DHBW
Duale Hochschule
Baden-Württemberg

Architekturdokument für den Programmentwurf

Native Android – App für persönliche Fitnessassistenz mit individuellen Plänen und Zielen

Stand:	14.06.2020
Autor:	Kai Müller
Kurs:	TINF18B5

Version	Autor	Änderungsvermerk	Zu Anforderungsdokument Version
1.0	K. Müller	Initiale Erstellung	1.0 – 02.12.2019

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Glossar

Tabelle 1 - Glossar

Wort	Bedeutung
Designpattern	Bewährte Lösungsmuster für wiederkehrende Entwurfsprobleme

Abkürzungsverzeichnis

CRUD	Create, Read, Update, Delete
DAO	Data Access Object
SSH	Secure Shell
ERM	Entity-Relationshop-Modell
BMI	Body-Mass-Index

1. Einleitung

Auf dem aktuellen App Markt gibt es derzeit keine App, welche sowohl die körperliche Fitness in Form von Kraft- und Ausdauertraining überwacht sowie gleichzeitig die Ernährung überwacht. Aktuell benötigt man für diesen Sachverhalt mindestens zwei Apps, was für die Endnutzer eine höhere Umständlichkeit bedeutet.

Die Idee der App ist diese Punkte kompakt in einer Android App darzustellen. Hierbei gilt es zu ermitteln welche Art von Plänen für die Endnutzer eine Rolle spielen. In Frage könnten hier jeweils verschiedene Krafttrainingspläne oder Ernährungspläne kommen. Eine hohe Wichtigkeit stellt ebenso die Erkennung weiterer Features dar, die potenzielle Endnutzer in der kompakten Fitnessapp als relevant sehen. Diese Features gilt es dann abzuwägen.

Der Endnutzer der App soll durch die Verwendung der Fitnessapp einen gesünderen Lebensstil führen. Die App soll einfach bedienbar sein, damit auch während des Trainings ohne eine hohe Anzahl an Klicks in der App zwischen Hauptaspekte navigiert werden kann. Alle Hauptaspekte der App sollen somit schnell und mit wenigen Klicks erreicht werden können.

In diesem Dokument wird die Architektur der Fitnessapp beschrieben. Es wird auf sehr granulare Themen, wie Datentypen, aber auch auf grob granularen Themen wie die systemweite Architektur eingegangen. Das Ziel dieses Dokuments ist es eine Architektur zu beschreiben, die eine schnelle und möglichst fehlerfreie Entwicklung gewährleistet.

2. Architekturstrategie

In diesem Kapitel wird genauer auf die Strategie der Architektur eingegangen. Das Hauptziel stellt hierbei eine hohe Testabdeckung, welche über automatisierte Tests erzielt wird, dar. Auf die Funktion sowie die Implementierung der automatisierten Tests wird in Kapitel 6 genauer eingegangen.
In den folgenden Kapitel wird zunächst auf die Architekturstrategie innerhalb des Programmcodes und danach auf die Architekturstrategie mit Bezug auf das komplette Projekt eingegangen.

2.1. Progammatische Architekturstrategie

Ein hoher Stellenwert in der Applikation genießt die Testbarkeit. Um eine hohe Testabdeckung gewährleisten zu können müssen einzelne Teile des Projektes, unteranderem mithilfe von Entwurfsmuster, separiert werden. Die leitende Architekturstrategie stellt die leichte Erweiterbar- sowie Testbarkeit dar.
Die genaue Implementierung der einzelnen Entwurfsmuster wird im UML-Klassendiagramm in Kapitel 4.1 visualisiert.

2.1.1. Humble Object - Pattern

In diesem Projekt wird ein Backend sowie ein Frontend entwickelt. Frontendtests gelten als schwer durchführbar, da schlecht automatisiert getestet werden kann ob ein bestimmtes Fenster im Frontend sich richtig verhaltet. Um dieses Problem zu minimieren wird in diesem Projekt das Humble Object - Pattern verwendet.
Hierbei wird Logik aus Teilen des Projektes, welche schwer zu testen sind (z.B. Frontend), entzogen. Diese Teile stellen die „Humble Objects“ dar. Also Objekte die nicht (gut) getestet werden können. Die Logik wird in separate Klassen ausgelagert, um unteranderem eine höhere Testbarkeit zu gewährleisten.
Durch die Verwendung des Humble Object – Pattern wird also die Testbarkeit, beziehungsweise die Testabdeckung, des Projektes erhöht.

2.1.2. Data Access Object - Pattern

Um die leichte Erweiterbarkeit des Projektes zu gewährleisten wird unteranderem das „Data Access Object“ Pattern implementiert. Hierbei wird für jede Datenbasis eine eigene Zugriffsklasse implementiert. Diese Zugriffklassen implementieren ein gemeinsames Interface. Somit kann leicht eine Datenbankart ausgetauscht oder eine neue Datenbasis

hinzugefügt werden. Bestehender Code muss nicht geändert werden. Das DAO-Pattern unterstützt gleichzeitig die Implementierung einer 3-Schicht Architektur. Hierbei wird das Projekt in die Schichten „Data-Access Layer“, „Business Layer“ und „Presentation Layer“ unterteilt. Auf die 3-Schicht-Architektur wird in Kapitel 2.2. eingegangen.

2.1.3. Singleton - Pattern

Als weiteres Design-Pattern wird das „Singleton-Pattern“ verwendet. Hierbei wird sichergestellt, dass maximal ein Objekt einer bestimmten Klasse zur Laufzeit existiert. Das Singleton-Pattern wird auf die Datenbank Verbindung angewendet, damit keine Möglichkeit besteht mehrere gleichzeitige Datenbankverbindungen aufzubauen. Dies steigert unteranderem die Performance der Applikation.

Es ist zu beachten, dass das Singleton Pattern oft als „Anti-Pattern“ bezeichnet wird, da die Testbarkeit der Applikation verringert wird. Das Singleton-Pattern wird jedoch in diesem Projekt nur für die Datenbankverbindung verwendet. Die Verbindung zur Datenbank wird in den automatisierten Tests jedoch durch Mocks weggekapselt und ist dadurch für die Tests irrelevant. Somit ist die Verwendung des Singleton – Patterns legitim.

2.1.4. Factory – Pattern

Um leichte Erweiterung der App zu garantieren, wurde sich für die Verwendung des Factory-Patterns entschieden. Durch die Verwendung besteht die Möglichkeit neue Speicherarten wie Cloudspeicherung später einzufügen. Das Factory-Pattern wird zusammen mit dem DAO-Pattern implementiert. Das Konzept des DAO Factory-Pattern ist in Abbildung 4 dargestellt.

2.2. Systemweite Architekturstrategie

In den letzten Jahren haben sich mehrere Systemarchitekturen etabliert. In nachstehenden Tabellen sind drei Systemarchitekturen mit Vor- und Nachteilen aufgeführt. Die Hauptanforderung an eine Systemarchitektur stellt ganz klar in diesem Projekt die Testbarkeit des Codes dar.

Tabelle 2 - 3-Layer-Architecture

Name	Aufbau	Vorteile	Nachteile
3-Layer-Architecture	• Data-Access-Layer: Zugriff auf Daten • Business-Layer: Verarbeitung von Daten, Anwendung von Logik • Client-Layer: Darstellung von Daten	• Horizontale Skalierung • Klare Kapselung der Logik • Hohe Testbarkeit • Steile Lernkurve	• Erhöhte Latenz

Tabelle 3 - Data-Context-Architecture

Name	Aufbau	Vorteile	Nachteile
Data-Context-Interaction-Architecture	• Data-Part: Domänenobjekte wie Fitnessübung, simpel gehalten, keine Methoden für Use-Cases • Interactions: Beinhaltet „Roles“, welche dynamisch die Daten erweitern. „Roles“ stellen die Implementierung von Use-Cases dar • Context: Abruf von Daten und Zuweisung von Rollen	• Anwendbarkeit auf Use-Cases durch Benutzerinteraktion • Hoher Entkopplungsgrad	• Sehr komplex • Keine Architektur für einfache CRUD-Anweisungen • Benötigt Frameworks für die meisten Programmiersprachen

Tabelle 4 - Hexagonal Architecture

Name	Aufbau	Vorteile	Nachteile
Hexagonal Architecture	• User-Side: Beinhaltet Frontend für Benutzer, Beinhaltet Code zum Interagieren • Businesslogic: Beinhaltet die komplette Logik • Server-Side: Beinhaltet Code für Zugriff auf Daten, wie Datenbank, Dateien, etc… • Unterschied zu 3-Layer-Architecture: Skalierung mithilfe von Adapter die an die Businesslogic „angeschlossen“ werden. Z.B. CLI, Frontend, Datenbank, Webservice	• Leichtes Anbinden / Entfernen von Modulen	• Flache Lernkurve, durch hohe Einarbeitungszeit

2.2.1. Entscheidung Systemarchitektur

Systemarchitekturen unterscheiden sich oft im Detail, haben aber meist das gleiche Ziel: die Separierung der Anwendung. Um das Ziel zu erreichen besteht jede Architektur aus unterschiedlichen Sichten. Alle genannten Architekturen können mehr oder weniger auf das Projekt angewendet, deshalb gilt es die Vor- und Nachteile der einzelnen Architekturen auszuwerten.

Zur Entscheidung wurde die nachstehende Entscheidungsmatrix verwendet.

Tabelle 5 - Kriterien Entscheidungsmatrix

Kriterium ID	Kriterium	Bewertungsskala	Gewichtung
KR-1	Kompatibilität mit Anwendungsfall	Nominalskala, Ja & Nein	K.O.
KR-2	Geschätzte Einarbeitungszeit	Intervallskala (subjektiv) gering, mittel, hoch	1
KR-3	Geschätzte Komplexität	Intervallskala (subjektiv) gering, mittel, hoch	3
KR-4	Umsetzung ohne extra Framework	Nominalskala, Ja & Nein	2
KR-5	Geschlossen für Erweiterungen	Intervallskala (objektiv) gering, mittel, hoch	2

Tabelle 5 zeigt die einzelnen Kriterien mit ihrer Gewichtung. Hierbei wurde die Kompatibilität als K.O. Kriterium festgelegt, da eine Systemarchitektur, die schlecht zu dem Anwendungsfall passt schlichtweg keinen großen Mehrwert für das Projekt darstellt. Die dreifache Gewichtung hat das Kriterium „Komplexität“. Da es sich bei diesem Projekt um ein vergleichsweises kleines Projekt handelt soll die Architektur schnell umsetzbar sein und dementsprechend erhielt die Komplexität die hohe Gewichtung. Eine zweifache Gewichtung erhielt das Kriterium „Umsetzung ohne extra Framework“, da dies für die Entwickler mit zusätzlichem Aufwand und somit auch mit zusätzlichen Kosten verbunden ist. Da eine gewisse Einarbeitung im Beruf des Entwicklers normal ist, wird die Gewichtung der Einarbeitungszeit auf 1 gesetzt. Kriterium KR-5 beschreibt die Geschlossenheit, bzw. Offenheit für Erweiterungen. Hoch stellt in diesem Kriterium den

schlechteren Wert dar. Dieses Kriterium wurde mit einer Gewichtung von 2 geplant. Dies ist zurückzuführen, dass es für ein Projekt wichtig ist, dass Erweiterungen ohne großen Aufwand durchgeführt werden können.

Die Kriterien sind in folgender Entscheidungsmatrix auf die 3 Architekturtypen angewandt. Der Wert „Hoch“ wird in der Rechnung mit dem Wert 1 eingefügt. „Gering“ entspricht dem Wert 3. Der Wert „Ja“ entspricht in der Endrechnung dem Wert 1. „Nein“ entspricht dem Wert 0.

Tabelle 6 - Entscheidungsmatrix Systemarchitektur

	3-Layer-Architecture	Data-Context-Interaction Architecture	Hexagonal Architecture
KR-1	Ja	Nein	Ja
KR-2	Gering	Mittel	Hoch
KR-3	Mittel	Hoch	Hoch
KR-4	Ja	Nein	Ja
KR-5	Mittel	Mittel	Gering

Nach Evaluierung der K.O. Kriterien wird deutlich, dass sich die Data-Context-Interaction Architecture aufgrund mangelnder Kompatibilität mit dem Anwendungsfall nicht eignet. In folgender Matrix werden die zwei übrigen Architekturen mit Rechnung genauer betrachtet, um eine Entscheidung zu fällen.

Tabelle 7 - Entscheidungsmatrix Ergebnis

	3-Layer-Architecture	Hexagonal Architecture
KR-2	Gering	Hoch
KR-3	Mittel	Hoch
KR-4	Ja	Ja
KR-5	Mittel	Gering
Rechnung	3*1 + 2*3 + 1*2 + 2*2	1*1 + 1*3 + 1*2 + 3*2
Ergebnis	15	10

Nach Auswertung der Tabelle 7 wird klar, dass die Entscheidung für die Architektur auf die 3-Layer-Architecture fällt. Dies ist zurückzuführen auf die besseren Werte in KR-2 und KR-3.

Die Hexagonale-Architektur stellt eine starke Alternative gegenüber der 3-Layer-Architecture dar. Ein Risiko, welches sich durch die Entscheidung gegen die Hexagonal Architecture ergibt, ist, dass keine hohe Offenheit gegenüber Erweiterungen herrscht. Somit ist eine Erweiterung durch eine Komponente, wie zum Beispiel ein Export der Daten an eine externe Homepage, mit höherem Aufwand verbunden. Der Grad der Sicherheit, der mit der 3-Layer-Architecture erreicht wird, ist jedoch als hoch anzusehen, da das Projekt mit hoher Sicherheit in der gewählten Architektur umsetzbar ist.

Nachstehend wird die 3-Layer-Architecture visuell dargestellt.

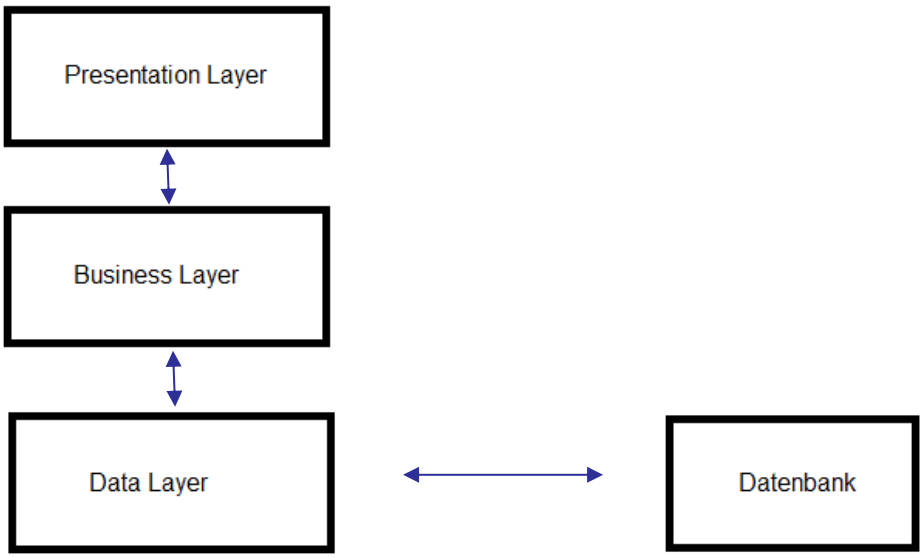

Abbildung 1 - 3-Layer-Architecture (Eigene Darstellung)

3. Statische Sicht – zu persistierende Daten

Die zu persistierende Daten sind in sehr strukturierter Form. Des Weiteren ist die Datenmenge der zu persistierenden Daten vergleichsweise klein. Aufgrund der vorher genannten Punkte wurde sich für ein relationales Datenbankmodell entschieden. Eine NoSQL-Datenbank würde aufgrund der strukturierten Daten keinen Sinn ergeben. Eine Graphen-Datenbank würde ebenso nicht sinnvoll sein, da keine/kaum Prozesse in Form eines Graphens darstellbar sind. Weitere Datenbankarten wie spaltenorientierte Datenbanken machen aufgrund der geringen Datenmenge ebenso wenig Sinn. Es gibt keine relevanten Daten in der App, die mithilfe der relationalen Datenbank nicht persistiert werden können.
Sollte die Oberfläche der App abgelöst werden, kann auf den persistierten Datenbestand selbstverständlich wieder zugegriffen werden.

3.1. ERM

Nachstehend ist das ERM visualisiert.

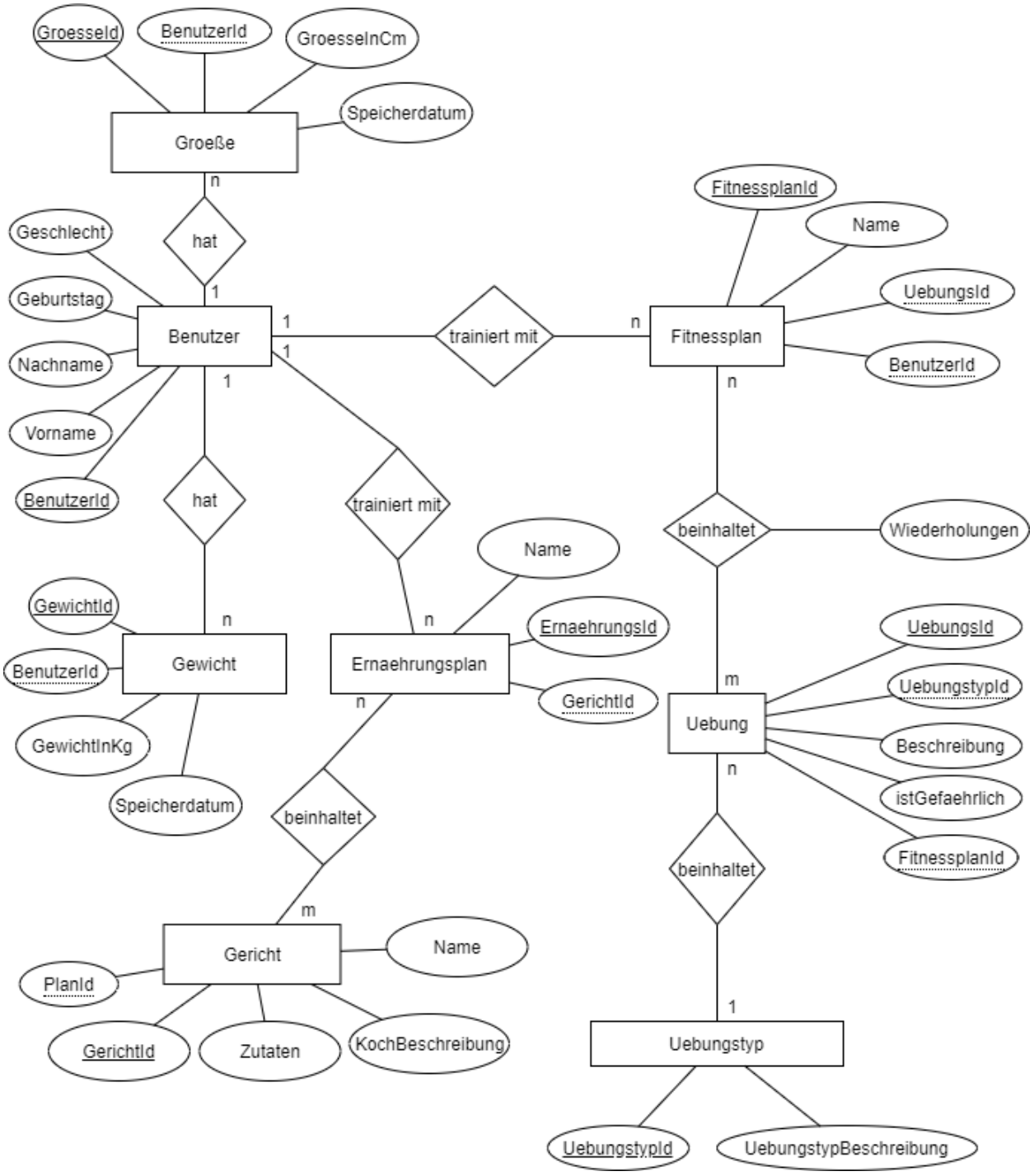

Abbildung 2 – ERM

Für die Fitnessapp wurde sich insgesamt auf 8 Entitäten festgelegt. Nach der Beseitigung der n:m Beziehung zwischen „Uebung“ und „Fitnessplan“ sowie zwischen „Ernaehrungsplan“ und „Gericht“ entstanden zwei Hilfsentität und somit sind 10 Entitäten insgesamt vorhanden. In folgender Tabelle werden die Entitäten beschrieben. Die entstandenen Hilfsentitäten sind kursiv geschrieben.

Tabelle 8 - Beschreibung der Entitäten

Entität	Beschreibung
Groeße	Wachstumsverlauf des Users, wird benötigt, um später unteranderem den BMI-Verlauf zu visualisieren
Benutzer	Allgemeine Informationen über einen Fitnessinteressierten
Gewicht	Gewichtverlauf des Users, wird benötigt, um später unteranderem den BMI-Verlauf zu visualisieren
Fitnessplan	Fitnessplan mit den einzelnen Übungen
Uebung	Informationen zu einer einzelnen Übung
Uebungstyp	Kraftübung, Cardioübung, …
Ernaehrungsplan	Ernaehrungsplan mit den einzelnen Gerichten
Gericht	Informationen über ein Gericht wie Zutaten
FitnessplanUebung	Hilfsentität um n:m Beziehung zwischen Fitnessplan und Uebung aufzulösen. Beinhaltet einen Plan mit allen Übungen sowie die Anzahl der Wiederholungen für eine entsprechende Übung

ErnaehrungsplanGericht	Hilfsentität um n:m Beziehung zwischen Ernaehrungsplan und Gericht aufzulösen. Beinhaltet Ernaehrungsplaene mit allen dazugehörigen Gerichten.

3.1.1. Beschreibung Relationen

Aus dem ERM lassen sich mithilfe von Transformationsregeln die Relationen der relationalen Datenbank erstellen. Diese sind nachstehend in tabellarischer Form aufgezeigt. Zu beachten gilt, dass **dick** markierte Spaltennamen keine Null-Werte zulassen, da diese Schlüsselfelder darstellen. Nach jeder Tabelle sind die einzelnen Datentypen begründet. Schlüsselfelder sind immer integer Werte und eindeutig. Auf sich wiederholenden Datentypen, die nicht von der Erklärung bereits erklärter Relationen abweichen, wird nicht eingegangen. Die Relationstabellen sind folgend aufgebaut:

1. Zeile -> Relationsname
2. Zeile -> Datentyp und Integritätsbedienung des Attributs
3. Zeile -> Attributnamen
4. Spalte -> Beispielwerte

Tabelle 9 - Relation Groeße

Groeße			
int	int	int, not null	Date, not null
GroesseId	**BenutzerId**	GroesseInCm	Speicherdatum
1	1	175	2020-06-05
2	1	176	2020-06-13
3	2	160	2020-06-05

Die Größe wird einheitlich in Zentimeter gespeichert. Somit werden keine Kommafehler, wie 17,5 Meter anstatt 1,75 Meter möglich sein. Speicherdatum benutzt den MySQL-Datentyp date, um eine einheitliche Form des Datums zu gewährleisten. Das Datum wird

später vom System und nicht vom User gespeichert. Somit ist die Korrektheit des Datums auf Seiten des Entwicklers.

Tabelle 10 - Relation Gewicht

Gewicht			
int	int	Double, not null	Date, not null
GewichtId	**BenutzerId**	GewichtInKg	Speicherdatum
1	1	75,7	2020-06-05
2	1	70,2	2020-06-13
3	2	60,0	2020-06-05

Hier wurde sich bewusst für einen double Wert für das Gewicht entschieden, da es einfach die übliche Form der Gewichtsangabe ist. Es ist unüblich sein Gewicht in Gramm anzugeben.

Tabelle 11 - Relation Benutzer

Benutzer				
int	varchar(255)	varchar(255)	date	varchar(20)
BenutzerId	Vorname	Nachname	Geburtstag	Geschlecht
1	Kai	Mueller	28.02.1997	Männlich
2	Max	Mustermann	01.01.1970	Männlich
3	Maxime	Mustermann	01.01.1970	Weiblich

Für die Namensfelder wurde sich für ein Varchar der Länge 255 entschieden, da Namen ziemlich lang sein können. Das Geschlecht wird ebenso mit dem Datentyp Varchar, jedoch mit Länge 20, versehen. Hier sind keine längere Felder nötig, da die gängigsten Geschlechtsangaben „Männlich“, „Weiblich“ oder „Divers“ sind.

Tabelle 12 - Relation Fitnessplan

Fitnessplan		
int	varchar(255)	int
FitnessplanId	Name	**BenutzerId**
1	Kais Fitnessplan	1
2	Get them abs	1
3	Nur Oberkörper	2

Tabelle 13 - Relation FitnessplanUebung

FitnessplanUebung			
int	int	int	int
FitnessplanUebung	**FitnessplanId**	**UebungId**	Wiederholung
1	1	1	10
2	1	2	5
3	1	3	3

Wiederholung hat hier den Typ int, da Übungen immer ganz durchgeführt werden können. Es gibt keine halben Wiederholungen.

Tabelle 14 - Relation Uebung

Uebung			
Int	Int	TEXT	Boolean, not null
UebungsId	**UebungstypId**	Beschreibung	istGefaehrlich
1	Butterfly	Gewicht mit ausgestreckten Armen von aussen nach innen bewegen	nein
2	Kreuzheben	Gewicht aus dem Hohlkreuz hochheben und kurz halten	ja
3	Liegestütze	Auf den Bauch liegen und dann den Körper mithilfe der Arme nach oben drücken	nein

Die Beschreibung der Uebung wird in dem MySQL Datentyp TEXT gespeichert. Mithilfe des Boolean Datentyps kann in Spalte „istGefaehrlich“ dargestellt werden ob eine Uebung gefährlich ist.

Tabelle 15 - Relation Uebungstyp

Uebungstyp	
int	varchar(100)
UebungstypId	UebungstypBeschreibung
1	Kraftuebung
2	Cardiouebung

Beschreibung des Übungstyps beinhaltet nur ein Wort und deshalb ist eine Begrenzung auf 100 Stellen des varchars ausreichend.

Tabelle 16 - Relation Ernaehrungsplan

Ernaehrungsplan	
int	varchar(100)
ErnaehrungsId	Name
1	Kais Ernaehrungsplan
2	Veggie Plan
3	Only Heathy Food

Die Begründung für 100 Stellen im varchar ist analog zu der Begründung in der Relation Uebungstyp.

Tabelle 17 - Relation ErnaehrungsplanGericht

ErnaehrungsplanGericht		
int	int	int
ErnaehrungsGerichtId	**ErnaehrungsId**	**GerichtId**
1	1	1
2	1	2
3	1	1

Diese Relation stellt eine Hilfsentität dar, die nur Schlüssel beinhaltet um eine n:m Beziehung aufzulösen.

Tabelle 18 - Relation Gericht

Gericht			
int	TEXT, not null	TEXT, not null	TEXT
GerichtId	Name	Zutaten	Kochbeschreibung
1	Gekochter Reis	500g Reis	Reis für 20 Minuten in kochendes Wasser
2	Obstsalat	2 Äpfel, 3 Birnen, 500g Joghurt	Alle Zutaten in einer Schüssel mischen und Joghurt oben drauf
3	Grüner Salat	1 Salatkopf	Salat waschen und in eine Schüssel

Die Zutaten werden in kommaseparierter Form mit dem Datentyp TEXT gespeichert. Name und Kochbeschreibung haben ebenso den Datentyp TEXT, da eine Kochbeschreibung auch länger sein kann.

4. Statische Sicht – Objekttypen zur Laufzeit

In Folgenden Kapitel werden die Objekte zur Laufzeit mithilfe eines Klassendiagramm dargestellt.

4.1. Klassendiagramm

Das Klassendiagramm wurde aus Gründen der Übersicht in 2 Teile unterteilt. Abbildung 3 beinhaltet größtenteils die Realisierung der einzelnen Objekte, die im Frontend dargestellt werden. Abbildung 4 zeigt die zuvor erwähnten Design-Pattern, beziehungsweise der Zugriff auf die Datenbank.

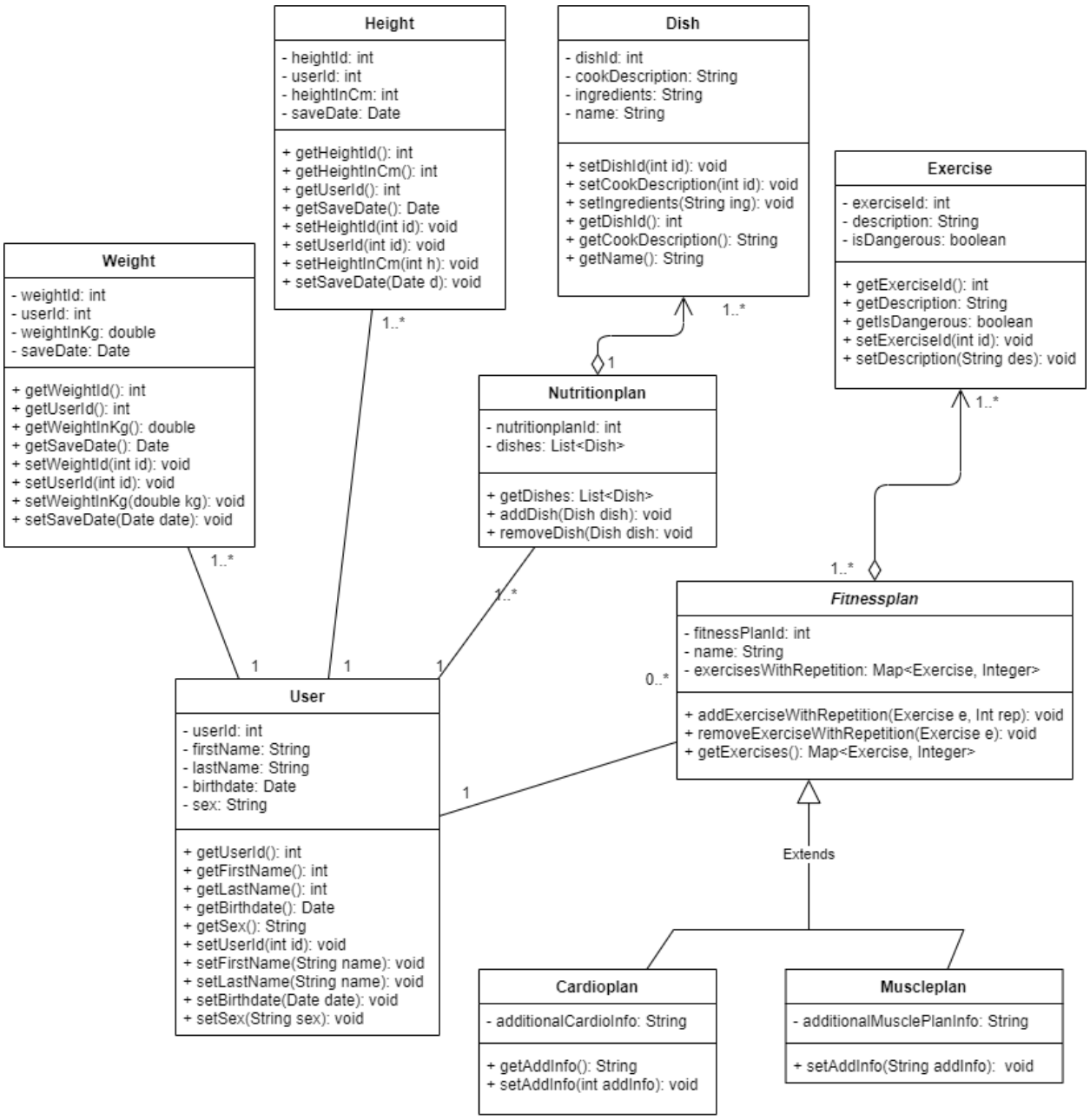

Abbildung 3 - UML-Klassendiagramm

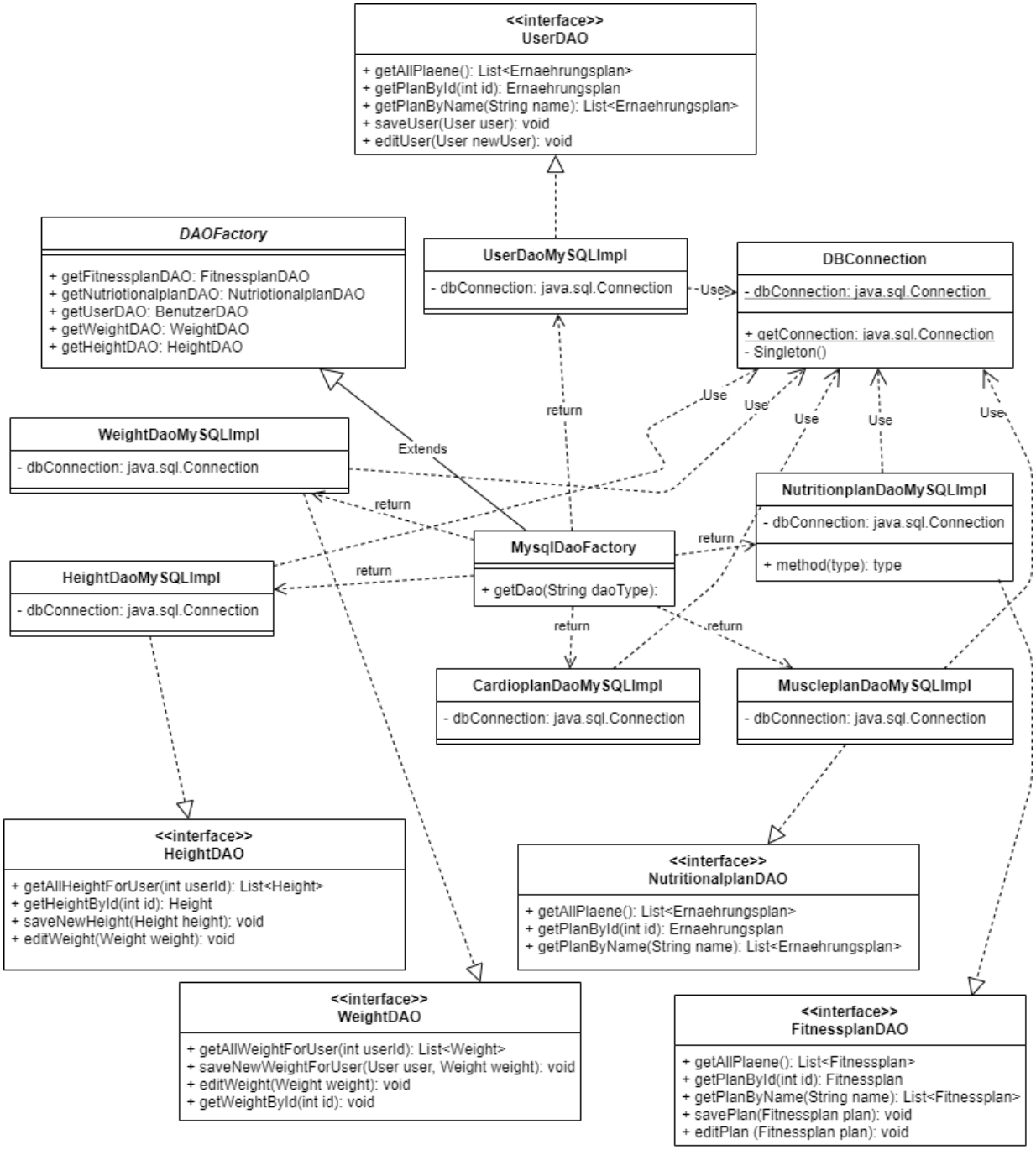

Abbildung 4 - UML-Klassendiagramm (Datenbankzugriff)

Die in Kapitel 2.1 beschriebene Design-Pattern sind in Abbildung 7 ersichtlich. Das Factory-Pattern ist in der Klasse „DAOFactory“ realisiert. Die abstrakte Klasse entscheidet zur Laufzeit um welche Datenbasis es sich handelt. Derzeit ist nur eine MySQL-Datenbasis vorhanden. Dies könnte jedoch mithilfe der Factory leicht erweitert werden. Das DAO-Pattern wurde mithilfe des diversen Interfaces sowie durch die Klasse „MysqlDaoFactory“ erzeugt. „MyqlDaoFactory“ liefert am Ende die konkreten Klassen zur Datenbank zurück.

Das Singleton Pattern ist in der Klasse „DBConnection“ implementiert. Es wird sichergestellt, dass nur maximal eine aktive Verbindung für die Anwendung besteht.
Als Alternative wäre es möglich die Factory wegzulassen, da diese aktuell keinen Mehrwert liefert. Durch die Factory kann jedoch ein wenig der Benefit der Hexagonal Architecture bezüglich der leichten Erweiterbarkeit erzeugt werden. Falls also eine weitere Speicherart der Daten, wie eine Clouddatenbank, gewünscht wird, kann dies mithilfe der Factory leicht implementiert werden. Der bestehende Code muss aufgrund der Factory und des DAO-Pattern nicht abgeändert werden.

5. Dynamische Sicht

Nachstehend werden die dynamischen Sichten beschrieben. Hierbei wird auf Funktionseinheiten sowie auf eine exemplarische Prozessbeschreibung eingegangen.

5.1. Funktionseinheiten

In folgender Tabelle werden die funktionalen Einheiten anhand des ursprünglichen Anforderungsdokument aufgezeigt. Die Anforderungen lassen sich dabei zu Funktionseinheiten zusammenfassen. Die Reihenfolge der Tabelle bezieht sich hierbei auf die Priorisierung.

Tabelle 19 - Funktionale Einheit (Trainingspläne erstellen)

Trainingspläne erstellen	
Anforderungsbeschreibung	**AnforderungsID**
Cardioplan erstellen	R-5
Krafttrainingsplan erstellen	R-6
Kompaktplan erstellen	R-7

Tabelle 20 - Funktionale Einheit (Trainingsplanumgang)

Trainingsplanumgang	
Anforderungsbeschreibung	**AnforderungsID**
Bestehender Trainingsplan bearbeiten	R-10
Bestehender Trainingsplan löschen	R-11

Tabelle 21 - Funktionale Einheit (Benutzerbezogene Werte)

Benutzerbezogene Werte	
Anforderungsbeschreibung	**AnforderungsID**
Erfassung Gewicht	R-16
Darstellung Gewichtsverlauf	R-17
BMI Berechnen	R-1
BMI visualisieren	R-2
BMI Infotext anzeigen	R-3

Tabelle 22 - Funktionale Einheit (Übungen)

Übungen	
Anforderungsbeschreibung	**AnforderungsID**
Stoppuhr für zeitabhängige Übungen	R-13
Vorzeitige Beendigung Stoppuhr	R-14
Warnhinweise zu Übungen anzeigen	R-15
Anzeige motivierender Texte	R-25

Tabelle 23 - Funktionale Einheit (Ernährungsplan)

Ernährungsplan	
Anforderungsbeschreibung	**AnforderungsID**
Ernährungsplan erstellen	R-19
Bearbeitung bestehender Ernährungspläne	R-22
Löschung bestehender Ernährungspläne	R-23

Die höchste Priorisierung genießt also die Einheit „Trainingspläne erstellen“, da dies die Hauptfunktion der Applikation darstellt.

5.2. Prozessbeschreibung Trainingspläne erstellen

Die Funktionseinheit „Trainingspläne erstellen“ ist für die korrekte Erstellung oder Generierung der Trainingspläne zuständig.
Um die am höchsten priorisierte Funktionseinheit während der Laufzeit genauer zu verdeutlichen dient folgendes Ablaufdiagramme.

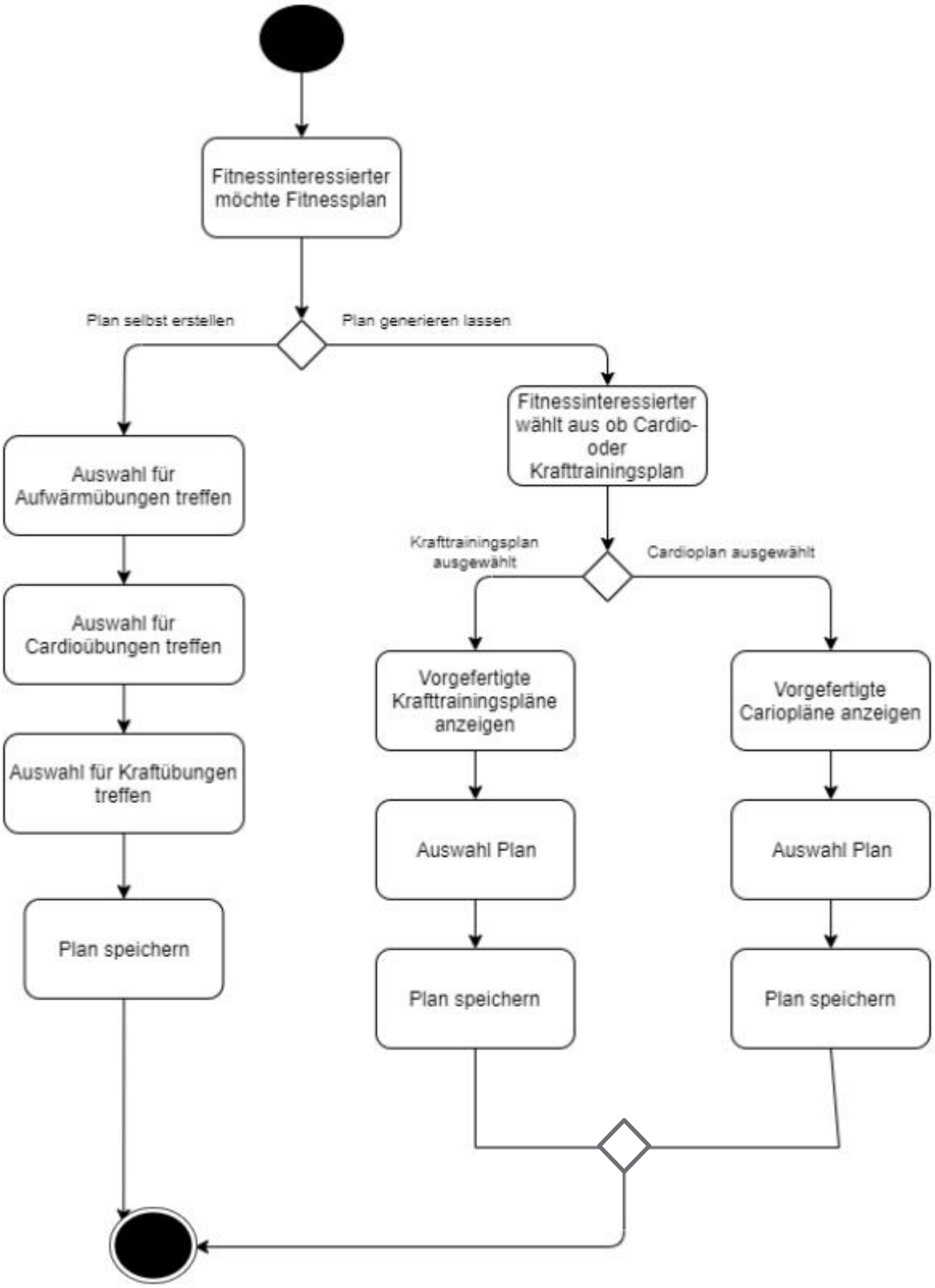

Abbildung 5 - Aktivitätsdiagramm Trainingspläne erstellen

In Abbildung 5 ist der Ablauf beim Anlegen eines Fitnessplanes visualisiert. Der Benutzer kann auswählen ob er einen Plan selbst anlegen möchte oder sich einen Plan generieren

lassen möchte. „Plan speichern“ stellt hierbei der Zugriff auf die DAO-Klasse zu und somit die Speicherung in das relationale Datenbanksystem.

5.3. Prozessbeschreibung Trainingsplanumgang

Die Funktionseinheit „Trainingsplanumgang“ beschäftigt sich mit der Bearbeitung sowie Löschung von bestehenden Trainingsplänen. Hierbei muss nach jeder Bearbeitung/Löschung die Datenbank aktualisiert werden.

5.4. Prozessbeschreibung Benutzerbezogene Werte

Unter benutzerbezogene Werte sind alle Anforderungen mit Bezug auf den Benutzer zusammengefasst. Ein Beispiel hierfür ist die Berechnung des BMI. Dies wird in folgendem Diagramm visualisiert.

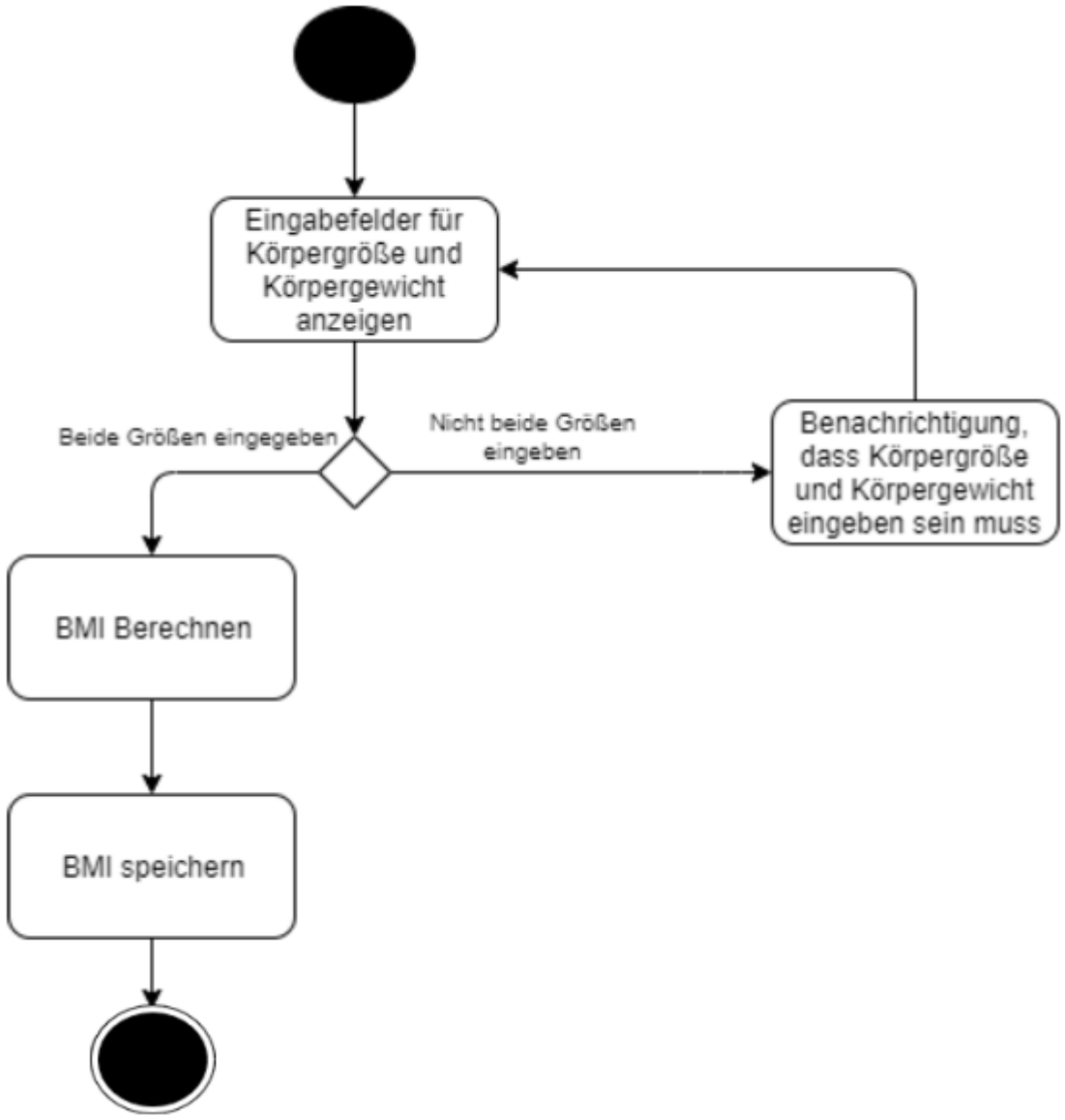

Abbildung 6 - Aktivitätsdiagramm BMI-Berechnen

Es wir deutlich, dass ein BMI nur berechnet wird, wenn der Benutzer Größe sowie Gewicht hinterlegt hat. Nach der Berechnung wird der BMI in der Datenbank gespeichert.

5.5. Prozessbeschreibung Übungen

In dieser Funktionseinheit werden Anforderungen mit Bezug auf eine Übung gespeichert. Ein Beispiel hierfür wäre, dass ein Motivationstext nach dem Beenden einer Übung angezeigt wird. Dieses Verhalten wird in folgendem Aktivitätsdiagramm dargestellt.

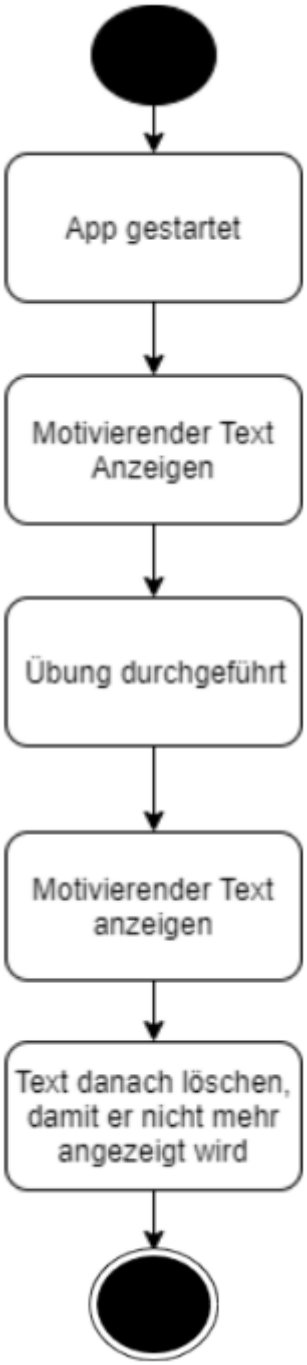

Abbildung 7 - Aktivitätsdiagramm Motivationstext

Abbildung 7 zeigt, dass nach einer durchgeführten Übung ein motivierender Text angezeigt werden soll. Nach dem der Text angezeigt wurde, wird dieser im weiteren Verlauf der App nicht mehr angezeigt.

5.6. Prozessbeschreibung Ernährungsplan

Diese Funktionseinheit ist die Zusammenfassung aus den Anforderungen zur Erstellung, Bearbeitung und Löschung von Ernährungsplänen.

6. Auswahl des Technologiestacks

In Kapitel 6 wird auf den Technologiestack eingegangen. Es werden Entscheidungen begründet sowie mögliche Alternativen aufgezeigt.

6.1. Programmiersprache

Die Programmierung einer Android App ist mit verschiedenen Programmiersprachen möglich. Die Auswahl an Programmiersprachen zeigt folgende Liste.

- Java
- Kotlin
- C und C++
- C#

Die letzten zwei Punkte benötigen spezielle Anpassung, damit die Android Programmierung möglich ist. Aufgrund vorhandener Erfahrung in der Programmiersprache Java wurde sich in diesem Projekt auf diese Sprache festgelegt. Eine Alternative stellt Kotlin dar. Aufgrund mangelnder Zeit kann sich jedoch nicht in Kotlin eingearbeitet werden, um eine sichere Evaluation zu treffen. Der Grad der Sicherheit für die Entscheidung ist jedoch trotzdem als hoch zu betrachten, da Erfahrung mit Java bereits vorhanden ist.

6.2. Buildvorgang

Die Kernaufgabe des Buildvorgangs stellt die Artefakterstellung sowie die Ausführung der automatisierten Tests dar. Zwei der bekanntesten Buildtools sind Maven und Ant. Aufgrund bereits vorhandener Erfahrung mit Maven wird für dieses Projekt Maven als Buildtool eingesetzt. Die vorhandene Erfahrung kann eingesetzt werden, um schnell ein Buildprozess zu konfigurieren, welcher automatisierte Tests ausführt sowie die Artefakte erstellt. Ant gilt jedoch als brauchbare Alternative.

6.3. Datenbank

In Kapitel 3 wurde bereits beschrieben, dass ein relationales Datenbankmodell verwendet wird. Es muss jedoch weiter entschieden werde, welches System für das relationale Datenbankmodell eingesetzt wird. MySQL und Microsoft SQL-Server stellen gängige Modelle der relationalen Datenbanken dar. Aufgrund der hohen Lizenzkosten wurde sich

jedoch für die MySQL-Datenbank entschieden. Es muss jedoch erwähnt werden, dass aufgrund mangelnder Zeit kein weiteres relationales Datenbankmodell evaluiert werden konnte. Dies senkt den Grad der Sicherheit, da unklar ist ob es ein Datenbankmodell gibt, welches besser für die Speicherung der Daten eignet. Es ist jedoch auch zu erwähnen, dass MySQL ein weit verbreitetes Datenbankmodell ist und eine hohe Erfahrung seitens der Entwickler vorhanden ist.

6.3.1. Umgebung der Datenbank

Die Datenbank könnte lokal auf dem Endgerät oder auf einem Online-Server implementiert sein. Vorteile für eine lokale Einrichtung der Datenbank ist, dass die App komplett im Offlinebetrieb funktioniert. Sollte die Datenbank online gehostet sein, können Funktionen, wie das Anlegen eines neuen Trainingsplans, nur mit einer aktiven Internetverbindung stattfinden kann. Der große Vorteil der Online-Hosting der Datenbank stellt die einfache Verwaltung durch die Entwickler dar. Deshalb wurde sich in diesem Projekt dafür entschieden die Datenbank online auf einem Server zu hosten. In zukünftigen Entwicklungsphasen könnte sich Gedanken gemacht werden neu angelegte Pläne im Offlinemodus in einer Art Cache vorzuhalten und sobald eine aktive Internetverbindung besteht die neu angelegten Pläne in die Datenbank zu speichern. Dadurch kann der Vorteil einer Datenbank, welche lokal auf einem Endgerät implementiert ist, auf die Variante mit der online gehosteten Datenbank angewandt werden. Zu beachten ist, dass bei einer online gehosteten Datenbank eine Zugriffsverwaltung implementiert sein muss, welche mindestens aus Username und Password besteht.

6.3.2. Datenbankzugriff

Der Zugriff auf die Datenbank kann über einfache SQL-Statements ausgeführt werden oder durch die Hilfe von Object-Relational Mapping Tools. Ein bekanntes ORM-Tool im Java Umfeld ist Hibernate. Mithilfe von Hibernate wird der Zugriff auf die Datenbank weiter vereinfacht, da die Datenbank in Java-Beans abgebildet wird. Nachdem man die Werte einer Bean gesetzt hat, kann über Hibernate das komplette Objekt in die Datenbank gespeichert werden. Dies sorgt für ein höheres Maß an Codetransparenz sowie für eine geringere Fehleranfälligkeit, da keine SQL-Statements im String-Format mehr geschrieben werden müssen.

Die Implementierung und die Konfiguration von Hibernate sind jedoch mit Aufwand verbunden. Aufgrund des erhöhten Aufwands wurde sich gegen die Verwendung eines ORM-Tools entschieden.
Eine Alternative, um die Fehleranfälligkeit in SQL-Statements zu senken, stellt die Verwendung von Prepared Statements dar. Auf diese Alternative wird in diesem Projekt zurückgegriffen.

6.4. Testtools

Da sich für die Programmiersprache Java entschieden wurde, fällt die Wahl des Testtools auf „JUnit“. JUnit ist ein seit Jahren im Java Umfeld etabliertes Testtool. Um eine Kapselung beim Testen zu gewähren wird das Tool „Mockito“ eingesetzt. Mithilfe von Mockito lässen sich Mock-Objekte erstellen, die Verhalten simulieren können. Ein Mock-Objekt wäre zum Beispiel die Datenbankverbindung. Die JUnit Tests werden als automatisierte Tests im Buildvorgang ausgeführt.

7. Verteilungsdiagramm

Nachstehend sind die einzelne Komponente grobgranular zur Übersicht dargestellt.

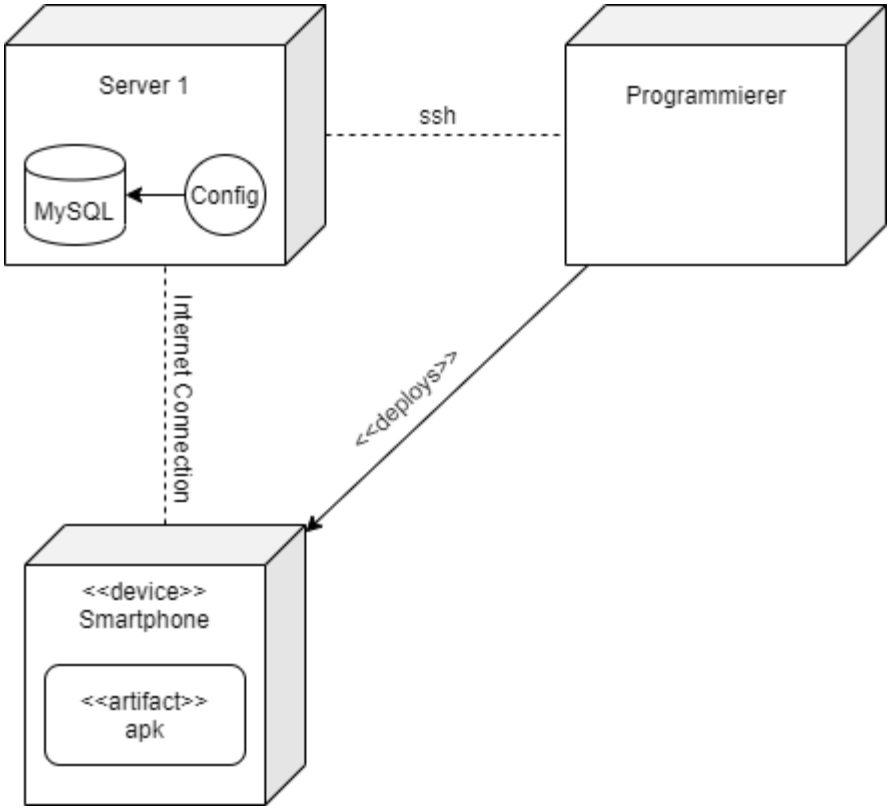

Abbildung 8 - Komponenten des Projektes (Grobgranular)

Es ist zu erkennen, dass auf einen online gehosteten Server die MySQL-Datenbank implementiert ist. Die Datenbank hat eine Konfiguration, welche ebenso auf dem Server liegt. Der Programmierer verbindet sich über SSH mit Server 1 und kann dort auf die Datenbank sowie auf die Konfiguration der Datenbank zugreifen. Des Weiteren deployed der Programmierer neue Updates der App auf die einzelnen Smartphones.

Das Smartphone beinhaltet die App als APK-Datei und bezieht Updates, welche vom Programmierer verteilt werden. Es besteht eine aktive Internetverbindung zwischen der App und der Datenbank.

8. Fazit

Es wurde eine mögliche Architektur konzipiert, die vor allem das Ziel auf hohe Testbarkeit legt. Die Architektur ist für die Umsetzung der zuvor gesammelten Anforderungen geeignet. Entscheidungen, die getroffen werden mussten, wurden mit verfügbaren Alternativen verglichen. Es konnte jedoch nicht jede Evaluation aufgrund mangelnder Zeit und Erfahrung in ausreichender Tiefe vollzogen werden.

Die in diesem Dokument aufgezeigte Architektur setzt auf Testabdeckung sowie auf die leichte Erweiterung von Funktionen. Aufgrund der erklärten Design-Pattern sowie der gewählten Systemarchitektur können leicht neue Komponente zum bestehenden System hinzugefügt werden. Die Auswahl der Technologien beruht größtenteils auf der kostenfreien Nutzung des Tools und auf Erfahrungswerte des Entwicklers.