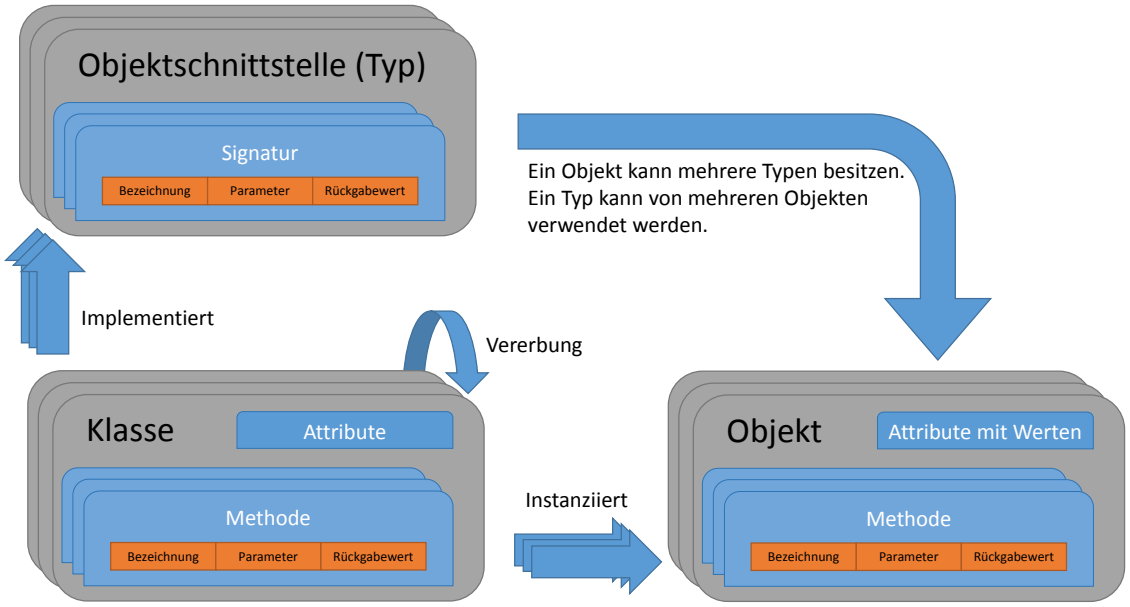

Abbildung 2.3: Zusammenhang von Objektschnittstelle, Klasse und Objekt

Nachdem Objektschnittstellen definiert sind, können dazugehörige Klassen implementiert werden. Klassischerweise entsteht eine Klassenhierarchie mit Unterklassen, welche die Implementierung sowie den Typ der Oberklasse erben und die Funktionalität erweitern oder modifizieren. Der Klassenhierarchieaufbau kommt der menschlichen Vorstellung von realen Objekten sehr nahe, wodurch die Umsetzung verein-

facht wird. Diese Technik wird als White-Box-Wiederverwendung bezeichnet, da die Struktur der Oberklasse eingebunden wird und somit der Unterklasse bekannt ist. Dadurch kann die Datenkapselung (engl. information hiding), das Verbergen von Informationen, nicht vollumfänglich eingehalten werden. Die Vererbung führt zu starken Abhängigkeiten, welche zur Übersetzungszeit (engl. compile time) festgelegt ist und somit während der Laufzeit nicht geändert werden kann. Hierdurch müssen Anpassungen oft in der Oberklasse vorgenommen werden oder eine zusätzliche Klassen eingeführt werden, dies schränkt die Flexibilität und Wiederverwendbarkeit ein.

Demgegenüber steht die Objektkomposition, hierbei werden bestehende Objekte als Referenz oder Aggregation in anderen Objekten verwendet. Diese können zur Laufzeit dynamisch festgelegt werden, da nur die Objektschnittstellen bekannt sind, spricht man auch von Black-Box-Wiederverwendung. Dadurch kann jedes Objekt ausgetauscht werden, wenn es den gleichen Typ besitzt. Die Datenkapselung bleibt mit dieser Technik vollständig erhalten, da keine Informationen der Implementierung nach Außen gelangen. Um die Vorteile der Vererbung, den Zugriff auf die Methoden der Oberklasse, zu realisieren wird das Entwurfsmuster *Delegation* genutzt. Die Erklärung des Entwurfsmusters befindet sich im Abschnitt A.3.

Die Delegation als Vererbungsalternative übergibt die Instanz (das Delegationsobjekt) als Reference an ein Empfänger-Objekt. Das Empfänger-Objekt kann auf die übergebene Referenz zugreifen und die Methoden vom Delegationsobjekt verwenden. Dadurch kann auch Mehrfachvererbung in Programmiersprachen umgesetzt werden, die selbst keine Mehrfachvererbung erlauben. Das Delegationsobjekt entspricht somit der Oberklasse und hält die notwendigen Methoden zur Ausführung bereit. Das Empfänger-Objekt kann somit die Aufgaben weiter delegieren. Dieses Schema wird auch in anderen Entwurfsmustern wie Besucher (engl. visitor), Beobachter (engl. observer) oder Strategie (engl. stategy) eingesetzt. Insgesamt ist das Muster komplizierter und erfordert einen höheren Lernaufwand als Klassenvererbung, zudem kann es zu Laufzeitineffizienzen kommen. Das ist der Flexibilität geschuldet, da die Objekte zur Laufzeit keine Aggregationsbeziehungen vorweisen, sondern lediglich eine Bekanntschaft. Dadurch ist nicht sicher gestellt, dass beide Objekte, Delegator und Empfänger zur Laufzeit gleich lange existieren. Es kann also vorkommen, dass ein Delegationsobjekt öfter erstellt und zerstört wird, was die Dauer der Operationen erhöht. [GHJV00, tut15]

2.3.2 Anwendungsschnittstelle

Um eine Kommunikation mit der Middleware herzustellen ist eine Schnittstelle zur Anwendungsprogrammierung (engl. application interface) kurz API notwendig.

> „Eine der zeitaufwendigsten Herausforderungen für Entwickler ist der Datenaustausch zwischen inkompatiblen Systemen über das Internet.“ [2]

Im Umfeld des *World Wide Web* (kurz WWW) empfiehlt sich eine Technologie die zum Transport das *Hypertext Transfer Protocol* (kurz HTTP), auf Grund der

[2] http://www.w3schools.com/xml/xml_usedfor.asp

hohen Verbreitung, verwendet. Daher soll kurz auf Webservice APIs eingegangen werden. In diesem Zusammenhang ist die *EXtensible Markup Language* (kurz XML) zu nennen, welche als Meta-Seitenbeschreibungssprache für die Datenübertragung im WWW optimiert wurde.

Lediglich die Baumstruktur mit einer einzigen Wurzel sowie der Prolog, die erste Zeile zur XML Deklaration, sind vorgegeben. Abbildung 2.4 auf der nächsten Seite zeigt eine vereinfachte Bestellung im XML Format auf der linken Seite. Der Prolog definiert die XML-Version mit 1.0 und die weit verbreitete Unicode-Zeichenkodierung UTF-8. Die Wurzel `<orders>` enthält mehrere Unterelemente, welche selbst wieder beliebig viele Elemente enthalten können. Jedes Element hat einen öffnenden (`<order>`) sowie schließenden (`</order>`) Tag in gleicher Groß- und Kleinschreibung. Der Tag-Name darf nur mit einem Buchstaben oder Unterstrich beginnen (außer dem Bezeichner `xml`), zusätzlich dürfen nur Buchstaben, Zahlen, Bindestriche, Unterstriche und Punkte verwendet werden. Zusätzlich kann jedes Tag mehrere Attribute in Anführungsstrichen (") enthalten (z.B. `<amount type="int">`), die Informationen in den Attributen sollten idealerweise kein Teil der zu übertragenden Daten darstellen, sondern einen optionalen Beschreibungscharakter vorweisen. Zwillings-Elemente (Elemente auf gleicher Baumebene) können unterschiedlich viele Unterelemente besitzen. Sie sind in ihrer Menge dehnbar, dies erlaubt eine einfache Erweiterung (zum Beispiel um `<acadamic_title>` in `<customer>`). Die Charakteristiken der Sprache sind optimal für eine Anwendung, welche flexibel und erweiterbar sein muss. Weist ein XML Dokument eine korrekte Syntax auf spricht man von *Well Formed*, kann es zudem gegen eine Dokumenttypdefinition (kurz DTD) validiert werden spricht man zusätzlich von *Valid*. Die Dokumenttypdefinition ist zwischen Prolog und Rootelement des XML Dokuments enthalten, Quelltext 2.2 veranschaulicht dies am Beispiel von *XHTML*, einem Vertreter der festgelegten XML Struktur. [Mic04b]

```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
3 "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
4 <html xmlns="http://www.w3.org/1999/xhtml">
```

Quelltext 2.2: XHTML Aufbau

Durch die flexible Datengestaltung hat XML sich als ideales Format zur Speicherung beliebiger Informationen etabliert und erlaubt durch die Standardisierung den Austausch zwischen verschiedenen Systemen. XML wird oft als schwergewichtige Sprache beschrieben, da jeder Tag einen schließenden Tag mit dem gleichen Namen benötigt. In Abbildung 2.4 auf der nächsten Seite ist auf der rechten Seite das kompaktere Datenformat JSON (JavaScript Object Notation) dargestellt. Es enthält alle Eigenschaften von XML in einer kürzeren Schreibweise. Es entsteht kein Informationsverlust bei der Übertragung der Daten in das JSON Format. Das Datenformat gewinnt in den letzten Jahren immer mehr an Bedeutung, sodass Microsoft die eigenen Technologien, welche auf XML basieren kontinuierlich zu JSON überführt. Die meisten Technologien werden aber auch in Zukunft weiterhin auf das XML-Format aufbauen, da eine Änderung der Dokumentenbeschreibung bei standardisierten Verfahren zu Problemen mit älteren Programmen führen kann.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<orders>
  <order>
    <number>1</number>
    <date>
      <year>2015</year>
      <month>02</month>
      <day>24</day>
    </date>
    <customer>
      <title>Herr</title>
      <firstname>Max</firstname>
      <lastname>Mustermann</lastname>
      <street>Musterweg 7</street>
      <zip>12345</zip>
      <city>Musterstadt</city>
    </customer>
    <articles>
      <article>
        <label>M1</label>
        <amount>3</amount>
        <price>2,99</price>
      </article>
      <article>
        <label>M2</label>
        <amount>1</amount>
        <price>19,99</price>
      </article>
      <article>
        <label>M3</label>
        <amount>1</amount>
        <price>14,95</price>
      </article>
    </articles>
  </order>
  <order>
    <number>2</number>
    <date>
      <year>2015</year>
      <month>02</month>
      <day>24</day>
    </date>
    <customer>
      <title>Frau</title>
      <firstname>Erika</firstname>
      <lastname>Mustermann</lastname>
      <street>Musterweg 7</street>
      <zip>12345</zip>
      <city>Musterstadt</city>
    </customer>
    <articles>
      <article>
        <label>M1</label>
        <amount>1</amount>
        <price>2,99</price>
      </article>
    </articles>
  </order>
</orders>
```

```json
{
 "orders": {
  "order": [
   {
    "number": "1",
    "date": {
     "year": "2015",
     "month": "02",
     "day": "24"
    },
    "customer": {
     "title": "Herr",
     "firstname": "Max",
     "lastname": "Mustermann",
     "street": "Musterweg 7",
     "zip": "12345",
     "city": "Musterstadt"
    },
    "articles": {
     "article": [
      {
       "text": "M1",
       "amount": "3",
       "price": "2,99"
      },
      {
       "text": "M2",
       "amount": "1",
       "price": "19,99"
      },
      {
       "text": "M3",
       "amount": "1",
       "price": "14,95"
      }
     ]
    }
   },
   {
    "number": "2",
    "date": {
     "year": "2015",
     "month": "02",
     "day": "24"
    },
    "customer": {
     "title": "Frau",
     "firstname": "Erika",
     "lastname": "Mustermann",
     "street": "Musterweg 7",
     "zip": "12345",
     "city": "Musterstadt"
    },
    "articles": {
     "article": {
      "text": "M1",
      "amount": "1",
      "price": "2,99"
     }
    }
   }
  ]
 }
}
```

Abbildung 2.4: Beispiel einer Bestellung in XML(links) und JSON(rechts)

Die *Webservice Description Language* (kurz WSDL), ein in XML geschriebenes Dokument, dient zur Beschreibung von Webservices. Es wurde von der W3C[3] als Empfehlung ausgesprochen, weshalb es kurz erläutert wird.

```
1  <message name="getTermRequest">
2    <part name="term" type="xs:string"/>
3  </message>
4
5  <message name="getTermResponse">
6    <part name="value" type="xs:string"/>
7  </message>
8
9  <portType name="glossaryTerms">
10   <operation name="getTerm">
11     <input message="getTermRequest"/>
12     <output message="getTermResponse"/>
13   </operation>
14 </portType>
15
16 <binding type="glossaryTerms" name="b1">
17    <soap:binding style="document"
18    transport="http://schemas.xmlsoap.org/soap/http" />
19    <operation>
20      <soap:operation soapAction="http://example.com/getTerm"/>
21      <input><soap:body use="literal"/></input>
22      <output><soap:body use="literal"/></output>
23   </operation>
24 </binding>
```

Quelltext 2.3: Webservice in WSDL

Quelltext 2.3 zeigt den Aufbau einer Webservice Beschreibung. Es enthält die Verbindungsdaten und Methodendefinitionen innerhalb des `<portType>` Tags. Die Bindings definieren das Nachrichtenformat und die Protokolldetails, der `type` bezieht sich dabei auf den `<portType>`. Für jede Operation muss eine Encodierung der Eingabe und Ausgabe angegeben werden. Der `<portType>` definiert den Namen `glossaryTerms` mit der enthaltenen Operation `getTerm`. Die Operation enthält eine Eingangsnachricht `getTermRequest` und eine Ausgangsnachricht `getTermResponse`. In den Nachrichten werden die Daten zur Kommunikation beschrieben, in beiden Fällen ist der Parameter vom Typ string. Zusätzlich enthält WSDL das Binding zum Übertragungsprotokoll *SOAP*, welches als industrieller Standard die W3C Recommendation (Empfehlung als Web Standard) erfüllt. Das Simple Object Access Protocol (kurz SOAP) basiert auf dem XML-Format, die zugrundeliegenden Daten hingegen können in verschiedenen Formaten wie *CSV* (comma-separated values) oder *XML* vorliegen. Im Beispiel wird `xmlsoap` verwendet und als Übertragungsprotokoll `HTTP`. Zusätzlich zur Übertragungsmöglichkeit kann noch der WSDL-Style beim SOAP-Binding angegeben werden: *RPC* (encoded/literal) oder *Document* (encoded/literal). [But05]

RPC bedeutet, dass die SOAP Nachricht einen Methodenaufruf in XML darstellt, woraus eine Struktur mit Methodennamen und Parameter generiert werden kann.

[3] http://www.w3.org

Document bedeutet, es handelt sich um ein XML Dokument zur Übertragung von Daten, welches eine Validierung durch ein vordefiniertes XML Schema erlaubt.

Da es sich um einen Transport von Bestelldaten handelt und eine Validierung ermöglicht werden soll, wird Document verwendet. Bei beiden WSDL-Styles kann zusätzlich zwischen encoded und literal entschieden werden.

SOAP Encoding stellt eine Erweiterung des SOAP Frameworks da und gibt an, wie Datenwerte im XML-Format codiert werden soll. Literal hingegen ist eine einfache Definition, welche die Serialisierung nach einem austauschbaren Schema erlaubt. Literal erhält den Vorzug, da encoded gegen die Richtlinien der *Webservices Interoperability Organisation* (kurz WS-I) verstößt und somit nicht als nachhaltig gilt. [w3s15]

2.3.3 Workflow

Dieser Abschnitt beschäftigt sich mit der Definition des Workflowbegriffes innerhalb dieser Arbeit. Ein Workflow ist die computergestützte Ausführung von wiederkehrenden Geschäftsprozessen, mit definierter Reihenfolge. Ziel ist die Optimierung, Beschleunigung und Einhaltung des korrekten Ablaufs von Geschäftsprozessen. Die einzelnen Geschäftsaufgaben und Beziehungen zueinander innerhalb des Geschäftsprozesses werden modelliert und durch ein Software-System ausgeführt. Hierbei wird sich auf die technischen und Ablaufs organisatorischen Aspekte bezogen, wodurch der Datenfluss durch die einzelnen Abarbeitungsschritte beschrieben wird. Der Workflow besteht aus einzelnen Aktivitäten, welche in einem Ablaufmodell spezifiziert werden. Um den Workflow zu modellieren ist ein Workflow-Management-System notwendig, dies erlaubt die Erstellung und Anpassung der Abarbeitungsschritte und deren Zuordnungen. Zu den Aufgaben des Workflow-Management-System zählt die Steuerung und das Monitoring der Prozesse.

Durch den Einsatz einer Workflow Automatisierung soll ein kontrollierter, optimierter Daten- und Informationsfluss gewährleistet werden. Zeit- und Ressourcen-Einsparungen reduzieren interne Kosten bei der Abarbeitung von Geschäftsprozessen. Durch die Wiederverwendbarkeit von Aktivitäten und der automatisierten Ausführung kann die Durchlaufzeit verringert und die Produktivität gesteigert werden. [TG03, Wei98, FG08]

2.4 Zusammenfassung

Dieses Kapitel hat Einblicke und Denkansätze vorgestellt, die für die Planung und Realisierung einer wiederverwendbaren Software notwendig sind. Angelehnt an die Grundzüge der serviceorientierten Architektur soll Funktions-, Daten- und Anwendungsschicht vereint werden um die Realisierung eines Geschäftsprozesses zu gewährleisten. Die Verwendung existierender Entwurfsmuster hilft bei der Entwicklung und Softwareintegration. Standards im Schnittstellenbereich fördern die Wartung und reduzieren die Integration von Software in bestehende Anwendungssysteme. Die Ansätze erlauben eine Abstrahierung eines spezifischen Kundenauftrags zur Lösung eines allgemeinen Problems.

3. Anforderungsanalyse

Im vorigen Kapitel wurden die Grundlagen beschrieben, die für die Umsetzung einer wiederverwendbaren Software notwendig sind. Mit der Verwendung von Entwurfspattern und dem Einsatz von Modellen zur Planung hat sich eine Erfolgsmethode (engl. best practice) für die Softwareentwicklung ergeben. Aufbauend auf dieser Erfolgsmethode soll innerhalb dieses Kapitels die aktuelle Situation des Augenoptikers eruiert werden. Die Erfassung der aktuellen Fulfillment-Situation soll die Abläufe und enthaltenen Probleme aufzeigen. Anhand dieser Erkenntnisse soll mit Hilfe der progressiven Abstraktion ein Lösungsentwurf entwickelt werden, welcher versucht ein allgemeineres Problem zu lösen um für zukünftige Kunden mit ähnlichen Problemen eine Standardlösung vorzuhalten.

3.1 Ist Analyse

Für die Endkunden des Augenoptikers soll zukünftig ein Kontaktlinsenabonnement geschaffen werden, mit dem der Kunde die Möglichkeit erhält, sich regelmäßig seine Kontaktlinsen per App oder Web nach Hause liefern zu lassen. Der neue Service führt erstmalig zur Abwicklung von Online-Bestellungen für den Auftraggeber, daher hat dieser noch keine automatisierte Fulfillment-Lösung im Einsatz. Durch die Ausdehnung der Testumgebung auf den Produktiveinsatz in einem Testland wird der Einsatz notwendig. Da täglich reale Bestellungen erzeugt und abgewickelt werden müssen.

3.1.1 Fulfillment-Prozess

Zur Erfassung und Beschreibung des aktuellen Fulfillment-Prozesses werden drei Diagrammtypen genutzt.

- Das Anwendungsfalldiagramm (engl. use case diagram) zur Visualisierung der groben Struktur, der beteiligten Akteure und der Systeminteraktionen.
- Das Sequenzdiagramm (engl. sequence diagram) zur Veranschaulichung der einzelnen Abläufe und Interaktionen.

- Das Ablaufdiagramm (engl. flowchart) zur Verbildlichung der einzelnen Statusübergänge der Bestellaufträge.

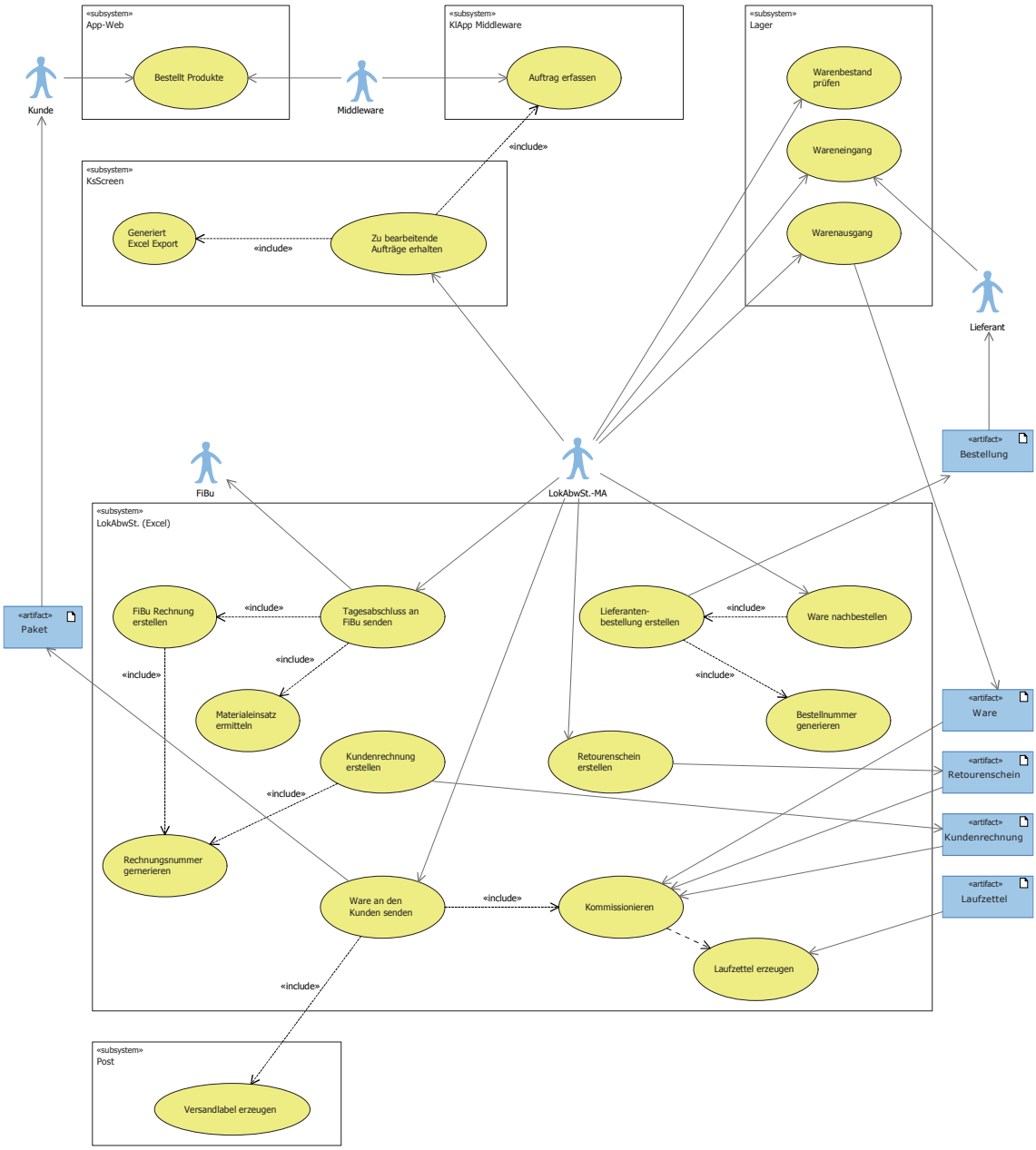

Abbildung 3.1: Anwendungsfalldiagramm IST-Situation

In Abbildung 3.1 ist der bestellende *Kunde*, die verarbeitende *Middleware*, der agierende Mitarbeiter der lokalen Abwicklungsstelle (im Diagramm *LokAbwSt.-MA*), der *Lieferant* für Nachbestellungen und die zu informierende Finanzbuchhaltung (im Diagramm *FiBu*) abgebildet. Die lokale Abwicklungsstelle (im Diagramm *LokAbwSt.*) als zentrales System bezeichnet das Service Center des Augenoptikers und umfasst die Aufgabenbereiche des Kommissionieren, Packen, Nachbestellen und Arbeiten mit Microsoft Office Dokumenten. Die mobile App beziehungsweise die Website (im Diagramm *App-Web*) stellt den Vertriebskanal für den Kauf beziehungsweise Abschluss eines Abonnements von Kontaktlinsen, Pflegemittel und Zusatzprodukte

dar. Zur Erfassung und Bereitstellung der Kundenbestellungen dient die *Middleware*. Das *Lager* hält Pflegemittel und Zusatzprodukte vor, Kontaktlinsen werden nach dem Prinzip der *Just-In-Time-Produktion* erst nach Eingang einer Kundenbestellung vom Kontaktlinsenhersteller bezogen. Die *Post* als fünftes System im Diagramm dient zur Auslieferung der Ware an den Kunden.

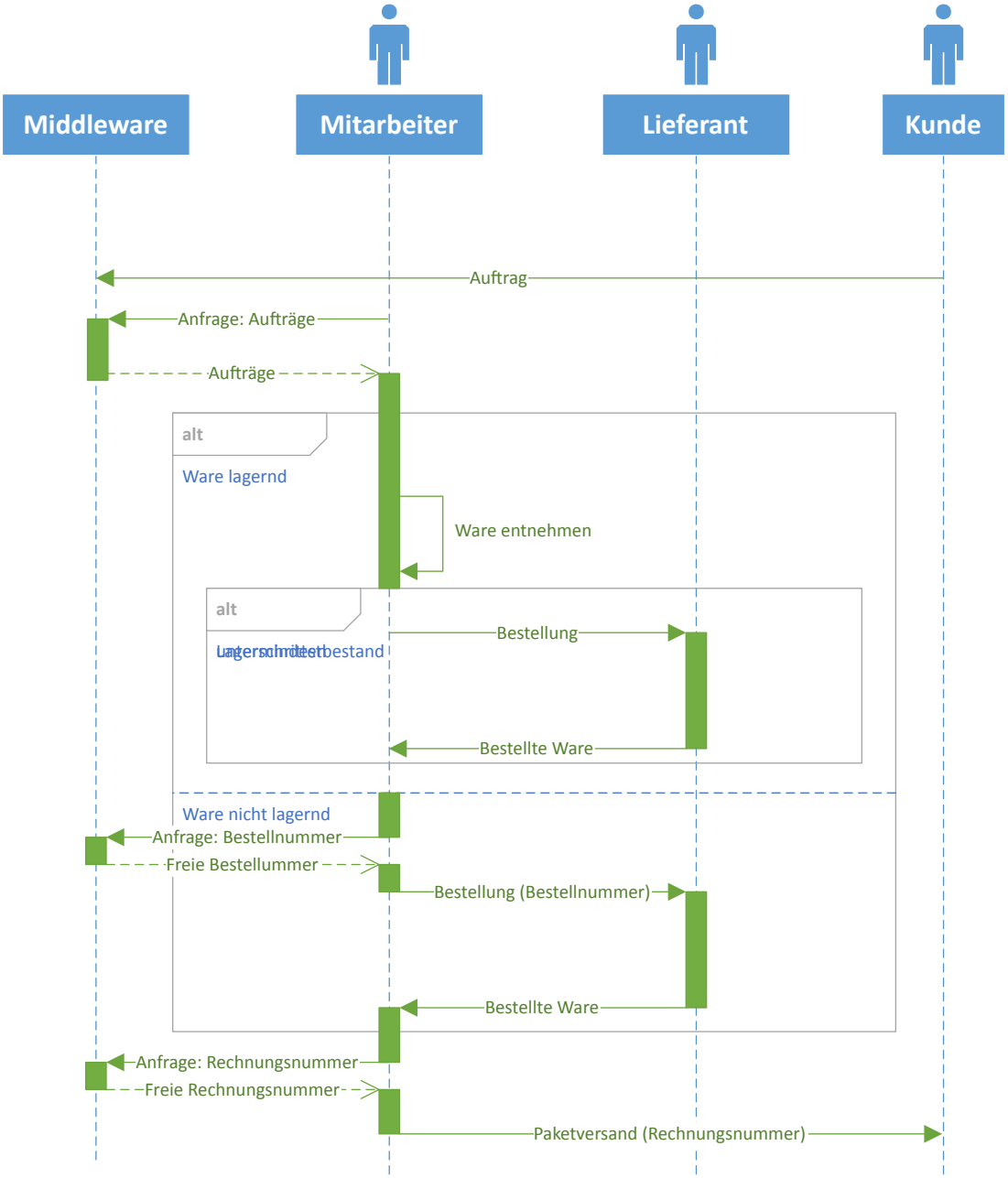

Abbildung 3.2: Sequenzdiagramm

Abbildung 3.2 stellt den Ablauf des Fulfillments dar, zur Vereinfachung wurde der Akteur *FiBu* und das System *Post* aus Abbildung 3.1 auf der vorherigen Seite nicht übernommen. Der Kunde löst direkt oder indirekt (durch das Abonnement) über die *App-Web* eine Bestellung aus. Die erzeugte Bestellung durchläuft mehrere vom

Augenoptiker definierte Zustände, die in Abbildung 3.3 dargestellt sind, angefangen mit *Vor Auslösen der Bestellung in der App*.

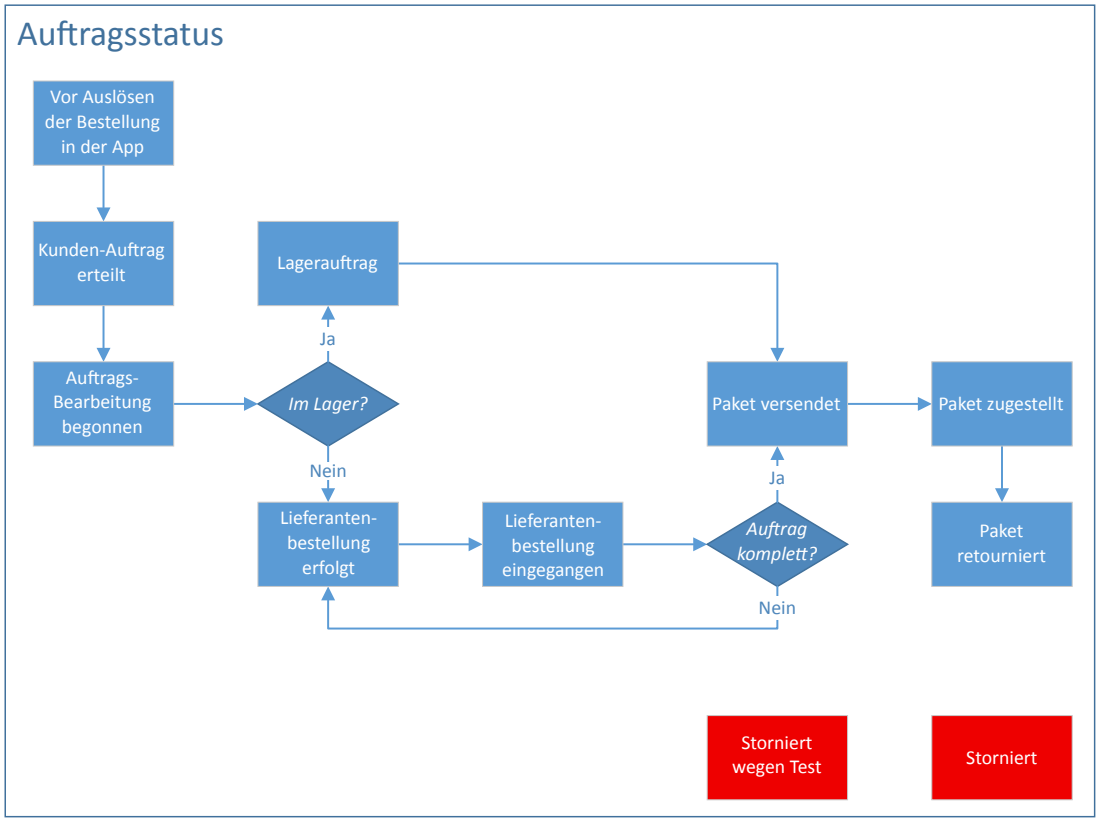

Abbildung 3.3: Auftragsstatus

Der erste Status definiert den Zustand der Bestellung, wenn Artikel in den Warenkorb verschoben wurden, die Bestellung aber noch nicht bestätigt ist. Nachdem der Kunde die Bestellung bestätigt hat, geht diese in den Status *Kunden-Auftrag erteilt* über und befindet sich innerhalb der Middleware, einer webbasierten Intranetlösung des Augenoptikers. Die Mitarbeiter der *LokAbwSt.* greifen einmal täglich auf die Middleware zu und exportieren die offenen Bestellungen in Form einer Excel-Datei. Der Status aller offenen Bestellungen wird händisch auf *Auftrags-Beabeitung begonnen* gesetzt. Enthält die Bestellung nur im Lager vorrätige Artikel, was bedeutet das die Bestellung nur aus Pflegemitteln und Zusatzprodukten besteht, geht die Bestellung in den Status *Lagerauftrag* und die Ware wird entnommen und kommissioniert. Enthält die Bestellung Kontaktlinsen oder nicht vorrätige Artikel erfolgt eine Lieferantenbestellung. Hierzu wird manuell in Excel eine Lieferantenbestellung angefertigt und an den Produkthersteller gefaxt, der Status wird wieder händisch von den Mitarbeitern innerhalb der Middleware gepflegt von *Lieferantenbestellung erfolgt* über *Lieferantenbestellung eingegangen* bis *Auftrag komplett*. Der Mitarbeiter erzeugt zusätzlich für jeden Kunden mit Excel eine Kundenrechnung. Sowohl für die Lieferantenbestellung als auch die Kundenrechnung existiert ein in Excel manuell gepflegtes Nummernbuch, damit jedes Dokument eine fortlaufende eindeutige Nummer erhält. Zum Abschluss erstellen die Mitarbeiter ein Retourenschein und ein Postlabel. Dadurch kann die Ware kommissioniert und an die Post übergeben werden, in der Middleware wird der Status *Paket versendet* und die Identifikationsnummer

des Pakets vom Mitarbeiter hinterlegt. Die Post bestätigt die Auslieferung, sodass die Bestellung automatisch in den Status *Paket zugestellt* überführt wird. Damit ist der Fulfillment-Prozess abgeschlossen. Am Ende des Tages prüft der Mitarbeiter die Lagerbestände und bestellt Artikel unter dem Meldebestand nach. An die Finanzbuchhaltung werden abschließend die Kundenrechnungen und die Materialeinsätze übermittelt.

3.1.2 Problemsituation

Der neue Kontaktlinsenabonnementservice befindet sich aktuell noch in einer Testphase, wird aber zum Eruieren von Praxiserfahrungen bereits produktiv in einem Testland betrieben. Da der Augenoptiker bisher keine Produkte online vertrieben hat, fehlt es dem Unternehmen an Erfahrung in dem Bereich und an einer automatisierten Fulfillment-Lösung. Daher wird innerhalb des Testsystems die Abwicklung mit Excel Dateien gelöst. Eine schnelle und kostengünstige Umsetzung, die für einige wenige Bestellungen pro Woche zum Evaluieren der Serviceidee ausreichend sind. Zum Zeitpunkt des Schreibens der Arbeit liegen täglich bereits vier bis fünf Bestellungen vor, wodurch die Fehlerquote und der zeitliche Aufwand durch den manuellen Ablauf stark zugenommen haben. Um die Kundenzufriedenheit durch geringe Lieferzeiten hoch zu halten und zusätzliche Arbeit für die Mitarbeiter zu reduzieren, ist eine Optimierungsmaßnahme unumgänglich.

Auf Grund der vielen Kombinationsmöglichkeiten (theoretisch über 50 Milliarden) und dem begrenzten Haltbarkeitsdatum können Kontaktlinsen nicht vorrätig gelagert werden. Die Tabelle 3.1 zeigt die Parametervielfalt einer Kontaktlinse. Dadurch müssen Kontaktlinsen bedarfsbezogen beim Hersteller bezogen werden, was die Zustellzeit für den Endkunden des Augenoptikers erhöht.

Bezeichnung	**Abkürzung**	**Bedeutung**
Radius	BC	Krümmung der Kontaktlinse. Wert zwischen 7,0 und 10,0
Stärke	PWR	Kontaktlinsenstärke in Dioptrien. Wert zwischen -30,0 und 30,0
Zylinder	CYL	Wert zwischen -0,25 und -10,00
Achse	AXIS	Wert zwischen 0 und 180
Durchmesser	DIA	Wert zwischen 13,0 und 15,0
Addition	ADD	Nahzusatz der Kontaktlinse. Wert zwischen 1,00 und 3,00

Tabelle 3.1: Parameter einer Kontaktlinse

Ein zusätzliches Problem ist die Vielzahl an verschiedenen Excel Dateien, die zur Bearbeitung der Bestellabwicklung notwendig sind. Der Hauptgrund ist die exportierte Datei aus der Middleware, welche eine Sammlung auftragsbasierter Bestellungen darstellt. Für die Fulfillment-Abwicklung sind kundenbasierte Aufträge notwendig und längst nicht alle Informationen des Exports relevant.

Die verwendeten Bezeichnungen der Status in der Middleware zur Erfassung des Zustands der Bestellung unterliegen keiner Namenskonvention und müssen aktuell von den Mitarbeitern manuell gesetzt werden. Dadurch entsteht ein großes Fehlerpotential im aktuellen Testsystem.

3.2 Anforderungen

Im vorausgehenden Abschnitt wurde das aktuelle Problem aufgezeigt, dass innerhalb der Testphase essentielle Bestandteile des Bestellabwicklungsprozesses mit sehr hohem menschlichem und zeitlichem Aufwand umgesetzt werden. Daran anknüpfend werden Anforderungen definiert, die für eine Verbesserung erforderlich sind. Die Anforderungen für den Soll-Zustand werden mit Hilfe des *Volere Requirements Specification Template* bestimmt, ein Hilfsmittel aus der Anforderungsanalyse des Softwareentwicklungsprozesses (Requirements Engineering).

3.2.1 Funktionale Anforderungen

Mit der Erfüllung der folgenden Anforderungen können unterschiedliche Geschäftsprozesse abgewickelt werden. Dadurch stellt der Prototyp einen Lösungsansatz für Probleme einer größeren Zielgruppe dar. Durch die Realisierung soll validiert werden, ob die Frage nach der Machbarkeit beantwortet werden kann.

3.2.1.1 Anbindungsmöglichkeit an eine Middleware

Damit das automatisierte Fulfillment implementiert werden kann, sind die Informationen aus der Middleware notwendig. Deshalb werden die erforderlichen Bestellungen aktuell aus der Datenbank exportiert und mit Excel verarbeitet. Mit dieser Export-Schnittstelle wurde ein unidirektionaler Weg geschaffen, sodass die Mitarbeiter der lokalen Abwicklungsstelle die Statusänderungen der Bestellungen händisch über das Webfrontend der Middleware eintragen müssen. Eine einfachere Möglichkeit wäre die notwendige Abwicklung des Fulfillments direkt in der bestehenden Middleware abzubilden. Die Middleware als Webanwendung wurde in .Net implementiert und besitzt ein ASP.NET Interface. Eine Erweiterung der Software würde neue Logikklassen und zusätzliche Unterseiten (engl. views) im Interface erfordern. Da diese nur Zusatzkomponenten darstellen ist die Umsetzung ohne großen Aufwand umsetzbar. Wie in Abschnitt 1.1 Motivation und Problemstellung beschrieben, wurde die aktuelle Middleware nicht optimal umgesetzt und soll in naher Zukunft ersetzt werden. Daher sollen sich die Middlewareerweiterungen in einem überschaubaren Rahmen bewegen. Durch die stetig wachsende Zahl an Bestellungen im Testland wird jedoch eine Fulfillment-Lösung benötigt, welche bereits in der bestehenden Middleware eingesetzt werden kann. Zusätzlich soll die entwickelte Lösung weiterhin in der zukünftigen Middleware betrieben werden können.

Die neue Middleware wird weiterhin eine Webanwendung sein, jedoch befindet sich die Umsetzung in einer sehr frühen Phase, sodass derzeit noch keine Informationen vorliegen, wie die Realisierung aussehen wird. Daher wird ein Schnittstellenkonzept benötigt, dass die grundsätzliche Anbindung an die Webapplikationen erlaubt um die Informationen sowohl aus der bestehenden als auch zukünftigen Middleware in die Fulfillment-Lösung zu übertragen. Zusätzlich wird eine Schnittstelle benötigt um bestehende Funktionalitäten aus anderen internen Systemen wie das ERP-System nutzen zu können.

3.2.1.2 Definition von Abarbeitungsschritten

Eine grundlegende Anforderung stellt das Definieren und Verknüpfen von Abarbeitungsschritten innerhalb einer definierten Abfolge dar. Die Abfolge entspricht einem

Workflows innerhalb bestimmter Geschäftsprozesse. Der definierte Workflow soll zur Laufzeit ausgeführt werden, dabei sollen die einzelnen Abarbeitungsschritte auf die Daten der aktuellen Bestellung sowie auf die Ergebnisse voriger Abarbeitungsschritte zugreifen können. Mit der Definition von mehreren Abarbeitungsschritten ergibt sich eine vorwärts gerichtete Abfolge, mit einem festgelegten Startpunkt. Jeder Abarbeitungsschritt kann beliebig viele Nachfolger haben. Des Weiteren soll es möglich sein, dass ein Abarbeitungsschritt die Ergebnisse von mehreren vorherigen Aufgaben benötigt. Dadurch wird eine Parallelität innerhalb des Workflows möglich. Der Workflow, sowie die darin enthaltenen Abarbeitungsschritte sollen jederzeit anpassbar sein. Die Abarbeitung des Workflows soll automatisch geschehen, sobald dieser durch ein Ereignis gestartet wird.

3.2.1.3 Recovery

Um auf unvorhersehbare Fehler reagieren zu können, soll die Anwendung aufgetretene Fehler sammeln und archivieren. Falls dieselbe Bestellung erneut mit derselben Konfiguration gestartet wird, soll die Abarbeitung automatisch von der Fehlerstelle fortfahren. Dabei ist zu beachten, dass bereits ausgeführte Aufgaben nicht erneut ausgeführt werden dürfen, da möglicherweise Prozesse in entkoppelten Systemen angestoßen wurden. Zu diesen entkoppelten Prozessen gehören die Nachbestellung notwendiger Kontaktlinsen oder das Erzeugen von Postlabels. Diese Vorgänge dürfen nur einmal durchlaufen werden, da ansonsten Zusatzkosten entstehen würden. Der Zustand der Maschine beinhaltet alle notwendigen Informationen zum Betrieb und der Wiederherstellung der Maschine.

3.2.1.4 Analysemöglichkeit

Kundenzufriedenheit spielt für den Erfolg des Auftraggebers eine große Rolle. Daher soll die Bestellabwicklung möglichst schnell abgewickelt werden, um dem Kunden die bestellte Ware möglichst zügig zu liefern. Da der Transport sowie die Herstellung der Kontaktlinsen von externen Partner übernommen wird, kann dieser Teil des Fulfillments nicht direkt beeinflusst werden. Durch eine intelligente Bestellvorhersage und Lageroptimierung soll versucht werden, die Lieferzeit zu reduzieren. Um das Optimierungspotenzial zu erfassen, muss eine generelle Analyse der Abläufe im System ermöglicht werden. Das Reporting ist nicht Bestandteil dieser Arbeit, es soll aber bereits die Grundlagen dafür schaffen.

3.2.1.5 Erweiterbarkeit

Die wohl größte Herausforderung des Projektes ist die Geschwindigkeit und Schwere der Veränderung die der Software ausgesetzt werden soll. Durch diesen Wandel ergibt sich die Anforderung, dass die Software bis auf das Grundgerüst zu späteren Zeitpunkten von Softwareentwicklern weiter entwickelt oder modifiziert werden kann. Die Erweiterungen können die Service-Schnittstellen, Modelle, Datenbanken oder den Workflow betreffen. Das Grundgerüst sollte trotz Erweiterungen nicht angepasst werden müssen. Hierzu muss die Software als Framework entwickelt werden, damit sie die notwendige Flexibilität mitbringt.

3.2.2 Nichtfunktionale Anforderungen

Die folgenden Anforderungen sollen als Qualitätsanforderungen verstanden werden. Sie dienen lediglich als Randbedingung und stehen nicht direkt im Fokus einer technischen Machbarkeitsanalyse.

3.2.2.1 Benutzerfreundlichkeit

Die zu entwickelnde Software dient zur technischen Überprüfung der Umsetzbarkeit. Sofern die Realisierung möglich ist kann dieser Prototyp als erste Version für die Projektrealisierung verwendet werden. Daher ist für die korrekte Verwendung und Weiterentwicklung eine hohe Quellcodequalität erforderlich. Das bedeutet, dass der Quellcode einheitlich formatiert ist und sowohl Bezeichner als auch Kommentare in englischer Sprache verfasst sind. Um das Verständnis zu erhöhen sollen alle Dokumentationskommentare vorliegen, damit anhand des Quellcodes automatisch eine Dokumentation erstellt werden kann. Dies erleichtert den Einstieg in die Software erheblich.

3.2.2.2 Leistung und Effizienz

Durch den Erweiterungscharakter der Software ist die Effizienz in erster Linie bei den Erweiterungen zu beachten. Dennoch darf die Summe der Verwaltungsdaten (engl. overhead), die durch das definieren von Aufgaben entsteht, mit jeder zusätzlichen Aufgabe nur maximal linear steigen. Der Overhead setzt sich dabei aus der nötigen Rechenzeit und dem nötigen Speicher zusammen. Die parallele Ausführung von Bestellungen stellt für den Auftraggeber eine grundlegende Anforderung dar und muss ohne weitere Anpassungen möglich sein.

3.3 Zusammenfassung

In diesem Kapitel wurde die Problemsituation eines speziellen Kunden gezeigt. Die Kernproblematik der anpassbaren automatischen Abarbeitung eines definierten Workflows wurde aus der IST-Analyse extrahiert. Die definierten Anforderungen erlauben im Anschluss die Erstellung eins Konzepts zur Lösung der Kernproblematik. Es ist nicht Ziel dieser Arbeit eine Individuallösung für den Augenoptiker zu entwickeln, sondern zu prüfen, ob es möglich ist durch Abstraktion der Problemsituation ein Konzept zu finden um eine möglichst große Gruppe von Kunden zu bedienen. Die benötigte Software mit Framework-Charakter soll eine Workflow-Automatisierung abbilden. Zur Kommunikation mit einer Middleware oder ähnlichen externen Anwendungen sind Schnittstellen erforderlich. Im Konzept soll die Anforderung für eine Wiederherstellung des aktuellen Zustands sowie die Möglichkeit für eine spätere Analyse beachtet werden.

4. Konzept

Dieses Kapitel baut auf die beschriebenen Anforderungen aus dem vorigen Kapitel auf. Es beschreibt Lösungsansätze für einen Validierungsansatz, der Zeigen soll ob eine wiederverwendbare Workflow-Automatisierung konzeptionell möglich ist. Das Konzept soll als Lösungsansatz für den Auftrag des Augenoptikers sowie für zukünftige Kunden mit ähnlichen Problemsituationen genutzt werden können. Um einen Überblick über die Realisierungsidee zu verschaffen, soll zuerst das grobe Konzept erläutert werden. Dabei soll auf die Anbindungsmöglichkeiten nach Außen und die Verwaltung und Steuerung innerhalb des Prototyps eingegangen werden. Anschließend sollen Lösungsansätze für die funktionalen Anforderungen, im speziellen die automatisierte Abarbeitung, beschrieben werden.

4.1 Konzeptstruktur

Die grundlegende Funktionalität der Anwendung ist das Entgegennehmen von Daten, sowie deren Abarbeitung nach einem festgelegten Workflow. Aufträge können Informationen jeglicher Art sein, die zur Abwicklung eines Geschäftsprozesses notwendig sind. Im Fall des Augenoptikers ist ein Auftrag eine Kundenbestellung im Rahmen des Fulfillments.

Die individuelle Anpassung der Anwendung soll von Anwendungsentwicklern vorgenommen werden, um unterschiedliche Geschäftsprozesse abzubilden. Hierfür ist die Erstellung und Anpassung von einzelnen Abarbeitungsschritten wie etwa das Erzeugen einer Rechnung notwendig. Um den realen Geschäftsprozess abzubilden ist eine Verknüpfung der einzelnen Abarbeitungsschritte zu einer definierten Abarbeitungsfolge notwendig. Dieser Arbeitsablauf soll im Folgenden *Workflow* genannt werden, die einzelnen Abarbeitungsschritte als *Activity*, dem englischen Wort für Tätigkeit, bezeichnet.

Die einzelnen Activities sollen flexibel gestaltet sein, damit eine Wiederverwendung in verschiedenen Workflows möglich ist. Der Workflow selbst soll auch adaptiv entworfen werden. Um den Anforderungen aus Abschnitt 3.2.1.5 Literaturerweiterbarkeit gerecht zu werden soll der Prototyp daher in Form eines Frameworks entworfen

werden. Dadurch werden Standards und Konventionen vorgegeben, die den Anwendungsentwickler bei der Abbildung des Geschäftsprozesses unterstützen. Wir haben uns für die Umsetzung einer Activity in Form einer Klasse (beziehungsweise Objekttyp) entschieden. Dadurch behält der Anwendungsentwickler im Folgenden Nutzer genannt die größtmögliche Freiheit bei der Programmierung und zugleich können Objektschnittstellen (siehe Abschnitt 2.3.1) vorgegeben werden.

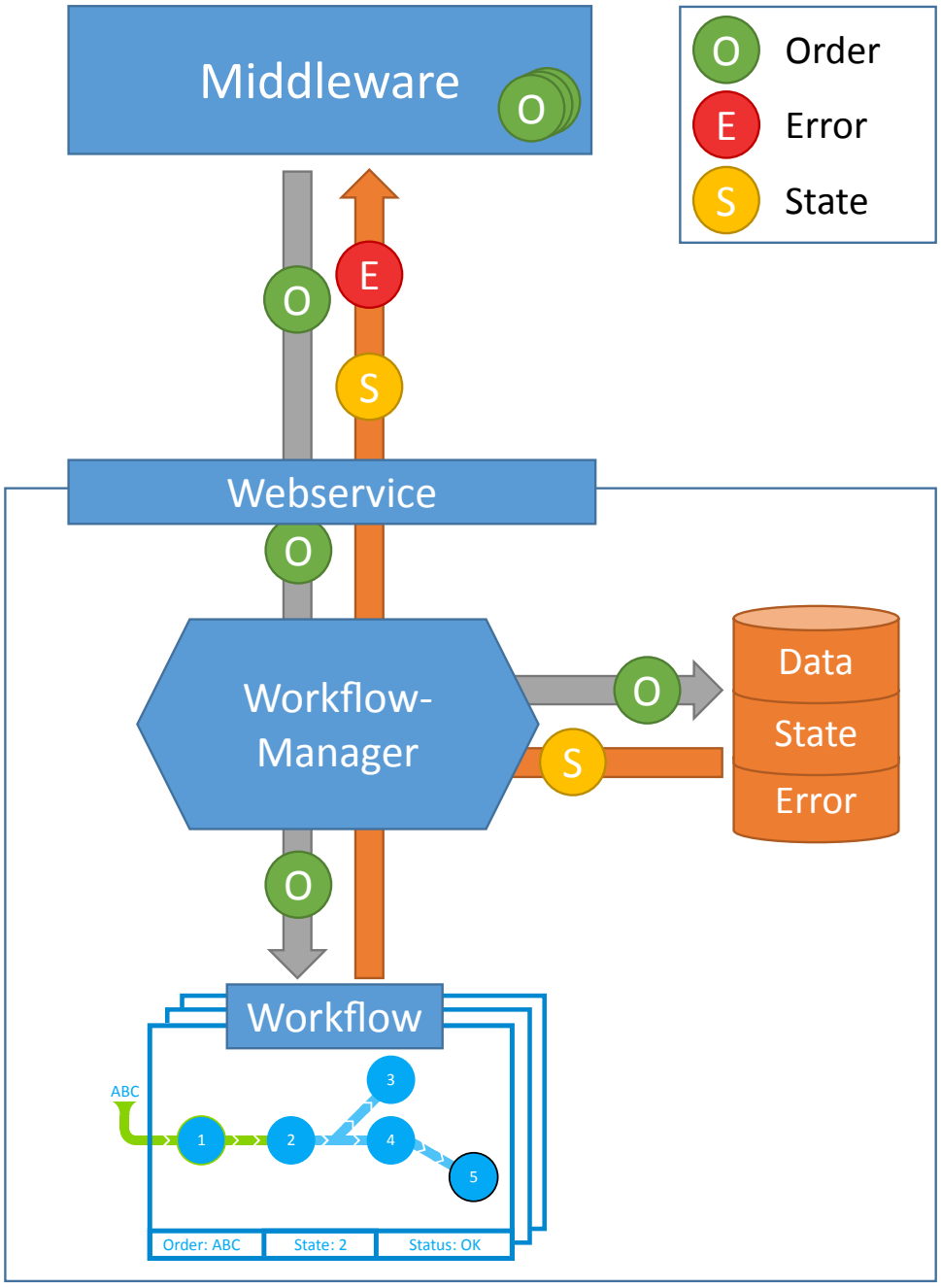

Abbildung 4.1: Überblick der Konzeptstruktur

Der Prototyp besteht aus vier Komponenten, dem Webservice, Workflow-Manager, persistenten Speicher und der Kernlogik zur Workflow-Abarbeitung. Die Abbildung 4.1 soll die Beziehungen untereinander verdeutlichen. Ein externes System

übermittelt Daten zur Abarbeitung an den Prototypen, im Falle des Augenoptikers sendet eine Middleware Bestelldaten (engl. order). Die Schnittstelle, die diese Bestellungen entgegennimmt soll durch einen Webservice realisiert werden. Mittels dieser Schnittstelle empfängt der Prototyp Bestellungen von der Middleware oder sendet Status und Fehler an die Middleware.

Der Workflow-Manager empfängt die Bestellungen und speichert diese in einem persistenten Speicher zur Analyse und Wiederaufnahme für den Fall eines Systemversagens ab. Zugleich werden die Bestelldaten an den entsprechenden Workflow weitergegeben und die Workflow-Abarbeitung gestartet. Sowohl den Zustand als auch auftretende Fehlernachrichten kann die Middleware anfragen.

4.1.1 Webservice

Interoperabilität (die Zusammenarbeit von verschiedenen unabhängigen Systemen) hat höchste Priorität, da die zukünftige Middleware unbekannt ist. Bekannt ist nur das es sich um eine Web-Applikation handeln wird, sodass ein Webservice die ideale Kommunikationsschnittstelle darstellt, was in Abschnitt 2.3.2 Anwendungsschnittstelle deutlich wurde. Die Anbindung von SOAP als typsichere Kommunikation soll über die Kommunikationsplattform *Windows Communication Foundation* aus dem Hause Microsoft bereitgestellt werden.

Um den Anforderungen der universellen Anbindung gerecht zu werden soll zusätzlich eine REST-Fassade zum SOAP Webservice integriert werden. REST als Alternative zu SOAP stellt eine weitverbreitete Web 2.0 Technologie dar, welche sich auf das JSON-Format stützt und deutlich einfacher zu implementieren ist, aber keine Typsicherheit wie SOAP bietet. Vor allem zur Anbindung mobiler Geräte hat sich REST durchgesetzt. Die Technologien SOAP und REST verwenden unterschiedliche Technologien und Prinzipien und der Vergleich der beiden Lösungen soll nicht Bestandteil dieser Arbeit sein. Beide Technologien werden als API angeboten und stellen somit den einzigen Kommunikationskanal nach Außen dar. Jegliche Dienste der Anwendung werden als Service bereitgestellt. Notwendig ist ein Service zum Starten der Abarbeitung des definierten Fulfillment-Workflows, welches die Bestellung als Übergabeparameter erwartet. Zusätzlich sind Services für die Auskunft des aktuellen Zustand und des Status der Bestellung notwendig. Alle Serviceanfragen werden an den Workflow-Manager (siehe Abschnitt 4.1.3) weitergeleitet.

4.1.2 Workflow

Der Workflow stellt die wichtigste Komponente für den Nutzer dar. Darin sind alle gewünschten Abarbeitungsschritte definierbar, die zu einem späteren Zeitpunkt automatisch ausgeführt werden sollen.

Der Geschäftsprozess muss dazu in einzelne Aufgaben (Activity genannt) unterteilt werden. Dieser Schritt ist Notwendig, damit sich Teilaufgaben wie zum Beispiel das Drucken eines Dokuments wiederverwenden lassen. In Abbildung 4.2 auf der nächsten Seite ist ein einfacher Workflow mit fünf Activities visualisiert. Jede Activity hat einen oder mehrere festgelegte Nachfolger, außer es handelt sich um eine End-Activity wie die Activities mit den Nummern drei und fünf. Die Activity fünf stellt außerdem die letzte Activity des Workflows dar, weshalb sie in dem Diagramm

schwarz umrandet ist. Nachdem die letzte Activity ausgeführt wurde, wird der Ursprung der Bestellung, in diesem Fall die Middleware, von der Fertigstellung der Bestellung informiert.

Neben den normalen Activities und den End-Activities gibt es in jedem Workflow genau eine Start-Activity, in Abbildung 4.2 die Activity mit der Nummer eins. Um die Resultate der Activities in den darauf folgenden Activities nutzen zu können, haben alle Activities bis auf End-Activities die Möglichkeit Daten an die folgende Activity zu übergeben. Im Umkehrschluss haben alle bis auf die Start-Activity einen Eingang (engl. Input) um die Daten zu erhalten. Die normale Activity hat somit einen Eingang und einen Ausgang (engl. Output) über den sie beliebige Daten übergeben kann.

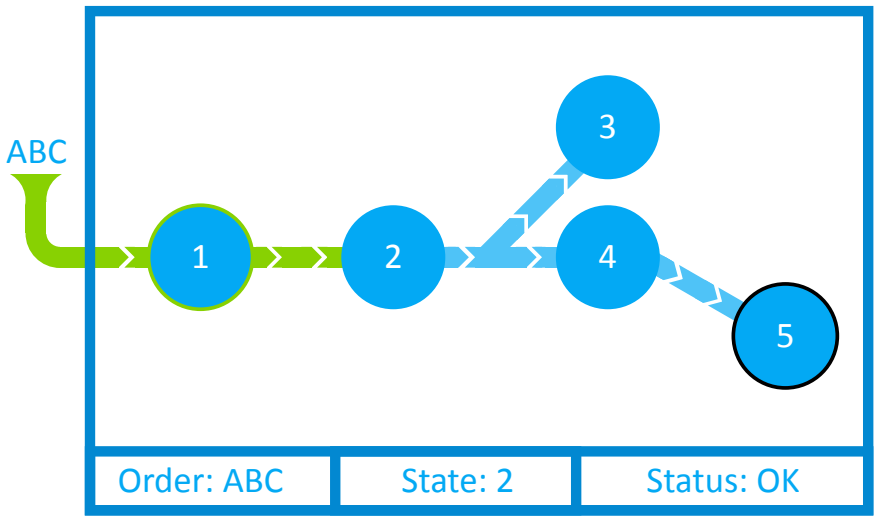

Abbildung 4.2: Workflow

Der Workflow wird durch das eingeben von Daten gestartet. Die Daten bestehen im Fulfillment-Prozess aus Bestellungen, in Abbildung 4.2 durch die Buchstaben *ABC* visualisiert. Da es vorkommt, dass die Bestellung in dem Workflow mehrfach benutzt wird, steht die eigehende Bestellung allen folgenden Activities zur Verfügung; kann jedoch nicht verändert werden. So wird sichergestellt, dass die von dem Nutzer geschriebenen Activities alle auf die ursprüngliche Bestellung zugreifen können.

Nachdem eine eingehende Bestellung den Workflow in Gang gesetzt hat, durchläuft er alle definierten Activities in der vom Nutzer festgelegten Reihenfolge. Dabei wird jede Activity nur einmal ausgeführt, was dazu führt, dass sich der Workflow zu jeder Zeit in einem einzigartigem Zustand (engl. State) befindet. Dieser ist gekennzeichnet durch die Bestellung und die aktuell ausgeführten Activities. Der Status des Workflows gibt an, ob ein Fehler während der Ausführung aufgetreten ist.

Oftmals kommt es in Workflows zu Verzweigungen, bei denen nach einer bestimmten Aufgabe mehrere Folgeaufgaben gestartet werden. Dies ist der Fall bei den Activities drei und vier aus Abbildung 4.2. Da die beiden Activities keine Veränderungen an der Bestellung vornehmen können und sie gänzlich voneinander unabhängig sind, können sie zu der gleichen Zeit ausgeführt werden. Alle Teil-Workflows die nach einer Verzweigung kommen werden parallel ausgeführt, wodurch die Effizienz deutlich verbessert wird.

Nicht nur Verzweigungen können in einem Workflow auftreten. Oftmals folgen auf Verzweigungen Zusammenführungen, bei denen die separat erarbeiteten Ergebnisse wieder zusammen geführt werden. Am Beispiel der Kommissionierung im Fulfillment wird dies deutlich. Das Drucken der Versandpapiere und packen des Paketes kann parallel ausgeführt werden. Für den Versand der Ware muss jedoch von all diesen Aufgaben eine Bestätigung erfolgen, dass sie erfolgreich ausgeführt wurden. Diese Bestätigung wird als Zusammenführung der einzelnen Activities angesehen und an die nachfolgende Versand-Activity gesendet.

Das Konzept voneinander unabhängiger Activities und dessen Ein- und Ausgang ermöglicht eine sehr einfache Wiederverwendbarkeit. Es ist möglich eine Activity sowohl in einem Workflow mehrfach zu benutzen, als auch dieselbe Activity in mehreren verschiedenen Workflows zu benutzen.

4.1.3 Workflow-Manager

Damit die über den Webservice eingehenden Bestellungen einen festgelegten Workflow erreichen, gibt es den Manager. Er nimmt eingehende Bestellungen an, weist ihnen eine eindeutige Identifikationsnummer zu die aus dem Hash der Bestellung generiert wird und speichert dann die komplette Bestellung ab. Die gespeicherten Bestellungen dienen der nachträglichen Analyse (Abschnitt 4.3) und zur Fehlerwiederherstellung (Abschnitt 4.2). Zusätzlich bekommt die Bestellung einen Eintrag in der *Status-Tabelle*. Hier wird der Workflow Status (siehe Abschnitt 4.1.2) zu jeder Bestellung noch während der Ausführung gespeichert. Im Falle eines Fehlers wird zusätzlich noch die Fehlernachricht gespeichert.

Nach dem Eingang einer Bestellung, wird in der Status Tabelle nach dieser Bestellung gesucht. Da jede Bestellung einzigartig ist, bedeutet ihr Finden, dass es sich um einen Fehler handeln muss, da die Bestellung bereits behandelt wurde oder wird. Sollte die Bestellung einem Fehler zugewiesen sein, so kann anhand des gespeicherten Status festgestellt werden, welche Activity als nächstes auszuführen ist. Der Manager führt diese Bestellung dann an der Stelle fort. So wird verhindert, dass bereits erfolgreich erledigte Activities nach einem Fehler erneut ausgeführt werden.

Von dem Webservice eingehende Statusanfragen werden ebenfalls von dem Manager beantwortet. Der Status wird aus dem laufendem Workflow genommen, oder falls er bereits fertig ist aus der *Status-Tabelle*. Die Konzepte zur Statusüberwachung werden in Abschnitt 4.4 Statusüberwachung erläutert.

4.2 Recovery

Das Halten eines Zustandes ist die Grundlage jedes Zustandsautomaten (engl. state machine). Der Zustand ergibt sich in diesem Fall aus der Bestellung und den Werten der Anwendung zu einem bestimmten Zeitpunkt. Da die Resultate der einzelnen Knoten des Automaten, also dessen Aktivitäten, erst nach dessen Ausführung fest stehen, gibt es Aktivitäten die zu einem Zustand bereits ausgeführt worden sind und welche die es noch nicht sind. Dieser Zustand sollte laut der Recovery Anforderung nach einem Fehler wiederherstellbar sein um nicht erfolgreich ausgeführte oder abgebrochene Bestellungen erneut auszuführen.

Um den nötigen Grad an Sicherungen zu ermitteln die für die Erfüllung der Anforderung aus Abschnitt 3.2.1.3 Recovery nötig sind, werden im Folgenden mögliche Fehlerursachen und Lösungskonzepte vorgestellt.

Der am einfachsten behandelbare Fehler ist eine Ausnahmebehandlung (engl. exception) die in einer vom Nutzer geschriebenen Aktivität geworfen wird. Diese kann das Framework abfangen und für den Nutzer aufbereitet speichern. Die Anwendung bleibt von dem Fehler unbetroffen und führt andere Bestellungen weiterhin aus. Der Fehler könnte alleine durch eine unerwartete Bestellung, beziehungsweise Eingabeparameter ausgelöst worden sein, oder durch einen Fehler in der Aktivität der unabhängig vom Eingabeparameter auftritt. Sollte ein externer Dienst, welcher durch eine Aktivität aufgerufen wird nicht erreichbar oder nicht erfolgreich ausgeführt werden, so wird auch das als Ausnahmebehandlung vom Framework abgefangen.

Schwer zu behandeln sind Fehler, die im Framework selber auftreten. Um diese zu minimieren sollte das Framework ausführlich getestet werden, da jeder implementierte Recovery-Algorithmus Fehler enthalten kann. Sollte dennoch ein Fehler auftreten, so werden alle betroffenen laufenden Bestellungen unterbrochen. Je nach betroffener Komponente könnten zusätzlich noch alle sich im Speicher befindlichen Daten gelöscht werden. Ein Fehler oder Ausfall der physischen Maschine hat die gleichen Auswirkungen und muss von daher nicht separat analysiert werden.

Die naheliegende Lösung ist das persistente Speichern aller erfolgreich ausgeführten Schritte und deren Ergebnisse. Nach einem Fehler müssen alle Aufgaben lediglich nach den erfolgreich ausgeführten Schritten fortgesetzt werden. Diese Lösung könnte jedoch im Widerspruch zu der Effizienz Anforderung stehen (siehe Abschnitt 3.2.2.2 Leistung und Effizienz), da das persistente Speichern in den meisten Fällen viel Zeit in Anspruch nimmt.

Eine schlankere Lösung wäre das persistente Speichern von aufgetretenen Fehlern. Dies würde es ermöglichen, nach einem Ausnahmefehler das System beenden zu können ohne dass die aufgetretenen Fehler gelöscht werden. Nach dem beheben der Fehler kann das System erneut gestartet werden und die fehlerhaften Bestellungen können fortgesetzt werden. Ein Systemversagen deckt diese Lösung zwar nicht ab, ist dafür jedoch sehr performant, da nur wenige persistente Speicherungen nötig sind.

Der Kompromiss ist das Verwenden der aus Abschnitt 4.3 gespeicherten Bestellungen und das Hinzufügen einer *Status-Tabelle*. In dieser werden nach jeder erfolgreichen Activity der Zeitpunkt, die Activity selber und das resultierende Ergebnis gespeichert. Wenn die Bestellung erfolgreich behandelt wurde, werden die Ergebnisse gelöscht und die Bestellung wird als erfolgreich markiert. Die resultierende Historie dient anschließend noch der Analyse und der Nachweisbarkeit. Diese Lösung nutzt die bestehenden Ressourcen und erweitert das Framework um eine Historie welche zur Analyse sehr hilfreich ist. Da sie zusätzlich noch die Anforderung aus Abschnitt 3.2.2.2 Leistung und Effizienz abdeckt, ist sie die bevorzugte Wiederherstellungslösung.

4.3 Analyse

Aus den Anforderungen aus Abschnitt 3.2.1.4 geht hervor, dass eine Möglichkeit zur Analyse angeboten werden soll. Theoretisch können die bestehenden Daten in der

Middleware bereits für eine Bestellvorhersage genutzt werden, da hierfür hauptsächlich der Eingang der Bestellung sowie die Bestellpositionen notwendig sind. Es ist jedoch gewünscht dies in die Abwicklung des Fulfillments auszulagern, damit der Erfolg messbar ist, da die Statusänderungen der Bestellung nur innerhalb der zu entwickelnden Anwendung enthalten sein sollen. Dadurch kann jede Statusänderung erfasst werden um zu erkennen, welche Prozesse innerhalb der Bestellbearbeitung am längsten dauern. Anhand dieser Daten können Prozesse optimiert werden und gegebenenfalls Vorhersagen zu möglichen zukünftigen Bestellungen getroffen werden.

Damit die gewünschte Analyse möglich ist, sind zwei Ansätze innerhalb des Frameworks denkbar. Die notwendigen Informationen beim Statuswechsel werden protokoliert (eng. logging) oder die Bestellungen werden komplett persistent gespeichert. Das Abspeichern von Statusänderungen ist in Logdateien oder einer Datenbank möglich und benötigt deutlich weniger Daten als die vollständige Erfassung der Bestelldaten. Um ein vollständiges Reporting anzubieten, empfiehlt sich jedoch die vollständige Speicherung jeder Bestellung. Zudem können diese Daten auch für das Wiederherstellen von Bestellungen genutzt werden. Dazu sollen alle übermittelten Daten zur Bearbeitung persistent in einer Datenbank abgelegt werden.

4.4 Statusüberwachung

Die Erkenntnis aus Abschnitt 4.1.1 hat ergeben, dass die Anwendung nur über eine Service-Schnittstelle zur Kommunikation mit der Middleware verfügt. Die Middleware übergibt beim Aufruf die Bestelldaten für das Fulfillment und delegiert die Abwicklung an den Prototyp, wodurch der Middleware die Information über den aktuellen Status der Bestellung fehlt.

Folgende Möglichkeiten sollen gegenübergestellt werden, um das Problem zu lösen:

- Beim Serviceaufruf wird auf das Ergebnis der Abfolge gewartet, um den Status als Rückgabewert zu erhalten.
- *Fire-and-Forget*: Der Service wird aufgerufen, wobei nicht auf den Rückgabewert gewartet wird. Es bedarf eine abschließende Activity innerhalb der Abfolge, welche den Status an die Middleware übergibt.
- Rückgabe eines eindeutigen Identifizierungskennzeichens: Beim Serviceaufruf wird eine ID zurückgegeben, die Anwendung stellt zusätzlich einen Service bereit, der anhand dieser ID den aktuellen Status zurück liefert.
- *Webservice Callback*: Beim Aufruf des Service wird ein Middleware-Service übergeben, welcher nach erfolgreicher Bearbeitung aufgerufen wird. Dadurch wird der finale Status bekannt.

Die erste Option ist sehr einfach zu realisieren und erfordert keine Speicherung der übermittelten Daten um die gewünschten Funktionalitäten zu bieten. Die Middleware muss unter Umständen sehr lange auf ein Ergebnis warten, da der Workflow beliebig komplex werden kann. Laufzeiten von Sekunden bis zu mehreren Monaten sind denkbar, wenn beispielsweise eine Nutzereingabe innerhalb der Abfolge notwendig ist. Die Verbindung muss über die gesamte Dauer der Bearbeitung offen

gehalten werden, da die Middleware auf eine Antwort (engl. response) wartet. Führt die Middleware den Serviceaufruf nicht innerhalb eines eigenen Tasks aus, kann dies zu einem blockierenden Verhalten (engl. blocking point) führen.

Beim Serviceaufruf sollte nicht auf das Ergebnis der Abfolge gewartet werden müssen, daher wartet die zweite Option auf keine Antwort und ignoriert eine eventuelle Rückgabe. Der Service wird aufgerufen, und der Erfolg oder Misserfolg ist für den Aufrufer uninteressant, weshalb man bei diesem Konzept von *Fire-and-Forget* spricht. Damit die Middleware trotzdem einen Status zurück geliefert bekommt, muss eine Aktion innerhalb der Abfolge den Status der Bestellung zurückmelden. Das Problem an diesem Vorgehen ist die Ungewissheit. Im Fehlerfall kommt keine Meldung und im Erfolgsfall wird nach einer unvorhersehbaren Zeit der Bestellstatus an die Middleware zurückgegeben. Somit verfällt die Middleware in das bekannte Halteproblem, da es ungewiss ist, ob eine Antwort zurückkommt. Es kann nicht davon ausgehen, dass die Bearbeitung fehlgeschlagen ist, da jede Sekunde die Rückmeldung erfolgen kann.

Die dritte Möglichkeit bietet der Middleware daher an, zu jeder Zeit den aktuellen Status der Bestellung zu erfragen. Beim Aufruf des Services wird ein Identifikator der Bearbeitung zurückgesendet. Anhand dieser ID kann jederzeit der Status abgefragt werden. Mit diesem Konzept erhält die Middleware nach Bedarf den Status der Bestellung. Sobald der Mitarbeiter innerhalb der Middleware den Status benötigt, wird die Anfrage nicht an die Datenbank der Middleware sondern an die Anwendung umgeleitet. Durch diesen Webservice kann nicht nur die Middleware, sondern auch jeder andere Dienst den Status erhalten. Daher empfiehlt sich die Implementierung dieses Konzepts. Wartet die Middleware hingegen auf den finalen Status um weitere Prozesse anzustoßen muss sie in regelmäßigen Zyklen den Status abfragen. Dies erzeugt unnötigen Netzwerkverkehr. Zudem erhält die Middleware den gewünschten Status erst verzögert.

Die vierte Option verfolgt das Prinzip der Delegationen, im Service-Bereich *Callback-Funktionalität* genannt. Dem Service wird eine Service-Adresse übergeben, die nach erfolgter Bearbeitung aufgerufen wird. Wird der übergebene Service aufgerufen, erhält die Middleware dadurch die Information, dass die Bestellung im gewünschten Status vorliegt. Zusammen mit der dritten Option stellen diese beiden Konzepte eine ideale Möglichkeit der Statusmeldung für die Middleware dar.

Auch im Fehlerfall muss die Middleware informiert werden. Hierzu könnte jede der vier genannten Möglichkeiten genutzt werden. Das Konzept der Delegation eignet sich am besten, da die Middleware dadurch sofort die Information erhält. Beim Serviceaufruf zur Fulfillment-Abwicklung wird als zusätzlicher Parameter eine Service-Adresse für den Fehlerfall übergeben. Sobald im Workflow ein Fehler auftritt, wird die hinterlegte Adresse aufgerufen. Dem Service kann nicht die Art des Fehlers mitgegeben werden, die Middleware erhält nur die Meldung, dass ein Fehler für eine bestimmte Bestellung aufgetreten ist. Daher muss das Framework der Middleware einen zusätzlichen Service anbieten, welcher den exakten Fehler ausgibt anhand des erhaltenen Identifikators, der auch zur Statusermittlung dient. Die entsprechende Fehlermeldung, sowie der finale Status einer Bestellung muss persistent gespeichert werden, damit die Middleware auch nach Beendigung der Bearbeitung der Abfolge den korrekten Fehler beziehungsweise Status erfragen kann. Das Konzept zur

Wiederherstellung des korrekten Zustands erfordert bereits einen persistenten Speicher, welcher bereits über den Bearbeitungsstatus verfügt. Zusätzlich soll auch eine eventuelle Fehlernachricht mit aufgenommen werden.

4.5 Zusammenfassung

Dieses Kapitel setzt sich mit den Anforderungen auseinander um eine konzeptionelle Lösung zu schaffen. Der Prototyp wird als Framework realisiert damit die nötige Flexibilität zur Erstellung des Workflows geboten werden kann. Dadurch ist es möglich verschiedenste Geschäftsprozesse abzubilden, wodurch eine große Gruppe von potentiellen Kunden bedient werden kann. Mit Hilfe der standardisierten SOAP und REST Schnittstelle kann eine Kommunikation mit externen Systemen, wie der Middleware erfolgen. Die Webservice sind durch einen Workflow-Manager mit der Kernlogik verbunden und leiten die Aufrufe an die hinterlegten Workflows zur Abarbeitung weiter. In Abschnitt 4.1.2 wird die Methodik zur Automatisierung der Abarbeitungsschritte beschrieben und stellt damit die Grundlage der Implementierung dar. Die Schnittstellendefinition für den Webservice gilt gleichzeitig als Datenbankschema, wodurch die übermittelten Informationen von außen direkt in eine Datenbank abgelegt werden können. Dadurch können die Daten persistent für Analysezwecke gespeichert werden ohne die Informationsstruktur zu verändern. Die Datenbank wird zugleich zur Erfassung der Zustände und eventueller Fehler genutzt. Das Framework sorgt für grundlegende Status- und Fehlerschnittstellen beim Anlegen eines neuen Webservices wie dem Fulfillment. Die Grundstruktur des Frameworks soll dem Entwickler bei der Workflow-Definition unterstützen.

5. Implementierung

Ziel dieses Kapitel ist die Vorstellung der Implementierung des Prototyps zum Nachweis der Machbarkeit. Vorgabe für die Realisierung ist die Verwendung von Microsofttechnologien, da sowohl der Auftraggeber SAB als auch der Augenoptiker über ein hohes Maß an Hintergrundwissen in diesem Bereich verfügen. Zusätzlich existieren bereits Lizenz- und Supportverträge mit Microsoft. Wir haben uns für die Programmiersprache C# mit der Verwendung des .Net Frameworks entschieden, da die Middleware des Kunden diese Komponenten bereits als Basis verwendet. Sollte sich in Kapitel 6 herausstellen, dass der Prototyp bereits als Grundlage für die Umsetzung des Kundenprojekts verwendet werden kann, stellt die Programmiersprache kein Hindernis dar. Aus der Wahl der Programmiersprache folgt die Nutzung der von Microsoft empfohlene Entwicklungsumgebung Visual Studio in der aktuellen Version 2013. Die Entwicklungsumgebung unterstützt uns bei der Planung und Realisierung, durch viele Automatismen und Konventionen wird die Entwicklung stark vereinfacht. Im ersten Teil wird der Aufbau der Anwendung beschrieben um einen Überblick über den Prototyp zu verschaffen, die folgenden Abschnitte nehmen darauf Bezug und beschreiben die einzelnen vorgestellten Komponenten näher. Die Konzepte aus dem vorigen Kapitel werden in den folgenden Abschnitten aufgegriffen und helfen beim Verständnis der Implementierung. Der Prototyp soll zeigen, ob die gewählten Anforderungen und erarbeiteten Konzepte realisierbar sind. Ziel ist es, eine Erkenntnis über die Realisierungsmöglichkeit zur allgemeinen Workflow-Automatisierung aus einem speziellen Auftrag ableiten zu können.

5.1 Struktureller Aufbau

Um die Machbarkeit der Wiederverwendbarkeit eines Kundenauftrags zu validieren, muss ein Prototyp entwickelt werden. Gewählt wurde hier die Optimierung eines Workflows und zwar der des Fulfillments, der sich abstrahieren lässt zu einem Workflow Framework, welches in Kapitel 4 näher beschrieben ist.

Die Struktur der Anwendung ist in der, von Visual Studio generierten *Code Map* in Abbildung 5.1 zu erkennen. Der Prototyp ist in die drei Komponenten *Services*,

Logik und *Daten* aufgeteilt, wobei sich die Logik wiederum in *Logic.Workflows*, *Logic.Activities* und die Kernlogik, *Logic.Core* unterteilt.

Die *Services.Core* implementiert den Webservice und dient als Schnittstelle nach außen. Hierüber geben externe Anwendungen neue Bestellungen ein und erhalten die Ergebnisse über deren Fortschritt. Der Service setzt sich dabei aus zwei Klassen zusammen, einem *BaseService* der die Statusanfragen und die Fehler nach außen weitergibt und die exemplarische Klasse *DefaultService* die vom Nutzer an das von ihm verwendete Datenmodell angepasst werden muss.

Die Serviceanfragen werden vom Webservice an die Logik (*Logic.Workflow*) weitergeleitet. Auch diese Komponente ist in zwei Klassen aufgeteilt, der *BaseWorkflow* der lediglich alle Anfragen des *BaseService* weiterleitet und der *DefaultWorkflow*. Hier kann der der Nutzer einen oder mehrere Workflows durch das Zusammenstellen von Activities konstruieren.

Die einzelnen Abarbeitungsschritte des Workflows kann der Nutzer in *Logic.Activities* anlegen, indem er für jede Activity eine eigene Klasse schreibt. Einmal definierte Activities können in dem Workflow beliebig oft wiederverwendet werden. Um das Schreiben der Activity-Klassen zu ermöglichen muss sich an einen von uns vorgegebenen Standard gehalten werden. Dieser wird in *Logic.Core* festgelegt, worauf in Abschnitt 5.5 genauer eingegangen wird.

In *Logic.Core* befindet sich der Workflow-Manager, der das Konzept des in Abschnitt 4.1.3 beschriebenen Workflow-Managers umsetzt und zusätzlich noch die Recovery (siehe Abschnitt 4.2) und Statusüberwachung (siehe Abschnitt 4.4) übernimmt. An den Workflow-Manager wird der in *DefaultWorkflow* festgelegte Workflow mitsamt der Bestellung, in Abbildung 5.1 als generischer Typ *TContext* dargestellt, übergeben. Auch die Funktionen aus dem *BaseWorkflow* werden an den Workflow-Manager weitergeleitet.

Der Workflow-Manager sammelt zusätzlich zu jeder Bestellung den dazugehörigen Workflow, startet ihn in einem eigenen Thread und merkt sich die Referenz in einer Liste. Für die Ausführung des Workflows wurde eine eigene Klasse namens *Execution-Manager* erstellt auf die in Abschnitt 5.6 genauer eingegangen wird. Jede Änderung, ob Statusänderung oder allgemeiner Fortschritt im Workflow wird von dem Execution-Manager an den Workflow-Manager gesendet, welche dieser in einer Datenbank persistent speichert. Eingehende Statusanfragen kann der Workflow-Manager anhand der gespeicherten Workflow-Referenzen direkt bei dem Execution-Manager anfragen, oder, falls der Workflow bereits durchgelaufen ist und sich daher nicht mehr im Speicher befindet aus der Datenbank geladen werden.

Der persistente Speicher wird durch eine relationale Datenbank abgebildet. Diese wird anhand der vom Nutzer geschriebenen Modelle, in dem Fall des Fulfillments die Bestellung, automatisch generiert. Dafür muss das Modell des Nutzers von einem von uns vorgegebenen Modell *BaseEntity* erben, siehe *Data.Core* in Abbildung 5.1 Durch die Vererbung erhält sein Modell eine interne *Id* zugewiesen und die Beziehung zur Status- und Recovery-Tabelle.

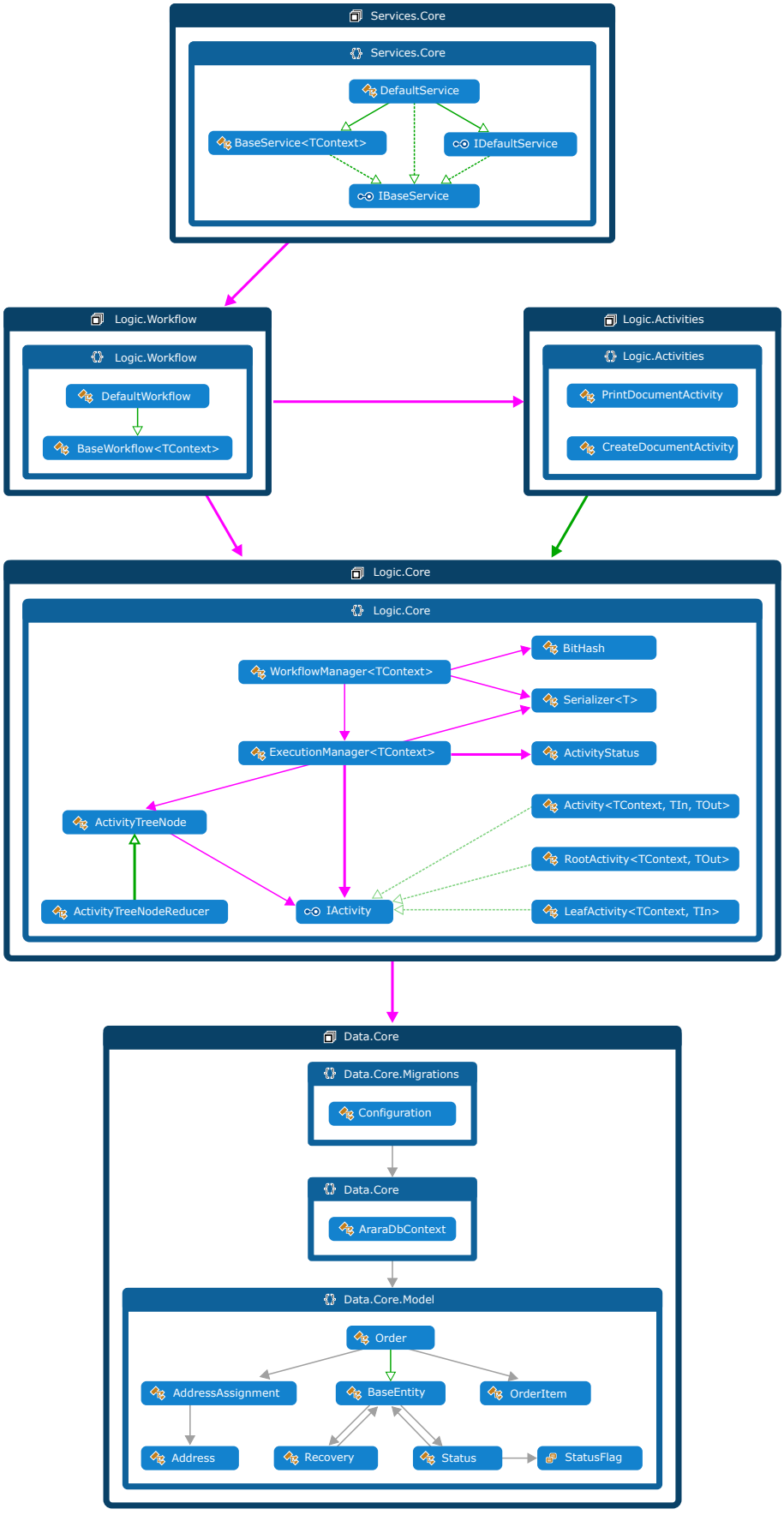

Abbildung 5.1: Code Map

5.2 Webservices

Aus Abschnitt 4.1.1 geht hervor, dass ein Webservice die ideale Technologie zur Realisierung der Anbindung an eine Middleware darstellt. Um den Service anzubieten soll ein SOAP-Webservice mit der Windows Communication Foundation (kurz WCF) erstellt werden. Mit Hilfe dieses Framework ist die Erstellung einer sicheren dienstorientierten Anwendung zur Verarbeitung von Geschäftstransaktionen möglich. Bei der Erstellung eines WCF-Services ist die abstrahierte Aufteilung des Endpunktes in drei Bereiche nach dem *ABC-Prinzip* zu beachten:

- **A**ddress – Zugriffsadresse
- **B**inding – Kommunikationsbeschreibung
- **C**ontract – Dienstleitungsbeschreibung

```
1  /// <summary>
2  /// Service to handle fulfillment.
3  /// </summary>
4  [ServiceContract]
5  public interface IDefaultService : IBaseService
6  {
7      /// <summary>
8      ///    Starts a Workflow to handle fulfillment.
9      /// </summary>
10     /// <param name="context">All information of an order.</param>
11     /// <param name="callBackUrlError">Url—Call on error.</param>
12     /// <param name="callBackUrlSuccess">Url—Call on success.</param>
13     /// <returns>Identifier of the Workflow.</returns>
14     [OperationContract]
15     long? Fulfillment(Order context,
16                       string callBackUrlError,
17                       string callBackUrlSuccess);
18 }
```

Quelltext 5.1: Fulfillment Interface

Aufbauend auf den Grundlagen der Anwendungsschnittstelle aus Abschnitt 2.3.2 soll die Implementierung betrachtet werden. Die Umsetzung mit C# und WCF folgt einem omnipräsenten Grundgedanken von Microsoft *Konvention vor Konfiguration* (engl. convention over configuration). Durch dieses Paradigma wird die Komplexität der Konfiguration reduziert und das KISS-Prinzip (engl. keep it short and simple) angewendet. Dadurch genügt es in C# den *Contract* zu definieren, siehe Quelltext 5.1. Der gezeigte *Contract* ist ein Interface bestehend aus zwei Teilen: dem `ServiceContract` und dem `OperationContract`. Beide Annotationen werden benötigt, um einen Service zu definieren. Visual Studio erstellt automatisch die notwendige WSDL-Datei, welche das *Binding* und die *Address* enthält, bei Bedarf können diese Angepasst werden, was für unseren Prototypen nicht notwendig ist. Abbildung 5.2 zeigt alle Schnittstellen des Prototyps, mit der Adresse *http://localhost:28375/DefaultService.svc*, dem *BasicHttp-Binding* und dem Contract `IDefaultService` auf der linken Seite. Die rechte Seite zeigt die geforderten Parameter: *context*, *callBackUrlError*, *callBackUrlSuccess* des Fulfillment-Webservices, welche auch im Codeabschnitt zu sehen sind.

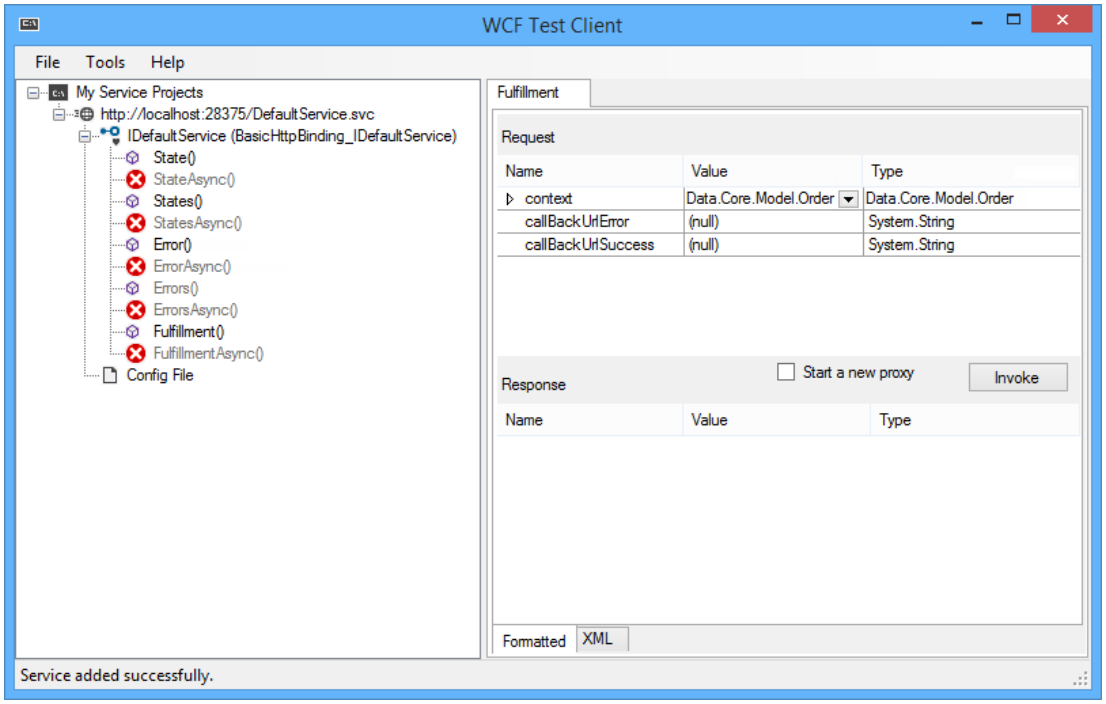

Abbildung 5.2: WCF Test-Client

Jeder Service, der von unserem Interface `IDefaultService` erbt muss die Fulfillment-Methode mit der gezeigten Signatur anbieten. Erwartet wird als Parameter eine Order (siehe Abbildung A.1 im Anhang), eine Service-URL für den Fehlerfall und eine Service-URL im Erfolgsfall. Die Service-URLs können für die beschriebene Callback-Funktionalität aus Abschnitt 4.4 angegeben werden, ist dies nicht gewünscht kann *null* als Parameter übergeben werden. Der Rückgabewert als *nullable long* ist der in Abschnitt 4.4 beschriebene Identifikator des Workflows zur Bearbeitung der Bestellung. Im Fehlerfall wird null zurückgegeben. Mit Fehlerfall ist an dieser Stelle ein Fehler innerhalb des Workflow-Managers gemeint und nicht ein Fehler im Workflow. Damit der Status beziehungsweise Fehler anhand des zurückgegebenen Identifikators wie in Abschnitt 4.4 gefordert zurückgegeben werden kann sind zusätzliche Methoden innerhalb des erstellten Service notwendig. Daher erbt `IDefaultService` vom `IBaseService`, welches die notwendigen Methoden fordert.

Die Serviceklasse `DefaultService` (siehe Quelltext 5.2) implementiert die geforderte Fulfillment-Methode, die Funktionalität ist im Logikbereich in der Klasse `Default-Workflow` enthalten. Diese logischen Trennungsebenen wurden bei der Programmierung bewusst gewählt, damit später Komponenten einfacher ausgetauscht werden können. Durch das Konzept der Dependency Injection (siehe Abbildung A.2 im Anhang) kann beispielsweise mit Ninject[1] die Objektabhängigkeit zur Laufzeit angepasst werden.

Da die Webservices die einzigen Schnittstellen zum Prototyp darstellen, werden neben dem Service zur Bearbeitung des definierten Workflows auch Services für die Statusüberwachung und der Fehlerausgabe angeboten. Diese zusätzlichen Services sind in Abbildung 5.2 abgebildet und heißen wie folgt:

[1] http://www.ninject.org/

```
1  /// <summary>
2  /// Service to handle fulfillment.
3  /// </summary>
4  public class DefaultService : BaseService<Order>, IDefaultService
5  {
6      /// <summary>
7      ///    Start a Workflow to handle fulfillment.
8      /// </summary>
9      /// <param name="context">All information of an order.</param>
10     /// <param name="callBackUrlError">Url-Call on error.</param>
11     /// <param name="callBackUrlSuccess">Url-Call on success</param>
12     /// <returns>Identifier of the Workflow.</returns>
13     public long? Fulfillment(Order context,
14                              string callBackUrlError,
15                              string callBackUrlSuccess)
16     {
17         return DefaultWorkflow.Fulfillment(context,
18                                            callBackUrlError,
19                                            callBackUrlSuccess);
20     }
21 }
```

Quelltext 5.2: Fulfillment Implementierung

- *State()* gibt den Status einer Bestellung anhand eines Identifikators zurück
- *States()* gibt alle Status aller Bestellungen als Liste zurück
- *Error()* gibt den Fehler einer Bestellung anhand eines Identifikators zurück
- *Errors()* gibt alle Fehler aller Bestellungen als Liste zurück

Damit diese Services angeboten werden, müssen sie im Contract definiert sein. Da sie jedoch bei jedem Webservice benötigt werden, haben wir uns für einen Contract `IBaseService` entschieden, welcher die Servicedefinitionen enthält. Quelltext 5.1 zeigt, wie `IDefaultService` vom `IBaseService` erbt und somit den Contract an den Service weitergibt. Die Serivcelogik ist in der Klasse `BaseService` definiert, Quelltext 5.2 zeigt das `DefaultService` von `BaseService<Order>` erbt und somit die notwendigen Funktionalitäten enthält. `BaseService` wurde generisch entwickelt, das bedeutet es kann mit jedem Datentyp umgehen, der für die Abwicklung eines beliebigen Workflows benötigt wird. Mit dieser Realisierung der Schnittstelle können alle möglichen Geschäftsprozesse empfangen und abgewickelt werden. [Mic15c, Rei03]

5.3 Workflow

Der Workflow bildet die Grundlage des Frameworks. Mit seiner Hilfe lässt sich ein automatisierbarer Ablauf von Abarbeitungsschritten darstellen. Die `Logik.Workflow` Komponente aus Abbildung 5.1 soll dem Nutzer dies ermöglichen, indem er vorher erstellte Activities zu einem Workflow kombiniert.

Die Activity ist so aufgebaut, dass sie einen Eingang und einen Ausgang hat. Die Kombination von Activities würde demnach nur zu einer Kette führen. Erst der

Einsatz einer Hilfsstruktur wie ein Graph, wobei jeder Knoten eine Activity ist erweitert das starre Konstrukt. Die Hilfsstruktur hat die Aufgabe die Ausgangsdaten beliebig vielen folgenden Activities zur Verfügung zu stellen und die Eingangsdaten zu sammeln, falls es mehrere Vorgänger gibt. Das bietet die Möglichkeit, mehrere Vorgänger und mehrere Nachfolger zu haben. Sollte eine Activity mehrere Nachfolger haben, kann auch von einer Verzweigung des Workflows gesprochen werden, wie links in Abbildung 5.3 dargestellt ist. Diese Verzweigung kann genutzt werden, falls die folgenden Aufgaben parallel ausgeführt werden können, da die folgenden Activities offensichtlich keine Abhängigkeiten zueinander haben. Verzweigungen können demnach auch zur Parallelisierung von Activities genutzt werden. Um die Ergebnisse mehrerer Activities zu sammeln, müssen sie alle denselben Nachfolger haben. Jede dieser Activities übergibt dann ihre Ausgabe an dieselbe Activity, welche alle Ergebnisse in einer Liste als Eingabe erhält. Diese Zusammenführung, die rechts in Abbildung 5.3 zu sehen ist, ermöglicht das konstruieren von Workflows, die dieselben Freiheiten wie ein gerichteter Graph besitzen. Das birgt zum einen das Potenzial nahezu beliebige Geschäftsprozesse abzubilden, aber auch die Gefahr unübersichtliche Workflows zu erstellen, was das Schreiben von Workflows erschwert.

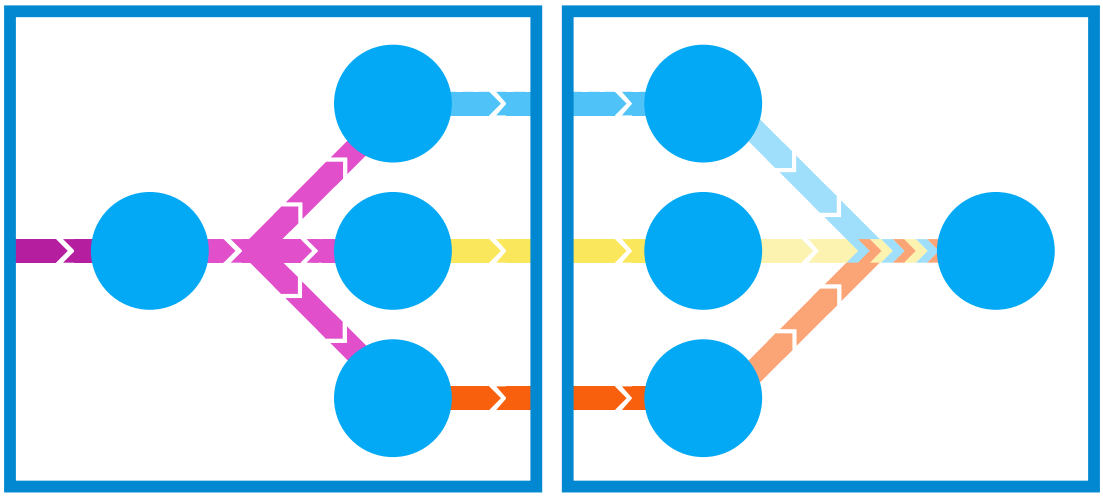

Abbildung 5.3: Workflowbeispiel

Um die Implementierung eines Workflows für den Nutzer möglichst einfach zu halten, haben wir uns gegen das Konzept eines Graphen und für einen Baum entschieden. Ein Baum bietet nur die Möglichkeit der Verzweigung, nicht aber der Zusammenführung. Das heißt es reicht, wenn jeder Knoten, in unserem Fall jede Activity ihre nachfolgende Activity kennt. Um die Logik der Workflowbildung von der, für den Nutzer sichtbaren, Activity Logik zu trennen, haben wir die Klasse `ActivityTreeNode` eingeführt, siehe Klassendiagramm in Abbildung 5.4. Jedes Objekt der Klasse `ActivityTreeNode` enthält als Eigenschaft genau eine Activity und eine Liste von Nachfolgern (ChildNodes). Da diese Klasse vom Anwendungsentwickler genutzt werden muss um den Workflow zu erstellen, wurden noch die Methoden *Count* und *Add* hinzugefügt. *Count* gibt die Anzahl der Kinder einer `ActivityTreeNode` wieder und beachtet dabei auch alle Kindeskinder. *Add* dient zum Hinzufügen von Activities und ist mehrfach überschrieben. Die Überschreibungen der Methode *Add* erlaubt es `Activities`, `ActivityTreeNode` oder eine Liste von Activities hinzuzufügen. Das

Hinzufügen eines `ActivityTreeNodes` ermöglicht das Anhängen bereits definierter Workflows.

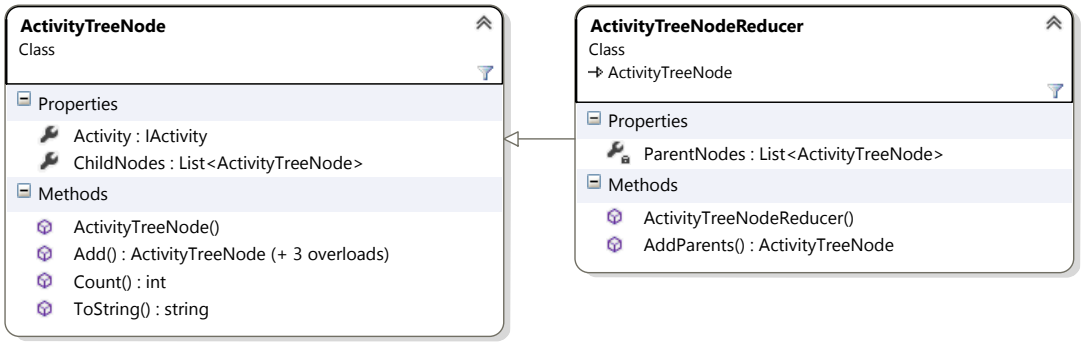

Abbildung 5.4: ActivityTreeNode

Um das Zusammenführen von Activities trotz Baumstruktur zu ermöglichen, haben wir die Klasse `ActivityTreeNodeReducer` eingeführt. Sie erweitert die Klasse `ActivityTreeNode` um die Liste `ParentNodes`. `ParentNodes` enthält `ActivityTreeNode` die vor der Ausführung der Activity beendet sein müssen, da die Ergebnisse für die Zusammenführung benötigt werden. Die Verwendung des `ActivityTreeNodeReducers` ist etwas umständlicher als der `ActivityTreeNode`, wird jedoch in den meisten von uns betrachteten Workflows nicht so häufig benötigt.

Der Nutzer schreibt für jeden gewünschten Workflow eine Klasse in die `Logik.Workflow` Komponente, ähnlich zum `DefaultWorkflow` aus Abbildung 5.1. Die mithilfe der `ActivityTreeNode` konstruierten Workflows werden nach dem Eintreffen von Bestellungen an den Workflow-Manager gesendet.

5.4 Activities

Die Activities bilden die einzelnen Abarbeitungsschritte eines Workflows ab und werden von den Framework Nutzern geschrieben. Damit das möglich ist, wird von dem Framework vorgegeben wie eine Activity auszusehen hat, beziehungsweise welche Funktionen implementiert sein müssen. Aus dem Konzept aus Abschnitt 4.1.2 geht hervor, dass jede Activity einen Eingang und einen Ausgang besitzt, über die sie Daten erhält und weitergibt. Der Datentyp der Ein- und Ausgabe soll der Nutzer selber festlegen können. Zusätzlich muss jede Activity noch eine Bestellung, im Folgenden als *Kontext* generalisiert, und einen Status enthalten.

Zusätzlich zu der Activity gibt es noch die `RootActivity`, die als Anfangs-Activity aus dem Konzept bekannt ist und die `LeafActivity`, auch als End-Activity bekannt. Die Namensgebung orientiert sich an eine Baumstruktur, bei der die Activities als Knoten angesehen werden. Wie aus Abbildung 5.5 hervorgeht, unterscheiden sich die beiden Activities nur durch das Fehlen der Eingabe- beziehungsweise Ausgabevariable.

Jede Activity hat demnach einen eigenen Eingabe- und Ausgabetyp. Das wird in C# mittels generischer Klassen dargestellt, die erst definiert werden wenn man sie benutzt, oder in diesem Fall wenn der Nutzer von ihnen erbt. Nachdem der Nutzer

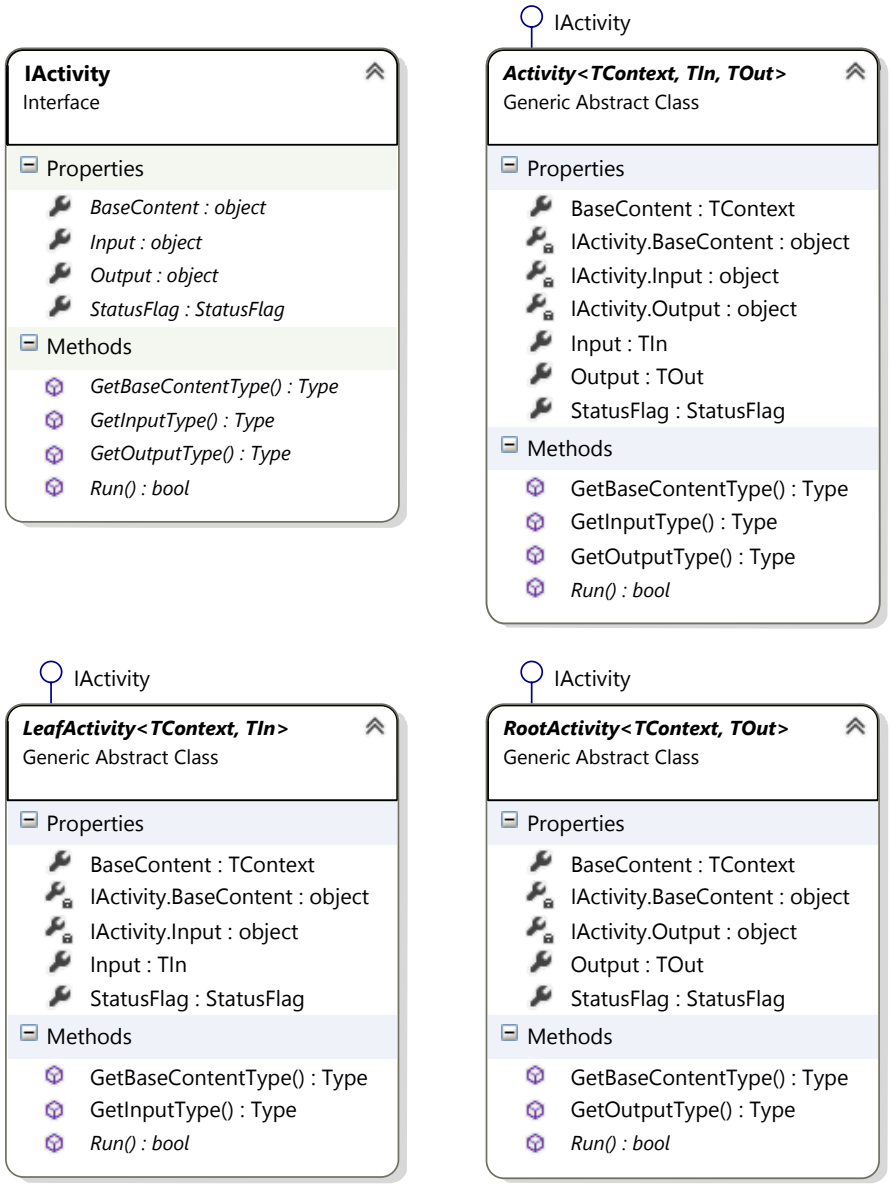

Abbildung 5.5: Activity Klassen

diese Vererbung durchgeführt hat, gibt es folglich viele verschiedene Klassen deren Typen alle unterschiedlich sein können. Da die verschiedenen Activities zur Ausführung generell angesprochen werden müssen und das bei verschiedenen Typen unter C# nicht möglich ist, wird eine abstrakte Activity erstellt. Das Interface `IActivity` bildet diese abstrakte Activity von der die Klasse `Activity` erbt.

In Abbildung 5.5 ist die `IActivity` zu sehen, die zusätzlich noch die Implementierung der nötigen Methoden erzwingt. Die wichtigste Methode für den Nutzer ist `Run`. In dieser Methode kann der Nutzer seinen Abarbeitungsschritt implementieren, welcher dann zur Laufzeit vom Execution-Manager ausgeführt wird. Über die Methoden `GetOutputType` und `GetInputType` kann sich der Nutzer zur Laufzeit die von ihm definierten Typen ausgeben lassen. Da die `IActivity` zwar vom Execution-Manager benutzt werden soll, aber keine Typen festgelegt werden können, sind alle

Eigenschaften von dem generellen Typ *Objekt*. Diese Eigenschaften müssen auch mit der Signatur (bestehend aus dem Namen der Methode sowie dem Typ und der Art aller ihrer formalen Parameter) von der Klasse `Activity` übereinstimmen. Das führt dazu, dass unabhängig von den Typen die der Nutzer für die Eingabe-, Ausgabe- und Kontextdaten festgelegt hat jedes dieser Objekte zum Typ *Objekt* umgewandelt wird.

Diese Typumwandlung nennt sich *boxing und unboxing* wobei es sich um die Konvertierung eines Wertetyps in den Typen *Objekt*, oder das Konvertieren eines Objekts in einen darauf aufbauenden Wertetyps handelt. Diese Konvertierung ist möglich, da in dem Typsystem von C# alle Werte eines Typs als Objekt behandelt werden können. Die Nachteile des *boxing und unboxing* sind, dass es einerseits speicherintensive Vorgänge sind und dass die Nutzung zu typunsicherem Code führt. Die Implementierung des *boxing* ist dafür relativ einfach, da die Konvertierung von einem speziellen zu dem generellen *Objekt* automatisch passiert. Lediglich die Rückführung benötigt den Typ in Klammern vor dem Objekt, damit der Compiler weiß welchen Typen der Wert des Objektes annehmen soll. Der Nutzer bekommt nichts von der Typumwandlung mit, außer dass die Methoden des `IActivity` dessen Rückgabetyp *Objekt* ist weiterhin für ihn zugänglich sind. Die Auswirkungen auf die Performance sind erfahrungsgemäß minimal, im Vergleich zum komplexeren Verfahren der *Reflections* bei dem der Objekttyp zur Laufzeit erkannt und die Methode demnach verändert wird.

5.5 Workflow-Manager

Nachdem die eintreffenden Daten einem passenden Workflow zugeordnet wurden, wird dieser mit den Daten an den Workflow-Manager geschickt. Er nimmt die Workflows entgegen, führt sie aus und liefert auf Anfragen hin Informationen über alle gespeicherten Workflows. Dazu verfügt der Workflow-Manager als einzige Komponente über einen Datenbankzugang. Hier werden neben den eintreffenden Workflows auch noch die betreffenden Daten, Status, auftretende Fehler und Wiederherstellungsinformationen gespeichert. Dadurch kann der Workflow-Manager jederzeit Auskunft über die Workflows liefern, aber auch Fehler an den Nutzer melden und fehlerhafte Activities wieder aufnehmen.

Der Workflow-Manager ist nach dem Singleton Pattern implementiert und lässt sich demnach nur einmal initialisieren. Das erklärt auch die Eigenschaft `Instance` und die private Variable *_instance* in Abbildung 5.6. Wenn der Service oder die Workflow-Klassen den Workflow-Manager aufrufen, so erhalten sie immer dieselbe Instanz. Dadurch wird verhindert, dass ein Workflow einem Workflow-Manager übergeben wird, die Abfrage nach dessen Status jedoch nicht denselben Workflow-Manager erreicht und somit nicht gefunden wird.

Die Übergabe eines Workflows geschieht durch die Funktion `Run`. Jede Bestellung bekommt eine eindeutige Identifikationsnummer zugewiesen, die sich aus dem Hash der Bestellung und dem Workflow zusammensetzt. Diese *ID* wird dem Aufrufer zurückgegeben damit er sie bei Anfragen als Referenz nennen kann. In der Methode `Run` wird daraufhin ein Execution-Manager (Abschnitt 5.6) erstellt, der Workflow als Parameter übergeben und anschließend ausgeführt. Die laufenden Workflows

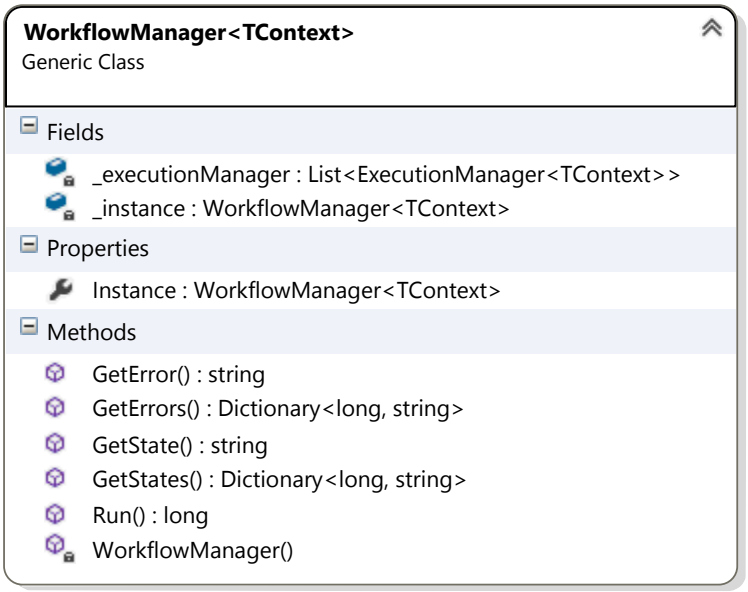

Abbildung 5.6: Workflowmanager

werden anschließend in der Liste `_executionManager` gesammelt bis sie vollständig durchgeführt wurden. Bei erfolgreicher Durchführung wird der Status des Workflows in die Datenbank geschrieben und die mit der Bestellung mit gegebene Callback-URL aufgerufen. Die Methoden `GetState` und `GetStates` geben den Status eines Workflows und die Status aller Workflows zurück. Die Methoden suchen dabei zuerst in der Liste der laufenden Workflows bevor sie in der Datenbank nach dem Status der gewünschten Bestellung suchen. Die Rückgabetypen von `GetStates` muss ein Dictionary mit der *ID* der Bestellung als Schlüssel sein, damit die Zuordnung zu dem Status erfolgen kann.

Die Fehlerbehandlung wird ebenfalls von dem Workflow-Manager geregelt. Während der Ausführung auftretende Fehler werden von dem Execution-Manager gesammelt und nach der Ausführung an den Workflow-Manager übergeben. Diese werden anschließend persistent in der Datenbank gespeichert und, sofern eine Callback-URL für Fehler der Bestellung mitgegeben wurde, wird auch diese aufgerufen. Die gespeicherten Fehler enthalten dabei sowohl die Fehlernachrichten als auch alle nötigen Daten um die betroffenen Activities nach der Reparatur wieder fortzusetzen. Diese umfassen die `ActivityTreeNodes` an denen der Fehler auftrat und die genutzten Eingabedaten. Die Daten werden dafür serialisiert als Binärdaten in der Datenbank gespeichert. Die Methode `GetError` liefert die Fehlernachrichten zu einer bestimmten Bestellung wohingegen `GetErrors` alle Fehlernachrichten und die entsprechenden Bestellungen ausgibt.

Um einen fehlerhaften Workflow wieder zu starten, muss dieselbe Bestellung erneut von der Middleware gesendet werden. Der Workflow-Manager überprüft nach Eingang jeder Bestellung, ob es in der Fehlertabelle eine Bestellung mit den gleichen Daten gibt. Konnte der Workflow einem Fehler zugeordnet werden, werden die gespeicherten Wiederherstellungsinformationen genutzt um den Workflow erneut auszuführen. Der Eintrag in der Fehlertabelle wird bei der Wiederaufnahme

des Workflows gelöscht, beim wieder auftreten eines oder des gleichen Fehlers wird ein neuer Eintrag erstellt.

5.6 Execution-Manager

Die Workflow-Ausführung besteht aus einer einfachen Klasse `ExecutionManager`, zu sehen in Abbildung 5.7, welche genau eine `ActivityTreeNode` entgegen nimmt und die öffentliche Methode `RunAll` bereitstellt. Neben dem Ausführen des Workflows kann noch die Ausführbarkeit des definierten Workflows überprüft werden. Die Überprüfung bietet dem Nutzer die Möglichkeit, seinen erstellten Workflow auf Kompatibilitätsfehler zu überprüfen. Damit sind nicht kompatible Objektschnittstellen gemeint, die auftreten, wenn die Ausgabe einer Activity und die Eingabe der darauf folgenden Activity nicht denselben Datentyp haben. Dies geschieht rekursiv für alle `ActivityTreeNodes` die in dem oberstem `ActivityTreeNode` enthalten sind. Dabei werden Activities welche nur einen Eingang oder nur einen Ausgang besitzen berücksichtigt.

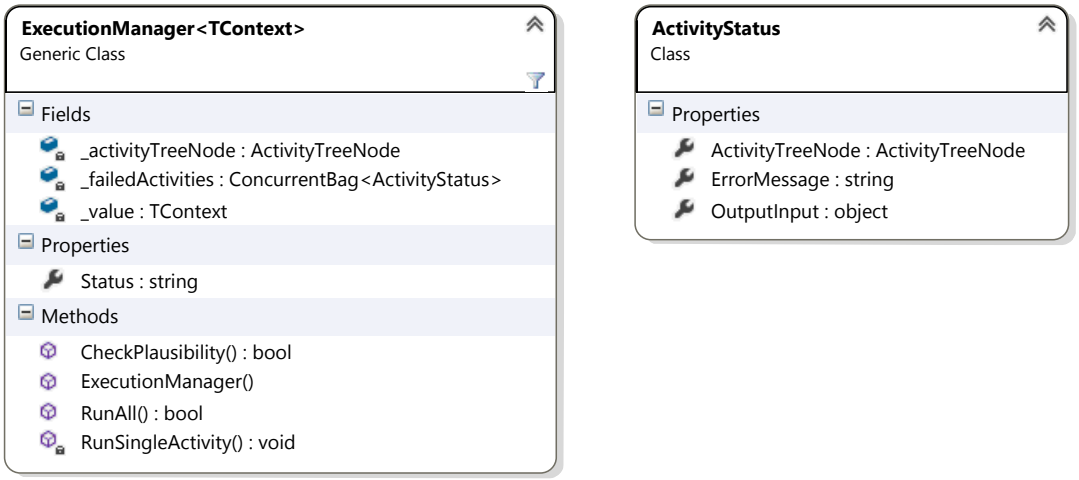

Abbildung 5.7: ExecutionManager

Um den Workflow zu starten muss die Methode `RunAll` aufgerufen werden. Nachdem sie den Workflow auf Kompatibilitätsfehler überprüft hat, startet sie die private Methode `RunSingleActivity`. Diese Methode benötigt einen `ActivityTreeNode` der ausgeführt werden soll, einen Kontext der jeder Activity mitgegeben werden soll und eine Eingangsvariable, die der nächsten Activity als Eingabe dient. Die Methode führt nun ihrerseits die Activity des `ActivityTreeNodes` in einem eigenem Thread aus. Nachdem die in der Activity enthaltende `run` Methode erfolgreich beendet wurde, wird für alle Kinder des gerade ausgeführten `ActivityTreeNodes` die Funktion `RunSingleActivity` aufgerufen. Dieses Mal mit jeweils einem Kind als `ActivityTreeNode`, demselben Kontext und der letzten Ausgabe des `ActivityTreeNodes` als neue Eingabe. Der Workflow endet sobald alle ausführbaren `ActivityTreeNodes` ausgeführt wurden.

Den aktuellen Fortschritt der Ausführung speichert der `ExecutionManager` nach jeder ausgeführten Activity. Diese Status können für Analysen des Workflows genutzt werden. Schlägt eine Activity fehl, wird der betreffende `ActivityTreeNode`,

die Fehlernachricht und das genutzte Eingabeobjekt in ein `ActivityStatus` überführt, siehe Abbildung 5.7. Das gespeicherte Objekt wird anschließend in einer Liste gespeichert um von dem Workflow-Manager abgerufen werden zu können. Diese Liste muss Threadsicher sein, da durch das Multithreading mehrere parallel ausgeführte Activities auf dieselbe Liste schreiben könnten. Von daher wird der von .Net bereitgestellte `ConcurrentBag` als Liste genutzt.

5.7 Persistenter Speicher

Bezugnehmend auf die Konzeptstruktur aus Abschnitt 4.1 soll im Folgenden auf den persistenten Speicher des Prototyps eingegangen werden. Der orange markierte Bereich in Abbildung 5.8 soll verdeutlichen, welcher Konzeptbereich umgesetzt werden soll.

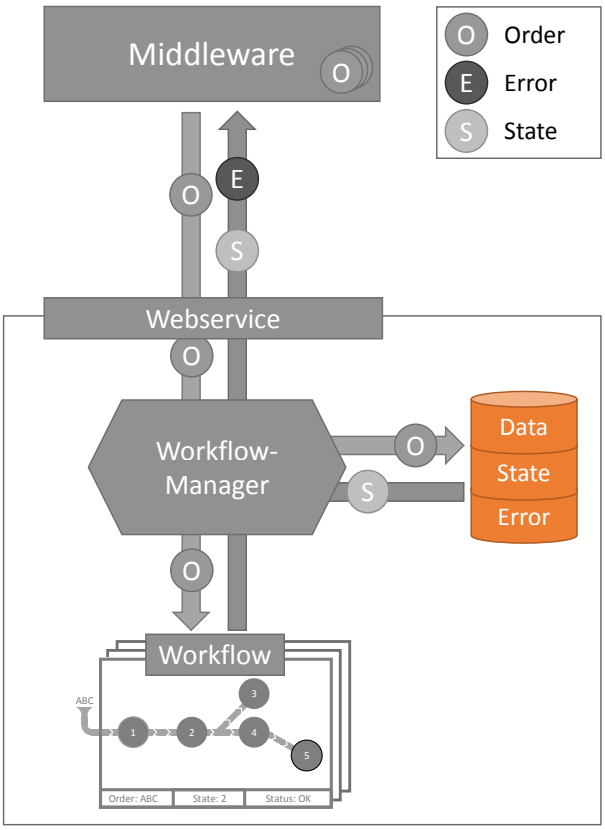

Abbildung 5.8: Persistenten Speicher des Prototyps

Mit der Vorgabe präferiert Microsoft Technologien einzusetzen, haben wir uns für das relationale Datenbankmanagementsystem Microsoft SQL Server[2] entschieden. Gespeichert werden sollen die übermittelten Daten, die Statuswechsel, die Fehler sowie die Zustände des Workflows.

Mit übermittelten Daten sind Informationen gemeint, die von einer externen Anwendung über einen Webservice an den Prototyp gesendet werden. Der Nutzer (in Form eines Anwendungsentwicklers) legt die Struktur des Daten-Modells für die Speicherung der zu übermittelnden Informationen selbst fest. Die Modelle basieren alle auf

[2] https://www.microsoft.com/de-de/server-cloud/products/sql-server/

dem gleichen Datenmodell, einem einfachen *Plain-old CLR objects* (kurz POCO), was zum Speichern und Transportieren beliebiger Daten dient. Ein POCO-Objekt ist lediglich eine Klasse mit Eigenschaften (engl. properties) ohne Methoden und Funktionalitäten. Voraussetzung ist, dass die Modell-Klassen von der abstrakten Klasse `BaseEntity` aus Abbildung 5.8 erben. Im Quelltext 5.3 ist unsere entwickelte `BaseEntity` abgebildet. Sie enthält eine intern (`Id`) sowie extern (`Identifier`) eindeutige Identifikationsnummer, eine Sammlung an Status (`Statuses`) und eine Wiederherstellungseigenschaft (`Recovery`). Durch diese globalen Eigenschaften können alle Statuswechsel erfasst und gespeichert werden. Die Wiederherstellung des Workflows wird durch den Typ Recovery (siehe Abbildung 5.8) gewährleistet. Dieser enthält den Workflow (`ActivityTreeNode`), den letzten Übergabewert für die Wiederaufnahme der Activity (`OutputInput`) sowie die Fehlermeldung (`ErrorMessage`) mit einem Zeitstempel (`DateTimeEvent`). Durch unser `BaseEntity` können alle geforderten Informationen erfasst werden und direkt in eine Datenbank überführt werden. Ermöglicht wird dies durch die Verwendung des *Entity Frameworks* mit der *Code First-Migration*. Microsoft erlaubt durch diese Migrationstechnik eine automatische Anpassung der Datenbank anhand des Modells. Der Nutzer hat somit keinen Kontakt mit der Datenbank und muss sich nicht um die Speicherung der Daten kümmern. Die Definition eines Modells genügt als Grundlage für die Workflow-Automatisierung.

Konkret für die Abwicklung des Fulfillments, stellt das Modell den Aufbau einer Kundenbestellung des Augenoptikers dar. Das Modell *Order* (siehe Abbildung A.1 im Anhang) stellt das Grundgerüst der Bestelldaten dar und erbt die Eigenschaften der `BaseEntity` um alle Funktionalitäten zu besitzen. Die externe Anwendung, welche die Daten zur Abarbeitung sendet, ist im Falle des Augenoptikers die Middleware. Die übermittelten Daten stellen die Informationen einer eindeutigen Bestellung dar. Das Modell *Order* definiert die Struktur und eine eindeutige Bestellung ist ein konkretes Objekt vom Typ *Order* inklusive der Bestelldaten.

Der Fulfillment-Webservice enthält als ersten Parameter den Typ Order wie in Abschnitt 5.3 gezeigt. Das Modell speichert die Bestellung für den gesamten Ablauf. Das *Order-Modell* als Transportobjekt der Bestellung wird vom Webservice gefordert und übergibt die Informationen in dieser Form an den Workflow zur Abarbeitung sowie Speicherung an die Datenbank. Abbildung 5.9 zeigt das automatisch erzeugte Datenbankschema des *Order-Modells*. Hierbei wird bewusst kein ER-Diagramm erzeugt, damit die erzeugten Datentypen erkennbar sind. Die `Order` (Bestellung) hat mehrere `OrderItems` (Bestellpositionen) und mehrere `AddressAssignments` (Adresszuordnungen). Die Bestellung kann beliebig viele Artikel umfassen und beliebig viele Kundenadressen aufnehmen, in unserem Fall existieren drei Typen, die Kundenadresse, die Lieferadresse und die Rechnungsadresse. Die Order ist nur für diesen speziellen Geschäftsprozess des Fulfillments notwendig und muss austauschbar sein. Daher erbt Order (oder jedes andere Modell) vom `BaseEntity`. `BaseEntity` gibt das Recovery sowie die Statusverwaltung vor. Dadurch kann der Workflow in `ActivityTreeNode` persistent gespeichert werden und jeder Statuswechsel für zukünftige Analysezwecke historisiert werden.

Diese Implementierung erlaubt somit die Definition eines beliebigen Modells als Struktur der Basisdaten für den entworfenen Workflow. Die automatische Integration

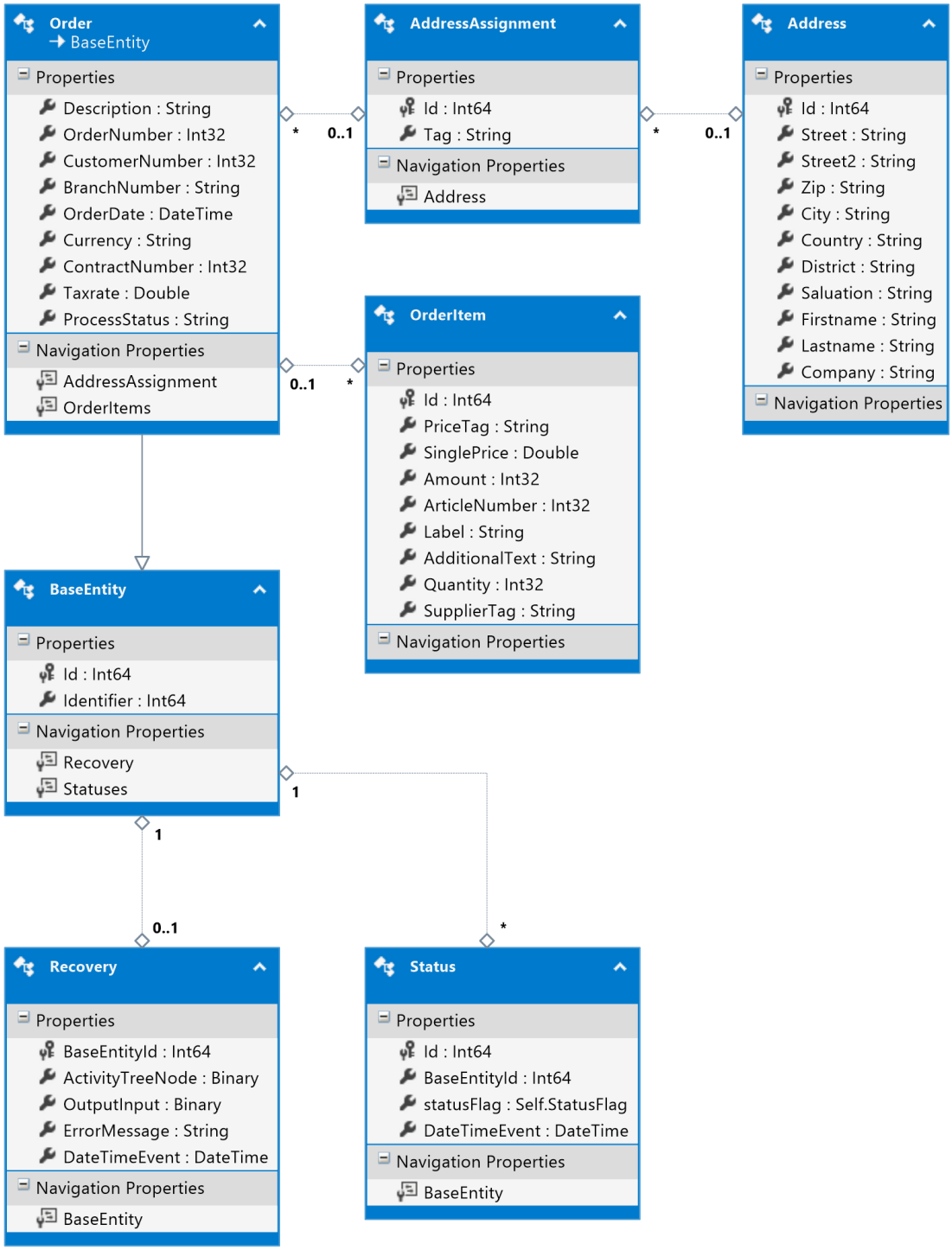

Abbildung 5.9: Datenbankschema

```
1  [Serializable]
2  public abstract class BaseEntity
3  {
4      [Key]
5      public long Id { get; set; }
6
7      public virtual ICollection<Status> Statuses { get; set; }
8
9      public virtual Recovery Recovery { get; set; }
10
11     public long Identifier { get; set; }
12 }
13
14 [Serializable]
15 public class Recovery
16 {
17     [Key]
18     [ForeignKey("BaseEntity")]
19     public long BaseEntityId { get; set; }
20
21     public virtual BaseEntity BaseEntity { get; set; }
22
23     public byte[] ActivityTreeNode { get; set; }
24
25     public byte[] OutputInput { get; set; }
26
27     [MaxLength(255)]
28     public string ErrorMessage { get; set; }
29
30     [Required]
31     public DateTime DateTimeEvent { get; set; }
32 }
```

Quelltext 5.3: BaseEntity

in allen Activities, ermöglicht jedem Abarbeitungsschritt den Zugriff auf die Daten. Die persistente Speicherung der übermittelten Daten, in einer relationalen Datenbank erlaubt komplexe Reporterstellungen für beliebige Analysen. Durch die persistente Speicherung von Statuswechsel und auftretenden Fehlern, können zusätzliche Services angeboten werden, die die Informationen auch nach Beendigung des Workflows an externe Anwendungen übermitteln können. Dadurch ist eine Historisierung der Status einer Bestellung möglich. Die Serialisierung und Wiederherstellung des gesamten Workflows realisiert einen Zustandsautomaten, der in Abschnitt 4.2 beschrieben wurde. [DGV+13]

5.8 Zusammenfassung

In den vorigen Abschnitten haben wir gezeigt, dass eine Implementierung auf Grundlage des erarbeiteten Konzepts realisierbar ist. Der Prototyp kann beliebige sowie beliebig viele Geschäftsprozesse erfassen und mittels einem definierten Workflow abarbeiten. Voraussetzung ist ein anpassbarer Typ je Geschäftsprozess. Dieser Typ definiert ein Modell für alle zu übertragenden Daten. Für den Augenoptiker ist der

definierte Typ die Bestellung (Order), da lediglich der Fulfillment-Geschäftsprozess abgebildet werden soll. Alle Bestellungen werden in diesem Format an den Prototypen übergeben. Die Schnittstelle ist ein Webservice, welcher in Abschnitt 5.2 beschrieben wurde. Jede Bestellung wird anhand des definierten Workflows bearbeitet. Der Workflow besteht aus einzelnen Activities, je Arbeitsschritt eine Activity. Die Überführung der empfangenen Daten vom Webservice an den Workflow übernimmt der Workflow-Manager und der Execution-Manager, welche sich zudem um einen reibungslosen Ablauf kümmern, hierzu gehören die Statusüberwachung, das Fehlerhandling und das Recovery. Zur korrekten Steuerung und um eine spätere Analyse zu ermöglichen werden alle notwendigen Daten persistent in einer relationalen Datenbank gespeichert. Der Aufbau der Daten entspricht wieder dem definierten Typ des Geschäftsprozesses.

6. Evaluierung

Im Kapitel 3 wurde anhand der gegenwärtigen Kundensituation eine Anforderungsanalyse durchgeführt. Das Augenmerk lag in der Findung von Voraussetzungen zur Erfüllung eines allgemeineren Problems, um ähnliche Kundenanforderungen mit einer zu implementierenden Lösung zu bedienen. Darauf aufbauend sollte überprüft werden, ob technisch eine Software entwickelt werden kann, die diese Anforderungen erfüllt. Kapitel 4 hat ausführlich unser Konzept einer wiederverwendbaren Workflow-Lösung vorgestellt. Die Konzeptrealisierung sollte Probleme und notwendige Komponenten aufzeigen, damit die geforderte Machbarkeit gezeigt oder widerlegt werden kann. Zu Beginn soll in diesem Kapitel auf die umgesetzten Anforderungen eingegangen werden, um zu zeigen welche Anforderungen vollständig, teilweise oder nicht umgesetzt wurden. Es soll gezeigt werden, ob das realisierte Konzept die notwendigen Eigenschaften besitzt, um den spezifischen Kundenauftrag zu erfüllen. Es soll gezeigt werden, ob der entwickelte Prototyp den geforderten Fulfillment-Prozess abbilden kann. Zugleich soll validiert werden, ob der Prototyp eine Workflow-Automatisierung darstellt. Im späteren Teil des Kapitels sollen Anregungen zur Verbesserung genannt werden, sowie die Nutzbarkeit des Prototypen kritisch hinterfragt werden. Durch den realisierten Prototyp sollte die Erkenntnis gewonnen werden, ob auf dem Markt bereits Produkte existieren um die ermittelten Kernprobleme zu lösen. Für nachfolgende Arbeiten sollen daher Softwarelösungen genannt werden, die dem Konzept unseres Prototyps nahe kommen um zu prüfen, ob die bestehenden Lösungen zur Umsetzung genutzt werden kann oder eine Weiterentwicklung unseres Prototypen sinnvoll ist.

6.1 Validierung

In der Validierung wird die Umsetzung aus Kapitel 5 den Anforderungen aus Abschnitt 3.2 gegenübergestellt und wertend diskutiert. Um die Leistung und Effizienz zu evaluieren wurden Messungen durchgeführt, die in Abschnitt 6.1.7 genauer beschrieben werden.

6.1.1 Anbindungsmöglichkeiten an eine Middleware

Für die Anbindung an die bestehende sowie an eine noch nicht näher bekannte Middleware sollte eine Schnittstelle geschaffen werden. Aus Abschnitt 4.1.1 ging hervor, dass ein Webservice durch die Implementierung von REST und SOAP ausreicht, um es aktuellen und zukünftigen Systemen zu ermöglichen mit der Anwendung zu kommunizieren.

Die Implementierung des SOAP Protokolls ermöglicht die Anbindung an die aktuelle Middleware. Die REST-Fassade hatte eine niedrigere Priorität und wurde aus Zeitgründen nicht umgesetzt. Einer Umsetzung steht jedoch technisch nichts im Wege, da es sich hierbei lediglich um eine REST Fassade handelt, die alle Anfragen intern an die SOAP-Schnittstelle weiterleitet. Bei den durchgeführten Tests mit der Middleware hat der SOAP-Service erwartungsgemäß funktioniert.

6.1.2 Definition von Abarbeitungsschritten

Einen Workflow durch die Definition seiner Abarbeitungsschritte darstellen und automatisch ausführen zu können stellt eine grundlegende Anforderung dar.

Die Definition eines Workflows kann von dem zukünftigen Anwendungsentwickler durch das Kombinieren von Activities vorgenommen werden. Die benötigten Activities schreibt der Entwickler in einzelne Klassen die einem von uns vorgegebenen Rahmen entsprechen. Der auf diese Weise entstehende Workflow ist in der Lage alle vorstellbaren betrieblichen Workflows abzubilden. Der benötigte Datenfluss zwischen den einzelnen Activities wird durch den Execution-Manager gewährleistet, selbst die parallele Ausführung von Activities funktioniert problemlos. Neben der Fähigkeit innerhalb eines Workflows Activities parallel zueinander auszuführen, können auch mehrere Workflows parallel ausgeführt werden.

Die größte Schwierigkeit bei der Nutzung liegt in dem Konzept des Frameworks selber. Hier wird von dem Nutzer erwartet eine Klasse zu schreiben, dessen Rahmenbedingungen wir vorgeben. Zwar helfen Beispiele an dem sich der Nutzer orientieren kann, jedoch würde eine graphische Oberfläche den Prozess deutlich vereinfachen. Die Anforderung einer Machbarkeitsanalyse erfüllt das entwickelte Framework. Für eine Weiterentwicklung unseres Prototyps wird eine graphische Oberfläche empfohlen. Damit das Userinterface ausgetauscht werden kann, empfehlen wir die Speicherung der definierten Abfolge in einer Datenbank.

6.1.3 Recovery

Unter dem Überbegriff Recovery sollten auftretende Fehler protokolliert werden, damit die Bearbeitung der betreffenden Bestellungen nach der Fehlerbehandlung fortgesetzt werden kann. Um die Anwendung nicht durch übermäßig viele Datenbankzugriffe zu verlangsamen, haben wir beschlossen nur nach einem auftretendem Fehler den Zustand der betreffenden Bestellung zu speichern. Das ermöglicht die Wiederaufnahme sobald die gleiche Bestellung erneut vom System erkannt wird, jedoch geht der Fortschritt der Bearbeitung verloren falls das komplette System versagt. In diesem Fall muss die Bestellung erneut bearbeitet werden, inklusive der

bereits durchgeführten Activities. Das führt in dem Unternehmen eventuell zu Zusatzkosten und einer längeren Bearbeitungsdauer, jedoch kann die Bestellung erneut ausgeführt werden. Es gehen keine Bestellungen durch ein Systemversagen verloren.

Das Recovery System hat sich in den Tests als sehr zuverlässig erwiesen. Die Bedenken bezüglich der Geschwindigkeit waren berechtigt, da sich in den Tests bestätigt hat, dass die Datenbankzugriffe einen Großteil der Durchführungszeit benötigen. Aus den durchgeführten Tests ergab sich dabei eine Dauer von ungefähr *150ms*, bei einem kleinem Workflow und einer kleinen Bestellung. Die Größe oder Dauer des Workflows hat keine Unterschiede gemacht. Alleine die Größe der Bestellung beeinflusst die Dauer des Datenbankzugriffes. Zwar werden nur fehlerhafte Activities gespeichert, doch wird bei jeder eingehenden Bestellung in der Datenbank gesucht, ob diese Bestellung bereits fehlerhaft durchgeführt wurde. Hinzu kommt noch, dass die Parallelität bei Datenbankschreibvorgängen nicht genutzt werden kann.

6.1.4 Analysemöglichkeiten

Um den Workflow des Kunden zu optimieren ist eine Anforderung an die Software die Möglichkeit der Workflow-Analyse. Um einen breites Spektrum an Analysen zu ermöglichen wird nach jeder ausgeführten Activity ein Eintrag in die Statustabelle der Datenbank geschrieben. Dieser enthält neben einem Zeitstempel und einer Statusbeschreibung auch den Identifikator (*ID*) der Bestellung um mögliche Abhängigkeiten zu der Bestellung aufzeigen zu können.

Durch den Umfang der gesammelten Daten ist die Anforderung einer Analysemöglichkeit erfüllt. Eine mögliche Analyse ist die benötigte Ausführungszeit der einzelnen Activities in einem Workflow, in Abhängigkeit zu der verwendetem Bestellung. Diese Analyse eignet sich besonders um einzelne Teilprozesse innerhalb eines Workflows zu optimieren. Möglicherweise erweisen sich auch bestimmte Bestellungen als Zeitintensiver, und sollten vorzeitig gelagert werden. Zusätzlich kann auch die Kundenbestellrate und die Auslastung des Systems ermittelt werden um die Hardware an die Auslastung anpassen zu können.

Aus unserer Sicht werden alle, für Analysen interessanten, Ereignisse protokolliert. Innerhalb der Activities muss sich der Anwendungsentwickler selber um die Protokollierung kümmern, wenn ihm eine Statusnachricht pro Activity nicht ausreicht. Hier könnte man ihm noch eine Protokollierfunktion bieten, die beliebige Nachrichten mit Zeitstempel und einem Kontext, wie die aktuelle Bestellung, in die Datenbank schreibt. Das bietet wiederum einen sehr flexiblen Ansatz der Protokollierung, die die Möglichkeit bietet große Datenmengen über den Inhalt einer Activity zu sammeln.

6.1.5 Erweiterbarkeit

Die Anforderung der Anpassbarkeit bezieht sich auf die Service-Schnittstelle, Modelle, Datenbank und den Workflow. Bei Anpassungen der genannten Komponenten soll die unterliegende Logik, das Grundgerüst, nicht verändert werden müssen.

Die Service-Schnittstellen erfüllen die Anforderungen, da dem Nutzer eine eigene Klasse zur Verfügung gestellt wird. Durch unsere vorgegebenen Konventionen enthält jede vom Nutzer angelegte Service-Schnittstelle bereits grundlegende Services.

Zwar kann die Funktionsweise der Workflows nicht erweitert werden, dennoch lassen sich durch die selbst erstellbaren Activities beliebige Workflows zusammenstellen. Auch hierfür muss die zugrundeliegende Logik nicht verändert werden.

Die Datenbank wird aus dem Datenmodell generiert, welche für die Service-Schnittstelle genutzt wird, und enthält die für das Framework notwendigen Tabellen. Anpassungen können hier nur an dem Datenmodell vorgenommen werden, da sie sonst von der Datenbankmigration überschrieben werden. Das Datenmodell kann durch beliebige Modelle erweitert werden. Somit ist auch hier die Anforderung erfüllt.

6.1.6 Benutzerfreundlichkeit

Damit die geschriebene Software als Grundlage zukünftiger Weiterentwicklungen genommen werden kann, sollte der Quellcode selbsterklärend sein. Dazu gehören aussagekräftige Methodenbezeichner, passende Variablen und vor allem ausreichend viele und präzise Dokumentationen. Eine einheitliche Formatierung und der Gebrauch der englischen Sprache ist ebenfalls eine Grundvoraussetzung für guten Quellcode. Wir haben uns vorrangig an die .Net Entwicklungsrichtlinien[1] von Microsoft gehalten, womit neben dem besagten Quellcodeverständnis auch die Architektur der Software empfohlen wird. Diese Richtlinie steigert das Verständnis und den Wiedererkennungswert der Software besonders für .Net Entwickler, aber auch für Programmierer ohne Microsoft-Affinität.

Einen weiteren Teil der Quellcodelesbarkeit macht die Struktur der gesamten Software aus. Um die Architektur möglichst schnell zu verstehen wurden eine Code-Map und ein Architekturdiagramm erstellt. Zusätzlich wurden noch das Datenbankschema und alle Klassendiagramme erstellt. Aus dem Quellcode und den Kommentaren kann zusätzlich, falls benötigt, eine automatische Dokumentation erstellt werden.

6.1.7 Leistung und Effizienz

Damit die von uns entwickelte Software im Betrieb eingesetzt werden kann, muss sichergestellt werden, dass sie sich über einen längeren Einsatzzeitraum vorhersagbar verhält. Das betrifft vorrangig die Ausführungsdauer von Aufträgen bei steigender Anzahl von Aufträgen.

Zur Simulation des Vorgangs haben wir eine Analyse durchgeführt. Die Bestellungen bestehen aus generierten Zufallsdaten wobei jede Bestellung eine Größe von *100 Byte* enthält. Zusätzlich wurde ein Workflow festgelegt, dessen Activities zum Teil parallel ausgeführt werden können. Da es innerhalb des Workflows auch Activities geben kann, die aufwendiger sind, bzw. länger dauern, wurde die Analyse mit kleinen und großen Activities ausgeführt. Der kleinere Workflow, im kommenden *Small Workflow* genannt, enthält eine Vielzahl von unterschiedlichen Rechnungen und Umwandlungen bei denen die Eingangsdaten und die Daten der Bestellung verwendet wurden. Um sicher zu stellen, dass die Ergebnisse der Rechnungen nicht wiederverwendet werden wenn sie sich noch im Cache befinden, werden alle Rechnungen um Zufallszahlen erweitert. Die Dauer der Berechnungen wurde durch `Threading.Sleep` künstlich auf *1ms* angehoben. Bei dem größeren Workflow, *Big Workflow* genannt,

[1] https://msdn.microsoft.com/en-us/library/hh156542(v=vs.110).aspx

wurde auf alle Activities gleichmäßig eine Arbeitslast von *49 ms* verteilt, sodass der Workflow insgesamt *50 ms* dauert. Die Simulation wurde auf einem Computer mit vier Prozessorkernen und acht Threads ausgeführt, was demnach maximal acht gleichzeitige Berechnungen erlaubt. Diese können auch vollständig von der entwickelten Engine genutzt werden.

Abbildung 6.1 zeigt die Ausführungsdauer, die die Workflow-Engine benötigt um eine bestimmte Anzahl an Aufträgen zu verarbeiten. Die Werte des Graphen bilden den optimierten Durchschnitt aus *100 Messungen* und können aus der Tabelle A.1 im Anhang entnommen werden. Als Orientierungsgröße dienen die reinen Berechnungen die in den Workflows durchgeführt werden. Sie enthalten weder Datenbankzugriffe noch Parallelität innerhalb der Rechnungen und sind in der Abbildung 6.1 als *Small Workflow* und *Big Workflow* gekennzeichnet. Um ein Verständnis für die Datenbankzugriffsdauer zu erhalten, wurden diese Workflows um die Recovery-Funktionalität erweitert und als *Small Workflow + Recovery* bzw. *Big Workflow + Recovery* gekennzeichnet. Das von uns entwickelte Framework erweitert die Recovery-Funktionalität um die Protokollierung, die ebenfalls in die Datenbank geschrieben wird und um die parallele Ausführung von einzelnen Activities. Das Framework ist in dem Diagramm als *Small Workflow + Framework* und *Big Workflow + Framework* aufgeführt.

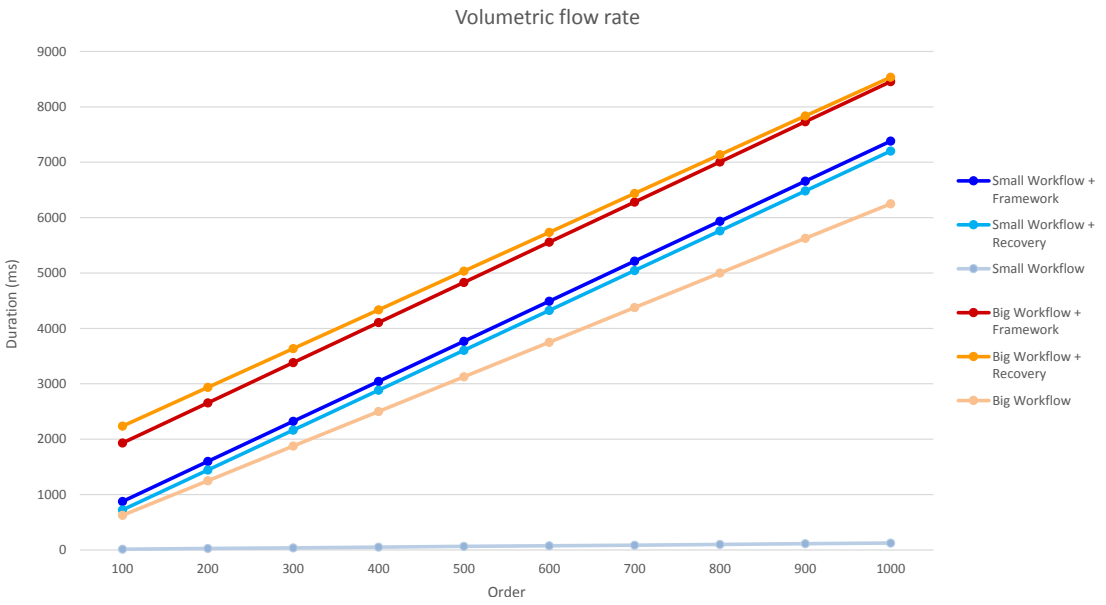

Abbildung 6.1: Benchmark

Primär geht aus den Messungen hervor, dass die Dauer pro Bestellung linear mit jeder weiteren Bestellung ansteigt. Das ermöglicht ein einfaches Vorhersagen der Dauer sowie die Nutzung der Software über einen langen Zeitraum.

Anhand der Durchläufe mit dem kleinen Workflow, in der Grafik die Kurven in Blautönen, kann man gut sehen, dass auch bei einem minimalen Rechenaufwand die Wiederherstellung einen großen Teil der Bearbeitungsdauer benötigt. Das Framework, welches zusätzlich zu der Wiederherstellung noch die Protokollierung und die Parallelität innerhalb des Workflows bietet, benötigt etwas länger. Das liegt daran,

dass die Parallelität bei sehr kleinen Workflows im Verhältnis zu dem Protokollierungsaufwand kaum einen Unterschied macht.

Der große Workflow, in der Grafik durch Gelb- bis Rottöne dargestellt, benötigt im Verhältnis zu der viel größeren Berechnung pro Bestellung nicht viel mehr Zeit als der kleine Workflow. Obwohl die Bearbeitungsdauer pro Bestellung um *500 ms* angehoben wurde, benötigt die gleiche Anzahl an Bestellungen nur *1055 ms* mehr Zeit. Der geringe Anstieg ist größtenteils der parallelen Abarbeitung der Workflows zu verdanken, die sehr effektiv durch das .Net Framework in Kombination mit dem Entity Framework durchgeführt werden. Bei dem großen Workflow ist bereits eine Optimierung durch unser Framework zu erkennen, da es etwas schneller als nur das Recovery ist. Da durch das Framework auch jeder Workflow in kleinere Teile unterteilt ist und parallel bearbeitet wird, nutzt es das Mehrkernsystem besser aus als ohne die Unterteilung. Das Recovery nähert sich jedoch der Optimierung des Frameworks an.

In der Praxis ist es zwar unwahrscheinlich, dass jede Sekunde mehr als *100 Bestellungen* eintreffen, dennoch hat die Untersuchung gezeigt, dass das Framework gut mit langwierigen Aufgaben zurechtkommt und auch über einen langen Zeitraum nicht an Effektivität verliert. Der nächste Verbesserungsansatz wären die Datenbankzugriffe, die aktuell noch auf der Festplatte stattfinden und somit den Großteil der Bearbeitungsdauer ausmachen. Abhilfe könnte ein intelligentes Caching oder eine Hauptspeicherdatenbank (engl. In-Memory-Database) liefern.

6.2 Probleme und grundlegende Erkenntnisse

Durch das entwickelte Konzept und die Implementierung wurde deutlich, welche Faktoren für die automatisierte Abarbeitung des spezifischen Kundenauftrags notwendig sind. Damit die Anwendung wiederverwendet werden kann, wurde eine Workflow-Automatisierung gewählt. Die erfolgreiche Machbarkeitsanalyse konnte zeigen, dass der Prototyp zur Abwicklung des Fulfillments fähig ist. Auf Grund der Abstraktion der Anforderungen und Anwendungsimplementierung können beliebige Geschäftsprozesse abgebildet und automatisiert werden. In einer späteren Arbeit soll geprüft werden, ob sich die Weiterentwicklung des Prototyps rentiert oder der Einkauf eines bestehenden Workflow Management Systems zur Realisierung der Kundenanforderungen vorzuziehen ist. Ziel ist es die optimale Lösung für den Augenoptiker zu finden und zugleich ein Produkt für ähnliche zukünftige Kundenanforderungen vorrätig zu haben.

Tabelle 6.1 verschafft einen Überblick über die erkannten Probleme und Vorteile des Prototyps. In diesem Kapitel soll auf die einzelnen Punkte näher eingegangen werden.

Der Prototyp befindet sich in einem sehr frühen Stadium und bietet dem Anwender, dem Anwendungsentwickler, wenig Komfort beim Erstellen des Workflows. Dies ist der Tatsache geschuldet, dass hierauf kein Schwerpunkt gelegt wurde. Im Fokus stand die Validierung der Realisierungsmöglichkeit, bezüglich der Frage, ob die beschriebene Kundensituation mit einer selbst entwickelten Workflow-Automatisierung umgesetzt werden kann. Der entstandene Prototyp erfüllt bereits die notwendigen

Vorteile	Probleme
Machbarkeit ist bestätigt	Geringer Komfort bei der Workflowerstellung
Abbildung des Fulfillments möglich	Programmieraufwand notwendig
Fehlerbehandlung enthalten	Hardware-Ausfälle wird nicht abgefangen
Typsichere SOAP-Schnittstelle	Fehlende REST-Schnittstelle
Ressourcenschonend	Keine Analyse implementiert
Ablaufprotokollierung enthalten	Kein automatisches Reporting enthalten

Tabelle 6.1: Vorteile und Probleme des Prototyps

Punkte zur Erreichung des Kundenziels, der Abwicklung des Fulfillments. Der aktuelle Ansatz stellt eine einfache, ressourcenschonende Komponente zur Erstellung von Workflows dar. Zur Erstellung eines Workflows ist derzeit ein geringer Programmieraufwand erforderlich. Es müssen Klassen zur Abarbeitung erstellt werden, welche einer vorgegebenen Objektschnittstellendefinition folgen. Bei kommerzieller Software ist eine Workflowerstellung zum Teil ohne Programmierkenntnisse möglich, sofern sich an die Geschäftsprozessmodellnotation BPMN (Business Process Model and Notation[2]) gehalten wird. Workflows lassen sich teilweise durch vorgefertigte Komponenten über eine grafische Oberfläche aus einer Sammlung von Aktivitäten zusammenstellen.

Für die Anbindung an mobile Services fehlt beim aktuellen Prototyp eine REST-JSON-Schnittstelle, als Ergänzung der aktuellen SOAP-XML-Schnittstelle. REST-JSON ist kompakter, effizienter und ermöglicht eine schnelle Implementierung, gegenüber dem typsicheren SOAP-XML. Für den Prototyp standen die Korrektheit und die Sicherheit im Vordergrund, statt Performance. Durch die verwendete Architektur ist einer Erweiterung um eine REST-Fassade sehr einfach zu integrieren.

Der Prototyp besitzt bereits eine Protokollierung und Fehlerbehandlungen, welche in professionellen Softwarelösungen standardmäßig enthalten ist. Die Protokollierung der Systemzustände ist für eine erste Version ausreichend umgesetzt, bei einem Absturz beziehungsweise Totalausfall der Hardware fehlt die korrekte Wiederherstellung der Daten. Der Prototyp gewährleistet jedoch eine Wiederholung der Abarbeitung, sodass der Betrieb des Unternehmens nicht zum Erliegen kommt, es aber zu Dopplungen in der Bestellungsabwicklung kommen kann. Dies ist für einen Prototypen im Testsystem akzeptable im Produktivsystem können hierdurch Zusatzkosten und Zeitverzögerungen entstehen.

Im Abschnitt 6.1.7 wurde gezeigt, dass die Abarbeitungszeit linear mit der Anzahl der Aufträge wächst, was positiv hervorzuheben ist. Die Wiederherstellungskomponente benötigt eine Evaluierung bezüglich der Optimierungsmaßnahmen zur Reduzierung der Abarbeitungszeit. Benötigt wird ein zeitlicher Vergleich mit etablierten Lösungen am Markt um die ermittelten Werte besser einschätzen zu können.

Unser Prototyp enthält kein automatisches Reporting, weder die Erfassung von neuen Bestellungen noch die Benachrichtigung über abgearbeitete oder fehlerhafte Bestellungen. Die Informationen werden derzeit nur in eine Datenbank geschrieben. Dies stellt bereits die Grundlage für zusätzliche Komponenten oder ein externes

[2] http://www.bpmn.org/

System dar, um die gewünschten Auswertungen, Analysen oder Benachrichtigungen zu generieren.

6.3 Zusammenfassung und Fazit

Der Prototyp enthält lediglich Grundzüge eines Workflow Management Systems, besitzt aber durchaus das Potential um ein einfaches, flexibles und günstiges Grundgerüst zur Erfassung und Abwicklung von Geschäftsprozessen zu bieten. Durch den Implementierungsaufwand und der fehlenden grafischen Oberfläche, ist eine Verwendung für Anwendungsentwickler empfehlenswert, die selbst eine derartige Softwarelösung für kleinere Bereiche implementieren wollen. Es bedarf einer Untersuchung, ob eine Weiterentwicklung sich als wirtschaftlich herausstellt um daraus ein eigenes Produkt zu erstellen. Der Cloud-basierte Ansatz mit generischen und standardisierten Ansätzen, welche sich nach etablierten Entwurfsmustern richten könnte die Basis einer zeitreduzierenden Prozessoptimierung bieten. Durch die verwendeten Microsoftkomponenten ist sowohl ein Supportvertrag während der Laufzeit zur Absicherung des Unternehmens möglich, als auch eine schnelle Weiterentwicklung der Software, da das Knowhow großflächig auf dem Arbeitsmarkt vertreten ist. Die enthaltene Quellcodedokumentation kann durch das Visual Studio automatisch in eine ausdruckbare Dokumentation der gesamten Software überführt werden. Die Realisierung komplexer Workflowkaskaden und paralleler Prozesse im gegenwärtigen Prototyp ermöglicht die Überlegung einer Weiterentwicklung der entstandenen Software.

7. Zusammenfassung und Ausblick

7.1 Zusammenfassung

In der vorliegenden Arbeit wurde eine Machbarkeitsanalyse durchgeführt, um zu überprüfen ob für einen konkreten Kundenauftrag eine wiederverwendbare Lösung in Form einer Workflow-Automatisierung realisierbar ist. Für den Kunden sollte eine Automatisierungsmöglichkeit für die Bestellabwicklung gefunden werden, da dieser Geschäftsprozess durch die Einführung eines neuen Serviceangebots zunehmend an Bedeutung gewinnt. Die Ermittlung des aktuellen Problems und die Anwendung der progressiven Abstraktion überführten den Kundenauftrag in ein allgemeines, fundamentales Problem. Die systemgestützte Abbildung von Geschäftsprozessen zur Automatisierung der Arbeitsabläufe mit Hilfe einer frameworkgestützten Workflow-Engine. Zur Validierung der Annahme und Nachweis der Machbarkeit sollte in dieser Arbeit ein Prototyp entwickelt werden, der die grundlegenden Anforderungen des Kunden erfüllt und zugleich das Potential besitzt beliebige Geschäftsprozesse abzubilden.

Aus dem Soll-Konzept des Kunden definierten wir funktionale und nichtfunktionale Anforderungen für den zu entwickelnden Prototypen. Eine zentrale Anforderung ist das Erstellen von einzelnen Abarbeitungsschritten, welche dann zu Workflows kombiniert werden können um einkommende Bestellungen automatisch abzuarbeiten.

Die automatisierte Steuerung wurde um Anforderungen zur Fehler- und Ereignisprotokollierungen erweitert, um im Fehlerfall eine Wiederherstellung zu gewährleisten, und um zukünftige Analysen zu ermöglichen.

Im drauf folgenden Kapitel wurde das Konzept vorgestellt. Es definiert die Architektur der Software und enthält Entwurfspläne für spätere Softwarekomponenten zur Realisierung aller Anforderungen. Es wurde großen Wert auf die Wiederverwendbarkeit und Anbindung an externe Softwaresysteme gelegt. Um diesen Forderungen gerecht zu werden, muss ein einheitlicher Entwicklungsrahmen für Anwendungsentwickler geschaffen werden, damit eine Automatisierung von definierten Prozessen ermöglicht wird.

Die hier diskutierten und getroffenen Konzeptentscheidungen wurden daraufhin in der Umsetzung mittels Microsoft Technologien implementiert. Die Workflow-Engine wurde aufgrund ihres erweiterbaren Charakters als Framework implementiert, mit dem der Nutzer seinen eigenen Code schreiben kann und dennoch alle Funktionalitäten der Workflow-Engine genießt. Hierbei wurde drauf geachtet, dass die Datenbank dem Code-First Ansatz entspricht, damit diese aus den Datenmodellen des Users generiert werden kann. Das definierte Datenmodell gibt auch das Format der einkommenden Daten von außen vor. Diese Schnittstelle an externe Middlewares wurde durch eine typsichere SOAP Schnittstelle realisiert wie es bei Unternehmenslösungen üblich ist. Bei der Umsetzung des Workflows wurde neben der Effektivität auch Wert auf die Benutzerfreundlichkeit gelegt um das Implementieren von eigenen Workflows so intuitiv wie möglich zu gestalten. Die resultierende Implementierung wurde anschließend mit den Anforderungen verglichen und festgestellte Schwachstellen diskutiert. Es stellte sich heraus, dass die von uns entwickelte Workflow-Engine die geforderten Anforderungen erfüllt, wodurch die Machbarkeit bestätigt werden konnte.

7.2 Ausblick

Der Prototyp hat gezeigt, dass alle Anforderungen realisiert werden können um eine automatisierte Abwicklung des Fulfillments zu gewährleisten. Zugleich kann die Lösung für ähnliche Kundenanforderungen wiederverwendet werden. In weiteren Arbeiten soll die Evaluation der Weiterentwicklung des Prototyps im Vordergrund stehen, um zu prüfen ob sich die Fortführung der Entwicklung rentiert oder ob eine kommerzielle Lösung gefunden werden kann, die die Anforderungen erfüllt.

Ein produktiver Einsatz mit der entstandenen Software ist möglich, es wird jedoch empfohlen den Recovery-Prozess zu optimieren um effizient mit einem Hardwareausfall umgehen zu können. Ein möglicher Optimierungsansatz ist der Einsatz einer auf das Problem zugeschnittenen Datenbanklösung. Hier sollte berücksichtigt werden, dass nur ein kleiner Teil der gespeicherten Daten jederzeit zur Verfügung stehen muss. Die gesammelten Ereignisdaten sind erst für größere Analysen nötig, weshalb möglicherweise ein Data Warehouse in Frage kommt.

Außerdem wird, um den Komfort der Konkurrenzprodukte zu bieten, eine grafische Oberfläche benötigt, die den Programmieraufwand bei der Erstellung eines Workflows senkt. Hier bietet das .Net Framework mit WPF eine sehr gute Möglichkeit eine graphische Oberfläche zu erstellen die per Datenbindung an die bestehende Logik anknüpft. Hier könnten auch die aktuellen Auswertungen und Nachrichten präsentiert werden.

Um das Produkt abzurunden, sollte es um die Anbindung an mobile Anwendungen erweitert werden. Hierzu wird die Implementierung einer REST-Fassade empfohlen, die auf die vorhandene SOAP Schnittstelle aufbaut.

Nach den hier genannten Implementierungen nähert sich das Programm dem großem Ziel, die Workflow Engine als kontinuierlich wachsendes Produkt auf den Markt zu bringen.

A. Anhang

A.1 Bestellmodell

Das Bestellmodell in Abbildung A.1 bildet das Modell des Fulfillments und definiert sowohl den generischen Typ *TContext* in unserem Framework als auch die Struktur der Datenbank zu Speicherung der Bestelldaten.

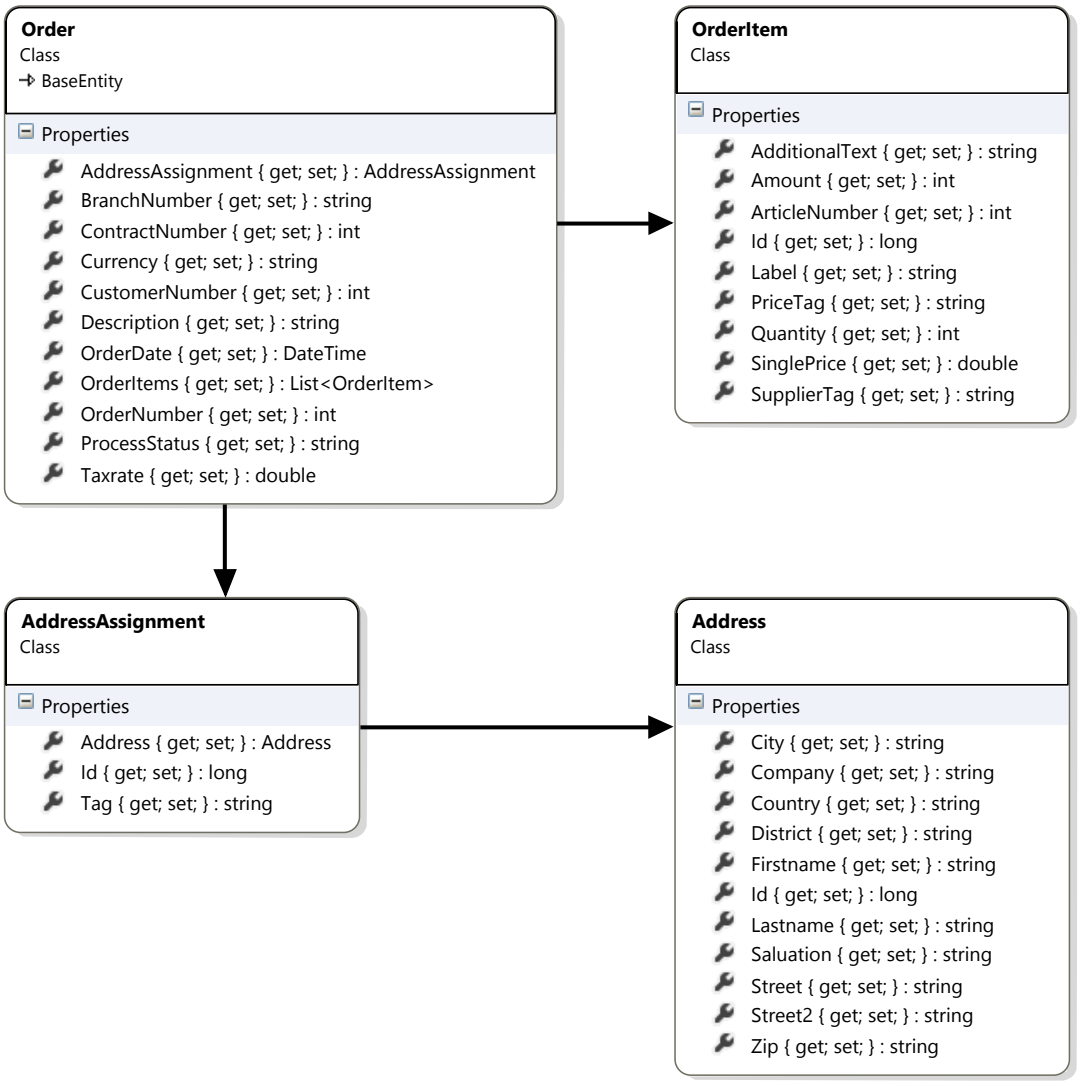

Abbildung A.1: Bestellmodell

A.2 Dependency Injection Muster

Mustername: Dependency Injection

Problemabschnitt: Abhängigkeit eines Objekts kann zur Laufzeit nicht geändert werden.

Lösungsabschnitt: Objekte erhalten ein bestimmtes Verhalten durch den Aufruf einer definierten Methode um einem definierten Typ zur Laufzeit eine externe Instanz zuzuweisen. Diese Injektion führt dazu, dass zur Laufzeit einem im Programmcode enthaltenen Typ (Interface) ein beliebiges Objekt dieses Typs verwendet werden kann. Ein Austausch des Objekts abhängig vom Typ ist sehr einfach möglich. Die Abbildung A.2 verdeutlich diesen Vorgang. Ein `Builder` erstellt die notwendige Klasse `ClassA`, worin der Typ `IServiceA` verwendet wird. Zur Laufzeit entscheidet sich, welches konkrete Objekt vom Typ `IServiceA` verwendet wird. Der `Builder` injiziert die notwendige Klasse `ServiceA` um der `ClassA` die gewünschte Funktionalität bereitzustellen. Dieses Muster kann auch für das *Mocking* (für Modultests) verwendet werden.

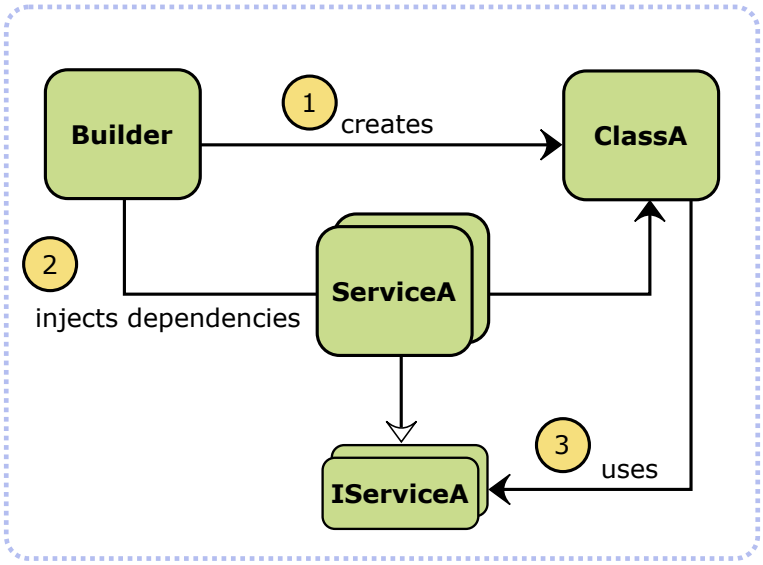

Abbildung A.2: Dependency Injection

Konsequenzabschnitt:
Vorteile:

- Lebensdauer einer Objektinstanz lässt sich leicht konfigurieren
- Automatische lockere Koppelung zwischen den Funktionen
- Komplexität von Klassen wird verringert
- Einhaltung von Seperation of concern

Nachteile:

- Höhere Einstiegshürde durch komplexere Konfiguration

[Mic15a, Cap15, Wis10]

A.3 Delegation Muster

Musternname: Proxy bzw. Adapter

Problemabschnitt: Fehlende Trennungsebene bei Vererbung zwischen Anbieter und Nutzer, wodurch ein nachträglicher Austausch der Funktionalität innerhalb einer Vererbungshierarchie kaum möglich ist.

Lösungsabschnitt: Die adaptierende Klasse `Client` leitet die Anfragen an den Delegator weiter, ohne selbst die Funktionen zu erben, es kennt lediglich die Schnittstelle `ITarget`. Dadurch kann der dahinterliegende Dienst `Adaptee` jederzeit ausgetauscht werden. Der `Adapter` leitet die Anfrage an den korrekten Dienst weiter und kann Fehler- und Ausnahmesituationen behandeln. Abbildung A.3 verdeutlicht diesen Vorgang.

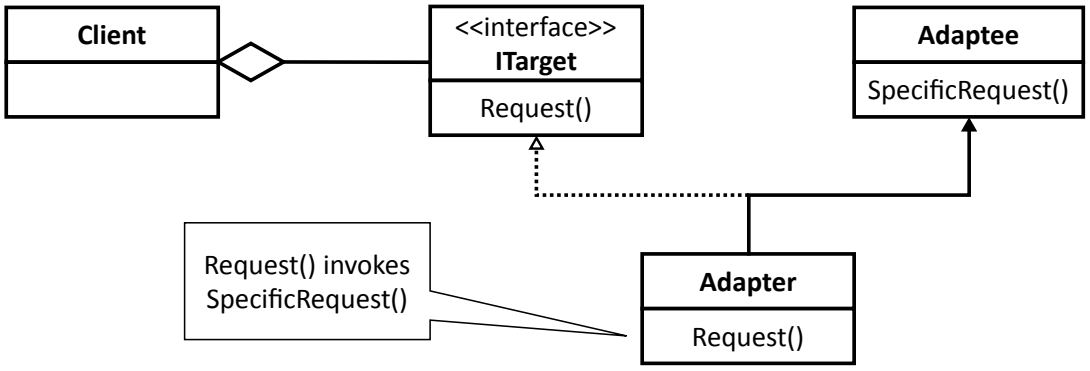

Abbildung A.3: Delegation

Konsequenzabschnitt:
Vorteile:

- Realisierung der Mehrfachvererbung
- Vermeidung von Vererbung
- Zusätzliche Trennungsebene zwischen Objektfunktionalitäten
- Erhöhte Wiederverwendung durch austauschbare Dienste
- Auslagerung komplexer Funktionen

Nachteile:

- Höhere Komplexität
- Laufzeitineffizienzen, da Objekte teilweise zu schnell zerstört werden, wodurch eine Neuerstellung notwendig wird.

[Mic04a, FH96]

A.4 Bestellungsfluss Messungen

	Duration (ms)					
	Small Workflow			Big Workflow		
Order	**FW**	**Recovery**	**Workflow**	**FW**	**Recovery**	**Workflow**
1	780	115	0,1	99	65	50
200	1626	1466	3	2667	2982	1249
300	2277	2146	4	3322	3655	1875
400	3062	2837	6	4073	4353	2503
500	3717	3597	7	4837	5043	3129
600	4434	4279	9	5562	5684	3746
700	5120	5034	11	6286	6391	4370
800	5948	5781	13	7013	7108	5003
900	6687	6461	13	7738	7825	5627
1000	7452	7261	14	8465	8592	6248

Tabelle A.1: Bestellungsfluss Messungen

Die Tabelle A.1 zeigt die Messdaten des Bestellflusses für den Benmarktest an, *FW* bezeichnet unser *Framework*.

Literaturverzeichnis

[BBG05] Sami Beydeda, Matthias Book, and Volker Gruhn. *Model-Driven Software Development*. Springer-Verlag Berlin Heidelberg, 2005. (zitiert auf Seite 10)

[But05] Russell Butek. Which style of WSDL should I use? Website, May 2005. Online verfügbar auf http://www.ibm.com/developerworks/library/ws-whichwsdl/; zugegriffen am 19.04.2015. (zitiert auf Seite 15)

[Cap15] Griffin Caprio. Dependency Injection. Website, May 2015. Online verfügbar auf https://msdn.microsoft.com/en-us/magazine/cc163739.aspx; zugegriffen am 25.05.2015. (zitiert auf Seite 65)

[DGV+13] Brian Driscoll, Nitin Gupta, Robert Vettor, Zeeshan Hirani, and Larry Tenny. *Entity Framework 6 Recipes 2nd Edition*. Apress, 2013. (zitiert auf Seite 50)

[FG08] Jakob Freud and Klaus Götzer. *Vom Geschäftsprozess zum Workflow*. Carl Hanser Verlag München, 2008. (zitiert auf Seite 16)

[FH96] Ulrich Frank and Sören Halter. Delegation: Eine sinnvolle Ergänzung gängiger objektorientierter Modellierungskonzepte. Website, September 1996. Online verfügbar auf http://www.wi-inf.uni-duisburg-essen.de/FGFrank/documents/Zeitschriftenartikel/Mobis96.pdf; zugegriffen am 12.05.2015. (zitiert auf Seite 66)

[GHJV00] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Entwurfsmuster*. Addison-Wesley, 2000. (zitiert auf Seite 5, 8, 11 und 12)

[GHR+14] Mickey Gousset, Martin Hinshelwood, Brian A Randell, Brian Keller, and Martin Woodward. *Professional Application Lifecycle Management with Visual Studio 2013*. John Wiley & Sons, 2014. (zitiert auf Seite 10)

[Mic04a] Microsoft. Structural Patterns: Adapter and Facade. Website, January 2004. Online verfügbar auf https://msdn.microsoft.com/en-us/library/orm-9780596527730-01-04.aspx; zugegriffen am 12.05.2015. (zitiert auf Seite 66)

[Mic04b] Microsoft. Vorteile von XML. Website, June 2004. Online verfügbar auf https://msdn.microsoft.com/de-de/library/cc431269.aspx; zugegriffen am 19.04.2015. (zitiert auf Seite 13)

[Mic15a] Microsoft. Dependency Injection. Website, May 2015. Online verfügbar auf https://msdn.microsoft.com/en-us/library/ff921152.aspx; zugegriffen am 25.05.2015. (zitiert auf Seite 65)

[Mic15b] Microsoft. Implementing Singleton in C#. Website, April 2015. Online verfügbar auf https://msdn.microsoft.com/en-us/library/ff650316.aspx; zugegriffen am 16.04.2015. (zitiert auf Seite 7)

[Mic15c] Microsoft. Was ist die Windows Communication Foundation? Website, May 2015. Online verfügbar auf https://msdn.microsoft.com/de-de/library/ms731082(v=vs.110).aspx; zugegriffen am 25.05.2015. (zitiert auf Seite 40)

[Nar09] Vijay Narayanan. Tips for Effective Software Reuse. Website, June 2009. Online verfügbar auf http://www.infoq.com/articles/vijay-narayanan-software-reuse; zugegriffen am 16.04.2015. (zitiert auf Seite 1)

[Rei03] Dipl.-Ing. Steffen Reichel. Anwendungen über einen SOAP-Web Service verbinden. Website, June 2003. Online verfügbar auf http://www.informatik.fh-lausitz.de/sreichel/projekte/soap-service.html; zugegriffen am 12.04.2015. (zitiert auf Seite 40)

[SS07] Sajjan G. Shiva and Lubna Abou Shala. Software Reuse: Research and Practice. Paper, 2007. Online verfügbar auf http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.471.5823&rep=rep1&type=pdf; zugegriffen am 16.04.2015. (zitiert auf Seite 1)

[TG03] Jacqueline Tran and Marco Glier. Geschäftsprozesse & Workflow: WfMC Referenzmodell. Website, February 2003. Online verfügbar auf https://vsis-www.informatik.uni-hamburg.de/oldServer/teaching/ws-02.03/ses/WFMC.pdf; zugegriffen am 16.04.2015. (zitiert auf Seite 16)

[tut15] tutorialspoint. C# - Generics. Website, 2015. Online verfügbar auf http://www.tutorialspoint.com/csharp/csharp_generics.htm; zugegriffen am 16.04.2015. (zitiert auf Seite 12)

[w3s15] w3school.com. WSDL Bindings. Website, 2015. Online verfügbar auf http://www.w3schools.com/webservices/ws_wsdl_binding.asp; zugegriffen am 16.04.2015. (zitiert auf Seite 16)

[Wei98] Dr. Dietmar Weiß. Workflow Management: Begriff und Abgrenzung. Website, 1998. Online verfügbar auf http://www.dr-weiss.com/index.php?id=53; zugegriffen am 16.04.2015. (zitiert auf Seite 16)

[Wis10] IT Wissen. Dependency Injection. Website, March 2010. Online verfügbar auf http://www.itwissen.info/definition/lexikon/Dependency-Injection-dependency-injection-DI.html; zugegriffen am 19.05.2015. (zitiert auf Seite 65)

Christian Steusloff Jens Wiemann

Hiermit erklären wir, dass wir die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet haben.

Magdeburg, den 05.06.2015