Inventar-Anwendung

Architekturentwurf von **Timo Klenk**

Version	**Änderungsvermerk**	**Bezieht sich auf Anforderungsdokument**	**Autor**
1.0	Initialer Architekturentwurf	1.0 – 01. Dez. 2019	Timo Klenk

Zielgruppe: Der Auftraggeber, Timo Klenk
Datum: 14.06.2020

Inhaltsverzeichnis

1 Einführung 4
2 Erläuterung der Architekturstrategie 5
3 Erhebung Funktionaler Einheiten 6
3.1 Vorstellung der relevanten Anforderungen 6
3.2 Vorstellung der funktionalen Einheiten 7
3.2.1 F-01 Revisionssichere Datensicherung mit Protokollierung der Änderungen 7
3.2.2 F-02 Verwaltung von Gegenstandstypen 7
3.2.3 F-03 Verwaltung von Lagerorten 8
3.2.4 F-04 Verwaltung von Gegenständen 9
3.2.5 F-05 Verwaltung von Nutzern 11
3.2.6 F-06 Durchführen einer Inventur 11
4 Architekturentscheidungen 12
4.1 E 01 Technologie-Stack: Entscheidung der Entwicklungsplattform 12
4.2 E 02 Datenspeicherung: Form der Datenspeicherung 14
4.3 E 03 Technologie-Stack: Entscheidung der zu benutzenden Datenspeicherung 16
4.4 E 04 Implementierung: Wahl der Grundlegenden Struktur des Codes 18
5 Sichten 22
5.1 Statische Sichten und Integritätsbedingungen 22
5.1.1 Statische Sicht: Zu Persistierende Daten 22
5.1.2 Statische Sicht: Daten zur Laufzeit 37
5.2 Dynamische Sichten 40
5.2.1 Prozesse innerhalb von F-01 40
6 Zusammenfassung 41
7 Referenzen 42

Tabellenverzeichnis:

Tabelle 1: Zusammenfassung relevanter Anforderungen 6
Tabelle 2: Entscheidungsmatrix für Entwicklungsplattform 12
Tabelle 3: Entscheidungmatrix für Datenspeicherform 14
Tabelle 4: Entscheidungsmatrix für Form der Datenspeicherung 16
Tabelle 5: Entscheidungsmatrix für Implementirungspattern 18
Tabelle 6: Auflistung der Kardinalitäten beim Konsolidieren 30
Tabelle 7: Umzusetzenden Events beim Konsolidieren 32
Tabelle 8: Attribute der Entitätstypen 33
Tabelle 9: Beispieldaten Datenbasis 36
Tabelle 10: Beispieldaten Eventtabellen 36

Abbildungsverzeichnis:

Abbildung 1: ERM Sicht für F-01 23
Abbildung 2:ERM Sicht für F-02 24
Abbildung 3: ERM Sicht für F-03 25
Abbildung 4: ERM-Sicht für F-04 26
Abbildung 5: ERM-Sicht für F-05 27
Abbildung 6: ERM-Sicht für F-06 28
Abbildung 7: Zwischenergebnis Sichten-Konsolidierung 31
Abbildung 8: Beispiel Konsolidierung Events an 1:N Relation 31
Abbildung 9: Konsolidiertes ERM 34
Abbildung 10: Grober Überblick über die Klassen zur Laufzeit, Bild. 1 38
Abbildung 11: Grober Überblick über die Klassen zur Laufzeit, Bild. 2 39
Abbildung 12: Prozessmodellierung Nachrichtenbasierter Datenänderungen 40

Sämtliche hier gezeigten Abbildungen wurden selbst mithilfe der freien Software diagrams.net erzeugt. Sie lassen sich zusätzlich unter nachfolgendem Link finden:
https://github.com/TINF18B5/SWE_TINF18B5_Timo_Klenk/tree/master/ArchitekturEntwurf%20Material

1 Einführung

Für kleine bis mittelgroße Unternehmen soll eine Anwendung für mobile Endgeräte erstellt werden, die es erlaubt, ihr real existierendes Lager mithilfe virtueller Lagerorte abzubilden und gelagerte Gegenstände darin einzupflegen.
Fokus der Anwendung ist das Reduzieren von Verwaltungsaufwand des Lagers sowie eine vereinfachte Protokollierung von Bestandsänderungen.
Dieses Architekturdokument wurde für eine erste Prototypische Implementierung geschrieben.
Es wird daher keinen Anspruch auf Vollständigkeit und Rechtskonformität geboten.

2 Erläuterung der Architekturstrategie

Für das Finden der zu benutzenden Architektur wurden zuerst das Anforderungsdokument in Funktionale Einheiten untergliedert. Anhand dieser funktionalen Einheiten wurde eine Abschätzung des zu verwendenden Technologie-Stacks getroffen. Hierbei wurde sich für ein asynchrones Implementierungsmuster entschieden.
Anhand dessen wurden im Anschluss erst die zu persistierenden Daten, und darauf aufbauend die Daten zur Laufzeit modelliert. Zuletzt wurden die Laufzeitprozesse modelliert.

3 Erhebung Funktionaler Einheiten

Für das Erheben der Funktionalen Einheiten wurde eine Horizontale Aufteilung der Anwendungsfälle vorgenommen. Im Folgenden werden die für dieses Dokument relevanten Anforderungen zuerst aufgeführt, und im weiteren Verlauf dieses Kapitels die daraus resultierenden Funktionalen in absteigend priorisierter Reihenfolge vorgestellt.

3.1 Vorstellung der relevanten Anforderungen

In Tabelle 1 werden die für dieses Dokument benutzten Anforderungen zusammen mit einer Referenz auf das ursprüngliche Anforderungsdokument sowie einer kurzen Zusammenfassung erläutert. Hierbei wurden nur Anforderungen mit aufgenommen, die sich auf die Arbeitsweise der Anwendung auswirken, Anforderungen die Integritätsbedingungen und Ausnahmefälle beschreiben, werden beim Vorstellen der entsprechenden Funktionalen Einheiten mitdiskutiert.
Die Tabelle wurde nicht nach der Nummer der Anforderung und auch nicht nach Reihenfolge der Anforderungen im ursprünglichen Dokument, sondern stattdessen nach der Priorisierung der daraus abgeleiteten funktionalen Einheiten sortiert.
Die aus den Anforderungen abgeleiteten Funktionalen Einheiten wurden hier der aufgrund dieser Priorisierung und für eine bessere Übersichtlichkeit mit aufgelistet, allerdings wird auf diese in erst später eingegangen.

Nummer	Beschreibung der Anforderung	Bestandteil von	Referenz
R-45	Datenmodell muss Revisionssicherheit aufweisen	F-01	[1, p. 32]
R-44	Änderungen am Datenbestand müssen protokolliert werden	F-01	[1, p. 32]
R-11	Gegenstandstypen anlegen	F-02.1	[1, p. 21]
R-13	Gegenstandstypen bearbeiten	F-02.2	[1, p. 22]
R-15	Gegenstandstypen suchen	F-02.3	[1, p. 27]
R-3	Lagerort anlegen	F-03.1	[1, p. 18]
R-4	Lagerort bearbeiten	F-03.2	[1, p. 19]
R-8	Übersicht der Lagerorte	F-03.3	[1, p. 33]
R-16	Gegenstand anlegen	F-04.1	[1, p. 23]
	Gegenstand einlagern	F-04.2	[1, p. 23]
R-22	Gegenstand auslagern	F-04.2	[1, p. 25]
R-23	Gegenstand löschen	F-04.2	[1, p. 26]
R-19	Gegenstandsinformationen anzeigen	F-04.3	[1, p. 33]
R-1	Nutzerauthentifizierung	F-05	[1, p. 17]
R-2	Nutzerberechtigungen	F-05	[1, p. 18]
R-32	Inventur erstellen	F-06	[1, p. 29]
R-34	Gegenstandsmenge bei Inventur eintragen	F-06	[1, p. 30]

Tabelle 1: Zusammenfassung relevanter Anforderungen

3.2 Vorstellung der funktionalen Einheiten

Im Nachfolgenden werden die aus den Anforderungen abgeleiteten funktionalen Einheiten aufgelistet und deren Funktion innerhalb der Anwendung erläutert. Die Reihenfolge der Anforderungen stellt hierbei auch deren Priorisierung in absteigender Reihenfolge dar. Diese Priorisierung wird ebenfalls bei der Vorstellung der entsprechenden Einheit mitdiskutiert.
Einige Funktionalen Einheiten sind für eine bessere Übersichtlichkeit selbst in mehrere Bestandteile geteilt. Diese Bestandteile sind durch Suffixe an die Funktionale Einheit gebunden.

3.2.1 F-01 Revisionssichere Datensicherung mit Protokollierung der Änderungen

Zur Datensicherung ist Revisionssicherheit gefordert.
Diese Anforderung wird als KO-Kriterium eingestuft, da es die einzige verpflichtende Angabe Seitens des Auftraggebers darstellt. Außerdem ist das Datenmodell ein für die weiteren funktionalen Einheiten wesentlicher Bestandteil, da eine Festlegung hier den Umsetzungsraum stark einschränkt.
Die Aufgabe dieser Funktionalen Einheit beschreibt den grundsätzlichen Aufbau des zu benutzenden Datenschemas sowie die Wahl, wie, wann und wo Anwendungsdaten gesichert werden.
Die zweite Aufgabe dieser Funktionalen Einheit ist das Protokollieren von Änderungen. Diese Aufgabe wurde dieser funktionalen Einheit zugeordnet, da durch gewisse Formate der Protokolle die Revisionssicherheit gewährleistet werden kann. Hierauf wird in E 02 näher eingegangen.

3.2.2 F-02 Verwaltung von Gegenstandstypen

Unter einem Gegenstandstyp versteht der Auftraggeber die „Beschreibung eines im Lager gelagerten Objekts“ [1, p. 3].
Der Umgang mit Gegenstandstypen wurde aufgrund der Anforderungen wiederum in drei Einheiten aufgeteilt, namentlich das Erstellen(F-02.1), das Verwalten(F-02.2) und das Suchen und Anzeigen (F-02.3) von Gegenstandstypen.
Der Umgang mit Gegenstandstypen wurde hoch priorisiert, da er Voraussetzung für eine funktionierende Implementierung der anderen Funktionalen Einheiten darstellt. Dies lässt sich dem Abhängigkeitsgraphen des Anforderungsdokuments entnehmen [1, p. 37], da darin die Anforderungen R-11, R-14 und R-7 direkte oder transitive Voraussetzungen für 20 der verbleibenden 42 Anforderungen sind.
Die Untergliederung dieser Funktionalen Einheit in drei Unterpunkte wurde der besseren Übersichtlichkeit sowie der Abtrennung von Zuständigkeitsbereichen getroffen. Da im Anforderungsdokument keine Löschung von Gegenstandstypen beschrieben war, wurde sie nicht als Funktionale Einheit aufgenommen.

3.2.2.1 Gegenstandstyp erstellen

Für das Anlegen von Gegenstandstypen wurde anhand von Figure 3.6 [1, p. 45] ein genereller Ablauf beschrieben. Hierin werden einem Gegenstandstypen beim Erstellen die Eigenschaften Name und Beschreibung gesetzt. Zusätzlich beschreibt R-12, dass der Nutzer beim Anlegen von mehreren Gegenstandstypen mit demselben Namen gewarnt werden soll. Hieraus lässt sich schließen, dass der Name für Gegenstandstypen nicht als eindeutig definiert ist, was beispielsweise bei Lagerorten gegeben ist. Da R-17 zusätzlich beschreibt, dass Gegenstandstypen beim Erstellen eines Gegenstandes miterstellt werden können sollen, muss das Erstellen von Gegenstandstypen so ausgestaltet werden, dass es aus verschiedenen Kontexten aufrufbar ist.

3.2.2.2 Gegenstandstyp bearbeiten

Für das Bearbeiten von Gegenstandtypen wurde anhand von Figure 3.7 [1, p. 46] ein genereller Ablauf beschrieben. Hierin werden bei einem Bearbeiten eines Gegenstands dessen Name und Beschreibung neu gesetzt. Die beim Anlegen von Gegenstandstypen vorhandene Prüfung auf doppelte Namen ist hier nicht aufgeführt, stattdessen wird anhand der Abnahmekriterien von R-13 definiert, dass der Name eines Gegenstandstypen nicht leer werden darf.

3.2.2.3 Gegenstandstyp suchen und Anzeigen

Figure 3.5 [1, p. 44] beschreibt die Funktionalitäten, die eine Gegenstandstypübersicht beinhalten soll. Hierbei wird eine Liste aller Gegenstandstypen eingefordert, was allerdings durch R-31 wiederum dahingehend eingeschränkt wird, dass diese Liste nur Gegenstandstypen enthalten soll, wie nicht für den jeweiligen Nutzer gesperrt worden sind.
Des Weiteren soll diese Liste filter- und sortierbar sein. Hierfür lassen sich dem Anforderungsdokument allerdings keine genaueren Informationen entnehmen.
Außerdem soll es möglich sein, einen Gegenstandstypen aus der Liste auszuwählen und sich eine Detailübersicht dessen anzeigen zu lassen. Hierbei beschreibt R-15, dass diese Detailübersicht eine Auflistung aller Gegenstände des jeweiligen Gegenstandstypen beinhalten soll. Da aus R14 nicht genauer ersichtlich wird, welche weiteren Informationen zu einem Gegenstand hier angezeigt werden sollen, wird im weiteren Verlauf des Dokuments davon ausgegangen, dass es sich hierbei um den Lagerort und die Menge des Gegenstands handelt.

3.2.3 F-03 Verwaltung von Lagerorten

Ein Lagerort wurde vom Auftraggeber als ein „im Lager existierender Ort, an dem Gegenstände gelagert werden können“ definiert [1, p. 3].
Der Umgang mit Lagerorten wurde an diese Stelle priorisiert, da die Funktionalen Einheiten, die sich mit dem Umgang mit Gegenständen (F-04) und der Inventur (F-06) beschäftigen, davon abhängen.

Anhand der gegebenen Anforderungen wurde der Umgang mit Lagerorten in drei Einheiten aufgeteilt, namentlich das Erstellen (F-03.1), Bearbeiten (F-03.2), sowie das Suchen und Anzeigen von Lagerorten (F-03.3).
Da im Anforderungsdokument keine Löschung von Lagerorten beschrieben wurde, wurde diesbezüglich keine Funktionalität aufgenommen.

3.2.3.1 Lagerort erstellen:

Für das Erstellen von Lagerorten wurde anhand von Figure 3.12 [1, p. 50] ein genereller Ablauf beschrieben. Hierin werden einem Lagerort beim Erstellen die Eigenschaften Name und Beschreibung gesetzt. Aus dem Aktivitätsdiagramm wird ersichtlich, dass der Name eines Lagerortes eindeutig sein muss, was sich mit R-5 deckt. Gleichzeitig darf der Name laut R-3 nicht leer sein. Hier wurde allerdings noch die zusätzliche Anforderung gestellt, dass bei Eingabe eines bereits existierenden Namens der Nutzer die Möglichkeit haben soll, stattdessen den Lagerort mit diesem Namen zu bearbeiten.

3.2.3.2 Lagerort bearbeiten

Wie Figure 3.13 [1, p. 51] zu entnehmen, besteht das Bearbeiten eines Lagerortes darin, den Namen und die Beschreibung dessen neu einzugeben. Hier wurde ebenso gefordert, dass der Name des Lagerortes nicht leer oder auf den eines bereits existierenden Lagerortes gesetzt werden darf. Es wurde allerdings davon ausgegangen, dass es möglich sein soll, den Namen eines Lagerortes beim Bearbeiten auf den des zu bearbeitenden Lagerortes zu setzen. Dies stellt eine Abweichung von den Anforderungen dar.

3.2.3.3 Lagerort suchen und Anzeigen

Figure 3.11 [1, p. 49] beschreibt die Funktionalitäten, die eine Lagerortübersicht enthalten soll. Hierbei wird eine Liste aller Lagerorte gefordert. R-8 spezifiziert dies als die Anforderung nach einer „Übersicht aller für ihn [den Nutzer] zugänglichen Lagerorte“ [1, p. 33]. Da hierbei allerdings keine weiteren Angaben gemacht wurden, wie einem Nutzer ein Lagerort zugänglich oder nicht zugänglich gemacht werden soll, wird im weiteren Verlauf des Dokuments davon ausgegangen, dass es sich hierbei um eine nicht spezifizierte Anforderung an die Nutzerverwaltung, die das Sperren von Lagerorten beinhaltet, handelt. Diese Vermutung liegt darin begründet, dass solche Anforderungen bereits für Gegenstände (R-28/R-29) und Gegenstandstypen (R-30/R-31) existieren.
Außerdem ist für diese Liste eine Sortiermöglichkeit gefordert. R-10 gibt hierbei die Sortierkriterien „Name, Anzahl an gelagerten Gegenständen, Gesamtzahl der gelagerten Objekte“ vor und beschreibt, dass die Liste nach einem oder auch mehreren dieser Kriterien auf einmal sortieren können soll [1, p. 12]. Zudem beschreibt R-9 die Möglichkeit, diese Liste nach den Kriterien Name und Beschreibung zu durchsuchen.
Außerdem beschreibt das Aktivitätsdiagramm noch eine Detailübersicht eines Lagerortes, die sich nach dem Anlegen oder Bearbeiten eines Lagerortes oder bei Auswählen eines Lagerorts aus dieser Liste öffnen soll. Der Inhalt dieser Übersicht wurde allerdings nicht näher spezifiziert. Anhand des Aktivitätsdiagramms kann davon ausgegangen werden, dass diese Übersicht eine Auflistung an Gegenständen beinhalten soll, sowie eine Möglichkeit, Gegenstände einzulagern. Da ein Gegenstand, wie in F-04 näher beschrieben, eine Kombination aus Lagerort und Gegenstandstyp darstellt, wird davon ausgegangen, dass diese Liste eine Auflistung aller Gegenstände, die diesem Lagerort zugeordnet sind, darstellen soll.

3.2.4 F-04 Verwaltung von Gegenständen

Ein Gegenstand wird als die Beschreibung „eine[r] Menge von Objekten eines Gegenstandstyp im Lager” dargestellt [1, p. 3]. Im Anforderungsdokument wurde nicht genauer erwähnt, ob ein Lagerort mehr als einen Gegenstand desselben Typs besitzen kann, daher wurde hierbei die Annahme getroffen, dass dem nicht so ist. Hierdurch lassen sich einige Abläufe der Anwendung vereinfachen, da zum Beispiel nicht entschieden werden muss, wie mehrere Gegenstände desselben Typs in der Detailansicht eines Lagerortes dargestellt werden sollen, um zwischen ihnen unterscheiden zu können.

Der Umgang mit Gegenständen wurde hierher priorisiert, da er von F-02 und F-03 abhängt.

Der Umgang mit Gegenständen wurde anhand der gegebenen Anforderungen in drei Einheiten aufgeteilt, namentlich das Erstellen (F-04.1), Bearbeiten (F-04.2), sowie das Suchen und Anzeigen von Gegenständen (F-04.3).
Hierbei wurde Anforderung R-16 in zwei geteilt, namentlich das Erstellen und das Einlagern von Gegenständen. Im ursprünglichen Anforderungsdokument wurden diese beiden Aktionen als eine behandelt. Dies hinterließ allerdings eine Inkonsistenz, da es zwar als möglich geschildert wurde, eine gegebene Menge eines Gegenstandes, die geringer als die Gesamtmenge ist, auszulagern, aber durch den beschriebenen Ablauf keine Möglichkeit bestand, zu einem existierenden Gegenstand eine Menge einzulagern. Daher führt eine Einlagerung eines neuen Gegenstands erst zu einem Erstellen des Gegenstands, und eine Einlagerung für einen bereits an diesem Lagerort gelagerten Gegenstandstypen zu einer Einlagerung in den bereits existierenden Gegenstand. Des Weiteren kann hierdurch eine Konsistente Untergliederung von F-02, F-03 und F-04 in funktional gleiche Untereinheiten ermöglicht werden, was der Übersichtlichkeit des Projekts zugutekommt.

3.2.4.1 Gegenstand erstellen

Für das Erstellen von Gegenständen wurde anhand von Figure 3.14 [1, p. 52] ein genereller Ablauf gegeben. Hierbei und anhand von R-17 wird ersichtlich, dass der Gegenstandstyp des zu erstellenden Gegenstandes entweder in irgendeiner Form auswählbar sein muss, sofern er bereits existiert, oder mit angelegt können werden muss. Des Weiteren wird in dem gegebenen Aktivitätsdiagramm beschrieben, dass die Menge eines zu erstellenden Gegenstandes größer als 0 sein muss.

3.2.4.2 Gegenstand bearbeiten

Um das Benennungsschema der anderen Funktionalen Einheiten beizubehalten wird das Einlagern, Auslagern sowie das Löschen eines Gegenstands unter dem Begriff „Bearbeiten“ zusammengefasst. Hierbei sollte vermerkt werden, dass das Löschen eines Gegenstands zwar keine Bearbeitung im eigentlichen Sinne darstellt, aufgrund der mit F-01 vorgestellten Revisionssicherheit und Protokollierung von Änderungen sich ein Löschen eines Gegenstandes eventuell als ein Inaktiv-setzen ausgestalten werden muss.
Für das Einlagern wurde anhand von Figure 3.14 [1, p. 52] ein genereller Ablauf gegeben. Gleiches gilt für das Auslagern, welches mit Figure 3.9 [1, p. 47] beschrieben wird. Beim Einlagern gilt als Bedingung für die eingelagerte Menge, dass sie größer al 0 sein muss. Beim Auslagern gilt diese Bedingung ebenfalls, wird allerdings durch R-21 dahingehend spezifiziert, dass der Nutzer beim Auslagern einer größeren Menge, als der Gegenstand besitzt, gewarnt werden muss, dies Aktion allerdings durch Bestätigen dennoch durchführen kann.
Des Weiteren spezifiziert R-22, dass ein Gegenstand, dessen Menge beim Auslagern 0 oder weniger erreicht, gelöscht werden muss. Dabei ist eine „spezielle Nachricht hinterlegt, die die Menge des Gegenstands unmittelbar vor dem Löschvorgang beinhaltet“ [1, p. 25]. Der genaue Inhalt dieser Nachricht wurde nicht weiter spezifiziert. Des Weiteren wird davon ausgegangen, dass die Bezeichnung „unmittelbar“ falsch gewählt wurde, und stattdessen die Menge des Gegenstands vor dem Auslagern in der Nachricht stehen muss. Grund hierfür ist, dass die Menge des Gegenstands unmittelbar vor einem hierdurch hervorgerufenen Löschvorgang immer 0 oder weniger sein muss und davon ausgegangen wurde, dass es für den Kunden eher von Interesse ist, wie viele Gegenstände ausgelagert wurden.
Für das Löschen von Gegenständen ist anhand von Figure 3.10 [1, p. 48] ein genereller Ablauf gegeben. Hierbei ist zusätzlich zu dem zu löschenden Gegenstand ein Löschgrund verpflichtend, im Wiederspruch zu R-24, wo der Löschgrund als Möglichkeit des Systems beschrieben wird. Um diesen Wiederspruch aufzulösen wird im weiteren Verlauf des Dokuments davon ausgegangen, dass jede Löschung von Gegenständen einen Löschgrund besitzt, dieser allerdings auch leer sein kann. Es wurde nicht verlangt, dass gelöschte Gegenstände nicht mehr in der Datenbasis zu finden sind, stattdessen wurde anhand von R-23 beschrieben, dass diese nicht mehr in den Detailinformationen des jeweiligen Lagerortes oder des jeweiligen Gegenstandstypen zu finden sein sollen, was daher auch durch das Verstecken dieser Gegenstände erfolgen kann.

3.2.4.3 Gegenstandsübersicht

Figure 3.5 [1, p. 44] und Figure 3.11 [1, p. 49] beschreiben die Übersicht von respektive Gegenständen und Lagerorten. Beide beschreiben eine Detaileinsicht eines einzelnen Gegenstandes beziehungsweise Lagerortes und bei beiden dieser Detailansichten wird eine Gegenstandsliste mit aufgeführt. R-15 spezifiziert, dass diese Liste in der Detailansicht eines Gegenstandstypen alle Gegenstände beinhalten soll, die von diesem Typ sind. Auf gleiche Weise spezifiziert R-18, dass die Liste bei einem Lagerort alle Gegenstände, die diesem Lagerort zugewiesen sind, angezeigt werden sollen. Diese beiden Anforderungen werden von R-29 dahingehend erweitert, dass die Listen nur Gegenstände beinhalten dürfen, die nicht für den Nutzer gesperrt sind.

Des Weiteren wird anhand der Aktivitätsdiagramme ersichtlich, dass der Nutzer einen Gegenstand in der Liste auswählen können muss und dass sich dabei eine Detailübersicht dieses Gegenstandes öffnen soll. Diese Detailübersicht wird von R-19 beschrieben. Über den Inhalt dieser Detailansicht wird allerdings nur beschrieben, dass auch Informationen über den Gegenstandstypen beinhaltet sein müssen. Es wird nicht genauer erläutert, was genau unter allen „relevanten Informationen eines Gegenstandes“ zu verstehen ist [1, p. 33]. Es wird daher im weiteren Verlauf des Dokumentes davon ausgegangen, dass die relevanten Informationen eines Gegenstandes dessen Typ, Lagerort und Menge zu verstehen sind.

3.2.5 F-05 Verwaltung von Nutzern

Unter der Funktionalen Einheit der Nutzerverwaltung wurden die Punkte Authentifizierung und Verwaltung von Nutzerrechten zusammengefasst. Der Grund ist, dass die Themen dieser beiden Punkte stark miteinander verwandt sind.

3.2.5.1 Authentifizieren von Nutzern

Es wird anhand von Figure 3.3 [1, p. 42] und R-1 sowie R-42 ersichtlich, dass ein Nutzer sich authentifizieren muss, um die Anwendung zu bedienen, allerdings so lange authentifiziert bleiben soll, bis er sich explizit ausloggt, auch über Beenden der Anwendung hinaus. Das Persistieren der Authentifikation nach Beenden der Anwendung wurde allerdings mit der Priorität „Niedrig“ eingestuft und kann daher zurückgeschoben werden. Da in den Annahmen des Anforderungsdokuments explizit ein Mehrbenutzerbetrieb als nicht notwendig empfunden wurde, wird ein solcher nicht in dem Architekturentwurf berücksichtigt.

3.2.5.2 Verwaltung von Nutzerrechten

R-2 beschreibt die Existenz von Benutzerberechtigungen, die Nutzern Zugang zu bestimmten Funktionen der Anwendung bieten sollen. Es wird allerdings nicht genauer beschrieben, was für Berechtigungen es geben soll, oder welche Funktionen mithilfe von Berechtigungen geschützt werden sollen.
Allerdings beschreiben R-28 und R-30, dass Gegenstände und Gegenstandstypen für bestimmte Nutzer gesperrt werden können sollen. Da keine Berechtigungen spezifiziert wurden und für das Sperren von Gegenständen und Gegenstandstypen diese erst einmal existieren müssen, wurde die Funktionale Einheit der Nutzerverwaltung prioritätsmäßig nach dem Umgang mit Gegenständen gesetzt.

3.2.6 F-06 Durchführen einer Inventur

Das Durchführen einer Inventur wurde Prioritätsmäßig an letzte Stelle gesetzt. Dies liegt daran, dass eine Inventur als eine Aneinanderreihung von Gegenstandsänderungen gesehen werden kann. Hierfür müssen also Gegenstände existieren und bearbeitbar sein.
Figure 3.15 [1, p. 53] beschreibt den generellen Ablauf einer Inventur. Nach R-32 überspannt eine Inventur eine Menge an Lagerorten. Diese werden sequenziell abgearbeitet. Hierbei werden je Lagerort alle Gegenstände sequenziell gezählt. R-33 spezifiziert, dass die Informationen und Lagerort des zu zählenden Gegenstandes angezeigt werden müssen. R 34 erweitert diese Anzeige um die Möglichkeit, die gezählte Menge des Gegenstandes einzutragen. Stimmt die vom Nutzer eingetragene Menge mit der im System hinterlegten überein, so geht es mit dem nächsten Gegenstand weiter, ansonsten spezifiziert R-35, dass der Nutzer die Menge erneut eingeben muss. Sollte hierbei einem Gegenstand die Menge 0 zugewiesen werden, so soll laut R-38 der Gegenstand gelöscht werden.

4 Architekturentscheidungen

Für das Finden der Architektur der Anwendung mussten mehrere offene Fragestellungen geklärt werden. Diese Fragestellungen werden in diesem Kapitel zusammen mit den untersuchten Alternativen aufgeführt. Getroffene Entscheidungen werden hierbei anhand der Alternativen diskutiert und begründet. Da für dieses Projekt mehrere Grundlegenden Fragen zu beantworten waren, wurden diese priorisiert und anhand dieser Priorisierung nacheinander abgearbeitet. Daher enthalten die Entscheidungsmatrizen aller Entscheidungen als KO-Kriterium einen Punkt, der angibt, ob diese Alternative mit den bisher getroffenen Entscheidungen vereinbar ist. Sofern notwendig enthalten die jeweils zu treffenden Entscheidungen eine Begründung ihrer Priorisierung.
Die Auflistung der getroffenen Entscheidungen ist hierbei absteigend priorisiert, um den Entscheidungsprozess transparent zu dokumentieren.

In den dargestellten Entscheidungsmatrizen sind die gewählten Alternativen für eine bessere Übersichtlichkeit unterstrichen. Die Zeilen der jeweiligen Entscheidungsmatrizen stellen hierbei die Vergleichskriterien in (sofern nicht anders beschrieben) absteigender Priorisierung dar.

4.1 E 01 Technologie-Stack: Entscheidung der Entwicklungsplattform

Das Anforderungsdokument beschreibt, dass die Anwendung für „mobile Endgeräte“ konzipiert ist [1, p. 4]. Des Weiteren soll die gesamte Anwendung inklusive der Datenbasis auf dem Gerät selbst laufen [1, p. 55]. Da allerdings nicht genauer spezifiziert wurde, was unter einem „mobilen Endgerät“ zu verstehen ist, ist hier ein großes Risiko festzustellen, da das Portieren der Anwendung bei Implementierung auf einer falschen Plattform mit hohem Aufwand verbunden sein kann.
Es wurde davon ausgegangen, dass sich mobiles Endgerät auf Tablet oder Smartphone bezieht, nicht auf beispielsweise Laptops, Notebooks oder PDAs.

Für diese Geräte gibt es mehrere Architekturen und Betriebssysteme, weswegen eine Unterstützung aller möglichen Geräte nicht lohnend erscheint. Des Weiteren gibt es innerhalb von Gerätefamilien mit gleichem Betriebssystem auch Displays mit verschiedenen Anzeigeauflösungen. Hierfür eine Möglichkeit zu finden, die gleiche Nutzererfahrung unabhängig von der Geräteauflösung zu finden, erweist sich als schwierig.

Es müssen also im ersten Schritt Rahmenparameter für die Plattform, auf der Entwickelt werden soll, festgelegt werden. Hierfür ist wichtig, zu evaluieren, auf welchen Typen von Endgeräten die Anwendung hierdurch funktionstüchtig zu machen ist, sowie ob eine Portabilität zu anderen Plattformen gegeben ist.

	Android Studio	**Xamarin**	**Flutter**	**Unity 3D**
Unterstützt Android	Ja	Ja	Ja	Ja
Unterstützt Apple iOS	Ja	Ja	Ja	Ja
Geschätzter Einarbeitungsaufwand	Gering	Mittelmäßig	Unbekannt	Mittelmäßig bis Hoch
Programmiersprache	Java/Kotlin	.NET	Dart	.NET
Unterstützt Windows Phone	Nein	Ja	Nein	Ja

Tabelle 2: Entscheidungsmatrix für Entwicklungsplattform

Laut statcounter GlobalStats liegt der derzeitige Marktanteil an Android OS in Deutschland bei 68.2%, und der von iOS bei 31.22% [2]. Daraus folgt, dass diese beiden Betriebssysteme einen Marktanteil von über 99% abdecken, weswegen andere Betriebssysteme bei der Auswahl der Entwicklungsplattform außer Acht gelassen wurden.
Da der Marktanteil von Android OS mehr als doppelt so hoch wie der von Apple iOS liegt, und da die Entwickler derzeit kein Gerät mit dem Apple iOS Betriebssystem besitzen, wird die erste Anwendung auf ein Android-Gerät konzipiert und darauf getestet. Da alle angeschauten Möglichkeiten allerdings Cross-Plattform Entwicklung für iOS erlauben wurde dies als nicht problematisch angesehen. Durch das Nutzen von Android ist es auch möglich, die vorher ausgeschlossenen Gerätetypen Laptop und Notebook mit einzubeziehen, da es für Windows, MacOS und Linux bereits Emulatoren gibt, mithilfe derer Android Anwendungen ausgeführt werden können [3].
Da derzeit keine Vorkenntnisse in einer dieser Entwicklungsumgebungen vorhanden sind, wurde der Einarbeitungsaufwand aufgrund der benutzten Entwicklungssprache eingeschätzt. Testprojekte wurde aufgrund von Zeitmangel nicht angelegt.

Android Studio:
Android Studio ist die offizielle Entwicklungsumgebung für Android OS und basiert auf Technologien der Firma JetBrains [3].
Aufgrund der gefestigten Vorkenntnisse der Entwickler in Java und dem benutzten Build-Tool Gradle ist eine vergleichsweise geringe Einarbeitungszeit geschätzt. Dies wird dadurch unterstützt, das bereits Produkte der Firma JetBrains verwendet wurden, was die Einarbeitung in die Entwicklungsumgebung erleichtert und die Produktivität aufgrund beispielsweise bereits bekannter Tastenkürzel erhöhen kann.

Xamarin:
Xamarin ist eine Quelloffene Entwicklungsplattform, die auf dem .NET Framework aufbaut und Cross-Plattform Projekte ermöglicht [4].
Da bereits einige Vorkenntnisse der Entwickler in .NET vorhanden sind, wird der Einarbeitungsaufwand mittelmäßig eingeschätzt.

Flutter:
Flutter ist ein Quelloffenes UI Toolkit von Google, das für die Entwicklung von Cross-Plattform Projekten konzipiert ist. Als Sprache benutzt es Dart, eine Objektorientierte Programmiersprache [5]. Da hierzu keine Vorkenntnisse bestehen, konnte der Einarbeitungsaufwand nicht ermittelt werden.

Fazit:
Aufgrund der geringsten geschätzten Einarbeitungszeit wurde sich für Android Studio entschieden. Dies wirkt sich insofern auf den Lösungsspielraum ein, als dass nun nur noch Bibliotheken und Frameworks in Frage kommen, die für Java oder Kotlin existieren. Es ist davon auszugehen, dass es keine Änderungen der Anforderungen geben wird, die im Konflikt mit dieser Entscheidung stehen werden. Hierfür müsste explizit ein bestimmtes Tool oder Framework vom Auftraggeber vorgegeben werden müssen, was zum derzeitigen Zeitpunkt für nicht wahrscheinlich gehalten wird.

4.2 E 02 Datenspeicherung: Form der Datenspeicherung

Für den Normalbetrieb der Anwendung müssen die im System hinterlegten Gegenstände, die existierenden Lagerorte, sowie die eingelagerten Gegenstände persistent gespeichert werden. Zudem verlangt das Anforderungsdokument in R-44, dass Bestandsänderungen protokolliert werden müssen.

Aufgrund der Revisionssicherheit muss auch sichergestellt werden, dass die Möglichkeit besteht, den Datenbestand zu einem früheren Zeitpunkt der Anwendung einzusehen, ohne den derzeitigen Datenbestand dabei zu verlieren.

	Direkte Speicherung	**Event Sourcing**	**Double Entry Book Keeping**	**Speicherung mit Versionierung**
Mit bisherigen Entscheidungen vereinbar	Ja	Ja	Ja	Ja
Frühere Zustände wieder herstellbar	Nein	Ja	Ja	Ja
Protokollierung	Nicht evaluiert	Gegeben	Ja	Schwierig
Anforderung an Hauptspeicher	Nicht evaluiert	Groß	Gering	Gering
Einfluss auf Startzeit	Nicht evaluiert	Groß	Gering	Gering
Anforderung an Sekundärspeicher	Nicht evaluiert	Mittel bis Groß	Groß	Mittel

Tabelle 3: Entscheidungmatrix für Datenspeicherform

Aufgrund dieser Anforderungen wurden verschiedene Ansätze erarbeitet, die Datenspeicherung aufzubauen:

Direkte Speicherung des jeweiligen Zustandes der Datenbasis:
Die erste Überlegung war, die Datenbasis zum jeweiligen Zustand der Anwendung direkt zu persistieren. Da hierbei allerdings die durch die Revisionssicherheit geforderte Möglichkeit, frühere Zustände der Anwendung wiederherzustellen, nicht sichergestellt wurde, wurde dieser Ansatz nicht weiterverfolgt.

Event-Sourcing Pattern:
Beim Event Sourcing Pattern wird nicht der jeweilige Zustand der Datenbasis persistiert, sondern eine Liste an Änderungen, die ab einem bestimmten Zeitpunkt der Anwendung durchgeführt werden. Diese Änderungen werden häufig als Events bezeichnet. Der jeweilige Zustand der Datenbasis kann also durch eine sequenzielle Projektion aller dieser Änderungen aufgebaut werden [6]. In einem einfachen Aufbau kann man in der Datenspeicherung also nur mit einer Liste an Events auskommen, die sequenziell auf eine „leere“ Datenbasis angewendet werden. Dieser Prozess müsste einmal beim Starten der Anwendung geschehen, da hierbei die Datenbasis lokal aufgebaut würde. Weiterhin müssten alle Änderungen, die im Verlauf der Anwendung entstehen einmal auf die Lokale Datenbasis angewendet werden und einmal in Form eines oder mehrerer Events persistiert werden.
Ein Vorteil dieses Patterns ist, dass im einfachsten Fall nur Events persistiert werden müssen. Des Weiteren ist es einfach möglich, einen früheren Zustand der Anwendung wieder herbeizuführen, indem nur die Events angewendet werden, die vor dem wiederherzustellenden Zeitpunkt ausgeführt worden

sind. Außerdem ist die Anforderung an eine Änderungsprotokollierung durch das Nutzen von Events bereits erfüllt, da Events genau diese Form an Änderungen darstellen.
Ein Nachteil dieses Patterns ist ein erhöhter Bedarf an Hauptspeicher, da die Datenbasis lokal abgebildet werden muss. Auch sorgt das Speichern aller Events dafür, dass langfristig immer mehr Events entstehen, da selbst eine „leere“ Datenbasis das Ergebnis von vielen Events sein kann. Außerdem ist durch das Anwenden aller Events zu Beginn der Anwendung eine höhere Startzeit gegeben.

Versionierung der jeweiligen Zustände der Anwendung:
Ein anderer Ansatz basiert darauf, den Ansatz der Direkten Sicherung des Anwendungszustandes zu erweitern. Hierbei muss die Möglichkeit, frühere Datenzustände abzurufen, geschaffen werden. Eine Möglichkeit, dies zu erreichen wäre die Erweiterung der zu persistierenden Objekte um ein Versionsfeld, das bei jeder Modifizierung des Objektes hochgezählt wird. Alternativ kann dieses Versionsfeld auch in Form eines Zeitstempels vorliegen. Fragt ein Nutzer dann Daten an, so muss sichergestellt werden, dass nur die „neuesten“ Werte der Daten, d.h. diejenigen Einträge mit der höchsten Versionsnummer, zurückgegeben werden. Für das Anfragen eines früheren Zustandes der Anwendung müssen dann die Einträge mit der höchsten Versionsnummer gewählt werden für die gilt, dass die jeweilige Versionsnummer vor dem abgefragten Zeitpunkt erstellt wurde. Dies ist bei dem Verwenden eines Zeitstempels als Versionsnummer einfach umzusetzen.
Ein Vorteil dieses Ansatzes ist, dass hierbei die Anwendung die gesamte Datenbasis nicht selbst im Hauptspeichers verwalten muss. Auch ist der Start der Anwendung nicht im Wesentlichen verlangsamt, da allerhöchstens die Verbindung zur Speicherform der Daten aufgebaut werden muss.
Ein Nachteil dieses Ansatzes ist, dass hierbei Daten redundant gespeichert werden. Des Weiteren könnten durch die hierdurch benötigte Datenaggregierung die Anfragen von Daten an die Speicherschicht langsamer werden. Auch ist hier die Frage, wie die Anforderung nach einer Protokollierung von Änderungen umgesetzt werden soll.

Double Entry Book Keeping:
Unter dem Begriff Double Entry Book Keeping beschreibt Eric Damtoft eine Kombination aus Event Sourcing und dem Speichern der jeweiligen Datenbasis der Anwendung [7]. Hierbei wird der letzte Zustand der Datenbasis parallel zu der Liste der Events persistiert. Hierdurch entfällt der Nachteil des Event-Sourcing, die gesamte Datenbasis im Hauptspeicher zu erfordern. Ebenso entfällt hierbei der Zeitaufwand, alle Events zu Beginn der Anwendung neu anzuwenden. Dafür ist hierfür die Protokollierung der Änderungen weiterhin gegeben. Weiterhin erlaubt das parallele Führen von Eventliste und Datenbasis eine Validierung der Datenbasis, indem überprüft wird, ob ein Anwenden aller Events zu dem derzeitigen Datenbestand führt. Hierbei kann allerdings nur eine Inkonsistenz erkannt werden, es obliegt dem Entwickler oder dem Nutzer, bei einer solchen Inkonsistenz zu entscheiden, welcher Datenstand zu benutzen ist.

Fazit:
Der Ansatz der direkten Datenspeicherung wurde direkt in der ersten Phase verworfen, da hier nicht die Möglichkeit gegeben war, frühere Zustände wiederherzustellen.
Als zweite Harte Anforderung wurde die Protokollführung gesetzt. Da diese bei den übrig gebliebenen drei Ansätzen gegeben war, wurde vorerst weiter gegangen, auch wenn sie bei der Versionierten Speicherung als schwierig eingeschätzt wurde.
Da es sich um ein mobiles Endgerät handelt, wurde die Anforderung des Hauptspeichers als wichtiges Entscheidungskriterium gewählt. Laut dem Web Intelligence Report von DeviceAtlas [8] waren im Februar 2019 der Großteil der Smartphones mit Hauptspeicher zwischen einem und vier Gigabyte ausgestattet. Auf längerfristige Sicht ist daher eine Speicherung der gesamten Datenbasis im

Hauptspeicher für diese Geräte nicht umsetzbar. Es sei hierbei angemerkt, dass Smartphones nur eine Teilmenge aller mobilen Endgeräte darstellen, es kann also durchaus Geräte geben, die genügend Hauptspeicher besitzen, um einen Hauptspeicherintensiveren Ansatz zu unterstützen. Dennoch wurde sich hierbei entschieden, den Speicherbedarf hierbei stärker zu berücksichtigen, um eine größere Menge an Geräten unterstützen zu können.
Da das Event Sourcing Pattern viel Hauptspeicher benötigt, wurde es in der Entscheidungsfindung erst einmal zurückgestellt. Damit war die Entscheidungsfindung auf die Versionierte Speicherung gegen das Double Entry Book Keeping reduziert.
Hierbei wurde sich für das Double Entry Book Keeping entschieden, da hier die Protokollierung als einfacher umzusetzen eingeschätzt wurde.
Zwar ist dies die Technologie, für die der Größte Aufwand an Sekundärspeicher geschätzt wurde, allerdings wurde dieser Punkt für den Fall einer nicht eindeutigen Entscheidung mit aufgenommen. Gleiches gilt für den Einfluss der Ansätze auf die Startzeit der Anwendung. Hierbei muss angemerkt werden, dass für den gewählten Ansatz hierbei ein guter Wert geschätzt wurde, und dieser Punkt damit ohnehin nicht gegen das Double Entry Book Keeping gesprochen hätte.
Da die Anforderung der Revisionssicherheit als zentrale Anforderung an die Anwendung gestellt wurde und mit dieser Entscheidung versucht wurde, dieser Anforderung gerecht zu werden, ist nicht davon auszugehen, dass diese Entscheidung im Nachhinein abgeändert werden muss. Selbst wenn der Kunde die Anforderung an ein revisionssicheres System zurücknehmen würde, würde dies nicht gegen den gewählten Ansatz sprechen.

4.3 E 03 Technologie-Stack: Entscheidung der zu benutzenden Datenspeicherung

Wie bereits in F-01 erläutert, ist die Wahl der Datenspeicherung maßgeblich von dem Punkt der Revisionssicherheit abhängig. Dazu kommt allerdings noch, dass die Datenbasis direkt auf dem Gerät sein soll. Daher muss ein Datenmodell gewählt werden, das auf dem mobilen Gerät selbst laufen kann. Falls möglich, sollte das benutzte Datensystem allerdings auch einfach auf einen Server oder ein anderes Speichermedium übertragbar sein, um für eine spätere Erweiterung der Anwendung weniger Aufwand betreiben zu müssen. Hierbei muss hinzugesetzt werden, dass dies keine Anforderung des ursprünglichen Architekturentwurfs war, und diesem Punkt dadurch eine geringe Priorität in der Entscheidungsfindung zugeordnet wurde.

Für die Analyse wurden verschiedene Speicherformen untersucht, welche in Tabelle 4 zusammengefasst und im Folgenden genauer erläutert werden.

	SQL-Server	**SQLite**	**Objekt-orientiert**	**NoSQL**	**Datei-System**
Mit bisherigen Entscheidungen vereinbar	Nein	Ja	Ja	Ja	Ja
Relational	Ja	Ja	Nein	Nein	Nein
Revisionssicherheit sichergestellt	Teilweise	Teilweise	Teilweise	Nur manuell	Nur manuell
Skalierbarkeit	Gut	Mittel	?	Mittel	Schlecht
Erweiterbar per Netzwerk	Ja	Ja	?	k. A.	Schwer

Tabelle 4: Entscheidungsmatrix für Form der Datenspeicherung

SQL-Server:
Unter dem Begriff „SQL-Server“ wurden verschiedene Gruppen an Datenbank Management Systemen, die auf der Client-Server Einteilung basieren und SQL als Datenabfragesprache benutzen, zusammengefasst. Wäre die Entscheidung auf diese gefallen, so würde in einer weiteren Analyse die zu benutzende Implementierung herausgearbeitet werden. Viele SQL-Server Implementierungen erlauben ein gesondertes Nutzerrechte System. Hierdurch könnte die Revisionssicherheit durch Serverseitiges Verpflichten einer Write Once, Read Many (WORM) Struktur vereinfacht werden. Da hierfür allerdings ein gesonderter Server benötigt würde, wurde diese Option nicht genauer evaluiert.

SQLite:
SQLite ist eine eingebettete SQL-Datenbank Engine, die durch den direkten Zugriff auf das Dateisystem operiert [9]. Android hat bereits einige Plugins, die das Speichern von Daten innerhalb einer SQLite Datenbank umsetzen. Als Beispiel wird hier die „Room“ API aufgeführt, die auch anhand eines gesonderten Guides ausführlich auf der Google Android Developer Page erklärt wird, und daher einen geringeren Einarbeitungsaufwand verspricht [10]. Des Weiteren ist die bei dieser API benutzte Datenbank innerhalb der Anwendungsdaten der Datei gespeichert, was dafür sorgt, dass die Datenbasis nicht von außen modifiziert werden kann. Dies kommt der Revisionssicherheit zugute, da diese verlangt, dass Daten im Nachhinein nicht mehr verändert werden dürfen.

Objektorientiert:
Unter dem Begriff „Objektorientiert“ wurden Datenbanksysteme zusammengefasst, die nach dem Prinzip der Objektorientierung aufgebaut sind. Diese Datenbanksysteme erlauben Vererbungshierarchien zwischen persistierten Datenobjekten [11, p. 395]. Da sich für die Entwicklung der Anwendung für eine Objektorientierte Programmiersprache entschieden wurde, würde auf diese Art auch der sog. *Impedance Mismatch,* der die Schwierigkeiten, Objekte einer Objektorientierten Programmiersprache auf ein Relationales Datenbankschema zu übertragen, beschreibt [12], durch ein Objektorientiertes Schema umgangen werden. Hierfür wurde zuerst eine Datenbankbibliothek „Objectbox“ evaluiert. Objectbox ist Eigentum von Objectbox Limited und unter der Apache 2.0 Lizenz lizensiert [13]. In einer Testimplementierung [14, p. branch oo_database_objectbox] wurde hierbei allerdings herausgefunden, dass diese Bibliothek nicht in der Lage ist, polymorphe Abfragen zu verarbeiten und damit für der durch das Nutzen einer OO-Datenbank versprochenen Vorteil des Nutzens von Vererbungshierarchien nicht umsetzbar ist. Aus diesem Grund wurde mit Perst eine weitere Bibliothek gefunden, die allerdings unter einer GNU-GPL Lizenz lizensiert ist. [15]. Da diese Lizenz eine Offenlegung des eigenen Quellcodes voraussetzt könnte sie nicht die richtige für ein Kommerzielles Produkt sein. McObject erlaubt auch eine kommerzielle Lizensierung, zu der allerdings keine weiteren Informationen auf der Website gegeben sind. Aufgrund des Zeitmangels wurde hierfür kein gesondertes Testprojekt erstellt. Laut Spezifikation erlaubt diese Bibliothek bis zu 2.000.000.000 Einträge und maximal 1 Terrabyte an Datenbankgröße. Da für die Anwendung Mittelständische Unternehmen als Endkunden spezifiziert wurden, wurden diese Werte auf mittel- bis langfristige Sicht für ausreichend eingeschätzt. Grund hierfür ist eine Überschlagsrechnung, in der 100 Lagerorte, 2000 Gegenstandstypen ein Gegenstand je Gegenstandstyp und Lagerort bei 1000 Event Einträgen pro Tag trotzdem noch über 5000 Jahre erreichen würde, ohne die angegebene Grenze zu erreichen (2.000.000.000 – 100 – 2000 – (100 * 2000)) / 365 / 1000 ≈ 5.479.

NoSQL:
Unter dem Begriff „NoSQL“ wurden verschiedene Gruppen an Datenbanksystemen zusammengefasst, die nicht Tabellenbasiert Daten speichern. Sie stellen einen Gegenpol zu Relationalen Datenbankmodellen dar [16]. Streng genommen sind auch die bereits vorgestellten Objektorientierten

Datenbanksysteme eine Teilmenge dieser Gruppe. Da diese allerdings bereits vorgestellt wurden, werden sie hier nicht miteinbezogen. Da die Form der zu persistierenden Daten allerdings als strukturiert angesehen wurde, wurde ein Nicht-Relationales Datenmodell für nicht notwendig empfunden und daher nicht weiter evaluiert.

Fazit:
Da das Verwenden eines gesonderten SQL-Servers nicht mit der Anforderung, dass die Datenbasis direkt auf dem Gerät gespeichert werden soll vereinbar war, wurde sie als erste verworfen.
Laut dem Ranking von Datenbanksystemen von solid IT ist der Großteil an verwendeten Systemen nach dem Relationalen Datenbankmodell modelliert [17]. Aus diesem Grund wurde davon ausgegangen, dass das Verwenden eines relationalen Entwurfsmusters langfristig eine bessere Wartbarkeit aufgrund der höheren Bekanntheit verspricht. Des Weiteren wird durch das Verwenden eines DBMS, das SQL spricht, auch eine einfachere Migration des Datenmodells auf einen externen Server in späteren Ausführungen der Anwendung erwartet. Aufgrund dieser Präferenz eines Relationalen Entwurfsmusters wurde sich für SQLite als Datenbanksystem entschieden.
Hierbei ist davon auszugehen, dass ein relationales System sich mit den Vorstellungen des Kunden vereinen lässt, und die Entscheidung darum im Nachhinein nicht angepasst werden muss. Lediglich die genaue Wahl, was für ein relationales Datenbankmanagementsystem zu benutzen ist, könnte im Nachhinein angepasst werden müssen, wie bereits bei dem Punkt „Erweiterbar per Netzwerk“ angesprochen wurde. Hier lässt sich eine höhere Variabilität dadurch erzeugen, dass man den Zugriff auf die Datenbank einkapselt, wodurch man das genaue Datenbanksystem in späteren Ausführungen der Anwendung auswechseln kann.

4.4 E 04 Implementierung: Wahl der Grundlegenden Struktur des Codes

Um eine gut erweiterbare und dokumentierte Architektur zu erstellen, bietet es sich an, zu diesem Zeitpunkt grundlegende Entscheidungen über zu verwendende Muster zu treffen.
Hierbei müssen die bereits getroffenen Architekturentscheidungen bedacht werden. Es wurde für eine Datenspeicherung nach dem Double Entry Book Keeping Pattern entschieden. Für die Abfrage früherer Zustände sowie eine Verifikation der Datenbasis möglich sein muss, die gespeicherten Events erneut auf eine Datenbasis anzuwenden. Aus diesem Grund wurde bereits eine Vorauswahl getroffen, um Systeme zu verwenden, die bereits intern durch Events oder Äquivalente Nachrichtenstrukturen arbeiten.

	Event Driven	**Akka**	**Message Driven**	**Synchrone Abarbeitung**
Mit bisherigen Entscheidungen vereinbar	Ja	Ja	Ja	Ja
Einfach mit Events zu vereinen	Ja	Ja	Mittel	Mittel
Erweiterbarkeit	Gut	Gut	Gut	Normal
Entwicklungs-overhead	Groß	Mittel	Groß	Gering

Tabelle 5: Entscheidungsmatrix für Implementirungspattern

Synchrone Abarbeitung:
Unter „Synchroner Abarbeitung“ wird hierbei das Muster verstanden, verschiedene Abläufe des Codes innerhalb eines oder mehrerer (verschachtelten) Methodenaufrufe zu realisieren. Gregor Hohpe beschreibt die Implikationen eines solchen Systems als die Charakteristiken eines Call Stacks [18, p. 3f]. Hierbei nennt er vier Annahmen, die in diesem Ansatz implizit getroffen werden: Die erste Annahme ist, dass immer nur eine Sache auf einmal geschieht, das bedeutet das sobald eine Funktion eine andere aufruft, sie so lange wartet, bis die aufgerufene Funktion fertig ist, die Abarbeitung also synchron verläuft. Die Zweite und Dritte Annahme ist, dass bereits im Vornherein bekannt ist, in welcher Reihenfolge die Ausführung zu geschehen hat und dass bekannt ist, welche Komponenten eine benötigte Funktionalität bereitstellen. Die letzte Annahme ist hier, dass die Ausführung auf einer einzelnen (virtuellen) Maschine erfolgt. Diese vier Annahmen sind mit den derzeitigen Anforderungen an die Anwendung vereinbar, da es sich derzeit um eine Anwendung handelt, die auf einem einzelnen Gerät laufen soll. Sollte beispielsweise die Datenbasis auf einen externen Datenbankserver verschoben werden, so müsste dieser Stil zumindest in Teilen für die Kommunikation mit dem externen Gerät gebrochen werden.
Hierbei ist der Vorteil, dass es sich innerhalb jeder Methode herausfinden lässt, woher die Anfrage kam, was der Fehlerbehebung während des Entwicklungsprozesses zugutekommt. Des Weiteren benötigt dieser Ansatz (theoretisch) keinen zusätzlichen Infrastrukturaufwand, wie es bei den anderen beiden Ansätzen der Fall ist. Daher kann die Entwicklung schneller begonnen werden. Ebenso besteht aufgrund der Synchronen Ausführung nicht die Chance, dass der Datenbestand aufgrund einer anderen Ausführungsreihenfolge der einzelnen Anweisungen in einen inkonsistenten Zustand kommt. Hinzu kommt, dass durch die fest gegebene Ausführungsreihenfolge die Ausführung stark optimiert werden kann, was zu einem schnelleren Programm führen kann.
Dafür ist in diesem System allerdings der Nachteil, dass Erweiterungen direkt in die entsprechenden Klassen eingeführt werden müssen, was die Wartbarkeit erschweren kann. Außerdem könnte durch dieses System eine parallele Benutzung der App von mehreren Benutzern erschwert werden.

Event Driven, Message Driven und Akka:
Event Driven und Message Driven Design sind zwei Komplementäre Ansätze der asynchronen Programmierung. Das Reaktive Manifest gibt hierbei als Differenzierung zwischen den beiden, dass eine Nachricht (Message) gerichtet an eine Zieladresse versandt wird. Ein Ereignis (Event) hingegen wird ungerichtet emittiert, sobald eine Komponente einen bestimmten Zustand erreicht hat. Events können von einer beliebigen Anzahl von Empfängern (inkl. keinem) empfangen werden [19]. Im Kontext der Anwendung könnte ein Event beispielsweise das Anlegen oder Bearbeiten eines Lagerortes quittieren. Eine Nachricht dagegen könnte gesendet werden, wenn der Nutzer die Änderungen eines Lagerortes anstoßen möchte.
Da Events erst bei Erreichen eines Zustandes emittiert werden, hat sich die Konvention gebildet, diese in der Vergangenheitsform auszudrücken. Ein Event könnte also ein „Lagerort wurde bearbeitet“ Event heißen. Diese Definition des Begriffs Event lässt sich mit der die bereits in diesem Dokument für das Event Sourcing und Double Entry Book Keeping verwendet wurde vereinen, weshalb nicht weiter zwischen den Begriffen differenziert wird.
Da bereits Events in der Datenbank persistiert werden sollen, bietet sich also eine Event Driven Architecture an. Wählt man eine rein Event-Driven Architektur, so muss festgelegt werden, welche Events persistiert werden sollen, und welche zwar für die Laufzeit der Anwendung von Relevanz sind, allerdings nicht persistiert werden sollen. Hierbei könnte als Faustregel gelten, dass nur Events, die die Datenbasis verändern, persistiert werden sollen. Als Event, das nicht persistiert werden muss, wäre in einer rein Event Driven Architektur beispielsweise ein „Knopf X wurde gedrückt“ Event oder je nach Implementierung auch ein „Der Text in Eingabefeld Y hat sich geändert“.

Ein Vorteil einer Event Driven Architektur ist, dass jede Komponente bestimmte Events empfangen und dementsprechend behandeln kann. Beispielsweise ist durch eine solche Architektur sehr einfach möglich, neue Komponenten hinzuzufügen, die beispielsweise jedes Mal, wenn ein Gegenstand gelöscht wird, eine E-Mail versendet, oder eine Bestellung anstößt, um den Gegenstand nachzubestellen. Hierbei muss hinzugefügt werden, dass solche Szenarien nicht als Anforderungen definiert wurden. Allerdings wurde ein solches Beispiel einer automatisierten Bestellung als mögliches Wachstumspotenzial der Anwendung im Anforderungsdokument vorgeschlagen [1, p. 55]. Der Unterschied ist allerdings, dass bei der Beschreibung dieser möglichen Erweiterung von einer Auswertung der Protokolldaten (welche zu diesem Zeitpunkt als Event modelliert geplant sind) erst im Nachhinein gesprochen wird, wohingegen eine Erweiterung anhand der Events bereits zur Laufzeit möglich wäre.
Ein Weiterer Vorteil dieser Architektur ist, dass sie durch das Nutzen von ungerichteten Eventemissionen eine sehr lose gekoppelte Architektur ermöglicht. Hierdurch könnte in Zukunft die Anwendung auf mehrere Systeme verteilt werden, und beispielsweise die mobile Anwendung nur noch die Nutzereingaben behandeln, wobei ein Zentraler (oder mehrere Verteile) Server die eigentliche Logik beinhalten. Es wird durch diese Architektur also eine hohe Variabilität versprochen.
Als Nachteil dieser Architektur ist allerdings ist, dass beim globalen Versenden von Events es zu Problemen bezüglich des Datenschutzes kommen kann, da Informationen anhand von Events abgefragt werden könnten, die nicht für alle Komponenten zugänglich sein sollen. Als Beispiel wäre hier beispielsweise das Event, das gesendet wird, wenn ein Nutzer sein Passwort ändern, oder sich einloggen möchte. Wenn dieses Event so gesendet würde, dass jeder es empfangen kann, könnte ein Angreifer an diese Daten kommen, indem er eine neue Komponente einführt, die nur diesen Event-Typ empfängt und die Daten anderweitig weiterleitet.
Aus diesem Grund werden in der Praxis häufig das Messaging Pattern und das Event Pattern zusammen angewandt. Hier ist die Apache-2.0 lizensierte Bibliothek Akka zu nennen [20]. Hier existieren ein bis mehrere Aktoren, die sich nach dem Prinzip der Message Driven Architektur gegenseitig Nachrichten zusenden. Parallel hierzu existiert auch ein Globaler Event-Stream über welchen es möglich ist, ungerichtete Events zu versenden. Hierbei können Aktoren über einen abstrakten Identifier eindeutig identifiziert werden. Somit können vertrauliche Informationen gerichtet versendet werden.

Fazit:
Für die Entscheidung wurde wieder zuerst das K.O. Kriterium, ob die Entscheidung mit den bisher getroffenen vereinbar ist, überprüft. Dies ist bei allen Strukturen der Fall. Danach wurde überprüft, wie einfach die verschiedenen Architekturen sich mit den aufgrund des Double Account Book Keeping Patterns gespeicherten Events vereinen lassen. Hierbei wurden der Event Driven Ansatz sowie das Nutzen der Akka Bibliothek aufgrund des Nutzens eines Event-Bus für einfacher zu integrieren gesehen. Danach wurde die Erweiterbarkeit an hohe Stelle gesetzt, da sie Anpassung an Änderungen an die Anforderungen der Anwendung vereinfachen kann. Somit wird hierdurch das Entwicklungsrisiko gesenkt.
Die letztliche Entscheidung zwischen dem Nutzen eines Event Driven Ansatzes und dem Nutzen der Akka Bibliothek fiel auf Akka, da durch das Nutzen einer vorexistierenden Bibliothek einen geringeren Entwicklungsaufwand verspricht, da nicht „das Rad neu erfunden“ werden muss.

Da bei dieser Entscheidungsfindung ein Hoher Wert auf Erweiterbarkeit gelegt wurde, besteht ein Risiko, dass diese Entscheidung im Nachhinein angepasst werden muss. Dies entspringt dem durch das Einbinden und Einarbeiten in Akka entstandenem zusätzlichem Arbeitsaufwand, der für eine längere Implementierungszeit sorgen kann. Hierbei muss vor dem Beginn der Implementierung überprüft werden, ob diese Entscheidung mit den Vorstellungen des Kunden zu vereinen ist. Sollte sich der Kunde während der Implementierung gegen diesen Ansatz entscheiden, so wäre das mit größeren Kosten

verbunden, sofern nicht innerhalb eines Events- oder Message Driven Ansatzes geblieben wird. In letzterem Fall könnten die Entstehenden Kosten sich im Rahmen halten, da Code eventuell weiterverwendet werden kann.

5 Sichten

In diesem Kapitel werden verschiedene Sichten auf in der Anwendung umzusetzende Anforderungen textuell und für vereinzelte Sichten auch grafisch dargestellt. Hierbei wird für eine vereinfachte Navigation in laufzeitunabhängige bzw. globale, sogenannte statische Sichten und laufzeitabhängige oder prozessbedingte, sogenannte dynamischen Sichten untergliedert.

5.1 Statische Sichten und Integritätsbedingungen

Unter den statischen Sichten werden von den genauen Daten und Prozessen unabhängige Integritätsbedingungen verstanden. Hierzu zählen die Schemata der zu persistierenden Daten sowie die Schemata oder Datentypen der in der Anwendung benutzten Objekte.
In dem Nachfolgenden Abschnitt wird zuerst die das Schema der zu persistierenden Daten analysiert und daraus die in der Anwendung verwendbaren Objekte abgeleitet.

5.1.1 Statische Sicht: Zu Persistierende Daten

Für die Persistierenden Daten wurde in E 02 festgelegt, Daten nach dem Double Entry Book Keeping Pattern zu speichern. Des Weiteren wurde in E 03 festgelegt, ein Relationales Datenschema zu verwenden. Aus diesem Grund wurde für die Anforderungsanalyse and das Datenbankschema das Entity Relationship (ER) Modell (auch mit ERM abgekürzt) für die Modellierung verwendet. Bezüglich der Grafischen Repräsentationen muss angemerkt werden, dass sich hierbei an der Syntax von Kemper orientiert wird. So werden beispielsweise Vererbungsbeziehungen mithilfe von Sechsecken und auf den Ober-Typ gerichteten Kanten dargestellt.
Im Nachfolgenden wird für die Anforderungsanalyse für jede Funktionale Einheit ein eigenes ERM modelliert das die Sicht auf die Datenbasis für die jeweilige Funktionale Einheit beschreibt. Diese einzelnen Modelle werden daraufhin zu einem Schema konsolidiert. Es ist hierbei anzumerken, dass die ERMs der einzelnen Zwischenschritte nicht vollständig sind. Des Weiteren wurden in diesen Modellen nur die Kardinalitäten aufgeführt, die aus der Analyse der jeweiligen Funktionalen Einheit direkt hervor gingen. Dieser Ansatz wurde gewählt, um eine spätere Konsolidierung zu vereinfachen.

5.1.1.1 F-01 Revisionssichere Datensicherung mit Protokollierung der Änderungen

Für F-01 wurde die Revisionssichere Datensicherung gefordert. Hierbei wurden keine besonderen Anforderungen gefunden. Für die ebenso geforderte Protokollierung der Änderungen wurden in E 02 festgelegt, Daten nach dem Double Entry Book Keeping Pattern zu speichern. Daher müssen also Events gespeichert werden.

Hierfür wurde Event als Abstrakter Ober-Typ mit mehreren Spezialisierungen modelliert. In einem ersten Entwurf wurde hier ein (unvollständiges) ERM erstellt, welches in Abbildung 1 dargestellt wird. Da zu diesem Zeitpunkt nicht alle Events sowie deren Eigenschaften ermittelt wurden, ist dieses ERM als unvollständig eingestuft. Dies wird in der späteren Konsolidierung mit den anderen ERMs entsprechend nachgebessert. Zu diesem Zeitpunkt wurde allerdings schon die Frage gestellt, wie diese Vererbungshierarchie am besten ausgedrückt werden kann. Es wird davon ausgegangen, dass diese Spezialisierungen vollständig und disjunkt sind, also es keine Instanzen geben wird, die nur „Event“ sind, und gleichzeitig kein Event mehrere dieser Spezialisierungen gleichzeitig sein kann. Diese Annahmen werden bei der Konsolidierung erneut diskutiert.

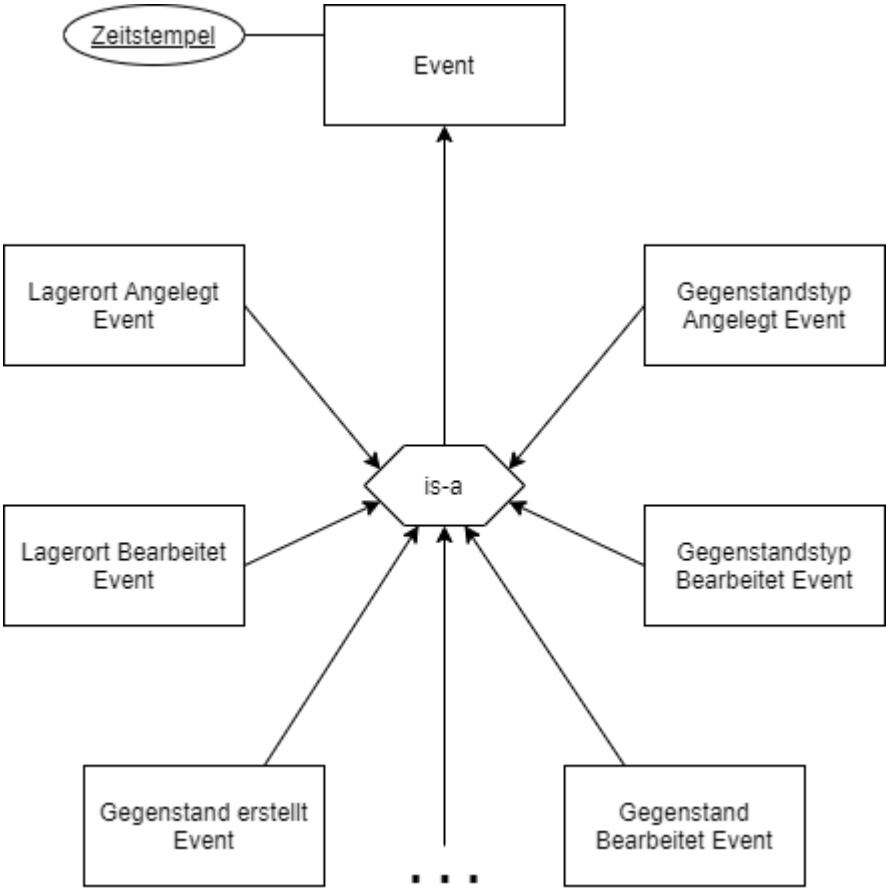

Abbildung 1: ERM Sicht für F-01

5.1.1.2 F-02 Verwaltung von Gegenstandstypen

In F-02 werden die Aktionen, die mit Gegenstandstypen durchführbar sein sollen, beschrieben. Hierbei wurden als Zentrale Abläufe das Anlegen, Bearbeiten und Suchen von Gegenstandstypen aufgeführt. Aus der der Analyse der Funktionalen Einheit ging hervor, dass Gegenstandstypen einen nicht-leeren Namen sowie eine Beschreibung besitzen. Des Weiteren ging aus der der Beschreibung der Gegenstandstypübersicht hervor, dass Gegenstandstypen eine Menge an Gegenstände ausmachen können. Aus dieser Formulierung ist zu folgern, dass es mehrere Gegenstände desselben Gegenstands geben muss, diese Relation also ein N als Kardinalität bekommen muss. Aus der Beschreibung der Gegenstandstypübersicht wird auch ersichtlich, dass Nutzer Zugriff auf Gegenstände haben sollen.

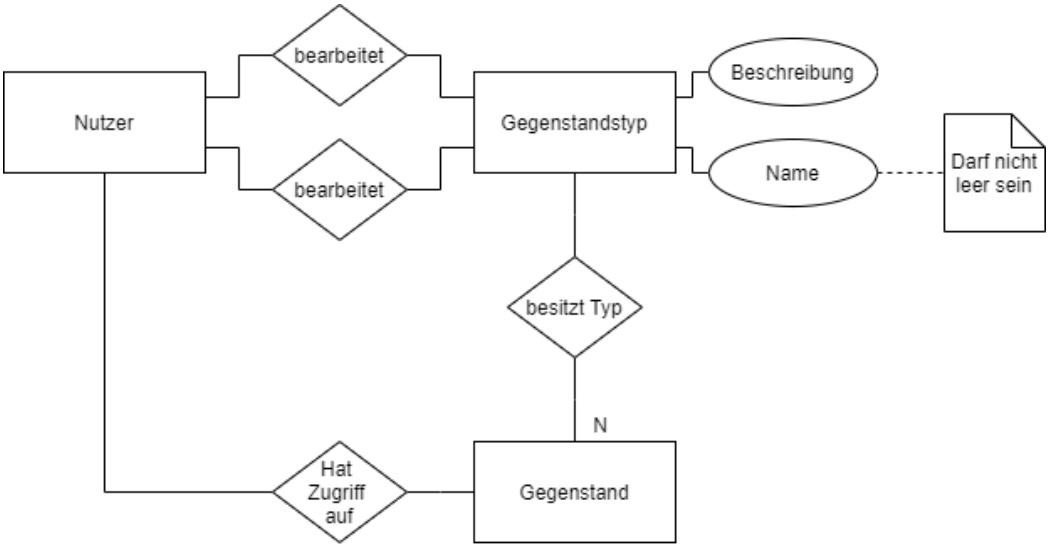

Abbildung 2:ERM Sicht für F-02

5.1.1.3 F-03 Verwaltung von LagerortenF-03

In F-03 werden die Aktionen, die mit Lagerorten durchführbar sein sollen, beschrieben. Hierbei wurden als Zentrale Abläufe das Anlegen, Bearbeiten und Suchen von Gegenstandstypen aufgeführt. Aus der Analyse der Funktionalen Einheit ging hervor, dass Lagerorte einen eindeutigen, nicht-leeren Namen sowie eine Beschreibung besitzen. In diesem ERM wurden die zusätzlichen Eigenschaften, nach denen die Lagerorte sortierbar sein sollen, als inferierte Attribute dargestellt. Der Grund hierfür liegt darin, dass diese Attribute in der späteren Anwendung vermutlich über Aggregatsfunktionen ermittelt werden, und nicht direkt in den jeweiligen Objekten zu finden sein werden. Sie wurden aber dennoch mit in das ERM mit aufgenommen, sodass sie später beim Erstellen der jeweiligen Sichten nicht vergessen werden. Da bei der Beschreibung der Lagerortübersicht wieder von einem Zugriff des Nutzers auf Gegenstände gesprochen wurde, wurde dieser hier als Relation eingeführt.

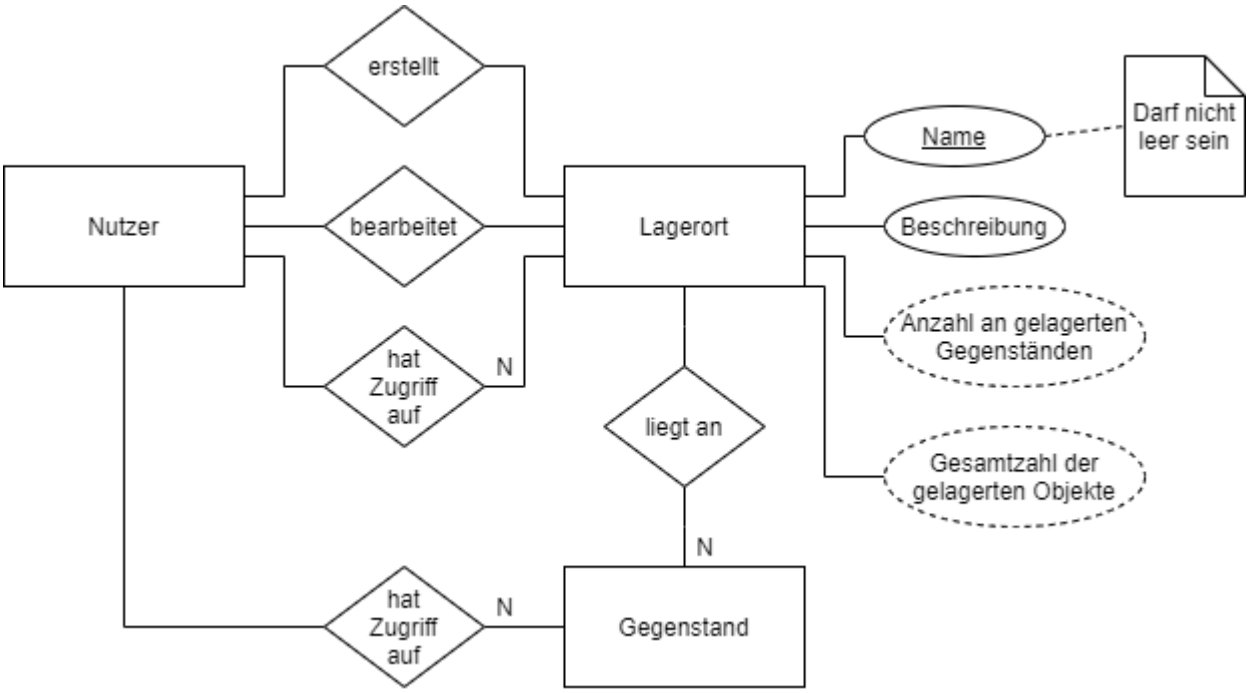

Abbildung 3: ERM Sicht für F-03

5.1.1.4 F-04 Verwaltung von Gegenständen

In F-04 werden die Aktionen, die mit Gegenständen durchführbar sein sollen, beschrieben. Hierbei wurden als Zentrale Abläufe das Anlegen, Bearbeiten und Suchen von Gegenständen aufgeführt. Aus der Analyse der Funktionalen Einheit ging hervor, dass ein Gegenstand immer aus genau einem Gegenstandstyp und genau einem Lagerort besteht. Erstere Bedingung entstammt dem Glossar, in dem ein Gegenstand als „eine Menge von Objekten eines Gegenstandstyp im Lager“ beschrieben wird [1, p. 3]. Hierbei wurde der Quantifizierer „eines Gegenstandstyp“ für ausschlaggebend befunden. Die zweite Bedingung wurde bereits in F-04 diskutiert. Aus diesen beiden Bedingungen geht hervor, dass sich ein Gegenstand eindeutig über die Kombination aus Lagerort und Gegenstandstyp identifizieren lässt, weshalb hierbei der Entitätstyp Gegenstand als schwach modelliert wurde. Als drittes Attribut von Gegenständen wurde die Anzahl genannt, welche immer positiv sein muss und nie „0“ erreichen darf. Letztere Integritätsbedingung geht daraus hervor, dass Gegenstände, sobald ihre Anzahl 0 erreicht, direkt gelöscht werden sollen. In Abbildung 4 wurde der Umstand, dass der Nutzer einen Gegenstandstyp beim Erstellen eines Gegenstandes mit erstellen können soll, durch eine dreistellige Relation ausgedrückt. Diese wird beim Umsetzen in die Anwendung vermutlich mit zwei Aufrufen, erst dem Erstellen des Gegenstandstypen und danach dem Erstellen des Gegenstands mit dem neuen Typ ersetzt. Sie wurde hier allerdings dennoch in dieser Form aufgenommen, da es der Beschreibung der Funktionalen Einheit am nächsten kam.

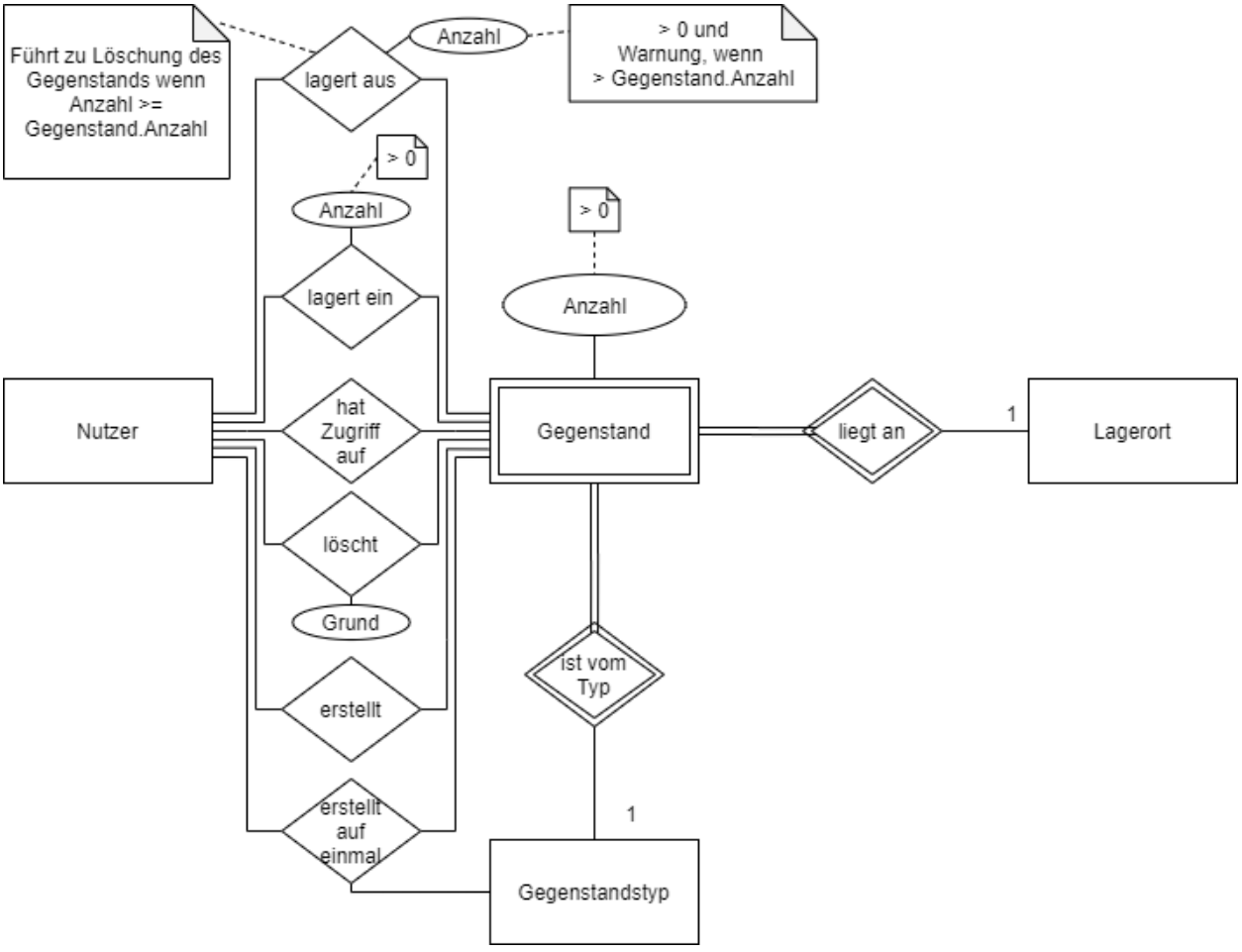

Abbildung 4: ERM-Sicht für F-04

5.1.1.5 F-05 Verwaltung von Nutzern

In F-05 wird das generelle Verwalten von Nutzern beschrieben. Hierbei wird beschrieben, dass ein Nutzer sich einloggen und ausloggen können muss, und so lange eingeloggt bleiben soll, bis er sich manuell ausloggt. Aus diesem Grund muss die Anmeldung des Nutzers in irgendeiner Form persistiert werden. Daher wurde in Abbildung 5 dem Nutzer ein Attribut namens Session-Token mitgegeben, das diese persistente Authentifizierung beschreiben soll. Der Grund, wieso dieses Attribut mit aufgenommen wurde, ist, dass es später nicht vergessen wird. Dazu werden Zugriffsrechte auf Gegenstände, Gegenstandstypen und Lagerorte erwähnt, welche durch Relationen modelliert wurden. Hierbei wurden die Relationen im Positiv-Sinn definiert, in der eigentlichen Implementierung könnte die Relation auch zu „hat keinen Zugriff auf“ invertiert werden. Dies wird in der späteren Konsolidierung der Schemata genauer diskutiert. Im Ursprünglichen Anforderungsdokument wurden noch andere Berechtigungen erwähnt, die Nutzern zu vergeben sein sollen. Da allerdings keine weiteren Angaben gemacht wurden, was solche Berechtigungen sein könnten, wurden sie bei der Modellierung weggelassen. Sie könnten jedoch im Nachhinein als eigenen Entitätstyp mit entsprechender Relation an den Nutzer hinzugefügt werden.

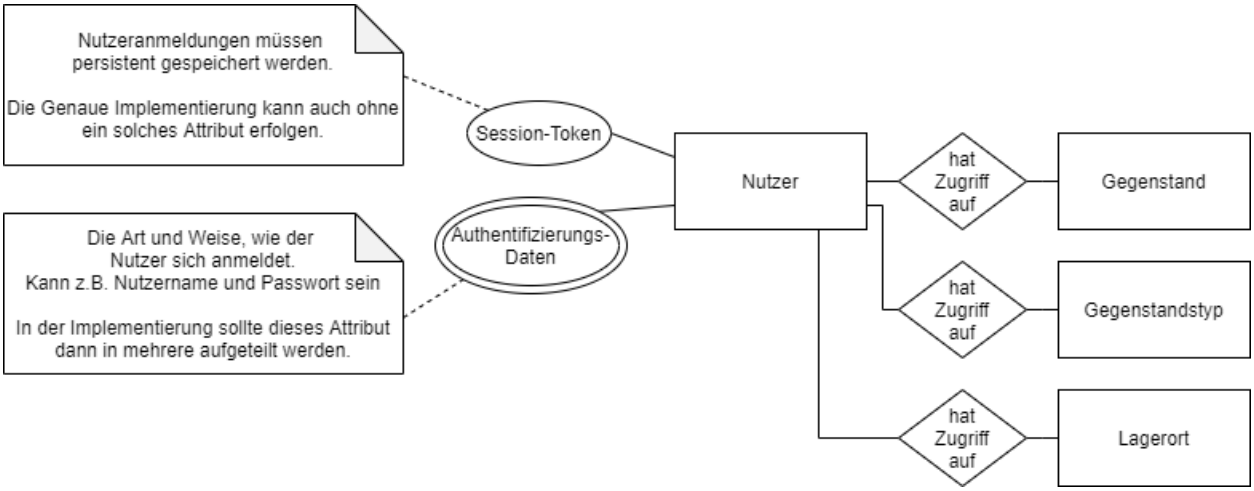

Abbildung 5: ERM-Sicht für F-05

5.1.1.6 F-06 Durchführen einer Inventur

In F-06 wird das Durchführen einer Inventur umrissen. Da eine Inventur als eine sequenzielle Bearbeitung von Gegenständen gesehen werden kann, müssen hierfür keine eigenen Entitätstypen modelliert werden. Um den Zusammenhang zwischen der Inventur und den jeweiligen Objekten wurden diese aber dennoch mit in Abbildung 6 mit modelliert. Diese Entitätstypen und die dazugehörigen Relationen wurden gestrichelt dargestellt, um sie von den zu persistierenden Entitätstypen abzugrenzen.

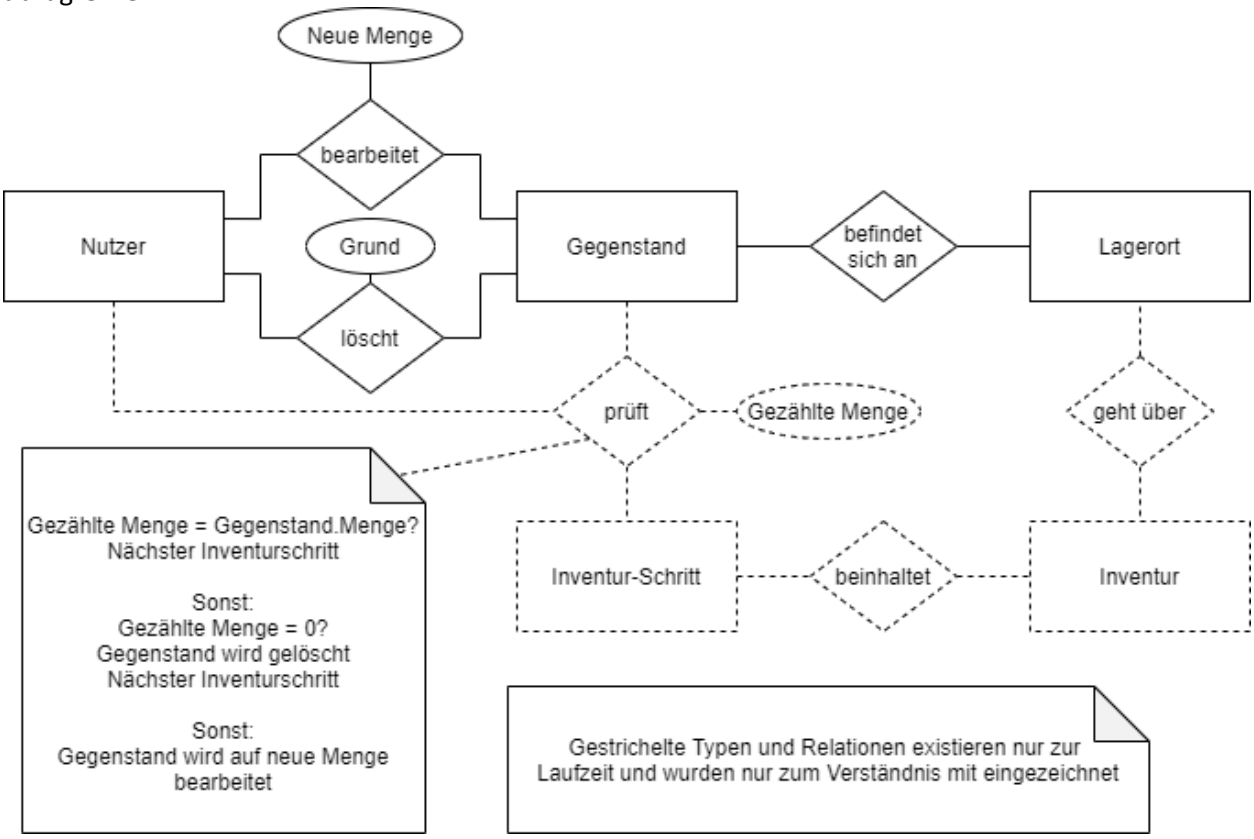

Abbildung 6: ERM-Sicht für F-06

5.1.1.7 Konsolidierung

Das Erste Problem, das beim Zusammenfügen der Modelle aufgetreten ist, ist dass der Entitätstyp Gegenstand in einem Modell als schwacher Entitätstyp modelliert ist, wohingegen er in den anderen als eigenständiger Typ modelliert ist. Die Begründung, wieso der Gegenstand als schwacher Typ modelliert wurde, war, dass jeder Gegenstand abhängig von dem jeweiligen Ort und Gegenstandtyp ist. Diese Information war nur beim Modellieren von F-04 Verwaltung von GegenständenF-04 bekannt, weswegen auch nur hier der Entitätstyp auf diese Weise modelliert wurde. Da auf diese Modellierung mehr Informationen über den Entitätstyp speichert als eine Modellierung als unabhängigen Typ, wurde sich hierbei für den schwachen Typ entschieden.

Für die Darstellung der Nutzerrechte auf Lagerorte, Gegenstände und Gegenstandstypen wurde bereits bei der Modellierung von F-05 angedeutet, dass hierbei die Zugriffsrelation auch invertiert sein könnte. Hierdurch würden die Relationen von „hat Zugriff auf“ zu „hat keinen Zugriff auf“ invertiert werden. Dies entspricht dem Umwandeln einer Whitelist auf eine Blacklist und ist vorteilhaft in Systemen, in denen der Nutzer standardmäßig Zugriff auf sehr viele Objekte hat, und nur sehr wenige oder gar keine explizit gesperrt werden müssen. In einem solchen Fall müsste beim Whitelist-Ansatz für jeden Nutzer die Berechtigung für den Zugang zum jeweiligen Objekt einzeln oder beim Anlegen des Objekts automatisch gegeben werden. Hierdurch resultiert auch ein erhöhter Speicherbedarf der Datenbank, da sehr viele Einträge in die Zugriffstabellen geschrieben werden müssen. Benutzt man stattdessen die invertierte Relation „hat keinen Zugriff auf“ und verfolgt damit den Ansatz einer Blacklist, so müssen nur für die Objekte, die explizit gesperrt werden, Einträge in die Datenbank aufgenommen werden. Analog ist beim Blacklist-Ansatz der Speicherbedarf höher, wenn nur sehr wenige Objekte für Nutzer zugänglich sind.
Da es sich hierbei um eine Inventaranwendung handelt, die auf einem Mobilen Endgerät laufen soll, ist es wahrscheinlicher, dass der Zugang nur für sehr wenige Objekte explizit gesperrt werden muss. Wenn der Nutzer bereits Zugang zu der Anwendung hat, ist davon auszugehen, dass er auch Zugang zum eigentlichen Lager hat und somit auch Zugang zu den meisten dort gelagerten Objekten hat.
Aus diesem Grund wurde sich für die Modellierung einer Blacklist entschieden, weswegen im konsolidierten Entwurf die Relationen-Namen zu „gesperrt für“ geändert wurden. Der Grund, für das „Umdrehen“ der Relation ist, dass „hat keinen Zugang zu“ so interpretiert werden könnte, als habe kein Nutzer Zugang zu irgendwelchen Objekten. Außerdem hält es den Namen der Relation kleiner.

Als nächstes müssen die Kardinalitäten der jeweiligen Relationen bestimmt werden. Hierfür wurde mit Tabelle 6 ein Überblick über die gefundenen Relationen und deren Kardinalitäten gegeben. Hierbei wurden die Beschreibungen der jeweiligen Funktionalen Anforderungen erneut durchgegangen, um zu überprüfen, ob Angaben hierzu noch nicht mit aufgenommen wurden. Für alle anderen Fälle wurde vorerst davon ausgegangen, dass jeweilige Ausprägungen der Entitätstypen mehrfach an der jeweiligen Relation beteiligt sein können. Eine Ausnahme hierzu bilden die Relationen namens „erstellt“ und „löscht“, da davon ausgegangen wurde, dass ein Objekt immer nur von einem Nutzer erstellt wird. Diese Annahme wurde für das Bearbeiten allerdings nicht getroffen. Hierbei wurden die Vererbungsrelationen, die für F-01 modelliert wurden, nicht mit aufgenommen, da sie in Tabelle 7 noch einmal gesondert aufgeführt werden. Ebenso wurde die für F-04 modellierte dreiwertige Relation, die das Anlegen eines Gegenstandstypen zusätzlich zu dem anzulegenden Gegenstand beinhaltet, nicht mit aufgenommen. Dies liegt daran, dass diese Relation in der eigentlichen Anwendung eher sequenziell abgearbeitet würde, auch wenn es für den Nutzer so aussehen kann, also würde sie auf einmal geschehen.
Es wäre möglich gewesen, die Relationen „Lagert ein“, „Lagert aus“ und „[Nutzer] bearbeitet [Gegenstand]“ zu einer gemeinsamen „Bearbeitet“ Relation zusammenzufassen, da sich alle drei durch eine Veränderung der Menge eines Gegenstandes auszeichnen. Da hierbei allerdings ein

Informationsverlust gegeben wäre, wurden die Relationen einzeln gelassen. Gleiches gilt für eine etwaige „Transaktion“ Relation, die das Ein- und Auslagern miteinander verbinden würde. In beiden Fällen wären Abfragen wie „wie viele Gegenstände werden durchschnittlich auf einmal eingelagert“ schwieriger möglich.
Sollten mehrere der erstellten Modelle die gleiche Relation mit unterschiedlichem Namen abbilden, so wurden sie in eine Zeile geschrieben und der verwendete Name fett geschrieben.

Von	**Name**	**Nach**	**Kardinalität**	**Teil von**
Gegenstand	Besitzt / **Ist vom Typ**	Gegenstandstyp	M:1	F-02, F-04
Gegenstand	Ist gesperrt für	Nutzer	N:M	F-02, F-05
Gegenstand	**Liegt an** / Befindet sich an	Lagerort	M:1	F-03, F-04, F-06
Gegenstandstyp	Ist gesperrt für	Nutzer	N:M	F-02, F-04, F-05
Lagerort	Ist gesperrt für	Nutzer	N:M	F-03, F-05
Nutzer	Bearbeitet	Gegenstandstyp	N:M	F-02
Nutzer	Bearbeitet	Lagerort	N:M	F-03
Nutzer	Bearbeitet	Gegenstand	N:M	F-06
Nutzer	Erstellt	Gegenstandstyp	1:M	F-02
Nutzer	Erstellt	Lagerort	1:M	F-03
Nutzer	Erstellt	Gegenstand	1:M	F-04
Nutzer	Lagert aus	Gegenstand	N:M	F-04
Nutzer	Lagert ein	Gegenstand	N:M	F-04
Nutzer	Löscht	Gegenstand	1:M	F-04, F-06

Tabelle 6: Auflistung der Kardinalitäten beim Konsolidieren

Anhand von Tabelle 6 wurde ein vereinfachtes Modell erstellt, in dem die darin aufgeführten Relationen benutzt wurden. Aus Platzgründen wurden hierbei die Attribute der Entitätstypen weggelassen. Ebenso wurden Relationen zwischen denselben Partnern in eine Raute gezeichnet. Aus diesen Gründen entspricht dieses Diagramm nicht der verbreiteten Darstellung von ER Modellen. In diesem Diagramm wurde auch auf das Einzeichnen von Kardinalitäten verzichtet, da Relationen zusammengefasst wurden, deren Kardinalitäten sich unterscheiden.

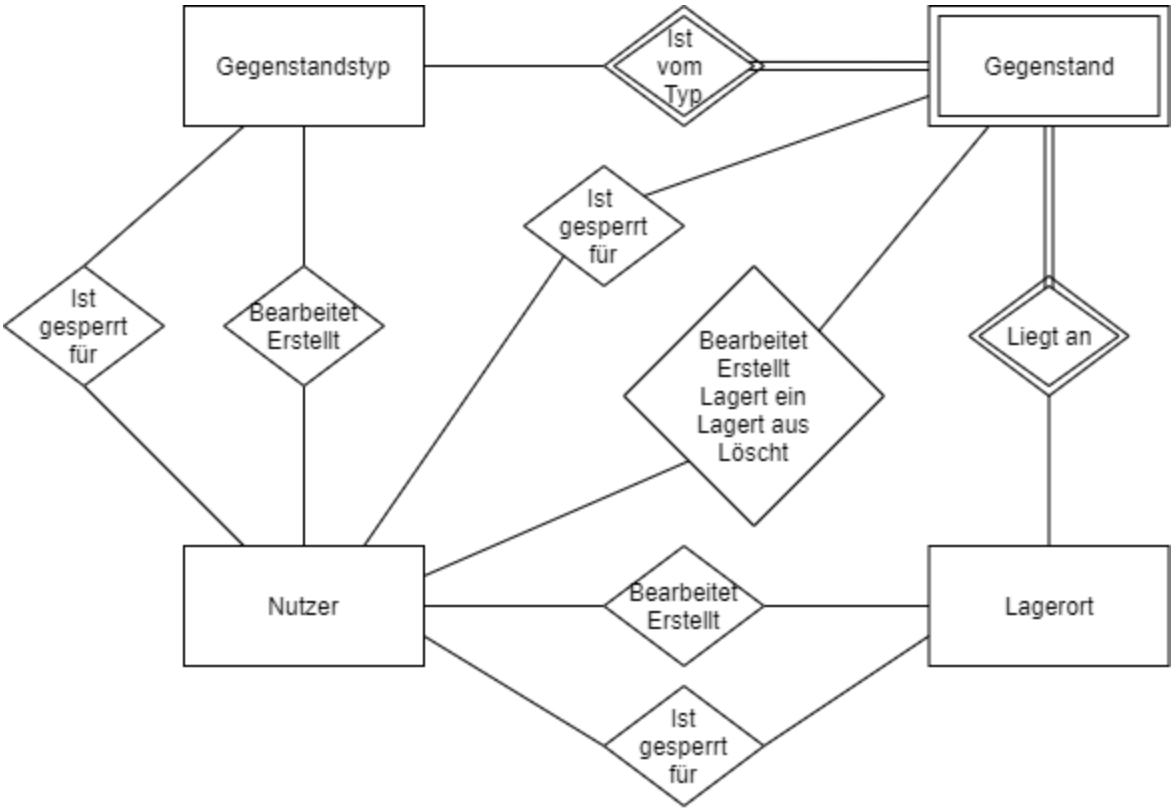

Abbildung 7: Zwischenergebnis Sichten-Konsolidierung

Als nächsten Schritt gilt, das bisher existierende ERM mit dem für F-01 erstellten Model zu vereinen. Hierbei fällt auf, das einige der als Event Entitäten im bisherigen Modell als Relationen modelliert wurden. Um diese Diskrepanz aufzulösen, können diese Binären Relationen zu dreistelligen Relationen erweitert werden, an die die entsprechenden Events mit beteiligt sind. Abbildung 8 demonstriert dies an einer 1:M und einer M:N Beziehung.

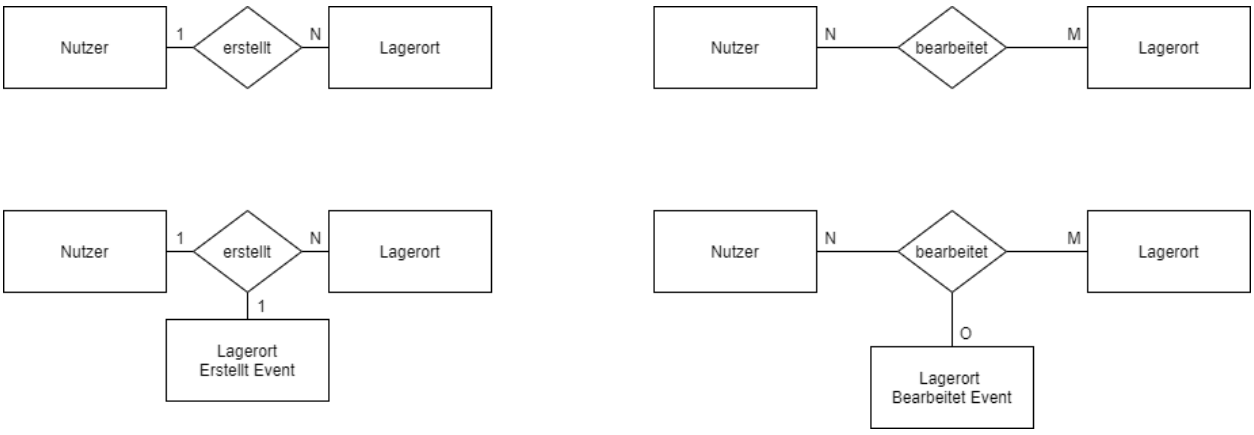

Abbildung 8: Beispiel Konsolidierung Events an 1:N Relation

1:M Beziehungen lassen sich so gut umsetzen, allerdings gilt dasselbe nicht für M:N Beziehungen, da hierbei der Lagerort ebenfalls nicht fixiert wird. Dies lässt sich darin begründen, dass ein Nutzer denselben Lagerort mehrmals bearbeiten kann. Dem lässt sich allerdings dadurch Abhilfe schaffen, dass man den Schlüssel des Events, welches ein Zeitstempel zu dessen Ausführung ist, stattdessen an die

Relation selbst bindet. Hierdurch ist es in der späteren Transformation zu einem Datenbankschema möglich, die Relation und das Event in einer Tabelle zu vereinigen.
Es bleibt also noch zu ermitteln, welche Relationen durch entsprechende Events erweitert werden müssen. Da Events per Definition Meldungen über Zustandsänderungen sind, müssen nur die Relationen berücksichtigt werden, die eine Änderung der Datenbasis bewirken. Daher können die Relationen, die modellieren, wie Gegenstände aufgebaut sind („Liegt an“ und „Ist vom Typ“) sowie die Gesperrt-Relationen hierbei ignoriert werden. Alle anderen Relationen benötigen ein entsprechendes Event. Tabelle 7 listet hierbei alle Relationen und den Namen des hinzuzufügenden Events auf. Hierbei bedeutet ein leerer Name, dass diese Relation kein Event benötigt.

Von	**Name**	**Nach**	**Event Name**
Gegenstand	Ist vom Typ	Gegenstandstyp	
Gegenstand	Ist gesperrt für	Nutzer	
Gegenstand	Liegt an	Lagerort	
Gegenstandstyp	Ist gesperrt für	Nutzer	
Lagerort	Ist gesperrt für	Nutzer	
Nutzer	Bearbeitet	Gegenstandstyp	Gegenstandstyp Bearbeitet Event
Nutzer	Bearbeitet	Lagerort	Lagerort Bearbeitet Event
Nutzer	Bearbeitet	Gegenstand	Gegenstand Bearbeitet Event
Nutzer	Erstellt	Gegenstandstyp	Gegenstand Erstellt Event
Nutzer	Erstellt	Lagerort	Lagerort Erstellt Event
Nutzer	Erstellt	Gegenstand	Gegenstand Erstellt Event
Nutzer	Lagert aus	Gegenstand	Gegenstand Eingelagert Event
Nutzer	Lagert ein	Gegenstand	Gegentand Ausgelagert Event
Nutzer	Löscht	Gegenstand	Gegenstand Gelöscht Event

Tabelle 7: Umzusetzenden Events beim Konsolidieren

Im nächsten Schritt werden die Attribute der einzelnen Entitätstypen aufgelistet und mit Datentypen und Integritätsbedingungen versehen. In Tabelle 8 wurden die bisher aufgeführten Entitätstypen aufgelistet und mit Integritätsbedingungen versehen. Diese wurden anhand der Beschreibungen der funktionalen Einheiten ermittelt. Für die Anmeldedaten des Nutzers wurde von einer Anmeldung mithilfe von Nutzernamen und Passwort ausgegangen, da hierzu keine genaueren Anforderungen vorlagen. Der Nutzername wurde hierbei ebenso als Schlüssel des Entitätstypen gesetzt, da für das Anmelden ein Nutzername im Allgemeinen eindeutig sein sollte. Für die Nutzerattribute „Passwort-Hash“ und „Session Token“ wurden keine Typen mit angegeben. Diese obliegen dem Entwickler.

Entitätstyp	Attributname	Identifizierend	Typ und Integritätsbedingungen
Gegenstand	Menge	Nein	INTEGER; Größer 0
Gegenstandstyp	Name	Nein	VARCHAR; Nicht leer
	Beschreibung	Nein	VARCHAR
	ID	Ja	INTEGER; Nicht leer; Eindeutig
Lagerort	Name	Ja	VARCHAR; Nicht leer; Eindeutig
	Beschreibung	Nein	VARCHAR
Nutzer	Name	Ja	VARCHAR; Nicht leer; Eindeutig
	Passwort Hash	Nein	Nicht spezifiziert; Nicht leer
	Session Token	Nein	Nicht spezifiziert;
Gegenstandstyp Bearbeitet Event	Zeitstempel	Ja	TIMESTAMP; Nicht leer; Eindeutig
	Neuer Name	Nein	VARCHAR; Nicht leer;
	Neue Beschreibung	Nein	VARCHAR;
Lagerort Bearbeitet Event	Zeitstempel	Ja	TIMESTAMP; Nicht leer; Eindeutig
	Neuer Name	Nein	VARCHAR; Nicht leer;
	Neue Beschreibung	Nein	VARCHAR;
Gegenstand Bearbeitet Event	Zeitstempel	Ja	TIMESTAMP; Nicht leer; Eindeutig
	Neue Menge	Nein	INTEGER; Größer 0
Gegenstandstyp Erstellt Event	Zeitstempel	Ja	TIMESTAMP; Nicht leer; Eindeutig
	Name	Nein	VARCHAR; Nicht leer;
	Beschreibung	Nein	VARCHAR;
Lagerort Erstellt Event	Zeitstempel	Ja	TIMESTAMP; Nicht leer; Eindeutig
	Name	Nein	VARCHAR; Nicht leer;
	Beschreibung	Nein	VARCHAR;
Gegenstand Erstellt Event	Zeitstempel	Ja	TIMESTAMP; Nicht leer; Eindeutig
	Menge	Nein	INTEGER; Größer 0
Gegenstand Eingelagert Event	Zeitstempel	Ja	TIMESTAMP; Nicht leer; Eindeutig
	Menge	Nein	INTEGER; Größer 0
Gegenstand Ausgelagert Event	Zeitstempel	Ja	TIMESTAMP; Nicht leer; Eindeutig
	Menge	Nein	INTEGER; Größer 0
Gegenstand Gelöscht Event	Zeitstempel	Ja	TIMESTAMP; Nicht leer; Eindeutig
	Grund	Nein	VARCHAR;

Tabelle 8: Attribute der Entitätstypen

Es fällt auf, dass die Attribute der hier aufgeführten Events sich mit denen der entsprechenden Relationen überschneiden. Dies ist ein später wichtiger Punkt für das Modellieren des Datenbankschemas. Aufgrund dieser Überschneidung wurden in der grafischen Darstellung des ERM die Attribute der Events weggelassen, da sie aus der Relation, der sie zugehörig sind, entnommen werden können. Auf diese Weise wird das Modell übersichtlicher. Da alle Events laut erster Modellierung Spezifizierungen eines Super-Typ „Event“ sind, würde durch das Einzeichnen dieser Relationen das

Modell nicht mehr übersichtlich wirken. Stattdessen wurde sich dafür entschieden, Events farblich zu hinterlegen, um diese Beziehung dadurch implizit auszudrücken. Abbildung 9 zeigt dieses Modell[1].

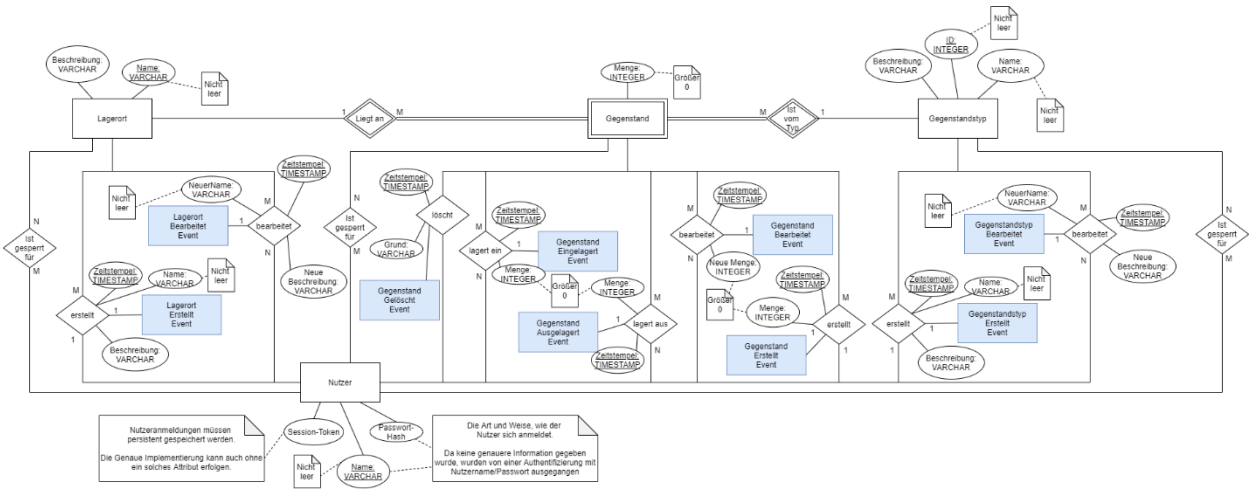

Abbildung 9: Konsolidiertes ERM

In der Modellierung von F-01 wurde davon ausgegangen, dass die Spezialisierungen von Event vollständig sind. Es soll also kein Event geben, das „nur Event“ ist und keine Spezialisierte Form. Da in Abbildung 9 nur die Spezialisierungen von Event benutzt wurden ist die Vollständigkeit der Spezialisierungen für gegeben erachtet. Die Folge für das Datenbankschema ist, dass keine eigene „Event“ Tabelle modelliert werden muss. Des Weiteren wurde davon ausgegangen, dass die Spezialisierungen disjunkt sind. Da hier keine Relationen gefunden wurden, die mehrere Events auslösen, und da kein Event von mehreren Relationen ausgelöst wird, wurden die Spezialisierungen als disjunkt erachtet. Die nicht modellierte Relation „erstellt auf einmal“, bei der ein Nutzer einen Gegenstandstyp und einen Gegenstand auf einmal erstellt, würde hierbei mit einem eigenen Event der Disjunktheit entgegenwirken. Da allerdings bereits beschrieben wurde, dass diese Anforderung durch das sequenzielle Auslösen von zuerst dem Erstellen des Gegenstandstypen gefolgt vom Erstellen des Gegenstandes ausgedrückt werden soll, ist hier auch keine Auswirkung auf die Disjunktheit gegeben. Dies ist auch der Grund, wieso diese Relation nicht in Abbildung 9 zu finden ist. Aufgrund der Disjunktheit und der Vollständigkeit der Spezialisierungen können die Events in voneinander unabhängigen Tabellen gespeichert werden, ohne dass Informationen doppelt gespeichert werden müssen. Möchte man allerdings alle Events abfragen, so muss hierfür ein Join über alle Eventtabellen durchgeführt werden. Dies kann allerdings durch das Nutzen von Sichten vor dem Anwender verborgen werden. Hinzu kommt, dass das Abfragen aller Events nur dann benötigt wird, wenn der Nutzer einen früheren Datenbestand wiederherstellen möchte, was nicht für regelmäßig erachtet wurde.

[1]Für ein besser lesbares Diagramm kann die Originaldatei unter https://raw.githubusercontent.com/TINF18B5/SWE_TINF18B5_Timo_Klenk/master/ArchitekturEntwurf%20Material/ERM_Konsolidierung_3.png eingesehen werden.

5.1.1.8 Beispielhafte Daten

Im Folgenden sollen beispielhafte Daten der Anwendung dargestellt werden. Der Prozess der Synthese der Datenbankstruktur anhand der Relationen wurde hierbei ausgelassen. Die wichtigste Abweichung vom ER Modell ist hierbei, dass Event und dazugehörige Relation als eine Tabelle modelliert wurden, was aufgrund der geteilten Attribute und der eindeutigen Zuordnung dank Schlüsselattribut der Relationen möglich ist.

Als Szenario wurde hierbei gewählt:
Es ist der 13.06.2020: Zur Vereinfachung liegt zwischen jedem Schritt, der die Datenbasis verändert, genau eine Stunde.

1) 9 Uhr: Die Datenbasis startet komplett leer. Sie hat einen Standard-Nutzer „DEFAULT“ mit einem Standard-Passwort
2) Nutzer loggt sich ein
3) 10 Uhr: Nutzer erstellt einen Lagerort namens „Fachboden 1-7“ mit Beschreibung „Kleinteile“
4) 11 Uhr: Nutzer bearbeitet den soeben erstellten Lagerort, und passt die Beschreibung auf „Schrauben und Muttern“ an.
5) 12 Uhr: Nutzer erstellt einen Gegenstandstyp namens „Torx 4x40mm“ mit der Beschreibung „Holzschraube“
6) 13 Uhr: Nutzer erstellt einen Gegenstand aus dem soeben erstellten Gegenstandstyp und dem davor erstellten Lagerort. Er weist dem Gegenstand die Menge 200 zu.
7) 14 Uhr: Nutzer bearbeitet den soeben erzeugten Gegenstand, sodass dessen Menge nun 300 beträgt.
8) 15 Uhr: Nutzer lagert 50 Elemente des soeben erzeugten Gegenstands aus.

Ein Auszug aus der Datenbank könnte nun also wie in Tabelle 9 und Tabelle 10 aussehen. Für eine einfachere Lesbarkeit wurden hier die Zeitstempel nur mit der Uhrzeit dargestellt. In einem echten System würden hier Jahr, Monat, Tag, Stunde, Minute und Sekunde benutzt werden. Es wurden „leere“ Tabellen nicht mit aufgeführt. Die leeren Tabellen wären in diesem Fall die Events für „Gegenstand Gelöscht“, „Gegenstand eingelagert“ und „Gegenstandstyp bearbeitet“.

Nutzer		
Nutzername	**Passwort Hash**	**Session-Token**
DEFAULT	XXXXXXXXXXXX	YYYYYYYYYYYYYYY

Lagerorte	
Name	**Beschreibung**
Fachboden 1-7	Schrauben und Muttern

Gegenstandstypen		
ID	**Name**	**Beschreibung**
1	Torx 4x40mm	Holzschraube

Gegenstände		
Gegenstands ID	**Lagerort Name**	**Menge**
1	Fachboden 1-7	250

Tabelle 9: Beispieldaten Datenbasis

Lagerort Erstellt Events			
Zeitstempel	**Name**	**Beschreibung**	**Nutzer**
10 Uhr	Fachboden 1-7	Kleinteile	DEFAULT

Lagerort Bearbeitet Events				
Zeitstempel	**Name**	**Neuer Name**	**Beschreibung**	**Nutzer**
11 Uhr	Fachboden 1-7	Fachboden 1-7	Schrauben und Muttern	DEFAULT

Gegenstandstyp Erstellt Events				
Zeitstempel	**ID**	**Name**	**Beschreibung**	**Nutzer**
12 Uhr	1	Torx 4x40mm	Holzschraube	DEFAULT

Gegenstand Erstellt Events				
Zeitstempel	**Lagerort**	**Gegenstandstyp**	**Menge**	**Nutzer**
13 Uhr	Fachboden 1-7	1	200	DEFAULT

Gegenstand Bearbeitet Events				
Zeitstempel	**Lagerort**	**Gegenstandstyp**	**Menge**	**Nutzer**
14 Uhr	Fachboden 1-7	1	300	DEFAULT

Gegenstand Ausgelagert Events				
Zeitstempel	**Lagerort**	**Gegenstandstyp**	**Menge**	**Nutzer**
15 Uhr	Fachboden 1-7	1	50	DEFAULT

Tabelle 10: Beispieldaten Eventtabellen

5.1.2 Statische Sicht: Daten zur Laufzeit

Zusätzlich zu den zu persistierenden Daten existieren zur Laufzeit noch weitere Strukturen, die beispielsweise für die Anbindung an die Datenbank sowie das Nutzer Interface zuständig sind. Des Weiteren müssen hier noch die Datentypen für die Inventur mit modelliert werden. Hierbei kann die Struktur, dir für das ER-Modell für F-06 bereits ermittelt wurde, wiederverwendet werden. Es wurde eine Inventur als eine Abarbeitung von vielen Inventurschritten verstanden, wobei jeder dieser Schritte einen Gegenstand prüft. Es bietet sich hierbei also das Nutzen eines Iterator-Patterns an. Hierbei wird die Inventur als eine Sammlung an Inventurschritten verstanden, die solange sequenziell abgearbeitet werden, wie es noch weitere Schritte gibt [21]. Das Pattern wird hierbei nicht in Reinform benutzt, das Iterator und Aggregat in der Inventur zusammenfallen. Allerdings lässt sich auch argumentieren, dass das Inventur-Objekt eine Möglichkeit bietet, über Gegenstände zu iterieren, dann wäre das Pattern erfüllt.
Da sich für die Bibliothek Akka entschieden wurde, existieren zur Laufzeit also auch Konstrukte dieser Bibliothek, beispielsweise der Event Stream. Akka ist eine Verbindung von Message Driven und Event Driven Architecture, das bedeutet, sie erlaubt sowohl das gerichtete Senden von Nachrichten als auch das ungerichtete Senden von Events. Dies führt in der Praxis zu einer Unterscheidung zwischen Events und Kommandos. Kommandos sind hierbei Änderungsanfragen an ein System, wohingegen Events die Quittierung bereits vollzogener Änderungen darstellen. Events sind dadurch unveränderbar, wohingegen Kommandos abgelehnt werden können [22].
Wird ein solcher Ansatz umgesetzt, müssen zusätzlich zu den Events also auch noch Kommandos hinzugefügt werden, die zur Laufzeit für Anfragen des Nutzers stehen. So kann eine Nutzeranfrage für das Erstellen eines Lagerortes in mehrere Phasen aufgeteilt werden:

1) Nutzer gibt Daten ein und bestätigt -> Ein Kommando „Erstelle Gegenstand“ wird erzeugt und gerichtet an eine Stelle in der Anwendung geschickt
2) Das Kommando wird von dieser Stelle empfangen und an den entsprechenden Prozessor weitergeleitet
3) Der Kommandoprozessor validiert die Eingaben
4) Der Kommandoprozessor führt die Änderungen aus
5) Der Kommandoprozessor sendet ein das die Änderung quittiert.
6) Nutzerinterface Komponente empfängt das versendete Event, und signalisiert dem Nutzer, wie er weiter machen kann.
7) Eine andere Komponente (oder der Kommandoprozessor selbst) speichert das Event in der Datenbank.

Eine solche Trennung von Kommandos und Events führt auch dazu, dass die Chance, Fehlerhafte Events in der Datenbank zu haben, minimiert wird, da Eingaben zuerst verifiziert werden müssen, bevor Events überhaupt entstehen. Durch das Nutzen dieser Aufteilung muss das bisher etablierte Datenmodell für die zu persistierenden Daten nicht angepasst werden, da Events als Quittierungen, was bereits geändert wurde, ausreichen, um den Zustand der Datenbasis wiederherzustellen. Es ist hierbei nicht notwendig, alle (eventuell auch abgelehnten) Kommandos zu wiederholen.
In der oberen Aufzählung wird abstrakt von einem „Kommandoprozessor“ gesprochen. Dieser ist die Komponente, die das entsprechende Kommando ausführt. Es kann sich hierbei um eine Komponente handeln, die für alle Kommandos zuständig ist, oder aber um Kontextabhängige Prozessoren, wie einen je Funktionaler Einheit. Letzterer Ansatz wurde hier gewählt, da hierbei eine bessere Modularität gewährleistet wird. Theoretisch wäre es ebenso gut möglich, einen Prozessor je Kommando einzuführen, dies würde allerdings zu einer Klassenexplosion führen, und ist in diesem kleinen Kontext

noch nicht notwendig. Eine solche Aufteilung ist auch in späteren Entwicklungszyklen noch einfach möglich.

Ein grober Überblick über die Klassen lässt sich in Abbildung 10 und Abbildung 11 finden. Hierbei behandelt letztere die durch das Nutzen von Kommandos entstehenden Klassen. Die Klassen, die für das Darstellen des Nutzer-Interfaces zuständig sind, wurden hierbei nicht mit aufgenommen, da hierfür weitere Recherchen notwendig gewesen wären.

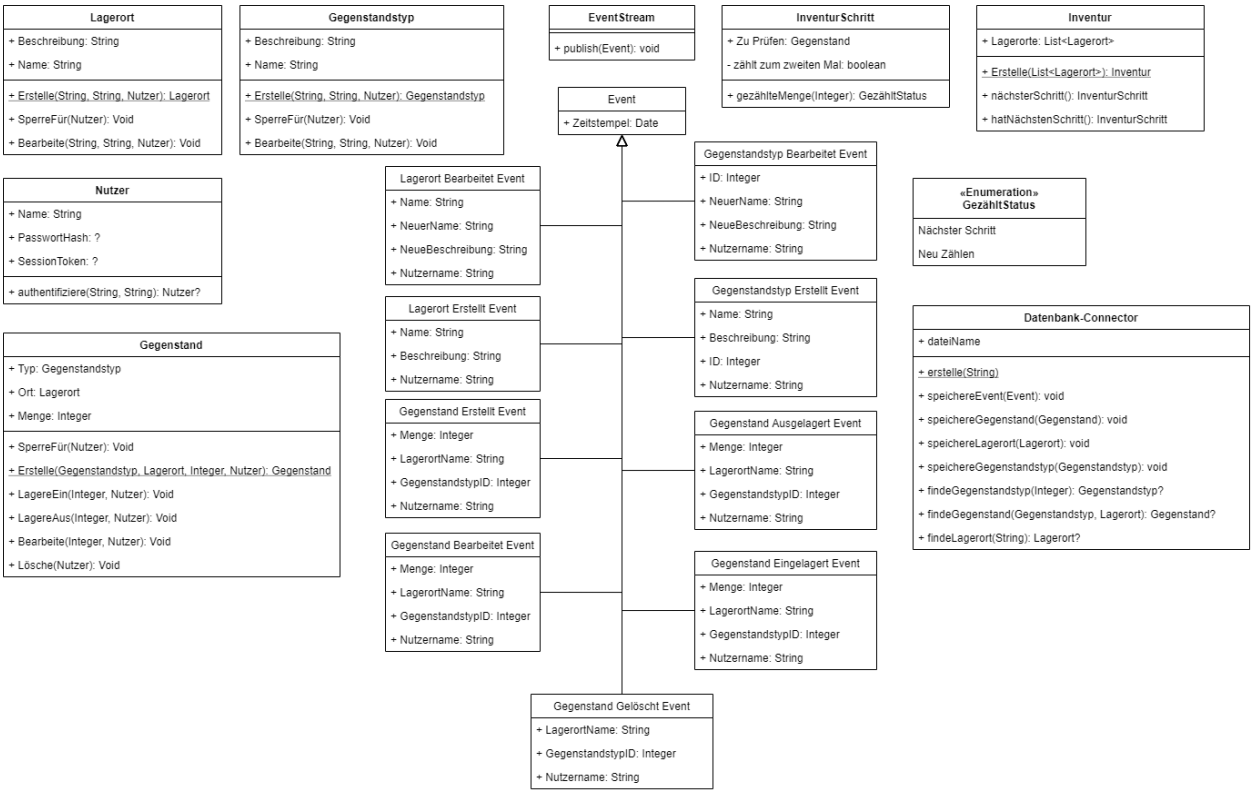

Abbildung 10: Grober Überblick über die Klassen zur Laufzeit, Bild. 1

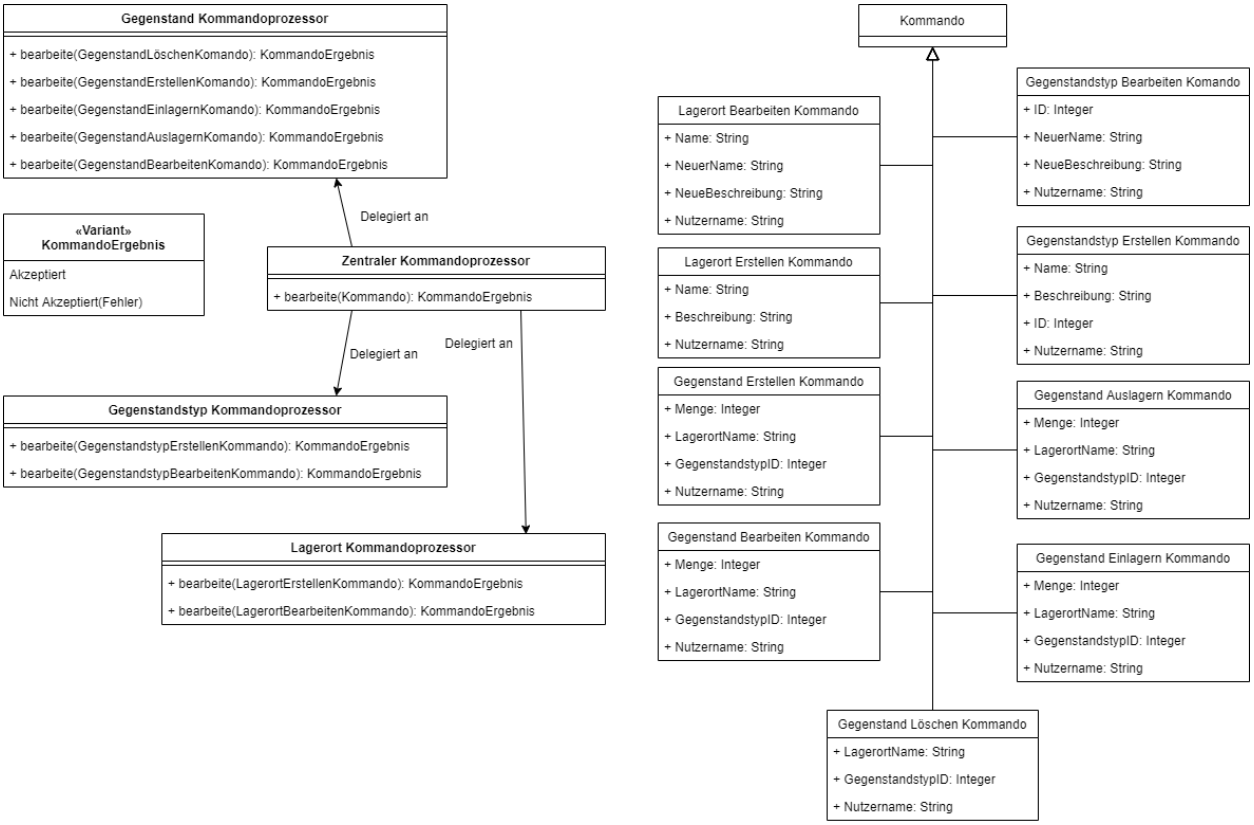

Abbildung 11: Grober Überblick über die Klassen zur Laufzeit, Bild. 2

5.2 Dynamische Sichten

Unter den dynamischen Sichten werden die Abläufe innerhalb der Anwendung modelliert. Hierbei werden zentrale Prozesse erneut vorgestellt und grafisch aufbereitet. Hierfür wird sich an den bereits in Kapitel 3 erhobenen funktionalen Einheiten orientiert.

5.2.1 Prozesse innerhalb von F-01

F-01 beschreibt das Protokollieren sowie das revisionssichere Speichern von Datenbestandsänderungen. Hierfür wurde die Revisionssicherheit mithilfe des Speicherns von sowohl den Datenbestand als auch der Events, die zu diesem Bestand führen umgesetzt. Wie bereits bei der Statischen Sicht erläutert, sendet der Nutzer die Daten anhand von Kommandos, die dann von einem Kommandoprozessor validiert und ausgeführt werden. In Abbildung 12 wird ein solcher Vorgang abstrakt dargestellt. Hierbei wurden die Aufgaben des Kommandoprozessors aufgeschlüsselt. In einer späteren Implementierung kann es mehr als einen Kommandoprozessor geben, wie beispielsweise in Abbildung 11 modelliert wurde. Da in der bisherigen Modellierung allerdings ein Zentraler Kommandoprozessor angesprochen wird, der selbst an die anderen delegiert, ist der Umstand, dass mehrere Prozessoren existieren für den Sender des Kommandos transparent, weswegen die untere Darstellung ihre Gültigkeit behält. Sind die Daten valide, so werden die Änderungen ausgeführt und aus der Nachricht ein Event erstellt, das an mehrere Event Handler geschickt wird. Der dargestellte „Event Handler 2“ soll hierbei demonstrieren, dass Events im Gegensatz zu Nachrichten ungerichtet gesendet werden und somit von mehreren unabhängigen Instanzen bearbeitet werden können. In späteren Ausführungen der Anwendung könnte hier beispielsweise ein weiterer Event Handler hinzugefügt werden, der nur Events vom Typ „Gegentand ausgelagert Event“ empfängt, und eine Tagesabrechnung aller Auslagerungen verwaltet.

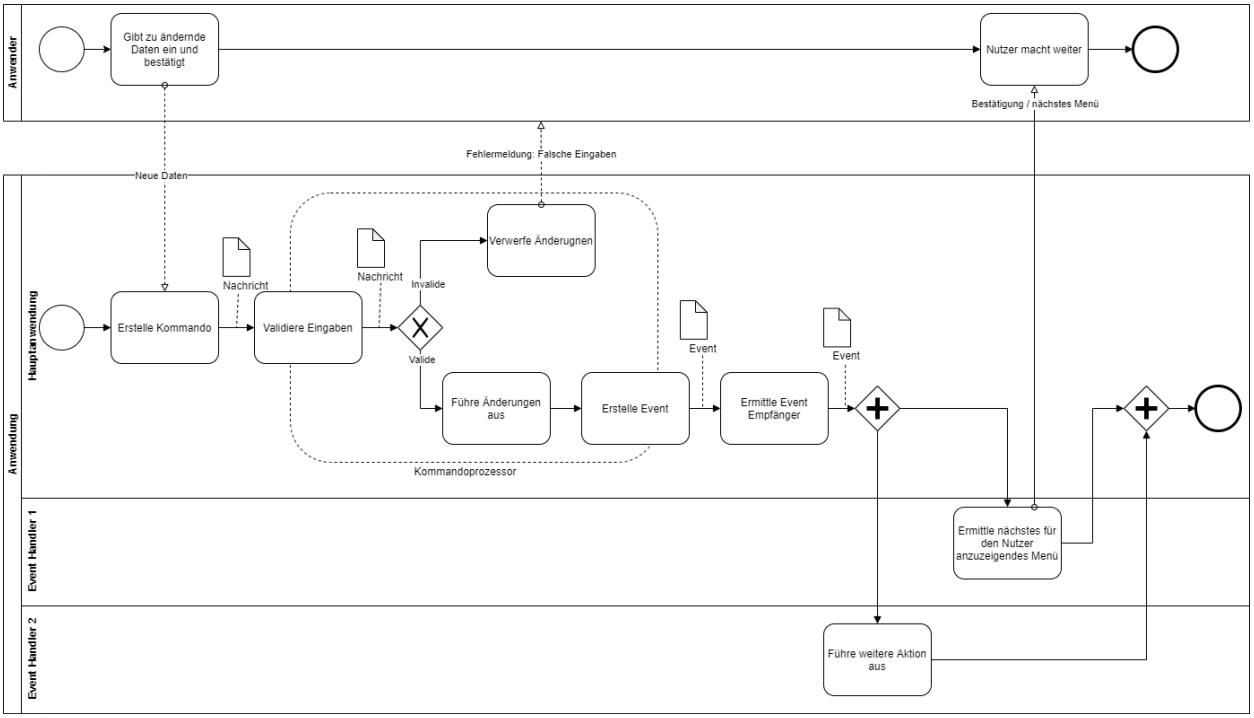

Abbildung 12: Prozessmodellierung Nachrichtenbasierter Datenänderungen

6 Zusammenfassung

In diesem Architekturdokument wurde eine mögliche Architektur entwickelt, die für das Umsetzen der Anforderungen des gegebenen Anforderungsdokumentes geeignet sein soll. Da dieses Dokument von einer Person im Rahmen einer Lehrveranstaltung erstellt wurde, ist davon auszugehen, dass nicht alle gängigen Technologien in ausreichender Tiefe betrachtet wurden. Bei offenen Fragen wurden hierbei auch (dokumentierte) Annahmen getroffen, was in einem echten Projekt durch eine Kommunikation mit dem Auftraggeber gelöst werden sollte.

Die entwickelte Architektur wurde mit dem Blick auf eine Erweiterbarkeit der Anwendung entwickelt, was einen großen Einfluss auf die Wahl der Datenspeicherung sowie das Zusammenspiel der einzelnen Komponenten hatte. Hierbei wurde sich für ein stark asynchrones System entschieden, was eine spätere Transition von Teilen der Anwendung auf beispielsweise einen Anwendungsserver vereinfachen soll. Aber auch Erweiterungen innerhalb der Anwendung sollen durch dieses System einfacher umsetzbar sein, da die Komponenten eine lose Kopplung aufweisen. Es wurde bei der Auswahl der Technologien für quelloffene und kostenfrei kommerziell nutzbare Technologien entschieden, wodurch eine Abhängigkeit von Drittanbietern minimiert wird.

7 Referenzen

[1] T. Klenk, „Inventar-Anwendung - Anforderungsdokument von Timo Klenk," DHBW, KA, 2019.

[2] StatCounter, „Mobile & Tablet Operating System Market Share Germany," 2020. [Online]. Available: https://gs.statcounter.com/os-market-share/mobile-tablet/germany#monthly-201805-202005. [Zugriff am 06 06 2020].

[3] Google, „Meet Android Studio," [Online]. Available: https://developer.android.com/studio/intro. [Zugriff am 07 06 2020].

[4] Microsoft, „Xamarin | Open-ource mobile app platform for .NET," Microsoft, [Online]. Available: https://dotnet.microsoft.com/apps/xamarin. [Zugriff am 07 06 2020].

[5] Google, „Flutter - Beautiful Native apps in record time," [Online]. Available: https://flutter.dev/. [Zugriff am 07 06 2020].

[6] M. Fowler, „Event Sourcing," 12 12 2005. [Online]. Available: https://martinfowler.com/eaaDev/EventSourcing.html. [Zugriff am 10 06 2020].

[7] E. Damtoft, „Event Sourcing and the History of Accounting," 26 07 2019. [Online]. Available: https://dev.to/dealeron/event-sourcing-and-the-history-of-accounting-1aah. [Zugriff am 10 06 2020].

[8] DeviceAtlas, „Most common smartphone RAM by country," DeviceAtlas, 02 09 2019. [Online]. Available: https://deviceatlas.com/blog/most-common-smartphone-ram-by-country. [Zugriff am 10 06 2020].

[9] sqlite.org, „About SQLite," [Online]. Available: https://www.sqlite.org/about.html. [Zugriff am 08 06 2020].

[10] Google, „Save data in a local database using Room," [Online]. Available: https://developer.android.com/training/data-storage/room. [Zugriff am 08 06 2020].

[11] A. Kemper und A. Eickler, Datenbanksysteme, 9 Hrsg., München: Oldenbourg Verlag, 2013.

[12] Wikipedia, „Object-relational impedance mismatch," 28 12 2019. [Online]. Available: https://en.wikipedia.org/wiki/Object-relational_impedance_mismatch. [Zugriff am 11 06 2020].

[13] Objectbox Limited, „objectbox/objectbox-java," Objectbox Limited, 2020. [Online]. Available: https://github.com/objectbox/objectbox-java. [Zugriff am 11 06 2020].

[14] T. Klenk, „SWE TINF18B5 Entscheidungsfindung," 11 06 2020. [Online]. Available: https://github.com/TINF18B5/SWE_TINF18B5_Timo_Klenk_Entscheidungsfindungen. [Zugriff am 11 06 2020].

[15] McObject, „Perst Open Source Object Oriented Database Specifications," McObject, 2020. [Online]. Available: https://www.mcobject.com/perst_database_spec/. [Zugriff am 11 06 2020].

[16] MongoDB, Inc, „NoSQL Databases Explained," 2020. [Online]. Available: https://www.mongodb.com/nosql-explained. [Zugriff am 08 06 2020].

[17] solid IT gmbh, „DB-Engines Ranking," solid IT gmbh, 01 06 2020. [Online]. Available: https://db-engines.com/en/ranking. [Zugriff am 11 06 2020].

[18] G. Hohpe, „Programming without a Call Stack - Event-driven Architectures," 2006. [Online]. Available: https://www.enterpriseintegrationpatterns.com/docs/EDA.pdf. [Zugriff am 11 06 2020].

[19] J. Bonér, D. Farley, R. Kuhn und M. Thompson, „Glossar - Das Reaktive Manifest," 16 09 2014. [Online]. Available: https://www.reactivemanifesto.org/de/glossary#Message-Driven. [Zugriff am 11 06 2020].

[20] Lightbend, Inc., „Akka: build concurrent, distributed, and resilient message-driven applications for Java and Scala | Akka,“ Lightbend, Inc., 2020. [Online]. Available: https://akka.io. [Zugriff am 12 06 2020].

[21] oodesign, „Iterator Pattern,“ [Online]. Available: https://www.oodesign.com/iterator-pattern.html. [Zugriff am 13 06 2020].

[22] Edument AB, „Commands and Events FSQ,“ Edument AB, 2018. [Online]. Available: http://www.cqrs.nu/Faq/commands-and-events. [Zugriff am 13 06 2020].