

Softwaretechnik II – Praktikum

Subsystem 4 – Zubereitung

Eine Dokumentation von:

J. Faßbender

J. Gobelet

L. Gobelet

E. Gödel

Team 4.4

Inhaltsverzeichnis

1	Meilenstein 1 – Datenzugriffsschicht	4
1.1	Teilaufgabe 1: Ausschnitt aus Logischem DM mit Entities und Value Objects	4
1.1.1	Klassendiagramm	4
1.1.2	Fachliches Glossar	5
1.1.3	Erweiterungen der Aufgabenstellung	5
1.1.4	Erläuterung	5
1.2	Teilaufgabe 2: Entities und Value Objects mit JPA-Annotierung	6
1.2.1	Annotationen der Entities und Value Objects	6
1.2.2	H2-Console	7
1.3	Teilaufgabe 3: Factories und Repositories	9
2	Meilenstein 2 – Komponentenschnitt	11
2.1	Teilaufgabe 1: Vorbereitung des Komponentenschnitts	11
2.1.1	Liste der Geschäftsobjekte	11
2.1.2	Liste der Use Cases	11
2.1.3	Liste der Umsysteme	12
2.2	Teilaufgabe 2: Ermittlung der verschiedenen Komponenten-Typen	13
2.2.1	Schritt 1: Geschäftsobjekte in zusammenhängende Gruppen einteilen	13
2.2.2	Schritt 2: Use Cases auf Daten/Logik analysieren	14
2.2.3	Schritt 3: Use Cases auf Nutzer-Interaktion analysieren	14
2.2.4	Schritt 4: Angebot von externen Schnittstellen	15
2.2.5	Schritt 5: Aufruf von externen Schnittstellen/Umsystemen	15
2.3	Teilaufgabe 3: Komponentendiagramm	16
3	Meilenstein 3 – Spezifikation, Implementierung und Demo eines REST-API	18
3.1	Teilaufgabe 1: Festlegen von Aggregates	18
3.2	Teilaufgabe 2: Design des REST-API	19
3.3	Teilaufgabe 3: Implementierung in Spring Data JPA / Web MVC	21
3.3.1	Code-Listing	21
3.3.2	Custom JSON Serializer	24
3.3.3	Nachweis der Lauffähigkeit	28
4	Meilenstein 4 – Microservices	30
4.1	Teilaufgabe 1: Context Map (Entities aus unserem fachlichen Datenmodell)	30
4.1.1	Context Map	31
4.1.2	Tabelle der Überlappungstypen	32
4.2	Teilaufgabe 1: Context Map (Entities aus unserem logischen Datenmodell)	32
4.2.1	Context Map	33
4.2.2	Tabelle der Überlappungstypen	34
4.3	Teilaufgabe 2: Aggregates	35
4.4	Teilaufgabe 3: Microservice-Architektur	37
4.4.1	Servicetabelle	37
4.4.2	Komponentendiagramm	38
4.4.3	Vergleich monolithisches Modell aus Meilenstein 2 mit Microservice-Modell	39

Abbildungsverzeichnis

1	Klassendiagramm	4
2	Gerichtstabelle	7
3	Speisentabelle	8
4	Zutatentabelle	8
5	Zutatenpositionstabelle	9
6	Zuordnungstabelle Gericht - Speise	9
7	Ausgabe in der Konsole	10
8	Komponentendiagramm	16
9	Aggregates	18
10	Ausschnitt Klassendiagramm für REST-API	19
11	Postman Collection	28
12	Postman Test	29
13	Context Map (fachliches Datenmodell)	31
14	Context Map (logisches Datenmodell)	33
15	Komponentendiagramm	38

1 Meilenstein 1 – Datenzugriffsschicht

1.1 Teilaufgabe 1: Ausschnitt aus Logischem DM mit Entities und Value Objects

1.1.1 Klassendiagramm

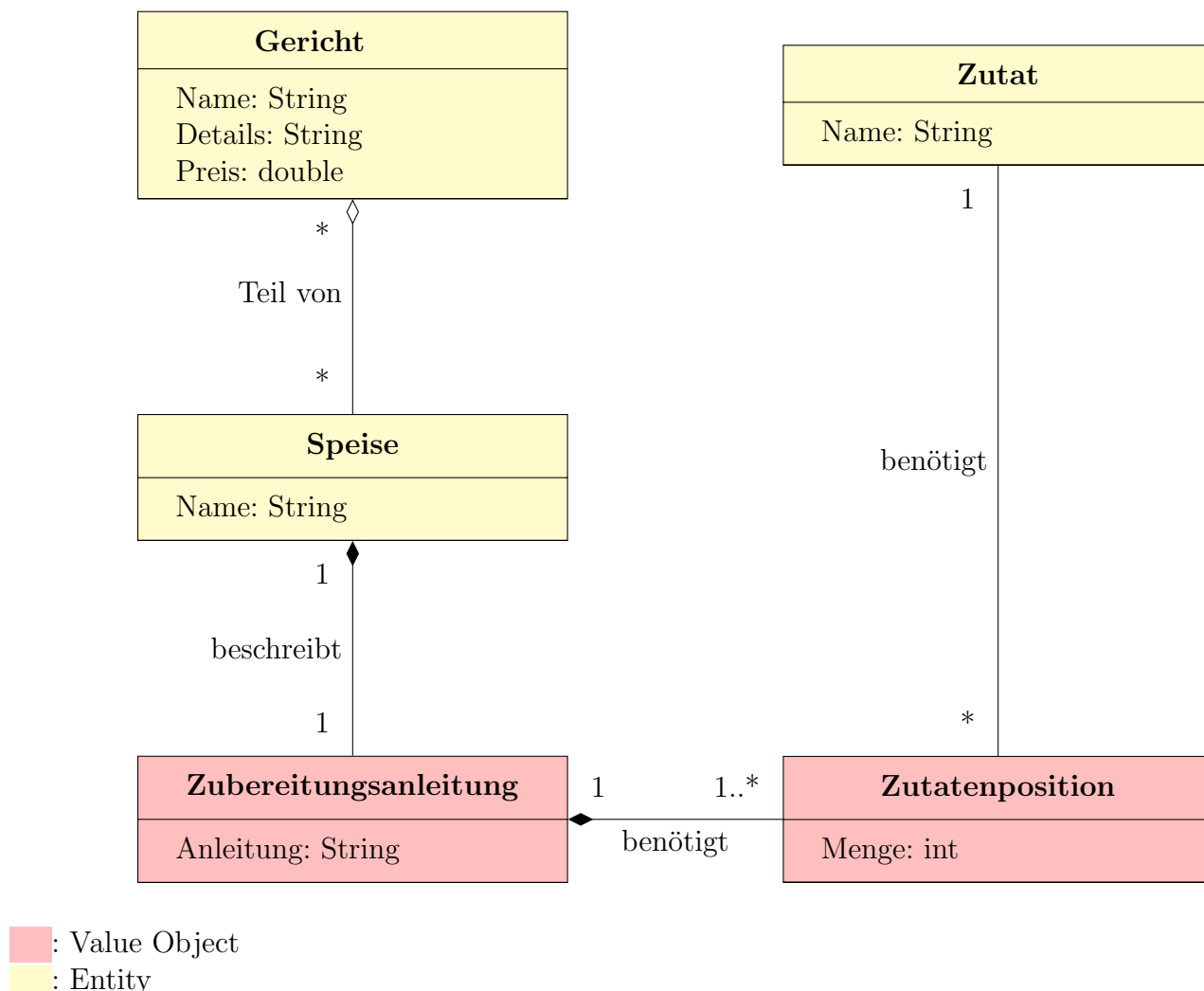


Abbildung 1: Klassendiagramm

1.1.2 Fachliches Glossar

Geschäftsobjekt	Attribut	Erklärung
Gericht		Vom Restaurant angebotenes Mahl.
	Name	Gerichtsbezeichnung.
	Details	Wird dem Gast angezeigt. Enthält nähere Angaben zu den Zutaten.
	Preis	Geldbetrag der für das Gericht zu bezahlen ist.
Speise		Teil eines Gerichts. Beispielsweise wäre eine Salatbeilage als Speise zu verstehen.
	Name	Bezeichnung der Speise.
Zubereitungsanleitung		Leitfaden zur Zubereitung einer Speise.
	Anleitung	Erklärender Text, der beschreibt, wie eine Speise zuzubereiten ist.
Zutat		Benötigt für die Zubereitung einer Speise.
	Name	Bezeichnung der Zutat.
Zutatenposition		Zuordnung zwischen Zutat und Zubereitungsanleitung. Gibt die Menge einer Zutat an, die für die Zubereitung notwendig ist.
	Menge	Die benötigte Menge.

1.1.3 Erweiterungen der Aufgabenstellung

Da es in unserem Logischen Datenmodell keine 1:1-Beziehung gab, haben wir eine zusätzliche redundante Entität eingebaut.

Hierbei handelt es sich um die Entität Speise. Diese Entität hätte genauso gut einfach Teil der Zubereitungsanleitung sein können und ist nur in unser Modell aufgenommen worden, damit wir die für die Aufgabenstellung benötigte 1:1-Beziehung in unserem Diagramm haben.

1.1.4 Erläuterung

Wir haben Zubereitungsanleitung als Value Object und nicht als Entity deklariert, da hier unserer Meinung nach Sharing nicht sinnvoll ist und ein Zubereitungsanleitungsobjekt deshalb persistent als Teil der zugeordneten Speise in der Datenbank gespeichert werden sollte.

Gleiches gilt für die Zutatenposition.

1.2 Teilaufgabe 2: Entities und Value Objects mit JPA-Annotierung

1.2.1 Annotationen der Entities und Value Objects

Gericht

```
@Entity
public class Gericht {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;
    private String name;
    private String details;
    private double preis;

    // Ein Gericht besteht aus mehreren Speisen und eine Speise kann
    // mehreren Gerichten zugeordnet sein.
    @ManyToMany
    @JoinTable(name = "gericht_speise",
        joinColumns = @JoinColumn(name = "gericht_id"),
        inverseJoinColumns = @JoinColumn(name = "speise_id")
    )
    private Set<Speise> speisen = new HashSet<Speise>();
}
```

Speise

```
@Entity
public class Speise {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;
    private String name;

    // bidirektionale Beziehung: Gericht kennt zugehoerige Speisen und
    // die Speisen kennen zugehoerige Gerichte
    @ManyToMany(mappedBy = "speisen")
    private Set<Gericht> gerichte = new HashSet<Gericht>();
}
```

Zubereitungsanleitung

```
@Embeddable
public class Zubereitungsanleitung {
    private String anleitung;

    // Die Anleitung enthaelt mehrere Zutatenangaben als Value-Objects
    @ElementCollection (targetClass = Zutatenmenge.class, fetch =
```

```

    FetchType.EAGER)
    @CollectionTable(name = "ZUTATENANGABE")
    private Set<Zutatenmenge> angaben = new HashSet<Zutatenmenge>();

```

Zutat

```

@Entity
public class Zutat {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;
    private String name;
}

```

Zutatenposition

```

@Embeddable
public class Zutatenmenge {
    private int menge;

    @ManyToOne
    private Zutat zutat;
}

```

1.2.2 H2-Console

SELECT * FROM GERICHT;

ID	DETAILS	NAME	PREIS
10	Voll das Oma-Essen!	Kartoffelbrei mit Möhren	7.5
11	Jede Erbse macht einen Knall!	Kartoffelbrei mit Erbsen	8.5

(2 rows, 9 ms)

Abbildung 2: Gerichtstabelle

SELECT * FROM SPEISE;

ID	ANLEITUNG	NAME
7	Möhren und Pfeffer umrühren!	Möhrengemüse
8	Erbsen, Salz und Pfeffer verbrennen lassen!	Erbsengemüse
9	Kartoffeln, Salz und Butter vermatschen!	Kartoffelbrei

(3 rows, 3 ms)

Abbildung 3: Speisentabelle

SELECT * FROM ZUTAT;

ID	NAME
1	Erbse
2	Butter
3	Salz
4	Möhre
5	Pfeffer
6	Kartoffel

(6 rows, 1 ms)

Abbildung 4: Zutatentabelle

SELECT * FROM ZUTATENMENGE;

SPEISE_ID	MENGE	ZUTAT_ID
7	1	5
7	3	4
8	100	1
8	2	3
8	5	5
9	6	6
9	5	3
9	2	2

(8 rows, 8 ms)

Abbildung 5: Zutatenpositionstabelle

SELECT * FROM GERICHT_SPEISE;

GERICHT_ID	SPEISE_ID
10	7
10	9
11	8
11	9

(4 rows, 1 ms)

Abbildung 6: Zuordnungstabelle Gericht - Speise

1.3 Teilaufgabe 3: Factories und Repositories

Factory für Erstellung von Gerichten

```
@Component
public class GerichtFactory {

    // Erstelle ein Gericht, das nur aus einer Speise besteht.
    public static Gericht createGerichtWithSpeise(String name, String
    details, double preis, Speise speise) {
        Gericht gericht = new Gericht(name, details, preis);
    }
}
```

```

    gericht.addSpeise(speise);
    // Rueckreferenz setzen
    speise.addGericht(gericht);
    return gericht;
}

// Erstelle ein Gericht, das aus mehreren Speisen besteht.
public static Gericht createGerichtWithSpeisen(String name, String
details, double preis, Collection<Speise> speisen) {
    Gericht gericht = new Gericht(name,details, preis);
    gericht.addSpeisen(speisen);
    for(Speise s : speisen) {
        // Rueckreferenz setzen
        s.addGericht(gericht);
    }
    return gericht;
}
}

```

Hier sieht man gut, warum Factories notwendig sind. Bei der Erstellung von Gerichten muss zugleich die Rückreferenz von Speise auf Gericht gesetzt werden.

Factory für Erstellung von Gerichten

```

public interface SpeiseRepository extends CrudRepository<Speise,
Integer> {
    // Die Abfrage ist in JPQL geschrieben - Eine objektorientierte
    // Abfragesprache, welche SQL aehnlich ist
    // Findet alle Speisen, die eine bestimmte Zutat enthalten
    @Query("select s from Speise s join s.anleitung a join a.angaben
ang where ang.zutat = :zutat")
    List<Speise> findByContainsZutat(@Param("zutat")Zutat zutat);
}

```

Ausgabe in der Konsole

```

// gib alle Speisen aus, die Salz enthalten
System.out.println("\nSalzige Speisen: ");
speiseRepository.findByContainsZutat(zutaten.get("Salz")).
forEach(s -> System.out.println(s.getName()));

```

Folgendes wird dann in der Konsole ausgegeben:

```

Salzige Speisen:
Erbsengemüse
Kartoffelbrei

```

Abbildung 7: Ausgabe in der Konsole

2 Meilenstein 2 – Komponentenschnitt

2.1 Teilaufgabe 1: Vorbereitung des Komponentenschnitts

2.1.1 Liste der Geschäftsobjekte

- Arbeitsplatz
- Bestellung
- Gericht
- Sitzplatz
- Speisekarte
- Zubereitungsanleitung
- Zutat
- Zutatenposition

2.1.2 Liste der Use Cases

- Am Arbeitsplatz an-/abmelden
- Gericht bestellen
- Gericht zubereiten

2.1.3 Liste der Umsysteme

Umsystem	Was geschieht zwischen Umsystem und unserem Subsystem?	Schnittstelle angeboten oder aufgerufen
Rezeptverwaltung	Rezeptverwaltung verwaltet die Geschäftsobjekte Gericht, Zubereitungsanleitung und Speisekarte. Der Gast fragt über das ihm zur Verfügung gestellte Frontend die Speisekarte und die Gerichte ab, während der Koch an seinem Terminal die Zubereitungsanleitung und die hiermit verbundenen Zutatenpositionen, angezeigt bekommt.	Aufruf einer Schnittstelle zur Rezeptverwaltung
Lagerverwaltung	Abfrage zum Zutatenbestand	Aufruf einer Schnittstelle zur Lagerverwaltung
Lagerverwaltung	Angabe zur Zutatenentnahme (kann auch über die gleiche Schnittstelle, die im obigen Tabelleneintrag spezifiziert ist, realisiert werden)	Aufruf einer Schnittstelle zur Lagerverwaltung
Buchhaltung	Abfrage der Bestellungen	Schnittstelle wird Buchhaltung zur Verfügung gestellt

Erläuterung

Wir legen redundant zur Lagerverwaltung unsere eigene Verwaltung mit Angaben zum Zutatenbestand an, um auch bei Nichterreichbarkeit der Lagerverwaltung funktionsfähig zu bleiben, da unser Subsystem essentiell für den Umsatz verantwortlich ist und ein Ausfall, das heißt in diesem Fall der Zustand, dass eine Zutat nicht mehr in benötigter Menge im Lager zur Verfügung steht, nicht auf Grund technischer Probleme eintreten sollte.

Allerdings stellen wir keinen Anspruch auf absolute Richtigkeit unserer Zutatenbestandsverwaltung, da wir nur die Ereignisse unseres Subsystems, das heißt in diesem Fall die Entnahme einer Zutat zur Zubereitung, protokollieren und die restlichen Angaben aus der Lagerverwaltung stammen.

Ist diese nun nicht erreichbar, verwendet unsere Zutatenbestandsverwaltung mitunter veraltete Daten, was wir nicht mit einbeziehen.

Der Lagerverwaltung wird die Entnahme von unserem Subsystem aus mitgeteilt.

Für den kompletten Synchronisationsprozess zwischen den beiden Systemen stellt uns die Lagerverwaltung zwei Schnittstellen (oder eine, die beide Aufgaben - Entnahme mitteilen und Zutatenbestand abfragen - zusammenfasst) zur Verfügung.

Zusätzlich haben wir eine Schnittstelle für die Buchhaltung angelegt. Diese ist zwar kein explizites Subsystem, wird aber, unserer Meinung nach, im Betriebsumfeld höchstwahrscheinlich als eigenes Subsystem existieren und unsere Schnittstelle zu den Bestellungen (im Endeffekt der Unternehmensumsatz aus dem Hauptgeschäft) nutzen wollen.

2.2 Teilaufgabe 2: Ermittlung der verschiedenen Komponenten-Typen

2.2.1 Schritt 1: Geschäftsobjekte in zusammenhängende Gruppen einteilen

Datenkomponente	Zugeordnete Geschäftsobjekte	Erklärung
Bestelldaten	Bestellung	Die einzigen Daten die in diesem Subsystem tatsächlich generiert werden. Da die Bestellungen sehr wichtig für das Hauptgeschäft der Firma ist, es das einzige Datenobjekt mit Implementierung eines Create-Interfaces (Factory) ist und auch sonst nicht in unsere sonstigen Datenkomponenten passt, wird die Bestellung, unserer Meinung nach, in einer eigenen Komponente implementiert.
Standortdaten	Arbeitsplatz, Sitzplatz	Diese Daten ändern sich äußerst selten (und auch nicht in unserem Subsystem) und umfassen im Vergleich zu anderen Komponenten wenig Datensätze und können deshalb, unserer Meinung nach, gut zusammengefasst werden.
Gerichtsdaten	Gericht, Speisekarte, Zubereitungsanleitung, Zutat, Zutatenposition	Stammdaten die für unseren Prozess der Zubereitung essenziell sind. Diese Daten stammen nicht aus unserem Subsystem, sondern sind über Schnittstellen abrufbar, sowohl von der Lagerverwaltung (Zutat), als auch von der Rezeptverwaltung (Gericht, Speisekarte, Zubereitungsanleitung, Zutatenposition). Unsere Datenkomponente greift über Adapterkomponenten auf diese Schnittstellen zu.

2.2.2 Schritt 2: Use Cases auf Daten/Logik analysieren

Daten-/Logikkomponente	Zugeordnete(r) Use Case(s)	Erklärung
Bestellabwicklung (Logik)	Am Arbeitsplatz an-/abmelden, Gericht bestellen, Gericht zubereiten	Unser „Backend“, was ab der Bestellaufgabe den Zubereitungsprozess steuert. Die Komponente umfasst die Vergabewarteschlange mit den besetzten und freien Arbeitsplätzen und übernimmt die Zuweisung, sobald eine Bestellung von einem Clienten eingeht. Sobald ein Gericht fertig zubereitet ist und der Koch dies seinem Terminal mitteilt, übernimmt diese Komponente auch die Anzeige der Ordernummer (im Gast-UI). Da dies alles vom Umfang her eher kleinere Aufgaben sind, haben wir uns dazu entschieden, diese Aufgaben in einer Komponente zusammenzufassen.

2.2.3 Schritt 3: Use Cases auf Nutzer-Interaktion analysieren

Dialogkomponente	Zugeordnete(r) Use Case(s)	Eigene Fassadenkomponente sinnvoll?	Erklärung
Zubereitungs-UI	Gericht zubereiten	Ja	Fassadenkomponente zur Orchestrierung der Gerichtszubereitung.
An-/Abmeldungs-UI	Am Arbeitsplatz an-/abmelden	Ja	Fassadenkomponente für den Zugriff auf Datenkomponente „Standortdaten“ (Read- und Updateoperationen auf den Arbeitsplatz) und um das „Strict Layering“ einzuhalten.
Gast-UI	Gericht bestellen	Ja	Fassadenkomponente zur Orchestrierung des Bestellvorgangs.

2.2.4 Schritt 4: Angebot von externen Schnittstellen

Umsystem/Schnittstelle	Eigene Fassadenkomponente sinnvoll?	Erklärung
Buchhaltung	Ja	Da die Buchhaltung lesenden Zugriff auf unsere Bestellungen haben soll, ist es notwendig eine spezialisierte Komponente hierfür anzulegen und nicht, wie intern in unserem Subsystem, den Zugriff über die Bestelldatenkomponente zu regeln.
Lagerverwaltung	Nein	Zugriff erfolgt nur aus der Gerichtsdatenkomponente über die Adapterkomponente der Lagerverwaltung, weshalb, unserer Meinung nach, keine Fassadenkomponente notwendig ist.
Rezeptverwaltung	Nein	Zugriff erfolgt nur aus der Gerichtsdatenkomponente über die Adapterkomponente der Rezeptverwaltung, weshalb, unserer Meinung nach, keine Fassadenkomponente notwendig ist.

2.2.5 Schritt 5: Aufruf von externen Schnittstellen/Umsystemen

Umsystem/Schnittstelle	Adapterkomponente sinnvoll?	Erklärung
Buchhaltung	Nein	Bereits spezialisierte Fassadenkomponente vorhanden.
Lagerverwaltung	Ja	Adapterkomponente für unsere Gerichtsdatenkomponente, die die Lese- und Schreibvorgänge zur Verfügung stellt und gleichzeitig bei Ausfällen als „Anti-Corruption-Layer“ fungiert.
Rezeptverwaltung	Ja	Adapterkomponente für unsere Gerichtsdatenkomponente, die die Lesevorgänge zur Verfügung stellt und gleichzeitig bei Ausfällen als „Anti-Corruption-Layer“ fungiert.

2.3 Teilaufgabe 3: Komponentendiagramm

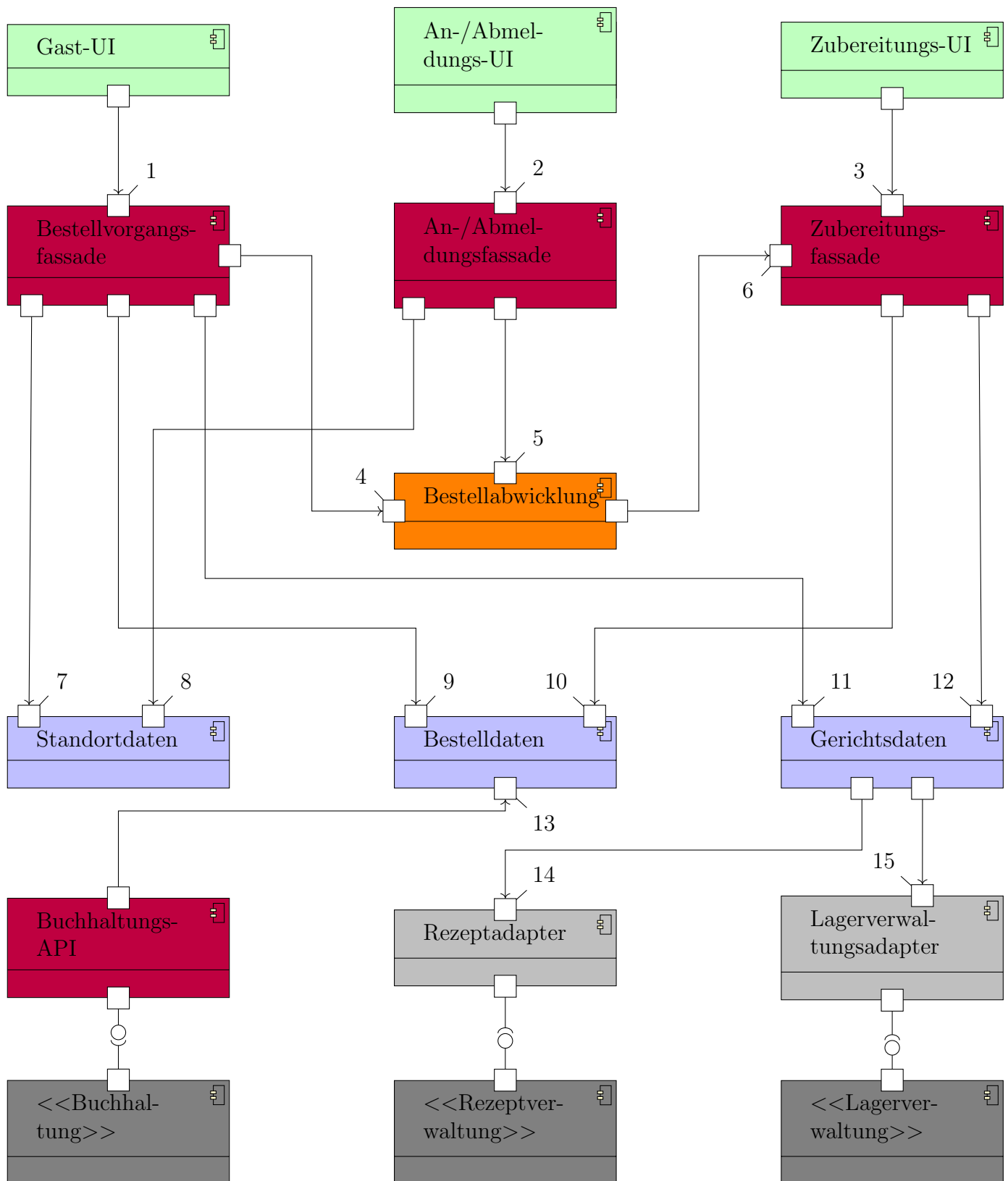


Abbildung 8: Komponentendiagramm

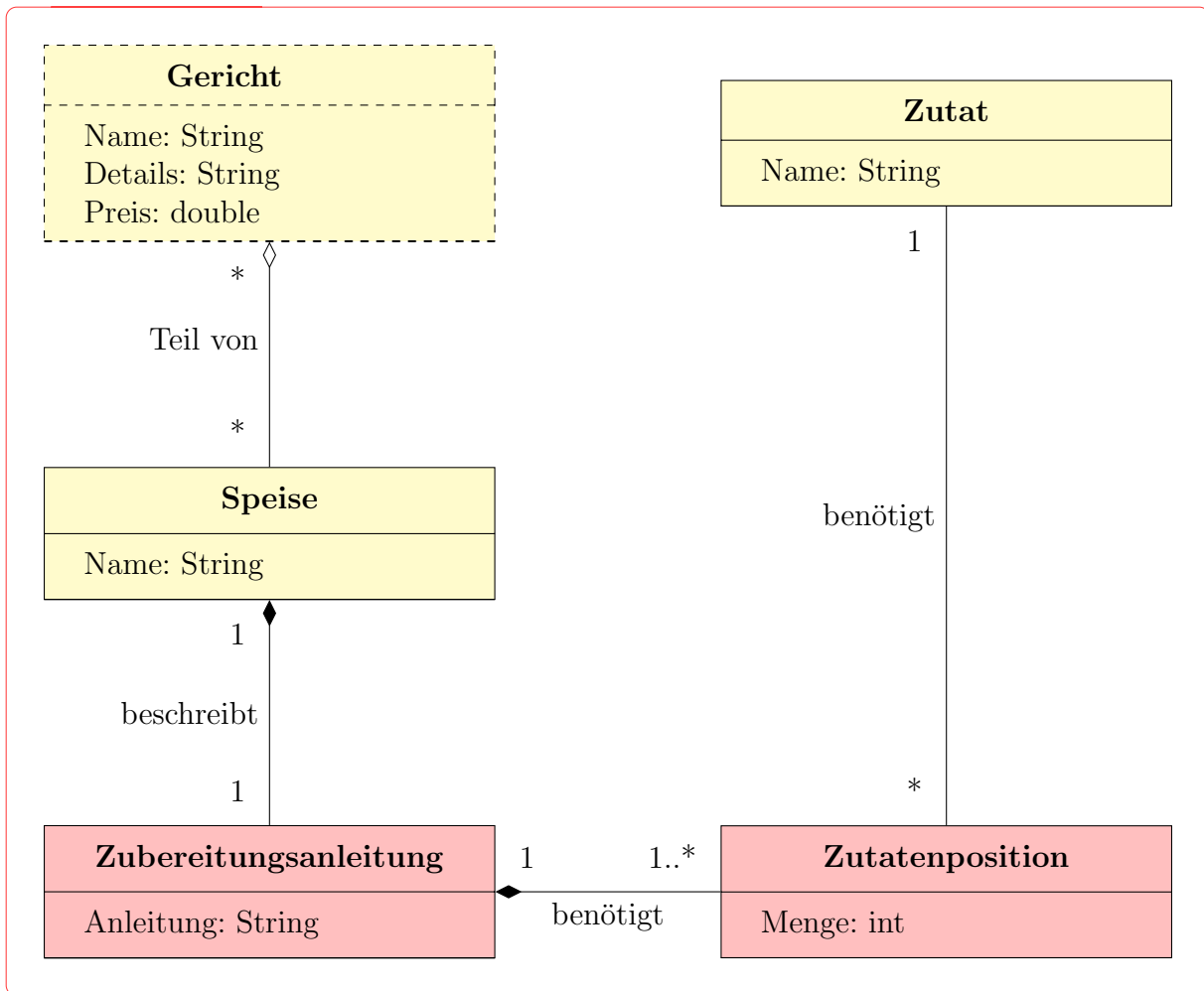
-  : Dialogkomponente
-  : Fassadenkomponente
-  : Datenkomponente
-  : Logikkomponente
-  : Adapterkomponente
-  : Umsystem

Portbeschriftungen

- | | | | | |
|-----------------------------------|---|--------------------------------|---|--|
| 1. Gast-UI | → | Bestellvorgangsfassade | : | Bestellvorgang beginnen. |
| 2. An-/Abmeldungs-UI | → | An-/Abmeldungsfassade | : | Am Arbeitsplatz an-/abmelden. |
| 3. Zubereitungs-UI | → | Zubereitungsfassade | : | Zubereitung beginnen. |
| 4. Bestellvorgangsfassade | → | Bestellabwicklung | : | Bestellung aufgeben. |
| 5. An-/Abmeldefassade | → | Bestellabwicklung | : | In Vergabewarteschlange aufnehmen. |
| 6. Bestellabwicklung | → | Zubereitungsfassade | : | Bestellung zubereiten. |
| 7. Bestellvorgangsfassade | → | Standortdaten | : | Sitzplatz abfragen. |
| 8. An-/Abmeldungsfassade | → | Standortdaten | : | Arbeitsplatz abfragen. |
| 9. Bestellvorgangsfassade | → | Bestelldaten | : | Bestellung speichern. |
| 10. Zubereitungsfassade | → | Bestelldaten | : | Zuzubereitende Bestellung abfragen. |
| 11. Bestellvorgangsfassade | → | Gerichtsdaten | : | Speisekarte und Gerichte abfragen. |
| 12. Zubereitungsfassade | → | Gerichtsdaten | : | Zubereitungsanleitung abfragen. |
| 13. Buchhaltungs-API | → | Bestelldaten | : | Bestellungen abfragen. |
| 14. Gerichtsdaten | → | Rezeptadapter | : | Daten aus Subsystem Rezeptverwaltung holen. |
| 15. Gerichtsdaten | → | Lagerverwaltungsadapter | : | Daten aus Subsystem Lagerverwaltung holen und synchronisieren. |

3 Meilenstein 3 – Spezifikation, Implementierung und Demo eines REST-API

3.1 Teilaufgabe 1: Festlegen von Aggregates



- : Value Object
- : Entity
- : Aggregate
- : Aggregate Root

Abbildung 9: Aggregates

Wir sind der Meinung, dass sich die Datenobjekte Gericht, Speise, Zubereitungsanleitung, Zutatenposition und Zutat als ein Aggregate mit Gericht als Aggregate Root zusammenfassen lassen, da keine Referenzen auf innere Entities existieren und ein fachlicher Zusammenhang besteht, da ein Gericht aus Speisen besteht, Speisen eine Zubereitungsanleitung haben und diese wiederum Zutatenpositionen, die auf Zutaten verweisen, ergibt sich hier ein enges fachliches Geflecht. Außerdem ist es so, dass wir in Meilenstein 2 alle diese Objekte in der Datenkomponente Gerichtsdaten (vgl. 2.2.1) zusammengefasst haben, weshalb wir uns überlegt haben, dass das Aggregate durchaus deckungsgleich sein könnte.

Eine mögliche Invariante wäre, wenn *Gericht.name* eine Kombination von den zugehörigen Speisen

wäre. Als Beispiel hierfür: *Gericht.name*: „Schnitzel mit Pommes und Salat“. Daraus lassen sich die Speisen Schnitzel, Pommes und Salat ableiten.

3.2 Teilaufgabe 2: Design des REST-API

Für unser REST-API verwenden wir folgenden Ausschnitt aus unserem Klassendiagramm aus Meilenstein 1:

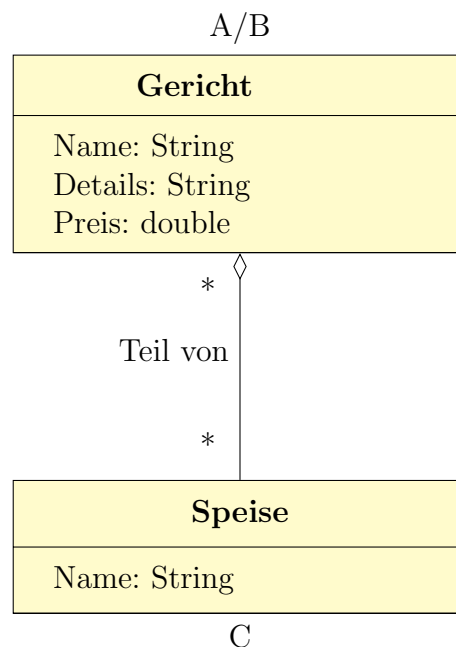


Abbildung 10: Ausschnitt Klassendiagramm für REST-API

Erläuterung

Diese Schnittstelle würde so in unserem Subsystem nicht implementiert, da die verwendeten Geschäftsobjekte nicht in unser Subsystem gehören und wir sie deshalb selbst über Schnittstellen aus anderen Subsystemen beziehen. Wir bieten nur eine Schnittstelle für das Geschäftsobjekt Bestellung für die Buchhaltung an (vgl. 2.3) und mit nur einem Geschäftsobjekt lässt sich das angegebene Szenario nicht durchführen, weshalb wir den obigen Ausschnitt verwenden.

Szen.-Nr.	URI	HTTP Verb	Request-Body	Ressource und Aktion
A1, BC1, BC4, BC7	/gerichte	POST, GET	Nur bei POST: { name=..., details=..., preis=...}	Neues Gericht anlegen, alle Gerichte ausgeben.
A2, A4	/gerichte?search= preis>{wert}	GET		Alle Gerichte <i>a</i> ausgeben, für die <i>a.preis > wert</i> gilt.
A3	/gerichte/{gericht_id}/ preis	PUT	{wert}	Preis von Gericht <i>a</i> mit <i>a.gericht_id</i> = {gericht_id} auf {wert} setzen.
A6	/gerichte/{gericht_id}	GET		Ein bestimmtes Gericht über die Id abfragen. 404 wird geworfen, falls Ge- richt nicht vorhanden.
BC2	/speisen	POST, GET	Nur bei POST: {name=...}	Neue Speise anlegen. Alle Speisen ausgeben.
BC3, BC6	/gerichte/{gericht_id}/ speisen/{speise_id}	PUT, DELETE		Speise einem Gericht hin- zufügen oder löschen.

Erläuterung

Bei Szenario BC3 und BC6 haben wir uns entschieden, dass die Beziehung zwischen einer Instanz von Gericht und einer Instanz von Speise über /gerichte/{gericht_id}/speisen hinzugefügt oder gelöscht werden kann. Man hätte dies auch über /speisen/{speisen_id}/gerichte tun können, was wir jedoch für unübersichtlicher und nicht so naheliegend wie unsere Variante gehalten haben.

3.3 Teilaufgabe 3: Implementierung in Spring Data JPA / Web MVC

3.3.1 Code-Listing

GerichtRestController

```
// BC4,BC7: Alle Gerichte ausgeben.
@GetMapping
public ResponseEntity<?> getAllGerichte(@RequestParam(value="
search", required = false) String query) {
    if(query == null)
        return ResponseEntity.ok().body(gerichtRepository.findAll());
;

    // query specified
    else {

        // Nur das Suchen nach Gerichten, mit einem Preis hoeher
        // einem bestimmten Wert wird implementiert.
        // Da wir fuer die Aufgabe nur die eine Option brauchen.
        try {

            if(!query.substring(0, 6).equalsIgnoreCase("preis>"))
                throw new Exception("Der erste Teil des Strings muss '
preis>' sein");

            String preisStr = query.substring(6);
            double preis = Double.parseDouble(preisStr);

            return ResponseEntity.ok().body(gerichtRepository.
findByPreisGreaterThan(preis));

        }
        // fange alle Exceptions auf die Eintreten koennen und gebe
        // einfach BadRequest zurueck
        catch(Exception e){
            return ResponseEntity.badRequest().build();
        }
    }
}

// A6: ein einzelnes Gericht ausgeben
@GetMapping("/{id}")
public ResponseEntity<?> getKundeById(@PathVariable("id") int id )
{
    Gericht g = gerichtRepository.findOne(id);
    // werfe 404 wenn Gericht nicht existiert
}
```

```

    if ( g == null ) return ResponseEntity.notFound().build();
    // gib Gericht zurueck
    else return ResponseEntity.ok().body(g);
}

// A5: Ein Gericht loeschen
@DeleteMapping("/{id}")
public ResponseEntity<?> deleteGericht(@PathVariable("id") int
id) {
    // gib leeren 200 zurueck, falls loeschen erfolgreich
    if ( gerichtRepository.exists(id) ) {
        // eigentliches loeschen
        gerichtRepository.delete(id);
        return ResponseEntity.ok().build();
    }
    // 404
    else return ResponseEntity.notFound().build();
}

// A1,BC1: Ein Gericht neu anlegen
@PostMapping
ResponseEntity<?> add( @RequestBody Gericht input ) {
    // lege neues Gericht mit Repository der Gerichtsentitaet an
    Gericht g = gerichtRepository.save(input);
    // location des neuen Gerichts
    URI location = ServletUriComponentsBuilder.
fromCurrentRequestUri()
    .path("/{id}").buildAndExpand( g.getId() ).toUri();
    // 201 mit der location im Header und als Body das
    // neue Gericht
    return ResponseEntity.created( location ).body( g );
}

// A3: Den Preis eines Gerichts aendern
@PutMapping("/{id}/preis")
ResponseEntity<?> change( @PathVariable("id") int id, @RequestBody
double preis) {
    Gericht g = gerichtRepository.findOne(id);
    // 404
    if ( g == null ) return ResponseEntity.notFound().build();
    else {
        // neuen Preis speichern
        g.setPreis(preis);
    }
}

```

```

gerichtRepository.save(g);
// 200 mit dem neuen Gericht im Body
return ResponseEntity.ok().body(g);
}
}

// BC3,BC6: Speisen einem Gericht hinzufuegen
@PutMapping("/{gericht_id}/speisen/{speise_id}")
ResponseEntity<?> addSpeise(@PathVariable("gericht_id") int
gericht_id, @PathVariable("speise_id") int speise_id) {
    // als erstes wird das Gericht mit gericht_id aus der
    Datenbank geholt
    Gericht g = gerichtRepository.findOne(gericht_id);
    // 404 wenn das Gericht nicht existiert
    if(g == null) return ResponseEntity.notFound().build();
    // dann die Speise mit speise_id
    Speise s = speiseRepository.findOne(speise_id);
    // auch hier 404 falls Speise nicht vorhanden
    if(s == null) return ResponseEntity.notFound().build();
    // Referenzen fuer beide setzen und beide speichern
    g.addSpeise(s);
    s.addGericht(g);
    gerichtRepository.save(g);
    speiseRepository.save(s);
    // 200 mit dem upgedatetem Gericht
    return ResponseEntity.ok().body(g);
}

// BC6: Speise (Verbindung) fuer ein Gericht loeschen
@DeleteMapping("/{gericht_id}/speisen/{speise_id}")
ResponseEntity<?> removeSpeise(@PathVariable("gericht_id") int
gericht_id, @PathVariable("speise_id") int speise_id) {
    // vgl. addSpeise
    Gericht g = gerichtRepository.findOne(gericht_id);
    if(g == null) return ResponseEntity.notFound().build();
    Speise s = speiseRepository.findOne(speise_id);
    if(s == null) return ResponseEntity.notFound().build();
    g.removeSpeise(s);
    s.removeGericht(g);
    gerichtRepository.save(g);
    speiseRepository.save(s);
    return ResponseEntity.ok().body(g);
}

```

SpeiseRestController

```
// BC5: Alle Speisen auslesen
@GetMapping
public List<Speise> getAllSpeisen() {
    return (List<Speise>) speiseRepository.findAll();
}

// BC2: Einen Speise neu anlegen
@PostMapping
ResponseBody<?> add( @RequestBody Speise input ) {
    // Die Zubereitungsanleitung erst mal leer lassen
    Speise s = new Speise(input.getName(), null);
    // speichern
    speiseRepository.save(s);
    // location der neuen Speise
    URI location = ServletUriComponentsBuilder.
fromCurrentRequestUri()
    .path("/{id}").buildAndExpand( s.getId() ).toUri();
    // 201 mit der location im Header und als Body die
    // neue Speise
    return ResponseEntity.created( location ).body( s );
}
```

3.3.2 Custom JSON Serializer

Um das Problem der Endlosserialisierung bei der m:n- Beziehung, die bei uns zwischen Gericht und Speise vorliegt, zu lösen, wurde vorgeschlagen die Annotation „@JsonIdentityInfo“ zu benutzen, mit der man die jeweiligen Klassen annotieren muss.

Die Serialisierung funktioniert dann so, dass beim ersten Vorkommen einer Entität diese ganz serialisiert wird, beim zweiten Vorkommen allerdings nur über die ID referenziert wird.

Beispielausgabe der Gerichte

```
[
  {
    "id": 2,
    "name": "Schnitzel mit Pommes",
    "details": "Der grandiose Klassiker!",
    "preis": 12.5,
    "speisen": [
      {
        "id": 2,
        "name": "Schnitzel",
        "gerichte": [
          2
        ],
      },
    ],
  },
]
```



```

    },
    {
        "id": 1,
        "name": "Pommes",
        "gerichte": [
            2,
            {
                "id": 3,
                "name": "Grosse Portion Pommes",
                "details": "Lecker fettig!",
                "preis": 6,
                "speisen": [
                    1
                ]
            }
        ],
    }
]
},
3
]

```

Aus unserer Sicht ist die schlecht lesbare Ausgabe ein Nachteil.

Ein Beispiel für die schlechte Lesbarkeit wäre unserer Meinung nach die „3“ ganz unten (vorletzte Zeile, s.o.), welche für das Gericht mit der ID 3 (Grosse Portion Pommes) steht, welches bereits zuvor aufgeführt wurde.

Eine andere Lösung, die wir gefunden haben ist ein Custom-Serialisierer.

Hierbei können wir dann selbst bestimmen, wie die Serialisierung funktionieren soll.

Hier haben wir uns dafür entschieden nur einen Serialisierer für die Gerichte, auf den die Speisen verweisen, zu programmieren, der anstatt die Gerichte in die „Tiefe“ zu serialisieren, nur die IDs der Gerichte auflistet. Damit wäre dann das Endlosschleifen-Problem gelöst.

Der Code dazu befindet sich in:

Relevanter Ausschnitt unseres Custom Serializers

```

@Override
public void serialize(
    Set<Gericht> gerichte,
    JsonGenerator generator,
    SerializerProvider provider)
    throws IOException, JsonProcessingException {

    List<Integer> ids = new ArrayList<>();
    for (Gericht g : gerichte) {
        ids.add(g.getId());
    }
    generator.writeObject(ids);
}

```

```

    }

}

```

Um den Serializer für die Gerichte einzusetzen, muss eine entsprechende Annotation mit dem Custom-Serialisierer beim der Getter-Methode der Gerichte in der Klasse Speise angebracht werden:

```

@JsonProperty
@JsonSerialize(using = CustomSetSerializer.class)
public Set<Gericht> getGerichte() {
    return Collections.unmodifiableSet(gerichte);
}

```

Beispielausgabe der Gerichte mit Custom Serializer

```

[
  {
    "id":2,
    "name":"Schnitzel mit Pommes",
    "details":"Der grandiose Klassiker!",
    "preis":12.5,
    "speisen":[
      {
        "id":2,
        "name":"Schnitzel",
        "gerichte":[
          2
        ]
      },
      {
        "id":1,
        "name":"Pommes",
        "gerichte":[
          3,
          2
        ]
      }
    ]
  },
  {
    "id":3,
    "name":"Grosse Portion Pommes",
    "details":"Lecker fettig!",
    "preis":6,
    "speisen":[

```

```
[
  {
    "id": 1,
    "name": "Pommes",
    "gerichte": [
      3,
      2
    ]
  }
]
```

Die Ausgabe ist, unserer Meinung nach, viel besser lesbar und es ist einfacher mit ihr zu arbeiten.

Bei den Speisen sehen wir die Gerichte nun als eine Liste von IDs.

Nachteil zu der vorigen Lösung ist, dass Speisen mehrfach aufgeführt werden, wie z.B. „Pommes“, weshalb die Ausgabe mit dem Custom Serializer nicht redundanzfrei ist.

Trotzdem ist unserer Meinung nach die Ausgabe des Custom Serializers besser als die Ausgabe von „@JsonIdentityInfo“, da aus unserer Sicht eine wohl- formatierte Ausgabe wichtiger ist als Redundanzfreiheit.

3.3.3 Nachweis der Lauffähigkeit

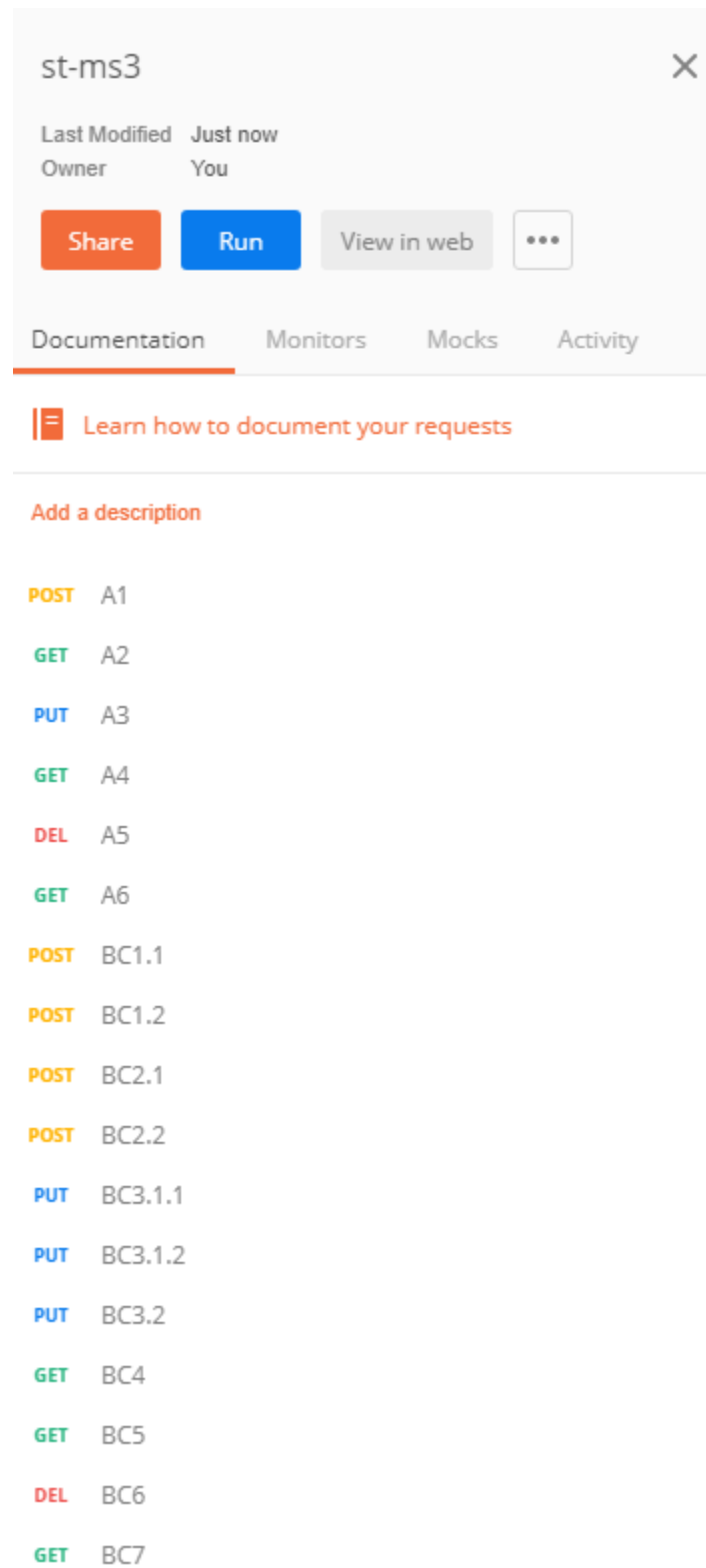


Abbildung 11: Postman Collection

<div> <div>18 PASSED</div> <div>0 FAILED</div> <div> st-ms3 33 mins ago </div> <div>No Environment</div> </div>		
<div> <div>Back</div> <div>1</div> </div>		
<div> <div> <div></div> <div></div> </div> <div></div> <div></div> </div>	<div> <div></div> <div>POST</div> <div>A1</div> </div>	<div>✓</div>
	<div> <div></div> <div>GET</div> <div>A2</div> </div>	<div>✓</div>
	<div> <div></div> <div>PUT</div> <div>A3</div> </div>	<div>✓</div>
	<div> <div></div> <div>GET</div> <div>A4</div> </div>	<div>✓</div>
	<div> <div></div> <div>DELETE</div> <div>A5</div> </div>	<div>✓</div>
	<div> <div></div> <div>GET</div> <div>A6</div> </div>	<div>✓</div>
	<div> <div></div> <div>POST</div> <div>BC1.1</div> </div>	<div>✓</div>
	<div> <div></div> <div>POST</div> <div>BC1.2</div> </div>	<div>✓</div>
	<div> <div></div> <div>POST</div> <div>BC2.1</div> </div>	<div>✓</div>
	<div> <div></div> <div>POST</div> <div>BC2.2</div> </div>	<div>✓</div>
	<div> <div></div> <div>PUT</div> <div>BC3.1.1</div> </div>	<div>✓</div>
	<div> <div></div> <div>PUT</div> <div>BC3.1.2</div> </div>	<div>✓</div>
	<div> <div></div> <div>PUT</div> <div>BC3.2</div> </div>	<div>✓</div>
	<div> <div></div> <div>GET</div> <div>BC4</div> </div>	<div>✓</div>
	<div> <div></div> <div>GET</div> <div>BC5</div> </div>	<div>✓</div>
	<div> <div></div> <div>DELETE</div> <div>BC6</div> </div>	<div>✓</div>
	<div> <div></div> <div>GET</div> <div>BC7</div> </div>	<div>✓</div>

Abbildung 12: Postman Test

4 Meilenstein 4 – Microservices

Da es auf unserer Seite zu Missverständnissen bezüglich der Entities der Context Map gekommen ist, haben wir die Context Map mit den Entities aus unserem logischen Datenmodell anstelle der des fachlichen Datenmodells gefüllt und unsere weitere Arbeit auf dieser Map durchgeführt, was, unserer Meinung nach, sowohl für die Aggregates als auch für das Modell besser ist, da es näher an der tatsächlichen Implementierung ist und somit auch der Vergleich zum monolithischen Modell aus Meilenstein 2 (vgl. 2.3) besser möglich macht, da beide Modelle auf den selben Entities (vgl. 2.1.1) beruhen.

Um konsistent mit Meilenstein 2 (Kapitel 2) zu bleiben, gehen wir auch hier von der von uns getätigten Annahme aus, dass es ein Subsystem Buchhaltung gibt.

4.1 Teilaufgabe 1: Context Map (Entities aus unserem fachlichen Datenmodell)

Diese Map und diese Tabelle wurden nachträglich hinzugefügt.

Anmerkung: im Folgenden ist der Ausdruck Subsystem equivalent zum Ausdruck Domäne.

4.1.1 Context Map

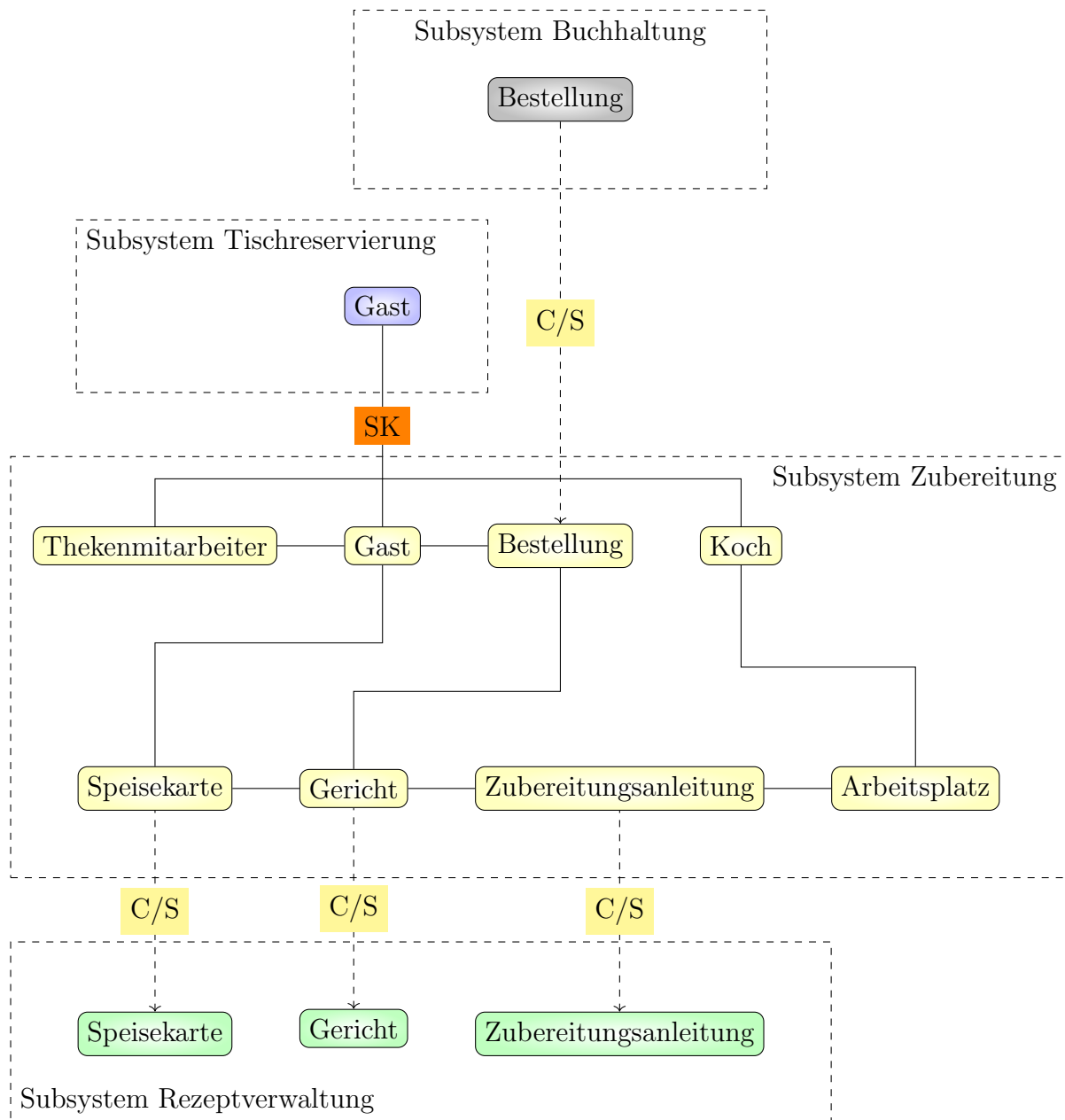


Abbildung 13: Context Map (fachliches Datenmodell)

C/S: Customer / Supplier

SK: Shared Kernel

--->: Customer → Supplier (Pfeilspitze auf Eigentümer gerichtet)

4.1.2 Tabelle der Überlappungstypen

Entity	Überlappung mit anderer Domäne	Überlappungstyp	Begründung
Bestellung	Buchhaltung	Customer / Supplier (unser Subsystem als Eigentümer)	Die Buchhaltung ruft die Bestelungsdaten bei uns ab. Da wir der Eigentümer der Entity sind und wir nicht wissen wie komplex die Anbinung unserer Schnittstelle an die Buchhaltungssoftware (wahrscheinlich proprietäre Anwendung) ist, Separate Ways (der Verzicht auf den Aufruf unserer Schnittstelle auf Seiten der Buchhaltung) allerdings im Kontext unmöglich ist, haben wir uns für Customer / Supplier entschieden.
Gericht	Rezeptverwaltung	Customer / Supplier (Rezeptverwaltung als Eigentümer)	Wir rufen die Gerichte beim Subsystem Rezeptverwaltung ab. Das Subsystem Rezeptverwaltung ist der Eigentümer und wir haben (brauchen) nur lesenden Zugriff auf das Entity Gericht. Customer / Supplier, da wir auf Augenhöhe mit der Rezeptverwaltung sind uns eine enge Zusammenarbeit möglich ist.
Speisekarte	Rezeptverwaltung	Customer / Supplier (Rezeptverwaltung als Eigentümer)	vgl. Entity Gericht.
Zubereitungsanleitung	Rezeptverwaltung	Customer / Supplier (Rezeptverwaltung als Eigentümer)	vgl. Entity Gericht.
Gast	Tischreservierung	Shared Kernel	Gast hat keinen klaren Eigentümer, beide Subsysteme sind gleichberechtigt und eine enge Zusammenarbeit ist möglich, ergo Shared Kernel.

4.2 Teilaufgabe 1: Context Map (Entities aus unserem logischen Datenmodell)

Auf dieser Map und dieser Tabelle beruhen die nachfolgenden Kapitel.

Anmerkung: im Folgenden ist der Ausdruck Subsystem equivalent zum Ausdruck Domäne.

4.2.1 Context Map

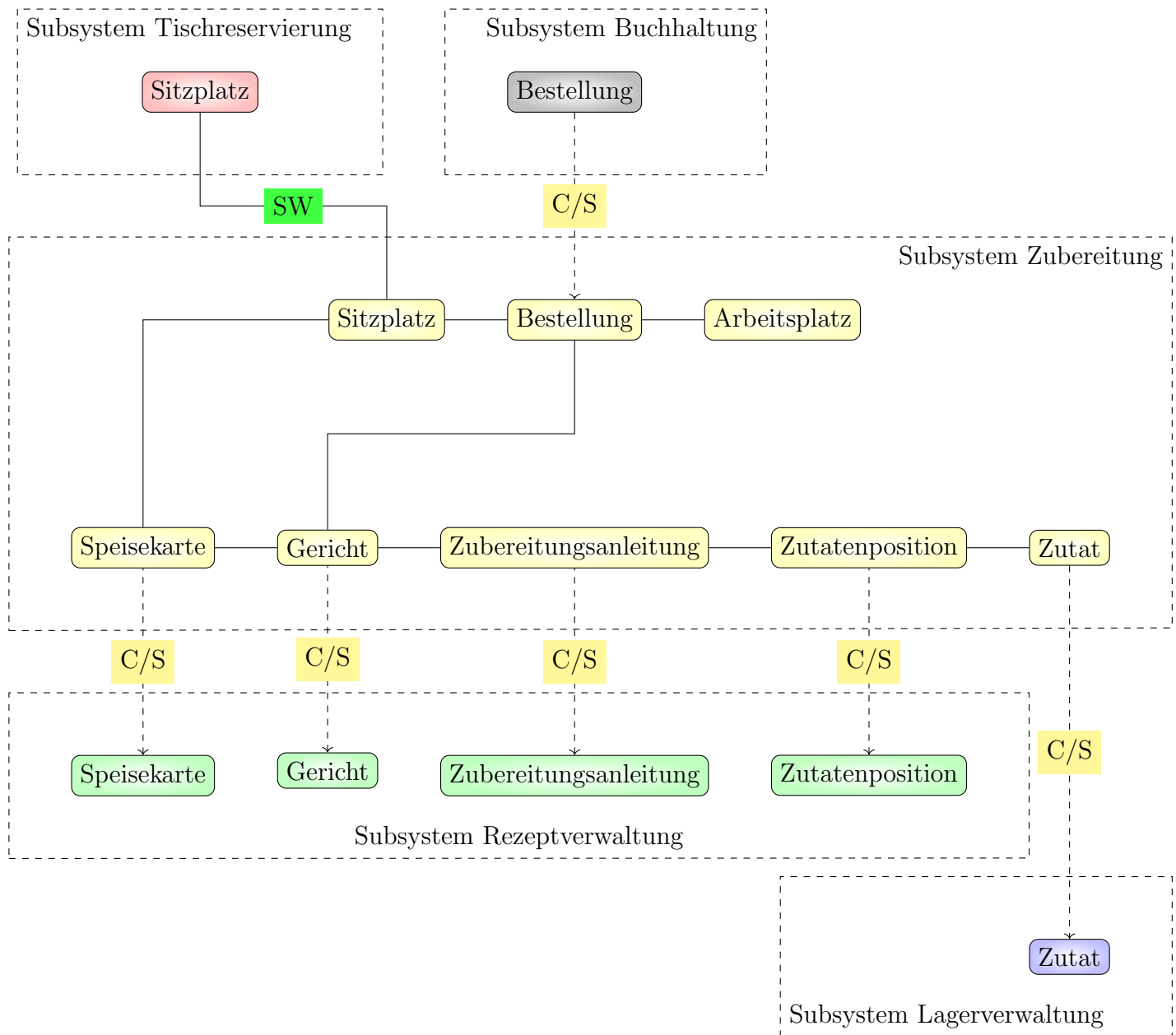


Abbildung 14: Context Map (logisches Datenmodell)

C/S : Customer / Supplier

SW : Separate Ways

--->: Customer → Supplier (Pfeilspitze auf Eigentümer gerichtet)

4.2.2 Tabelle der Überlappungstypen

Entity	Überlappung mit anderer Domäne	Überlappungstyp	Begründung
Bestellung	Buchhaltung	Customer / Supplier (unser Subsystem als Eigentümer)	Die Buchhaltung ruft die Bestelungsdaten bei uns ab. Da wir der Eigentümer der Entity sind und wir nicht wissen wie komplex die Anbindeung unserer Schnittstelle an die Buchhaltungssoftware (wahrscheinlich proprietäre Anwendung) ist, Separate Ways (der Verzicht auf den Aufruf unserer Schnittstelle auf Seiten der Buchhaltung) allerdings im Kontext unmöglich ist, haben wir uns für Customer / Supplier entschieden.
Gericht	Rezeptverwaltung	Customer / Supplier (Rezeptverwaltung als Eigentümer)	Wir rufen die Gerichte beim Subsystem Rezeptverwaltung ab. Das Subsystem Rezeptverwaltung ist der Eigentümer und wir haben (brauchen) nur lesenden Zugriff auf das Entity Gericht. Customer / Supplier, da wir auf Augenhöhe mit der Rezeptverwaltung sind uns eine enge Zusammenarbeit möglich ist.
Speisekarte	Rezeptverwaltung	Customer / Supplier (Rezeptverwaltung als Eigentümer)	vgl. Entity Gericht.
Zubereitungsanleitung	Rezeptverwaltung	Customer / Supplier (Rezeptverwaltung als Eigentümer)	vgl. Entity Gericht.
Zutat	Lagerverwaltung	Customer / Supplier (Lagerverwaltung als Eigentümer)	Zutat ist in unserem Fall einfach die Menge der Zutat, welche im Lager zur Verfügung steht und wird mit der Lagerverwaltung abgeglichen. Zutat verhält sich analog zu Gericht.
Zutatenposition	Rezeptverwaltung	Customer / Supplier (Rezeptverwaltung als Eigentümer)	vgl. Entity Gericht.

Sitzplatz	Tischreservierung	Separate Ways	Aus dem einfachen Grund, dass Sitzplatz bei uns etwas völlig anderes ist, als beim Subsystem Tischreservierung, haben wir hier Separate Ways gewählt. Während bei der Tischreservierung der physikalische Sitzplatz gemeint ist, bezieht sich unsere Definition von Sitzplatz auf die Instanz eines Gast-Clientprozesses, welches auf einem Tablet im Restaurant läuft. Der Gast gibt mit Hilfe dieses Programms seine Bestellungen auf. Da dies zwei völlig unterschiedliche Dinge sind und eigentlich nur der Name der Entities gleich ist, haben wir uns hier für Separate ways entschieden.
-----------	-------------------	---------------	---

4.3 Teilaufgabe 2: Aggregates

Im Folgenden beziehen wir uns unter anderem auf unsere Aggregates aus Kapitel 3.1. Die einzige Unterscheidung zu diesem Aggregate ist, dass wir die redundante Entity Speise (in Kapitel 1.1 dem Klassendiagramm des logischen Datenmodells hinzugefügt, um die Aufgabenstellung zu erfüllen) entfernen und das Attribut *Speise.name* in *Zubereitungsanleitung.name* überführen.

Aggregate Root	Weitere beteiligte Entities	Invarianten	Begründung, dass das ein Aggregate ist
Gericht	Zubereitungsanleitung, Zutatenposition, Zutat	<i>Gericht.name</i> wird aus <i>Zubereitungsanleitung.name</i> zusammengesetzt (Schnitzel, Pommes, Salat \Rightarrow <i>Gericht.name</i> = Schnitzel mit Pommes und Salat)	vgl. 3.1
Bestellung			Entity Bestellung als eigenständiges Aggregate, da es keine sinnvollen Zuordnungen gibt.
Speisekarte			vgl. Aggregate Bestellung.
Arbeitsplatz			Entity aus unserem logischen Datenmodell, ohne Überlappung zu anderen Subsystemen. Alleinstehend, da auch hier, wie beim Aggregate Bestellung keine sinnvollen Zuordnungen existieren.
Sitzplatz			vgl. Aggregate Bestellung.

4.4 Teilaufgabe 3: Microservice-Architektur

4.4.1 Servicetabelle

Service	Bildet ab	Kommentar
Bestellungen	Bestellung	Dient als API Gateway, da unser Subsystem ein Supplier für die <u>Bestelldaten</u> ist.
Gerichte	Gericht (vgl. 3.1)	Adapterservice, da unser Subsystem ein Customer für die <u>Gerichtsdaten</u> ist.
Speisekartenverwaltung	Speisekarte	Adapterservice, da unser Subsystem ein Customer für die <u>Speisekartendaten</u> ist.
Restaurantverwaltung	Sitzplatz, Arbeitsplatz	Service der die Sitz- und Arbeitsplätze der Standorte verwaltet.
Gast-UI		UI-Service für den Gast. Hier wird unserer Meinung nach kein API Gateway benötigt, da das UI nur Daten aus dem Speisekartendaten-Service benötigt und Bestellungen an den Bestelldaten-Service schickt, was weder eine besondere Darstellung der Daten ist, noch als viele verschiedene Aufrufe zu charakterisieren wäre.
Koch-UI		UI-Service für den Koch. Hier wird unserer Meinung nach kein API Gateway benötigt, da nur Daten aus dem Gerichtsdaten-Service abgerufen werden müssen.

4.4.2 Komponentendiagramm

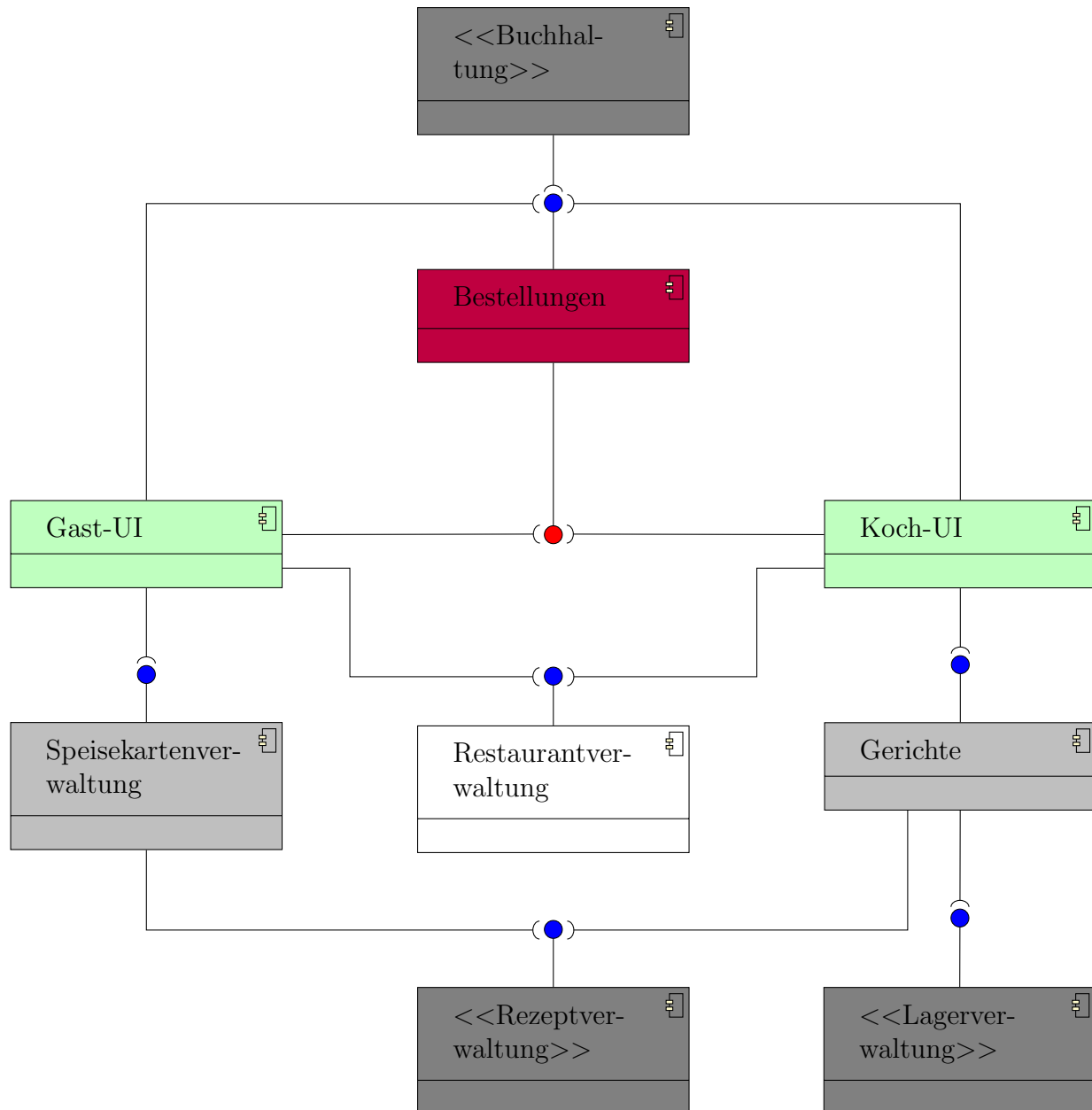








Abbildung 15: Komponentendiagramm

- : Service
- : UI-Service
- : API Gateway
- : Adapterservice
- : Umsystem
- : REST-API

4.4.3 Vergleich monolithisches Modell aus Meilenstein 2 mit Microservice-Modell

Service	Entspricht / bildet ab auf Komponente aus MS 2
Bestellungen	Bestelldaten: Datenkomponente, Buchhaltungs-API: Fassadenkomponente, Bestellabwicklung: Logikkomponente
Gerichte	Gerichtsdaten: Datenkomponente, Rezeptadapter: Adapterkomponente, Lagerverwaltungsadapter: Adapterkomponente
Speisekartenverwaltung	Entspricht Teil der Gerichtsdaten: Datenkomponente, Rezeptadapter: Adapterkomponente
Restaurantverwaltung	Standortdaten: Datenkomponente
Gast-UI	Gast-UI: Dialogkomponente, Bestellvorgangsfassade: Fassadenkomponente
Koch-UI	An-/Abmeldungs-UI: Dialogkomponente, An-/Abmeldungsfassade: Fassadenkomponente, Zubereitungs-UI: Dialogkomponente, Zubereitungsfassade: Fassadenkomponente

Erläuterung

Anstelle der Bestellabwicklung, eine Logikkomponente unseres monolithischen Modells (vgl. 2.3), die für die Vergabe von Bestellungen an die angemeldeten Arbeitsplätze und die Mitteilungsversendung an den Gast (dass seine Bestellung abholbereit ist) verantwortlich war haben wir eine Event-Schnittstelle an unseren Bestellungen-Service angebunden, welche Gast- und Koch-UI benachrichtigt, sobald entweder eine Bestellung eingeht und diese bearbeitet werden soll (Event für das Koch-UI) oder eine Bestellung fertig zubereitet und abholbereit ist (Event für das Gast-UI).

Was sind aus Ihrer Sicht die Vorteile der einen oder anderen Architektur?

Unserer Meinung nach hat die Microservice-Architektur den Vorteil der Übersichtlichkeit und der losen Kopplung. Allein vom Modell wirkt die Microservice-Architektur überschaubarer (zumindest im Rahmen unseres Subsystems). Des weiteren gefällt uns die klare Definition von Schnittstellen und deren Implementierung über Netzwerkprotokolle wie HTTP.

Hier sehen wir allerdings auch einen Vorteil der monolithischen Anwendung, da diese nicht auf die eher allgemein gehaltenen Schnittstellen angewiesen ist und so, unserer Meinung nach, mehr Freiraum bietet.

Welche Architektur würden Sie umsetzen, wenn Sie das als Informatikprojekt implementieren müssten und wieso?

Die Microservice-Architektur, da sie, allein vom Modell, übersichtlicher wirkt. Die lose Kopplung der Services und die einheitlichen Schnittstellen gefallen uns sehr gut. Unser Subsystem ist vom Implementierungsaufwand, unserer Meinung nach, eher gering, weshalb die Menge an Komponenten des monolithischen Modells abschreckend wirkt.

Unabhängig davon erfreut sich die Microservice-Architektur momentan großer Beliebtheit, was wir auch als Vorteil sehen.