Technologische Fachoberschule „Max Valier" Bozen

Fachrichtung Industrieinformatik

Klasse 5A

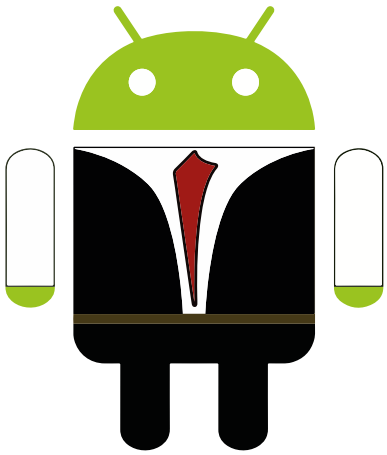
Dokumentation zum Maturaprojekt

Free Orderman System

Autor:
Florian Mahlknecht

Tutor:
Karl Lunger

10. Juni 2014



Vorwort

Am Ende der Sommerferien 2013 stellte mir mein Schulkamerad und Freund *Alex Mazzon* seine Vision eines *freien Kassensystems* für den Gastronomiebereich vor und bat mich um Zusammenarbeit. Vor allem aufgrund der Tatsache, dass Teamarbeit in der Informatik sowie auch in unzähligen anderen Bereichen von elementarer Wichtigkeit ist, habe ich nach reiflicher Überlegung meine eigenen Ideen zugunsten dieses größeren Projektes fallen lassen.

Vermutlich ist nicht zuletzt das gute kollegiale Verhältnis, welches wir pflegen, das, was zum Erfolg geführt hat. Durch zwei verschiedene Meinungen konnten wegweisende Entscheidungen wohl überlegter und objektiver vollzogen werden. Auch in schwierigen Momenten haben wir durch unsere Zusammenarbeit zentrale Probleme gemeistert und neue Hoffnung geschöpft.

Beide zählen wir Arbeitserfahrung im Servicebereich der Gastronomie zu unseren Lebensläufen, wodurch wir einen realitätsnahen Blick auf die Probleme und Anforderungen im Umgang mit Kassensystemen als Stärke vorweisen können. *Alex* hat in seiner Arbeit effektiv mit automatisierten Lösungen gearbeitet und mir wurden in einem kleineren Betrieb die Probleme der Anschaffungskosten sowie auch die reizvollen Vorteile, welche die besagten Systeme mit sich bringen würden, klar.

Das Schönste an Projekten ist wohl der Eifer und die Motivation, die – nicht ständig aber dennoch – aufkommen und den enormen Arbeitsaufwand dem Auge entziehen. Nach jedem Teilerfolg blickt man noch weiter über die unzähligen Arbeitsstunden hinweg. Darüber hinaus kommt eine ganze Flut an neu Erlerntem und neuen Erfahrungen hinzu. Von der Planung bis zur Durchführung haben sich bei diesem Projekt Themenfelder quer durch die ganze Informatik erschlossen, wodurch es sich als würdige Abschlussarbeit erwiesen hat.

Zusammenfassung

Immer öfters werden automatisierte Kassensysteme in der Gastronomie genutzt, um die ausschlaggebende Service-Qualität zu steigern. Durch die technischen Hilfsmittel wird nicht nur verhindert, dass Bestelltes vergessen oder unleserliche Einträge nicht verrechnet werden, sondern auch der Arbeitsablauf wird entscheidend verbessert und dadurch die Effizienz im Einklang mit einem optimierten Personaleinsatz gesteigert.

Die Anschaffungskosten für handelsübliche Systeme liegen für viele kleinere Betriebe jedoch weit über deren Möglichkeiten. Entscheidende Erfolgsfaktoren werden diesen somit unterschlagen. In einer Zeit, in der junge Generationen flächendeckend mit Smartphones und anderen Taschencomputern ausgestattet sind, könnte der Einsatz dieser Geräte eine kostengünstige Alternative bieten, allerdings scheitert es an der fehlenden *Software*.

Das Free Orderman System macht sich somit zum Ziel, *vorhandene* Infrastrukturen wie Smartphones und WLAN-Netze zu nutzen, um so die Vorteile eines Bestellsystems auch kleineren Pubs, Restaurants und Bars zugänglich zu machen.

Die große Herausforderung lag dabei in der Vernetzung autonomer Systeme und insbesondere in der Verteilung der Informationen über Tische, Bestellungen und Produkte. Dafür wurde ein Protokoll entwickelt, welches die Verbreitung der Daten von einem zentral stationierten Server gemäß dem effizienten Model-View-Prinzip ermöglicht.

Nach sechs Monaten Entwicklungszeit ist im Rahmen dieses Maturaprojektes eine einsatzsichere, *freie* Lösung entstanden. In einem Gastronomiebetrieb wurde das System bereits erfolgreich im Tagesgeschäft getestet und soll dort künftig als solide Dauerlösung eingesetzt werden.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Wie allseits bekannt ist in Bars und Restaurants der Kunde König. So sollte die freundlich entgegengenommene Bestellung *schnell*, *unkompliziert* und *sicher* verarbeitet werden. Das Erfüllen dieser Forderung ist mitunter die größte Herausforderung in der Gastronomie. Häufig kommen Gäste zur selben Zeit im Restaurant an, sollten möglichst zusammen bedient werden und Bestelltes sollte nicht auf sich warten lassen. Es ist somit wenig verwunderlich, dass immer öfters von Block und Feder auf computergestützte Lösungen zurückgegriffen wird.

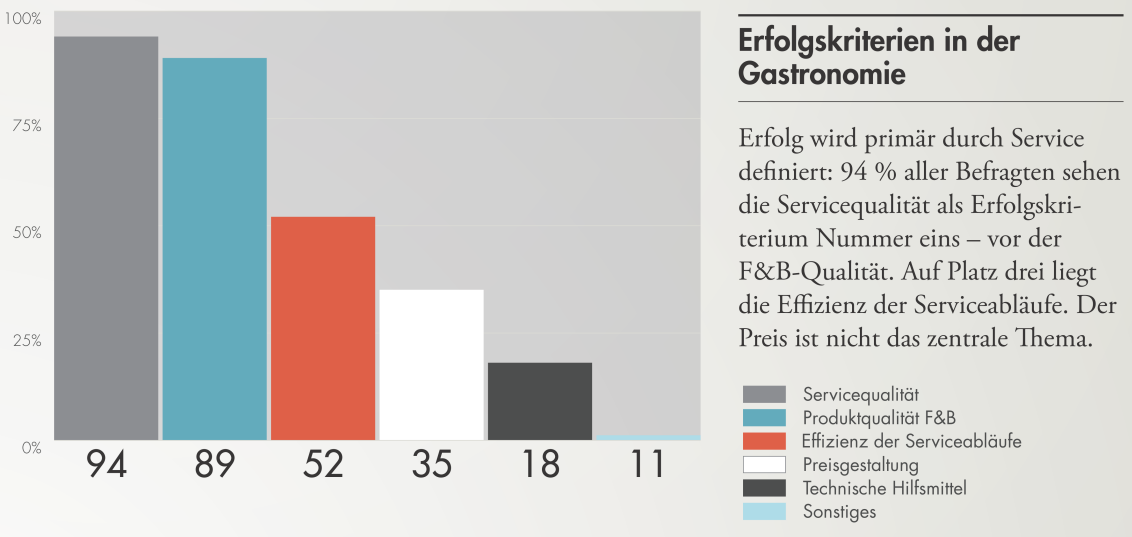


Abbildung 1.1: Orderman Gastrostudie 2012 – Erfolgsfaktoren in der Gastronomie[1]

Wie man aus dem Ausschnitt der von Orderman® beauftragten Gastrostudie in Abbildung 1.1 ablesen kann, zählt Service-Qualität zum primären Erfolgsfaktor in der Gastronomie.[2] Die Optimierung dieser ist somit das A und O für den Fortschritt des Betriebes.

Die 352 befragten europäischen Gastronomen ergriffen im Zuge dieser wichtigen Optimierungen unterschiedliche Maßnahmen, so z. B. Schulung der Mitarbeiter, Ablaufoptimierung, Koordination Küche & Service und *technische Hilfsmittel*. Als wichtigsten Vertreter der »technischen Hilfsmittel« nannten 67% ein „Funkboniersystem“ und damit das Anliegen dieses Projektes.

[1] siehe [GaSt], Seite 7

[2] Die Studie wurde von Ploner Hospitality Consulting durchgeführt. Die Objektivität und Qualität hinsichtlich der unvoreingenommenen Ergebnisse dieser Studie soll in diesem Kontext unbeachtet bleiben, zumal nur die Relevanz von Kassensystemen erörtert werden soll

Mit Hilfe der automatisierten Kassensysteme wird die Geschwindigkeit im Service zweifelsohne erhöht. Beim Abschicken der Bestellung können die Arbeitskollegen in Bar und Küche schon mit der Zubereitung dieser beginnen und weitere Bestellungen aufgenommen werden. Auch bei der Abrechnung bietet das System bedeutende Geschwindigkeitsvorteile, sowie mehr Sicherheit für Kunden und Betreiber.

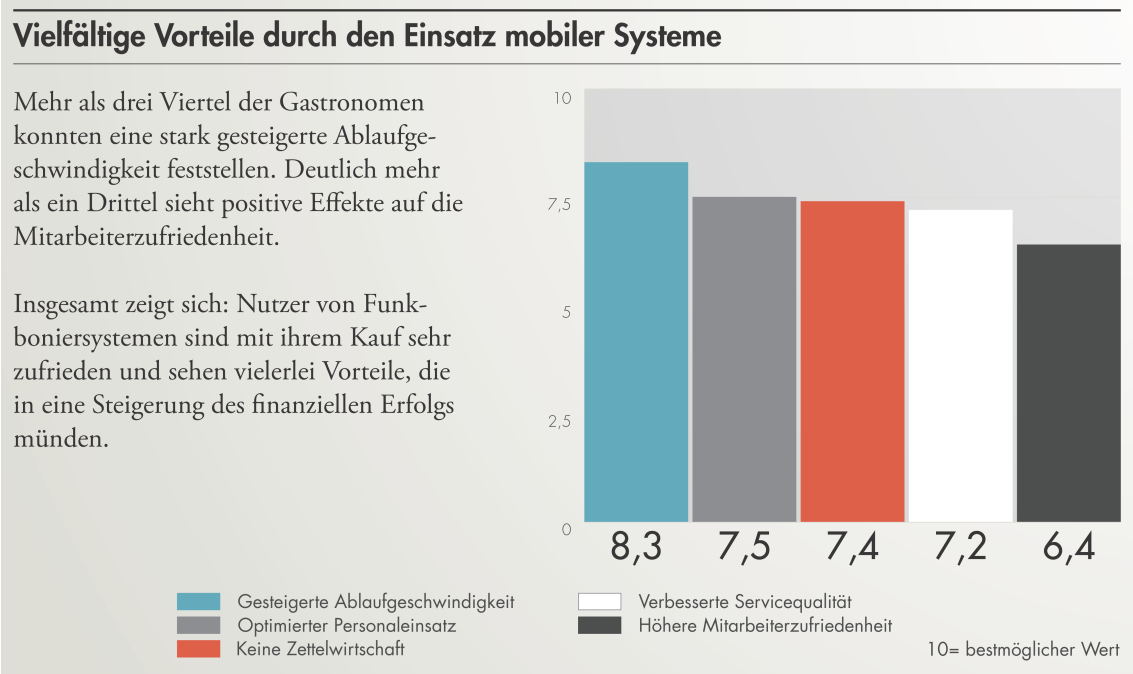


Abbildung 1.2: Orderman® Gastrostudie 2012 – Vorteile mobiler Systeme

Diese Vorteile schlugen sich auch in den Ergebnissen der Studie nieder, wie man im 2. Auschnitt (Abbildung 1.2) erkennen kann.

Zusammenfassend greifen diese Systeme an ausschlaggebender Stelle im Arbeitsfluss eines Gastronomiebetriebes ein und können Produktivität, Geschwindigkeit und Effizienz maßgeblich steigern.

1.1 Idee

Wenn man sich die Anschaffungskosten für ein handelsübliches Kassensystem zu Leibe führt, so wird schnell klar, dass dies für kleinere Betriebe wenig rentabel ist. Somit werden diesen Betrieben u. U. entscheidende Wettbewerbsvorteile vorenthalten, obwohl durch die rasanten Entwicklungen in den letzten Jahren inzwischen weit verbreitete elektronische Geräte die notwendigen Voraussetzungen schon mitbringen.

So basiert die Idee nun darauf, *vorhandene* Geräte und Infrastrukturen durch die geeignete *Software* zu verbinden und diese mithilfe offener und frei zugänglicher Spezifikationen für jedermann zugänglich zu machen.

1.2 Motivation

In einer Zeit, in der immer mehr Leute mit berührungsempfindlichen mobilen Endgeräten alias Smartphones ausgestattet sind und WLAN-Netze die Straßen erobern, fehlt für den spezifischen Einsatz der Geräte häufig nur noch die passende *Software*. Nicht von unge-

fähr überfluten Applikationen für jedes erdenkliche Einsatzgebiet die »App«-Plattformen[3] mobiler Betriebssysteme.

Solch ein System ermöglicht es Betrieben, die vor den Anschaffungskosten eines Kassensystems zurückschrecken, bestehende Infrastrukturen zu nutzen und dadurch ihre Produktivität ohne große finanzielle Hürden zu steigern.

Eine weiterer wichtiger Ansporn ist das Softwareangebot der *Open Source Community*[4] um ein Projekt mit *leicht zugänglicher* Dokumentation und Implementation zu erweitern.

1.3 Ziel

Erklärtes Ziel ist es somit eine *offene*[5] Implementierung eines Kassensystems auszuarbeiten und volle Funktionstüchtigkeit zu erreichen. Dabei wird eine Komplettlösung für Bars, Restaurants und Pubs mit Bestellverarbeitung und Zahlungssystem angestrebt. Als Abschluss soll das System im Produktiveinsatz in einem Gastronomiebetrieb getestet werden.

1.4 Übersicht

Die vorliegende Arbeit ist in mehrere Kapitel unterteilt. Im Kapitel Kassensysteme sollen zunächst vorhandene Lösungen *analysiert*, und allgemeine Anforderungen an die Systeme ausgearbeitet werden. Es soll geklärt werden, *was* ein solches System leisten muss und das Problem dadurch eingegrenzt werden.

Daraufhin wird im Kapitel Konzeption eine Lösung erarbeitet. Die Software wird *modelliert* und *designt*; durch wichtige Entscheidungen wird bestimmt *wie* die Umsetzung erfolgen soll. Dabei wird das Problem in mehrere beherrschbare Teilprobleme zerlegt und das Modell iterativ[6] verfeinert.

Anschließend folgt im Kapitel Implementierung eine Erläuterung zur Umsetzung, wobei Schlüsselstellen beschrieben und entscheidende Probleme und Erfolge bei der Implementierung aufgezeigt werden.

Schließlich findet im Kapitel 5 die Evaluierung mit einem Bericht zum Praxistest und seinen Erkenntnissen und Bewertungen seinen Platz.

Im Schlusswort wird dann ein Fazit mit einer Stellungnahme zu den Zukunftsperspektiven des Projektes gegeben.

[3] Online Vertriebsstellen v. a. für Klein-Applikationen, bekannt geworden durch mobile Betriebssysteme
[4] Gemeinschaft, die sich die Unterstützung und Verbreitung *quelloffener* Software zum Ziel macht
[5] Es soll unter der *GPL* – General Public License veröffentlicht werden, mehr dazu im Abschnitt 4.1.1
[6] schrittweise

Kapitel 2

Kassensysteme

Ein Kassensystem ist eine „EDV-Lösung zur Einbindung von [...] Registrierkassen und Scannerkassen [...] in ein Warenwirtschaftssystem“[1]. Unterschieden wird dabei zwischen Kassensysteme für die Gastronomie und den Handel.

Kassensysteme für die Gastronomie haben im Gegensatz zu Systemen für den Handel eine vollständige Bestellverarbeitung integriert. Bestellungen werden beim Kunden aufgenommen, der entsprechende Bestellschein gedruckt und erst anschließend kommt das Zahlungssystem zum Tragen. Dieses übernimmt je nach gesetzlichen Rahmenbedingungen nicht nur den Druck des Kassenbons, sondern auch das *persistente* Speichern der Rechnungsdaten.

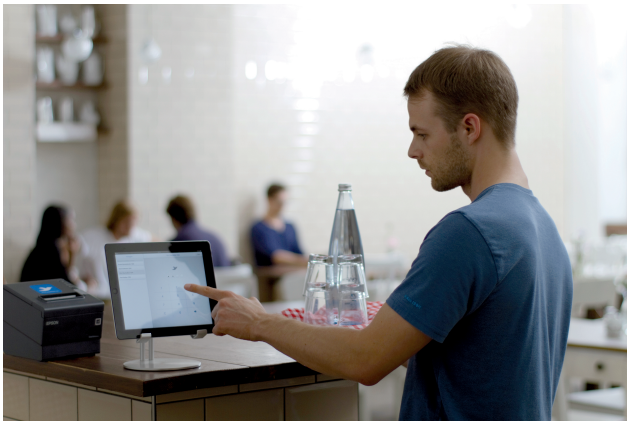


Abbildung 2.1: Komponenten eines Kassensystems.[2]

Zu den elementaren Komponenten zählen:

- Tragbare Geräte für das Service-Personal
- Bondrucker
- Vernetzung
  - Drahtlos-Vernetzung der tragbaren Geräte
  - LAN für Drucker und statische Geräte

Häufig hinter der Theke anzutreffen sind zudem computergestützte Registrierkassen, durch welche sich meist über eine berührungsempfindliche Bedienoberfläche – zusätzlich zu

[1] Zitiert aus der Einleitung von [KasSys]
[2] Urheberrechte: orderbird AG

den mobilen Geräten – Bestellungen, Rechnungen, etc. durchführen lassen. In Abbildung 2.1 erkennt man bspw. einen Tabletcomputer, welcher als Kombi-Gerät sowohl hinter der Theke, als auch beim Kunden eingesetzt wird.

Eine kabelgebundene Vernetzung der Handhelds[3] ist angesichts der oft eiligen Lage und schlicht unmöglichen Verbindung ausgeschlossen. Denn ausschlaggebend und damit auch der *wesentliche* Vorteil ist die Optimierung der *Kommunikation* zwischen Service und Küche bzw. Bar.

2.1 Grundlegende Funktionsweise

In Abbildung 2.2 ist die grundlegende Funktionsweise übersichtlich dargestellt. Dafür wurde der Bestellvorgang auf verschiedene Akteure bzw. Systeme aufgeteilt:

- Kunde
- Bedienung
- Kassensystem
- Bar / Küche

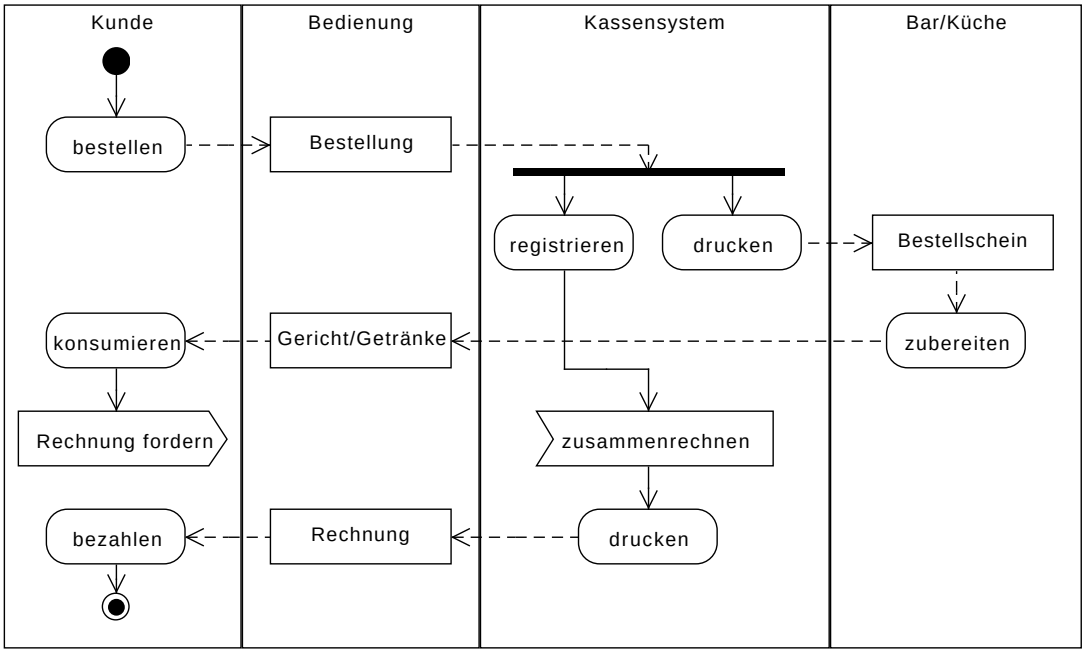


Abbildung 2.2: Funktionsweise Aktivitätsdiagramm

Der Kunde startet den Bestellvorgang, indem er bei der Bedienung seine Bestellung aufgibt. Die Bestellung wird dann von der jeweiligen Bedienung in das Kassensystem eingetragen (durch *Interaktion* mit dem Kassensystem) und gelangt dadurch über ein Ausgabegerät (Bon-Drucker) in die Bar bzw. Küche. Dort werden die Gerichte bzw. Getränke zubereitet und schließlich serviert. Äußert der Kunde dann den Zahlungswunsch, so wird dieser durch das Service-Personal an das Kassensystem weitergeleitet, welches die Abrechnung übernimmt.

[3] Taschencomputer

Die Bedienung fungiert also prinzipiell als *Vermittler* zwischen den „Systemen“. Das Kassensystem arbeitet so indirekt mit den Kunden und liefert die Bestellaufträge an die Bar/Küche. Hauptnutzer des Kassensystems ist somit der *Kellner*.

Die Aufgabenabgrenzung des Personals ist klar geregelt, so können sich die Bediensteten in Bar/Küche auf die Zubereitung konzentrieren und die Fachkräfte im Service auf die Betreuung der Kunden. Gegenüber herkömmlichen Mitteln mit Block und Feder ist somit neben dem Geschwindigkeitsvorteil durch Reduzierung der Laufwege eine Fokussierung auf die essenzielle *Kundenbetreuung* im Service möglich.

2.1.1 Grenzen und Ausbaufähigkeit

Grenzen in der Optimierung des Arbeitsflusses kristallisieren sich bei näherem Betrachten schnell heraus: Das Kassensystem kann nicht erkennen, ob ein Gericht bereits serviert wurde. Dieser harmlos erscheinende Punkt hat in der Praxis oft einen fundamentalen Stellenwert. Insbesondere bei hoher Auslastung ist es für Service-Kräfte oft schwer den Überblick zu behalten und die elementare Frage *ob* der Kundenwunsch bereits erfüllt worden ist oder nicht, bedeutet oft zusätzlichen Stress.

Des Weiteren ließe sich das hier analysierte „klassische“ Bestell-Prinzip erweitern, indem eine Benachrichtigung von abholbereiten Gerichten ermöglicht würde. Dazu müsste das Personal hinter der Theke und in der Küche durch einen speziellen Zugriff auf das System einem Kellner eine Servier-Aufforderung zuschicken können.

2.2 Integration in den Betrieb

Um einen Überblick der Aufgaben eines Gastronomiebetriebes zu erhalten, ist im Prozesshierarchiediagramm (Abbildung 2.3) eine Auflistung und Einteilung der *möglichen* Tätigkeiten abgebildet.

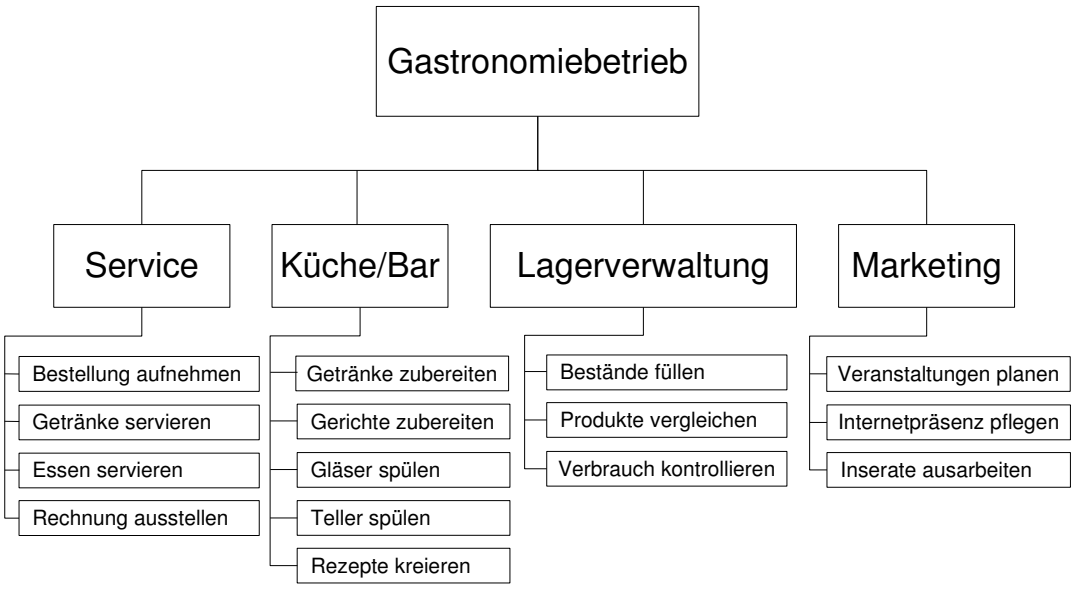


Abbildung 2.3: Gastronomiebetrieb Prozesshierarchiediagramm

Nach kurzer Überlegung lässt sich schnell herausfinden in welchen Bereichen ein Kassensystem eingreifen kann:

- Service

- Küche/Bar
- Lagerverwaltung

Eindeutig stützt es die Kommunikation zwischen Service und Küche/Bar, kann aber auch durch statistische Dienste der Lagerverwaltung dienen. Insbesondere können Statistiken zur Erfassung der Beliebtheit von Produkten verwendet werden, wodurch möglicherweise unbeliebtes aus dem Sortiment genommen werden kann.

Nur schlecht denkbar ist hingegen die Einbindung des Marketingsegments. Bestenfalls könnte die Erfassung beliebter Produkte als Marketinganstoß dienen, jedoch sind diese Szenarien für die Praxis irrelevant.

2.3 Marktanalyse

Der Markt für Kassensysteme ist belebt, so gibt es nicht etwa *den* Global-Player, welcher alle Betriebe mit seinen Produkten ausstattet, sondern es gibt die verschiedensten Lösungen, verschiedene Geräte und unterschiedliche Ansätze.

Als wohl Bekanntester und laut eigenen Angaben Weltmarktführer[4] unter den Anbietern für Bestell– und Kassensysteme für den Gastronomiebetrieb gilt das österreichische Unternehmen Orderman® mit Sitz in Salzburg.

Abbildung 2.4: Orderman®[5]

Orderman® bietet dabei eine *Komplettlösung* an:

- Handhelds
- PC Kassen

[4] siehe [OrdUG]
[5] Urheberrechte: Orderman GmbH

- Zuverlässiges Funknetzwerk[6]

Auch in Südtirol sind die Produkte von Orderman® weit verbreitet. So vertreibt das Südtiroler Unternehmen Giacomuzzi GmbH bspw. bevorzugt diese Produkte und bietet Support dazu an.[7]

2.4 Gesetzliche Rahmenbedingungen

Das Drucken eines Kassenbons muss durch geeignete Maßnahmen, welche vom Finanzsystem vorgeschrieben werden, begleitet werden. Ansonsten hätte ein Kassenbon keinen Wert; wäre nur ein Stück Papier.

Wie in [KasSys] geschildert, herrschen je nach Staat unterschiedliche gesetzliche Bedingungen vor.

In Italien ist das Gesetz dazu recht strikt, sodass die Rechnungsdaten auf unlöschbaren Speicher eingetragen werden müssen. In anderen EU-Ländern wie bspw. Deutschland kann die Software der Computer-Kassensysteme zertifiziert werden und dadurch offiziell gültige Kassenbons drucken. In Rahmen des Maturaprojektes soll die gesetzliche Lage jedoch unbeachtet bleiben und das Augenmerk auf das System gelegt werden.

2.5 Anforderungen

Durch das Use Case Diagramm verschafft man sich sehr leicht einen Überblick über die Funktionalitäten, die das System dem *Endnutzer* – sprich dem Kellner – zur Verfügung stellen muss.

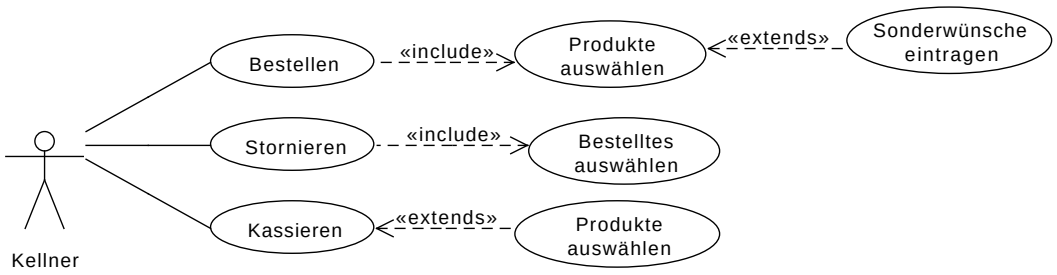


Abbildung 2.5: Kellner Use Case Diagramm

Wie man aus der Abbildung 2.5 durch das bewusst einfach gehaltene Use Case Diagramm sehr leicht ablesen kann, sind folgende grundlegende Funktionalitäten und damit *Anforderungen* an das System gegeben:

- Bestellen
- Stornieren
- Kassieren

Bei jeder dieser Aktionen ist stets die Auswahl eines Tisches inbegriffen (hier nicht abgebildet). Ein Tisch bildet somit die *Grundlage* für weitere Aktionen.

[6] Durch den eingesetzten (Industrie-) Standard RS485 über eine 433Mhz Funkverbindung wird ausreichende Reichweite und eine zuverlässige, nahezu störungsfreie Verbindung ermöglicht. Siehe [RS485]

[7] siehe [GiaMuz]

Bestellen beinhaltet stets («include») das Auswählen von Produkten und kann («extends») mit dem Eintragen von Sonderwünschen zu den Produkten verbunden sein, sofern der Kunde diese äußert. Eine Cola *mit Eis und Zitrone* wäre bspw. eine mögliche Bestellung eines Produktes und des dazugehörenden Sonderwunsches.

Zum Stornieren müssen hingegen bereits bestellte Produkte und ihre Sonderwünsche ausgewählt werden.

Das Zahlen kann »zusammen« oder »getrennt« erfolgen. Die Auswahl einzelner Produkte ist somit nur dann notwendig, wenn die Gäste eine getrennte Bezahlung wünschen.

Diese *grundlegenden* Funktionen müssen möglichst effizient und problemlos durchgeführt werden können, denn sie werden je nach Betrieb tagtäglich hunderte Male gebraucht. Inkonsistenzen oder Fehler in diesem Bereich sind folglich ein absolutes Tabu.

2.5.1 Daten

Es kristallisiert sich also die Notwendigkeit folgender Daten heraus:

- Tische
- Produkte
- Bestellungen

Tisch

Ein Tisch hat konventionell eine eindeutige Nummer und ist durch diese identifiziert. Es ist aber durchaus übliche Praxis, dass ein Tisch durch seine Nummer *und* durch seine Position (Terrasse, Bar, Stube) identifiziert wird. Man stelle sich dazu einen Betrieb vor, welcher eine Terrasse und eine Stube als Lokalität aufweist. Es kann sehr wohl sein, dass das Personal sich auf eine Tischeinteilung einigt, in der bspw. der Tisch 3 in Stube und in der Bar existiert.

Außer der Nummer und ggf. der Zugehörigkeit zu einer Lokalität benötigt der Tisch selbst keine zusätzlichen Informationen.

Produkte

Produkte könnten folgende Eigenschaften aufweisen:

- Zugehörigkeit zu einer Kategorie
- Name
- Abkürzung
- Preis

Produkte müssen bei einer Bestellung ausgewählt werden können und in einer Bestellung (auch mehrmals) enthalten sein.

Bestellungen

Bestellungen sind das Bindeglied zwischen Tischen und Produkten. Folgende Eigenschaften sind relevant:

- Zugehörigkeit zum Tisch
- Zugehörigkeit zum Produkt

- Anzahl
- Extras (für eventuelle Sonderwünsche)
- Extrapreis

Bei Annahme der Kundenwünsche erstellt der Kellner eine *Liste* von Bestellungen, welche dann gedruckt werden kann.

2.6 Kernproblem

Aus technologischer Sicht liegt das Kernproblem in der *Verteilung* von Informationen über Bestellungen, Tische und Produkte auf autonomen Endgeräten.

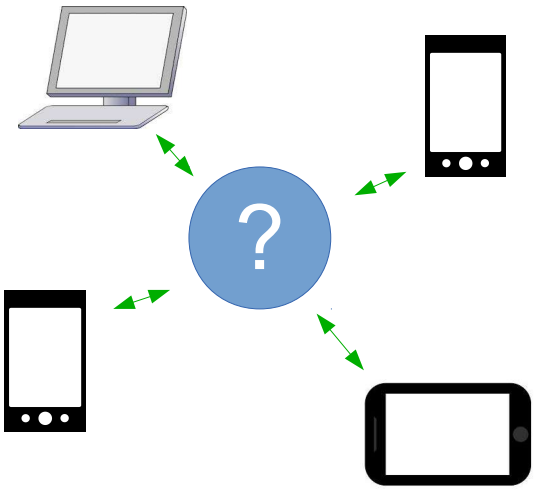


Abbildung 2.6: Kernproblem

Es handelt sich folglich um ein *verteiltes System*, welches auf allen Endgeräten die angepassten Informationen für den jeweiligen Kellner bereitstellen muss.

2.6.1 Vernetzung

Es muss somit eine *Vernetzung verschiedener Systeme* stattfinden, um die Anforderung erfüllen zu können. Dies bedeutet dass die Aufgabe der Konzeption darin besteht, eine Software zu entwerfen, welche autonome Systeme miteinander vernetzt und dadurch ein *vereintes* System schafft, welches jedem Benutzer *dieselben* Daten zur Verfügung stellen kann.

Kapitel 3

Konzeption

3.1 Prinzipien

Um einen roten Faden durch das Projekt zu ziehen, werden nun einige Leitlinien definiert. Zu jeder Zeit liegt das Hauptaugenmerk in den Schlagworten

- einfach
- sicher
- schnell

Anders als so mancher Werbespruch werden diese Prinzipien konkret in folgende Programmierkonzepte umgesetzt:

- DRY
- KISS
- RAII

3.1.1 Keep It Simple, Stupid!

Das KISS-Prinzip besagt, man soll die Lösung eines Problems so einfach wie möglich halten und unnötige Komplexität vermeiden. Das unnötige Verkomplizieren provoziert nicht nur das Auftreten von Fehlern, sondern macht es allen Beteiligten *schwerer*, das Verständnis und damit den Überblick zu behalten.

3.1.2 Don't Repeat Yourself

Mit der Kernaussage »Wiederhole dich nicht« setzt sich das Programmierprinzip *DRY* die Reduzierung der Redundanz zum Ziel. Besonders in der Programmierung, wo Veränderung an der Tagesordnung steht, können mehrfach realisierte Lösungen des gleichen Typs schnell zu Problemen führen. Kommt der Wunsch nach Erweiterung oder gar ein Fehler in diesen Segmenten auf, so wird es schnell unübersichtlich, langwierig und heikel.

Man kann sich das anhand eines Kochbuches vorstellen, in dem bei *jedem* Gericht mit Zwiebeln erklärt wird, wie man diese schneidet. Abgesehen davon, dass das Buch unnötig dick wird und sich der Autor grundlos die Mühe macht stets neue Formulierungen für *dasselbe* Probleme zu finden, muss bei einer hypothetischen Änderung in der Technik des Zwiebelschneidens (sei es ein besserer Trick für Anfänger) *jede einzelne* Stelle verbessert werden.

Die Ursache von unnötigen Wiederholungen ist meist ein Mangel an Abstraktion. Leider ist die Lösung dieses Problems in der Programmierung nicht immer so trivial wie beim

Schreiben des Kochbuches. Umso wichtiger ist es, dass der Gedanke im Hinterkopf verfestigt wird, jegliche Art von Wiederholung zu vermeiden und abstraktere Lösungsansätze zu favorisieren – solange sich deren Komplexität in Grenzen hält und nicht in Konflikt mit dem KISS-Prinzip gerät.

3.1.3 Resource Acquisition Is Initialization

Das schon recht spezielle Programmierkonzept RAII, zu deutsch »Ressourcenbeschaffung ist Initialisierung« hilft in der Programmierung wohl definierte *Beschaffung* und *Bereinigung* von Ressourcen zu erzielen.

Dadurch wird nicht nur *sauberer* Quelltext geschaffen, sondern auch die Garantie, dass Ressourcen nach ihrem Gebrauch freigegeben werden und somit unter Umständen auch Performance, welche durch nicht freigegebenen Ressourcen unnötig beansprucht wird.

3.2 Herangehensweise

Zunächst soll in Form einer sauberen Kommunikationsstruktur das Fundament des Projektes geschaffen werden. Es wird also an das in Abschnitt 2.6 proklamierte *Kernproblem* herangegangen. Danach kann die Teilung des Problems vorgenommen werden und schließlich mit dem iterativen[1] Lösen der Teilprobleme begonnen werden.

3.3 Kommunikation

Die *Datenverteilung* über das Bestellsystem muss durch eine *Kommunikation verschiedener Systeme* realisiert werden. In der Informatik haben sich für dieses elementare Problem *verteilte Systeme* klarerweise unterschiedliche Lösungsansätze etabliert. Die in diesem Zusammenhang relevanten Kommunikationsmethoden werden im Folgenden kurz analysiert.

3.3.1 Peer to Peer

Die Kommunikation zwischen *gleichberechtigten* Kommunikationspartnern wird bei Rechnernetzen als *Peer to Peer* Kommunikation bezeichnet. Ausschlaggebend ist dabei, dass die Kommunikationspartner *diesselbe* Vorgehensweise zur Kommunikation benutzen und sich in ihrem Wesen nicht unterscheiden.

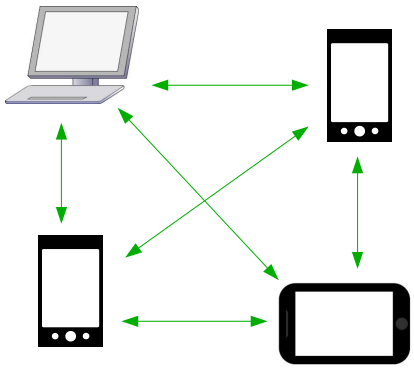


Abbildung 3.1: Peer to Peer

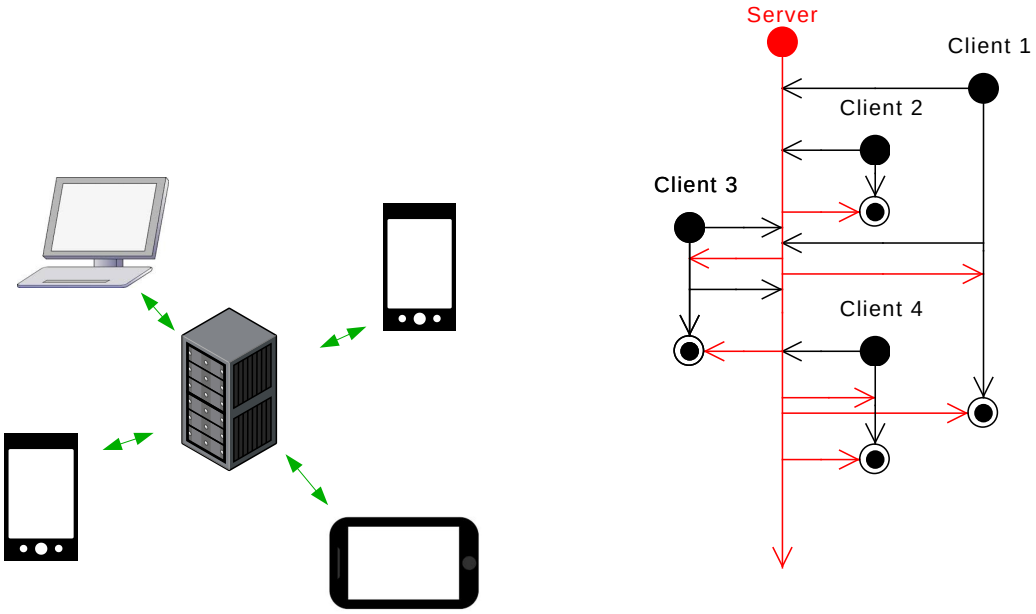
In diesem Modell gibt es *keinen* Verwalter als zentrale Anlaufstelle, sondern es wird auf eine *dezentrale* Verwaltung gesetzt, in dem jeder Kommunikationspartner das zur Verfügung stellt, was andere brauchen und sich das von anderen holt, was er braucht. Die dezentrale Verwaltung hat – wie in Abbildung 3.1 dargestellt – eine vollkommene Masche[2] zur Folge. Jeder muss sich also mit jedem verbinden, damit alle Daten auf allen Geräten korrekt wiedergegeben werden können.

[1] schrittweisen

[2] Verbindung zwischen jedem der Kommunikationspartnern

3.3.2 Client Server

Das Client-Server-Modell hat im Unterschied zum Peer to Peer Modell einen *zentralen, passiven* Kommunikationspartner – den Server – zu dem sich mehrere autonome Systeme – die Clients – verbinden und Dienste beanspruchen können.



(a) Infrastruktur (b) Lebenszyklus

Abbildung 3.2: Client Server

Durch den Server als zentrale Anlaufstelle für alle Clients ergibt sich der in Abbildung 3.2b dargestellte Lebenszyklus. Der Server – einmal gestartet – läuft ständig. Die Clients können sich *unabhängig* voneinander beim Server melden, Informationen erhalten und die Verbindung ggf. wieder trennen.

Erwähnenswert ist das unterschiedliche Verhalten von Client und Server. Als *Dienststeller* nimmt der Server eine *passive* Rolle ein. Er *wartet* auf Verbindungen von Clients. Der Client hingegen wird aktiv, sobald die Notwendigkeit besteht. Er verbindet sich und nimmt den Dienst in Anspruch.

3.3.3 Anwendung

In Anbetracht der sensiblen Daten – es geht schließlich um Geld – soll beim Kassensystem die *zentrale* Speicherung und Verwaltung der Daten bevorzugt werden. Durch den Server wird in diesem bewährten Modell nicht nur die Administration erleichtert, sondern auch der Verwaltungsaufwand bei den Clients auf ein Minimum reduziert.

3.4 Netzwerk

Die vorausgesetzte Vernetzung[3] bietet eine Netzwerktopologie, welche *allen* beteiligten Geräten (Server, Handhelds, Drucker) eine Kommunikation ermöglicht.

Konkret kann dazu eine WLAN-Verbindung dienen, welche das gesamte Lokal abdeckt und die Smartphones oder Tablets durch das bereits integrierte WLAN-Modul erreicht.

[3] siehe Abschnitt 2.6.1

Netzwerkdrucker können in dieses System leicht integriert werden, indem sie am Router[4] durch die vorhandenen Schnittstellen eingebunden werden.

3.5 Datenaustausch

Wie auch bei der Kommunikation gibt es verschiedene Ansätze um Daten über mehrere Systeme – insbesondere im Client-Server-Modell – zu verteilen. Der Server besitzt nämlich die gesamten Informationen über Tische, Produkte und Bestellungen. Welches Prinzip verwendet werden soll, um diese Daten »an den Mann« zu bringen, wird nun geklärt.

3.5.1 Request Response

Die wohl am häufigsten verwendete und damit bekannteste Methode zur Datenverteilung ist das Request-Response-Modell. Der Client schickt – sobald er die Daten benötigt – dem Server eine *Anfrage* mit der Spezifizierung *welche* Daten er benötigt und erhält dann eine *Antwort* mit den entsprechenden Daten. Durch dieses Frage-Antwort-Prinzip, wie auch in Abbildung 3.3 dargestellt, kann ein einfacher Datenaustausch ermöglicht werden, der besonders gut in den Lebenszyklus des Client-Server Modells[5] passt. So wartet der Server auf eine Anfrage, wird dann aktiv und liefert durch die Antwort die benötigten Daten.

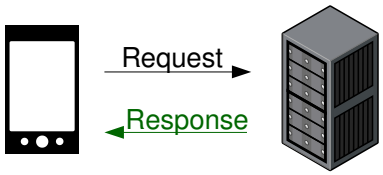


Abbildung 3.3: Request Response

Als prominentester Vertreter dieses Prinzips ist sicherlich das HTTP Protokoll zu nennen. Es bietet die Grundlage für das heutige »Internet«. Jeder Browser schickt einen HTTP-Request, eine Anfrage, an einen bestimmten Server und erhält die Website als Antwort.

3.5.2 Model-View

Das Model-View-Modell ist ein in der Anwendungsprogrammierung beliebtes und viel genutztes Prinzip. Es abstrahiert die *Darstellung* von den *Daten*, was zahlreiche Vorteile mit sich bringt. Das Model enthält und verwaltet dabei die Daten und die View kümmert sich um die Darstellung.

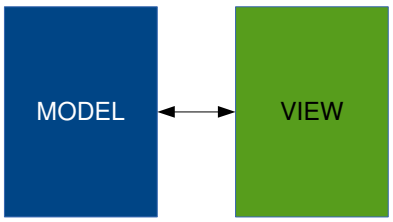


Abbildung 3.4: Model View

Mehrere Views können mit *einem* Model verbunden werden. Ein und dieselben Daten können so durch verschiedene Darstellungen (simultan) visualisiert werden. In den meisten Implementierungen dieses Konzepts findet sich eine Besonderheit in der Datenbereitstellung: Die View wird insofern an das Model gebunden, als dass Datenänderungen im Model der View mitgeteilt werden. Dies bedeutet, dass eine View beim Binden an das Model *einmalig* die anzuzeigenden Daten abfragt und dann über Änderungen im Model informiert wird.

Die Übertragung dieses Prinzips der Datenbereitstellung auf die Client-Server-Struktur bedeutet, dass der Server die Daten – also das Model – bereitstellt und die Clients zur Datenanzeige – als View – verwendet werden.

[4] oder auch ein anderes Gerät, bspw. ein PC, welcher das WLAN-Netz zur Verfügung stellt
[5] siehe Abbildung 3.2b

3.5.3 Anwendung

Der Datenaustausch nach dem Model-View-Konzept hat überzeugende Vorteile gegenüber einem klassischen Request-Response-Modell:

- geringere zu übertragende Datenmenge
- sofortige Bekanntgabe von laufenden Änderungen
- Kontrolle über verteilte Daten

Im Unterschied zum Request-Response-Modell kann beim Model-View-Ansatz eine entsandte Datenmenge durch das Melden von Änderungen korrigiert werden. Jeder verbundene Client hat so stets aktuelle[6] Daten und kann damit konsistente Änderungsanfragen schicken.

Konkret kann durch diesen Ansatz bspw. verhindert werden, dass Client A eine Kassier-Anfrage schickt, die bereits von Client B ausgeführt wurde: Schickt Client B die Anfrage zuerst, so wird durch das Model-View-Konzept Client A darüber informiert und ist nicht mehr im Glauben, der Tisch hätte noch unbezahlte Produkte, wie es beim Request-Response-Modell nach der letzten Anfrage von Client A der Fall wäre.

3.6 Datenbank

Um nun die aus Abschnitt 2.5.1 bekannten notwendigen Daten besser in die Lösung zu integrieren, sollen an dieser Stelle die verschiedenen Datengruppen und ihre Beziehungen zueinander definiert werden. Da alle Daten am Server verwaltet werden müssen, soll hiermit auch die Struktur der Datenbank des Servers geklärt werden.

3.6.1 Datenbankmanagementsystem

Die Verwaltung der Daten am Server soll durch ein DBMS – Datenbankmanagementsystem erledigt werden. Die Vorzüge gegenüber anderen Lösungen wie eigenen Dateiformaten oder ausschließliche Verwendung des flüchtigen Speichers überwiegen klar, setzt man doch mit relationalen Datenbanken auf eine standardisierte, sichere und Jahrzehnte lang erprobte Lösung.

Als DBMS soll PostgreSQL zum Einsatz kommen. Diese Entscheidung wird weniger durch objektive Argumente gerechtfertigt, sondern schlicht durch Sympathie an diesem Open-Source-Projekt und durch die Erweiterung der persönlichen Erfahrungen um ein neues DBMS.

Am Server läuft also parallel zum FOS[7] Dienst, welcher die Clients versorgt, der PostgreSQL Dienst – wie in Abbildung 3.5 ersichtlich. Die Server Applikation *synchronisiert* somit die laufenden Änderungen mit der Datenbank.

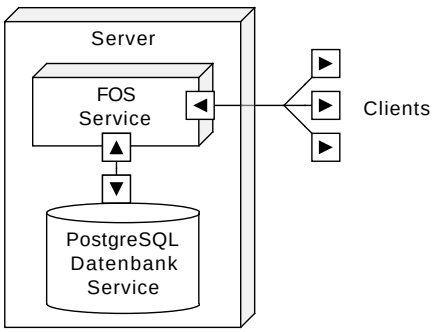


Abbildung 3.5: Server Datenbank Interaktion

[6] lediglich die Übertragungslatenz einer Änderungsmeldung stellt eine Verzögerung und somit eine kurzzeitige Inkonsistenz in der Datenverteilung dar

[7] FOS steht hier und im folgenden als Abkürzung für Free Orderman System

3.6.2 Datenbankdesign

Zum Datenbankdesign und damit zur Ausarbeitung der Datenstrukturen eignet sich wohl am Besten das Entity-Relation-Ship Diagramm. In Abbildung 3.6 dargestellt, zeigt es die Objekte, deren Eigenschaften und Beziehungen auf.

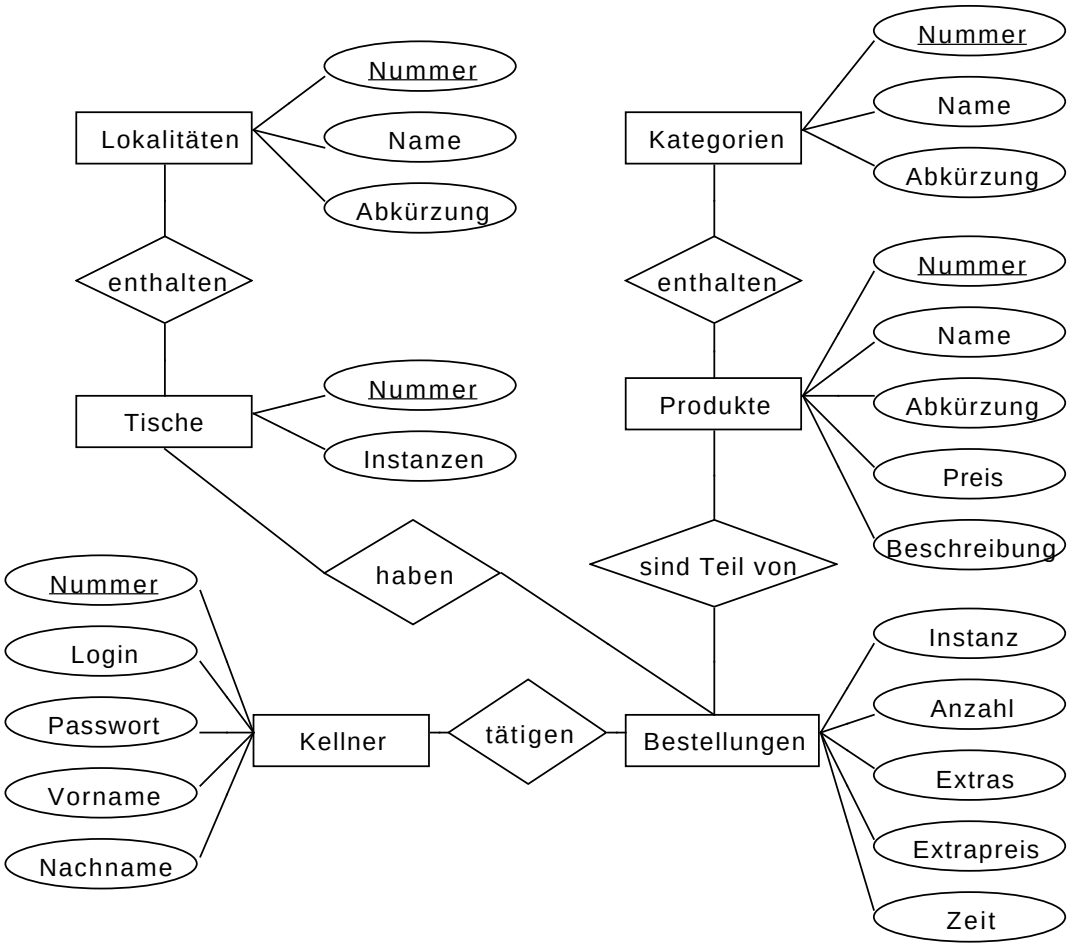


Abbildung 3.6: Datenbank Design ER-Diagramm

Um eine hierarchische Struktur und damit bedeutende Übersicht zu gewinnen, werden Tische in Lokalitäten und Produkte in Kategorien unterteilt. Außerdem wird für die optimale Nutzung der begrenzten Anzeigefläche auf den Endgeräten eine Abkürzung für etwaige Bezeichnungen zur Verfügung gestellt.

Identifikation der Tische

Um das Konzept nicht festzufahren, muss – wie so oft beim Software Design – so weitsichtig wie möglich geplant werden. Als nicht zu vernachlässigende Funktionalität ist in dieser Hinsicht das *Teilen* von Tischen zu betrachten.

Gesellen sich zwei unterschiedliche Gruppen an den selben Tisch, müssen für *denselben* Tisch zwei *unterschiedliche* Bestelllisten und damit Rechnungen angelegt werden. Das Vergeben einer neuen Nummer für den nun benötigten zweiten Tisch wäre dafür ungeeignet, da es sich nicht um einen neuen Tisch handelt – sie sitzen am selben Tisch – und der Bezug zur tatsächlichen Tischnummer nicht verloren gehen darf.

Somit bedarf es einer zweiten Identifikationsnummer und damit dem Konzept der Instanzen. Ein Tisch wird dabei durch seine Nummer *und* durch seine Instanz identifiziert. Die Instanz ist wiederum eine Nummer, welche den Bestellungen angehängt wird. Am Server muss für einen Tisch die zur Zeit höchste Instanz-Nummer gespeichert werden. Hat der Tisch nur eine Instanz – normalerweise der Fall – so ist die höchste Instanz-Nummer 1 und den Bestellungen für diesen Tisch wird als Instanz die Nummer 1 mitgegeben.

Bestellungen

Bestellungen haben als Besonderheit *keine* Identifikationsnummer. Dies hat den großen Vorteil, dass so *keine* inkrementierende Laufnummer benötigt wird, welche evt. an das Ende ihres Bereiches[8] gelangt. Die Bestellungen dienen also dazu, die *aktuellen* Bestellungen zu speichern – nicht aber als Datenarchiv.

Diese aktuellen Bestellungen werden demnach eindeutig durch den Kellner, den Tisch sowie dessen Instanz-Nummer, die Produktnummer, die Extras (Sonderwünsche) und den Extrapreis identifiziert.

3.6.3 Datenverteilung

Die gesamten Daten sind durch die Datenbank nun am Server gespeichert. Die Daten, welche den Clients zugespielt werden, können nun als *Teilmenge* der Daten des Servers betrachtet werden. Bspw. besteht ohne Weiteres die Möglichkeit einem bestimmten Kellner nur das Arbeiten (Bestellen, Stornieren, Kassieren) auf der Terrasse zu erlauben, in dem die anderen Daten schlicht nicht versandt werden.

3.7 Protokoll

Das Protokoll ist sicherlich das mit Abstand wichtigste Teilproblem des Systems. Es ist die Grundlage der Kommunikation und definiert im Detail *wie* die Geräte miteinander kommunizieren.

3.7.1 Basis

Um größtmögliche Unabhängigkeit und Sicherheit zu erlangen, soll als Kommunikationsbasis eine IP/TCP Verbindung dienen, welche vom Netzwerk[9] ohne Weiteres bereitgestellt werden kann. Der IP/TCP Standard bietet durch seine enorme Verbreitung[10] weitreichende Unterstützung von nahezu allen Netzwerken und Netzwerkgeräten.

Zudem kann die Transportschicht TCP leicht durch die konforme *verschlüsselte* 4. Schicht TLS (Transport Layer Security) ersetzt werden. Dies ist ist eine Sicherheitsmaßnahme von größter Bedeutung, können doch auch verschlüsselte WLAN-Netze mit dem entsprechenden Know-How und genügend Zeit u. U. »geknackt« werden.[11]

Die Verbindung zum Server soll also über einen IP/TCP Socket auf Port 2107[12] erfolgen.

[8] Im Gegensatz zur Mathematik haben die darstellbaren Zahlenbereiche in der Informatik sehr wohl eine Obergrenze vor dem Unendlichen: bei üblicherweise verwendeten 32 Bit für natürliche Zahlen (ohne Vorzeichen) wäre dies $2^{32} - 1 = 4.294.967.295$

[9] siehe Abschnitt 3.4

[10] er ist das Fundament des HTTP-Protokolls und damit des heutigen Internets

[11] siehe [DrKS]

[12] Die Wahl der Portnummer ist willkürlich getroffen

3.7.2 Datendarstellung

Die Daten sollen durch UTF-8 kodierte Zeichenketten repräsentiert werden. UTF-8 bietet als populärste Unicode Kodierung Unterstützung für Zeichensätze fast jeder Schriftkultur. Damit wird der Internationalisierung und Übersetzung der Software der Weg frei behalten.

Die Zeichenketten sollen in begrenzte Abschnitte eingeteilt werden, welche *Nachrichten* darstellen. Eine Nachricht ist durch *ein* JSON-Objekt begrenzt.

JSON

JSON ist ein weit verbreitetes Datenformat, welches zur plattform– und programiersprachen-unabhängigen Darstellung von Informationen eingesetzt wird. Als Grundlage bietet es die Kodierung von *Objekten* und *Listen* sowie einigen elementaren Datentypen wie Zeichenketten, Wahrheitswerten und Zahlen. Genau ein solches JSON-Objekt mit seinen Eigenschaften wird als atomare Nachricht übermittelt.

3.7.3 Sitzung

Eine Sitzung[13] stellt einen Zeitraum der nutzbringenden Kommunikation zwischen Server und Client dar. Für das Kassensystem hat Sicherheit einen großen Stellenwert, sodass es von großer Bedeutung ist, die (geschäftliche) Kommunikation erst nach einer erfolgreichen Authentifizierung mit Benutzername und Passwort zu erlauben.

Aufbau

Das »Login«-Objekt[14] nimmt dabei das in Listing 3.1 angeführte Aussehen an.

```
1 {
2   "type": 6,
3   "login": "benutzername",
4   "password": "geheimespaSsw.rt"
5 }
```

Listing 3.1: Json Login Object

Die Eigenschaft »type« spezifiziert durch eine Zahl den Typ des Objektes – in diesem Fall ein Login-Objekt – und ermöglicht so die Klassifizierung des ankommenden JSON-Objektes. »login« und »password« geben die in der Datenbank festgehaltenen Benutzernamen und Passwörter an (Abbildung 3.6).

Abbau

Beim Abschluss der Sitzung wird vom Client ein »Logout«-Objekt versandt (Listing 3.2).

```
1 {
2   "type": 7,
3   "printClearance": false
4 }
```

Listing 3.2: Json Logout Object

[13] geläufig unter dem englischen Begriff »Session«

[14] Hier und im Folgenden werden stets englische Begriffe für die technischen Umsetzungen verwendet, da dies in der Informatik mittlerweile als De-Facto-Standard gilt – insbesondere bei Open-Source-Projekten. Siehe dazu als Anregung [LeProg]

Der Wahrheitswert »printClearance« gibt dabei an, ob die (Tages-)Abrechnung für den betreffenden Kellner gedruckt werden soll. Sollte durch einen Verbindungsabbruch ein vorzeitiges Ende der Kommunikation auftreten, so wird keine Abrechnung gedruckt.

3.7.4 JSON Objekte

Wie bereits erwähnt muss jedes JSON-Objekt eine Nummer bereitstellen, welche dessen Typ angibt. Dadurch können ankommende Objekte korrekt analysiert und konvertiert werden. Diese Zahl wird durch einen Aufzählungstypen[15] programmiertechnisch umgesetzt.

Server

Durch Typ-parametrisierte Klassen[16] kann das Model-View-Prinzip[17] den Programmierprinzipien *Don't Repeat Yourself* und *Keep It Simple, Stupid!* entsprechend umgesetzt werden.[18] Ganz rechts in Abbildung 3.7 ersichtlich, bilden die Typen »KeyValuePair« und »Update« die Grundlage dafür.

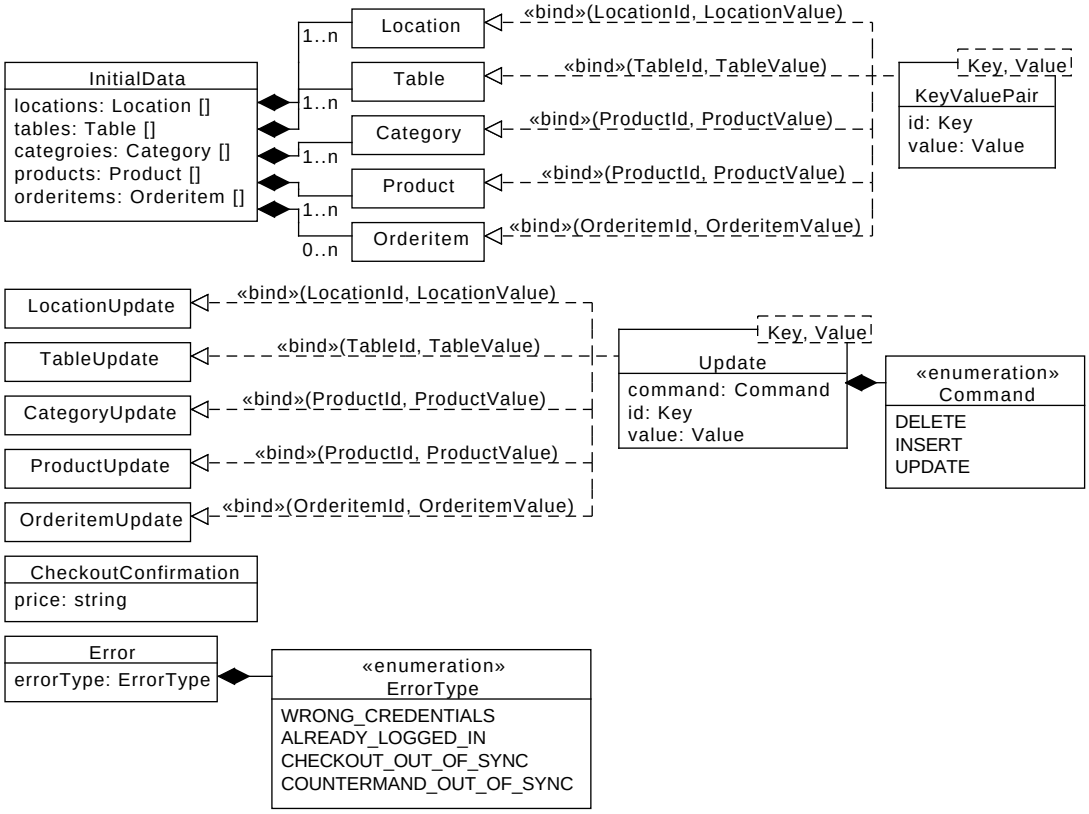


Abbildung 3.7: Json Objekte Server

Zur Übermittlung von Änderungen muss jedes Objekt identifizierbar sein. Aus dieser Überlegung heraus impliziert sich die Aufteilung eines Objektes in Identifizierung und

[15] für die Umsetzung in C++ siehe S. 71 [BREY]
[16] für die Umsetzung in C++ (Templates) siehe S. 246 [BREY]
[17] siehe Abschnitt 3.5.2
[18] siehe Abschnitt 3.1

Wert. Im Englischen als »Key« und »Value« bekannt, können damit die Typ-unabhängigen Klassen »Update« und »KeyValuePair« bedient werden.

Der »Update«-Typ kann bei jeglichen Objekten als Änderungsmeldung eingesetzt werden. Dafür ist neben der Identifizierung und dem Wert des betreffenden Objektes die Spezifikation der Änderungsart von Nöten. Diese kann – als Aufzählungstyp »Command« umgesetzt – folgende Merkmalsausprägungen annehmen:

- INSERT: Einfügen
- UPDATE: Aktualisieren
- DELETE: Löschen

Beim Löschen ist der Wert des Objektes unnötig, doch gemäß dem KISS-Prinzip (Abschnitt 3.1.1) soll er unbeachtet mitgeschickt werden.

Der »KeyValuePair«-Typ dient hingegen der Notwendigkeit, die Anfangsdaten zu verschicken. Nach einer erfolgreichen Authentifizierung müssen nämlich gemäß dem Model-View-Prinzip dem Client die aktuellen Daten zugespielt werden. Konkret bedeutet dies, dass der Client als Antwort auf ein valides »Login«-Objekt ein »InitialData«-Objekt zugesandt bekommt.

Die »CheckoutConfirmation« hingegen ist die Bestätigung für eine erfolgreiche Abrechnung und beinhaltet den zu entrichtenden Preis als Zeichenkette. Durch »Error« kann der Server nach einer Mitteilung des Clients einen aufgetretenen Fehler melden, darunter „Falsche Anmeldedaten“, „Bereits angemeldet“, „Bereits bezahlt“ und „Bereits storniert“.

Datendarstellung in JSON

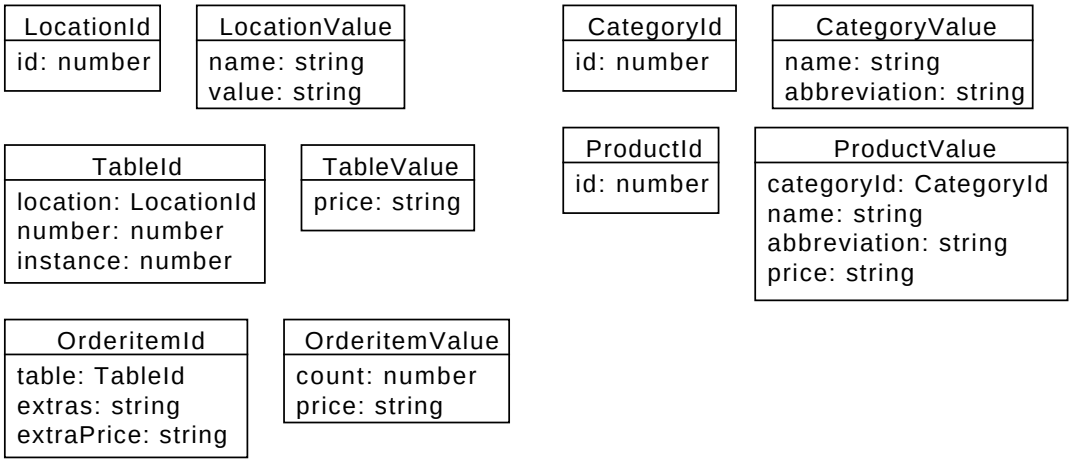


Abbildung 3.8: Json Daten Objekte

In Abbildung 3.8 ist die Ausführung der Daten Objekte in JSON dargestellt. Als diskutierbar erweist sich dabei die Aufteilung der Bestellung »Orderitem« in »OrderitemId« und »OrderitemValue«. Eindeutig identifizierbar ist eine Bestellung nämlich nur durch die Kombination aus:

- Tisch
- Produkt

- Extras
- Extrapreis

Die Zusammenführung von Extras *und* Extrapreis ist dabei durch den Anwendungsfall zu erklären, dass unterschiedliche Kellner (leider) für dasselbe Extra beim selben Produkt (in der Theorie) einen unterschiedlichen Preis angeben können. Diese müssen dann separat gelistet werden.

Äußerst wichtig ist zudem das Datenformat der Preise. Da Rundungsfehler unbedingt ausgeschlossen werden müssen, erhält der Client alle für die Anzeige notwendigen Preise als *Zeichenkette*. Dies ist eine Sicherheitsmaßnahme, was den den Client davor bewahrt, fehlerbehaftete Rechnungen mit Gleitkommazahlen durchführen zu müssen. So ergibt sich, dass sowohl »OrderitemValue« als auch »TableValue« einen Preis enthalten, obwohl sich diese Informationen errechnen lassen würden.

Client

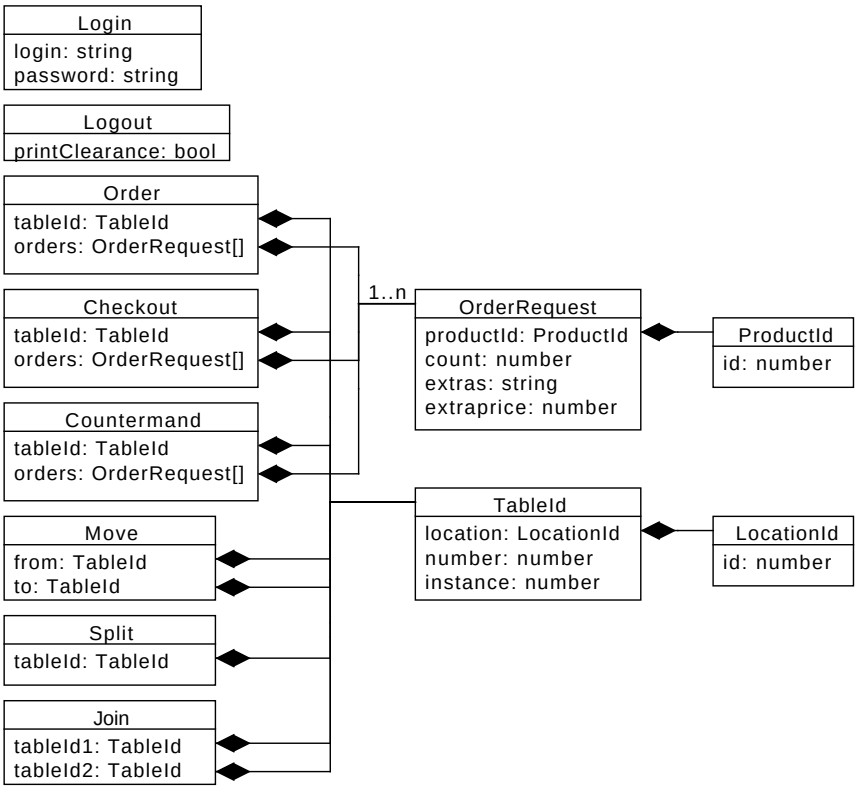


Abbildung 3.9: Json Objekte Client

Wie in Abbildung 3.9 aufgelistet kann der Client folgende Objekte verschicken:

- Login – Sitzungsaufbau
- Logout – Sitzungsabbau
- Order – Bestellung
- Countermand – Stornierung

- Checkout – Bezahlung
- Move – Bestellungen verschieben
- Split – Tisch teilen
- Join – Tisch wieder zusammenführen

Wie man leicht erkennen kann, hat – außer Sitzungsauf– und -abbau – jedes Objekt eine »TableId«, Tisch-Identifizierung, in sich. Diese Ausführung deckt sich mit der Analyse der grundlegenden Funktionalitäten in Abschnitt 2.5.

3.7.5 Kommunikationsverlauf

Es handelt sich also zusammenfassend um eine *asynchrone* Kommunikation. Dies bedeutet, dass nicht das klassische Frage-Antwort-Spiel überwiegt, sondern viel mehr ein „zufälliges“ Ankommen von »Updates« des Servers und Anfragen des Clients geschieht.

3.8 Zielplattformen

Um einer weiten Verbreitung von FOS nicht hinderlich zu sein, soll es auf möglichst vielen Plattformen laufen. Besonders im Mobilbereich, wo sich zunehmend eine Fragmentierung der Betriebssysteme abzeichnet,[19] ist es aus der Sicht von Entwicklern von großer Bedeutung auf mehreren Plattformen vertreten zu sein.

Dabei muss unterschieden werden zwischen der Client– und der Server-Applikation. Die Client-Applikation sollte auf Tablets, Smartphones und PCs hinter der Theke laufen, um eine vernünftige Abdeckung des Marktes zu erreichen. Bei der Server-Applikation hingegen reicht es, wenn sie auf PCs betrieben werden kann.

3.8.1 Qt

Um diese hochgesteckten Ziele ohne viel Mehraufwand erreichen zu können, kommt Qt[20] zum Einsatz. Qt ist eine C++ Bibliothek mit einem ausgereiften Ereignis System und zahlreichen Features.

Abbildung 3.10: Qt Logo und Leitmotiv

Mit der offiziellen Unterstützung von Android ab der Version 5.2, erschienen im Dezember 2013, erweiterten die Entwickler die Unterstützung einmal mehr um eine bedeutende Plattform. Damit erweist sich Qt ideal für dieses Projekt, da Client *und* Server – mit dem selben Framework geschrieben – gemeinsamen Code verwenden können. Konkret kann bspw. mit der Unterstützung von JSON, die Analyse und Konvertierung eingehender JSON-Objekte *einmal* implementiert und beiderseits verwendet werden.

[19] Android, Windows Phone 8, iOS, Tizen, ...
[20] englisch ausgesprochen »cute«

Die Verwendung von Qt impliziert die Verwendung der Programmiersprache C++, welche als umfangreiche Sprache auch objektorientierte Programmierung unterstützt. Das definierte Protokoll, welches auf JSON-*Objekten* basiert, verlangt geradezu nach einer objektorientierten Lösung. Infolgedessen soll mithilfe von Qt und C++ der objektorientierte Ansatz realisiert werden, dessen Grundzüge noch in der Konzeption ausgearbeitet werden sollen.

3.9 Problemteilung

Um die Anforderungen[21] nun alle meistern zu können, kann das Problem an dieser Stelle leicht in einen *Client*- und einen *Server*-Teil zerlegt werden.

Nachfolgend soll der Server-Teil behandelt und anschließend in der Implementierung realisiert werden.

3.10 Asynchrones Clienthandling

Zu der Hauptaufgabe eines jeden Servers gehört die Abwicklung von Clients und ihren Anfragen, im Englischen und somit im Fachjargon als *Clienthandling* bezeichnet. Das möglichst effiziente Erfüllen dieser Aufgabe ist der entscheidende Faktor für die Qualität eines Servers. Ebenso haben dabei Ausfallsicherheit und Stabilität einen hohen Stellenwert.

Immer öfters steht in der Informatik das Problem der *parallelen* Abarbeitung[22] im Mittelpunkt. Die Größe der Halbleiter schrumpft stetig[23] und immer mehr Rechenleistung wird verfügbar. Es wird jedoch zunehmend auf *mehrere* Rechenkerne gesetzt, sodass verschiedene Aufgaben simultan anstatt sequentiell erledigt werden können. Vor allem im Server-Bereich wird dieses Konzept häufig angewandt, sodass es prädestiniert für die Verwendung im Server des Free Orderman Systems ist.

Im Objekt-Sequenz-Diagramm in Abbildung 3.11 ist skizziert wie das *asynchrones* Clienthandling umgesetzt wird. Dabei werden in der dargestellten Situation folgende Objekte in parallel laufenden Ausführungssträngen – als *Threads* realisiert — betrieben:

- Loginhandler
- Clienthandler #1
- Clienthandler #2

Wie man erkennen kann, wird die Authentifizierung vom »Loginhandler« übernommen. Dieser konsultiert die Datenbank durch das Objekt »DatabaseConnection« und erhält einen Wahrheitswert »successful« (erfolgreich) oder »failed« (fehlgeschlagen) als Antwort. War die Authentifizierung erfolgreich (wie beim Client #1), so wird ein neuer Ausführungsstrang – als »ClientHandler #1« abgebildet – erstellt und durch diesen *jede weitere* Anfrage des entsprechenden Clients verarbeitet. Schlug die Authentifizierung fehl (wie beim ersten Versuch des Clients #2), so wird eine Fehlermeldung vom »Loginhandler« (im »Login«-Thread) versandt.

Es wird also erst dann ein neuer Thread angelegt, wenn die Authentifizierung erfolgreich war. Dies hat eine schonende Behandlung von Systemressourcen und einen bedeutenden Sicherheitsgewinn zur Folge. Hypothetische Angreifer würde ohne gültige Zugangsdaten so höchstens eine Auslastung des »LoginHandler«-Threads erreichen können, ohne das restliche System merklich zu stören – abhängig vom Ausmaß der Attacke.

[21] siehe Abschnitt 2.5

[22] Für dieses Konzept und dessen Umsetzung sind mehrere Bezeichnungen gebräuchlich, unter anderem Nebenläufigkeit und asynchrone Verarbeitung.

[23] siehe [MoGe]

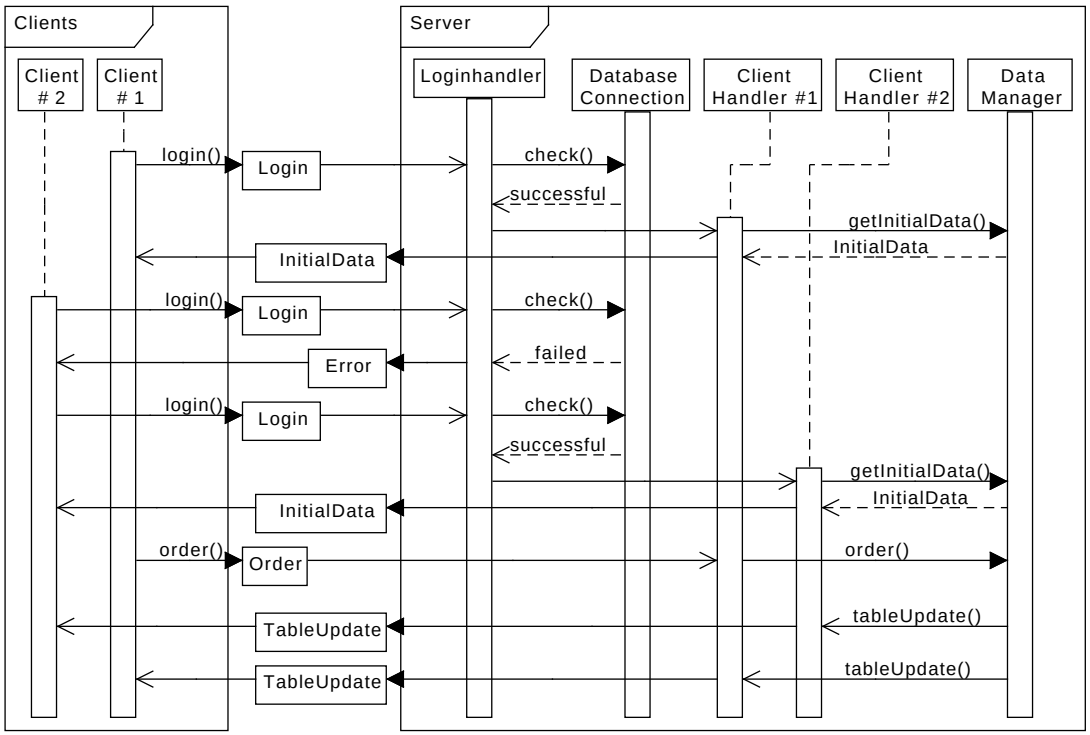


Abbildung 3.11: Client Handling

Des weiteren ist ersichtlich, dass jegliche Datenabfragen und -änderungen durch das »DataManager«-Objekt vollzogen werden. Dieses *zentrale* Objekt benachrichtigt alle laufenden »ClientHandler«-Threads über Veränderungen (»Updates«) der Datenmengen und implementiert so das Model-View-Konzept.

3.11 Daten-Manager

Die Daten-Manager-Klasse »DataManager« ist integraler Bestandteil der Implementierung des Model-View-Konzepts. Um dabei möglichst DBMS-unabhängig zu bleiben, soll ein Konstrukt *abstrakter* Klassen entstehen, welche den Datenbankzugriff soweit abstrahieren, dass die Verwendung dieser alle nötigen Funktionalitäten bietet.

Die Instantiierung der Klasse »DatabaseFactory«, welche als Ursprung der anderen Instanzen dient, kann mit Hilfe eines Dialogfensters, welches erst nach erfolgreicher Verbindung und Initialisierung der Datenbank die Fortsetzung des Server-Starts erlaubt, durchgeführt werden. Die Klassen »PSQLDatabaseFactory«, »PSQLDatabaseConnection« und »PSQLDataManager« bilden die Implementierung für den PostgreSQL-Service.

Durch die Signale[24] »locationUpdate«, »tableUpdate«, ... , wird es ermöglicht *Änderungen* am Datenbestand allen Clients mitzuteilen. Damit vom »DataManager« Signale entsandt werden können, muss dieser von »QObject« erben, was den Unterschied zu den Interfaces »DatabaseFactory« und »DatabaseConnection« erklärt.

Die Methoden »getInitialData«, »order«, »counter« und »checkout« sind den Clients bzw. »ClientHandler« vorbehalten. Sie beschränken sich auf das Ändern, Hinzufügen und Löschen von *Bestellungen*.

[24] von Qt-bereitgestellte „Erweiterung“ von C++, welche essenzieller Bestandteil des Ereignis-Behandlungs-System ist

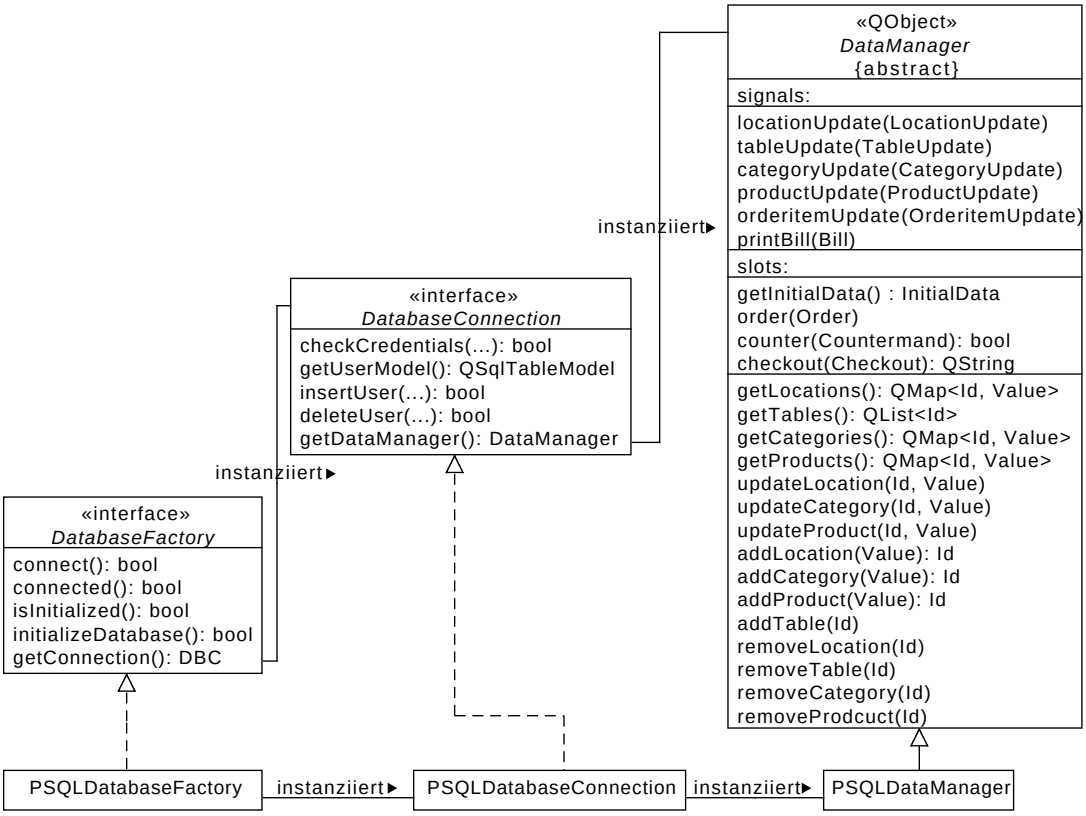


Abbildung 3.12: Klassen zum Datenbankzugriff

Die übrigen Methoden hingegen sind für die Änderung und Einsicht betriebsspezifischer Daten vorgesehen: Lokalitäten, Tische, Kategorien und Produkte. Eine Besonderheit stellen dabei Tische dar, deren Wert (»TableValue«, siehe Abbildung 3.8) nicht geändert werden kann, wodurch die Methode »updateTable(...)« entfälllt.

3.12 Erscheinungsbild

Entgegen den Konventionen wird der Server des Free Orderman Systems zunächst an eine grafische Anwendung gebunden sein. Dies hat den Vorteil, dass die Integration der Administrationsoberfläche zur Verwaltung des Datenbestandes bedeutend einfacher ist.

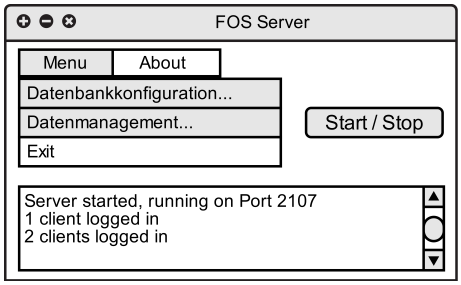


Abbildung 3.13: Skizze der Benutzeroberfläche des Servers

Der Server wird also durch eine Benutzeroberfläche repräsentiert (Abbildung 3.13),

welche ein einfaches Starten und Beenden des Dienstes ermöglicht und Konfigurationsmöglichkeiten sowie Verwaltungsoptionen bietet.

Insbesondere muss die Oberfläche ein Datenverwaltungsfenster bieten, welches die Datenbestände bzgl. Kellner, Lokalitäten, Tische, Kategorien und Produkte verändern kann. Durch die Anbindung an den »DataManager« werden Änderungen – bpsw. das Hinzufügen von Tischen – an die verbundenen Clients mittels »Update«-Objekten weitergeleitet.

Durch diese Maßnahmen entsteht ein einfach zu bedienendes System, was leicht durch den Besitzer des Gastronomiebetriebes eingerichtet werden und sogar im laufenden Betrieb auf Produktänderungen reagieren kann.

Kapitel 4

Implementierung

4.1 Open Source

Im Deutschen als *quelloffene* Software bekannt, ist Open Source mehr als nur das Offenlegen des Herzstücks einer Software, dem Quelltext. Es ist vielmehr eine Philosophie. Durch Open Source ergibt sich das Potenzial, dass das Projekt Anklang in einer Community findet. Damit ermöglicht sich eine potenzielle Weiterentwicklung und vor allem Langlebigkeit des Projekts. Vor allem aber wird es durch die freie Verfügbarkeit jedem ermöglicht das System zu nutzen, ohne sich über Anschaffungskosten Gedanken machen zu müssen.

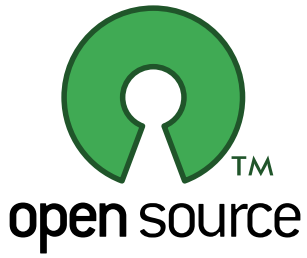


Abbildung 4.1: Logo der Open Source Initiative

4.1.1 Lizenz

Als Lizenz kommt die GPLv3 – General Public License Version 3 zum Einsatz. Sie garantiert, dass die Software von jedem verwendet, kopiert und verändert werden kann, unter der Bedingung, dass jede Veränderung wiederum als GPLv3 lizenziert veröffentlicht wird. Damit wird ein Aufgreifen und geschlossenes Weiterentwickeln der Software verhindert.[1]

Um die Lizenz wirksam zu verwenden, reicht es nicht, sie auf der Projektseite anzugeben: An den Anfang jeder Quelldatei muss ein vorgegebener Text eingefügt werden, wie er als Exempel dem Listing 4.1 zu entnehmen ist.

```
1  /*
2   * author:          Florian Mahlknecht <mail@fmdevelop.com>
3   * begin:           12/18/2013
4   * copyright:       (c) 2013 Alex Mazzon and Florian Mahlknecht
5   *
6   *
7   * This file is part of FOS.
8   * FOS is free software: you can redistribute it and/or modify
9   * it under the terms of the GNU General Public License as published by
10  * the Free Software Foundation, either version 3 of the License, or
11  * (at your option) any later version.
12  *
13  * FOS is distributed in the hope that it will be useful,
14  * but WITHOUT ANY WARRANTY; without even the implied warranty of
15  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
16  * GNU General Public License for more details.
17  *
18  * You should have received a copy of the GNU General Public License
19  * along with FOS. If not, see <http://www.gnu.org/licenses/>.
20  *
```

[1] Dieses Prinzip wurde von Richard Stallmann, bedeutenster Free-Software Pionier, als Copyleft benannt

```
21 */
```

Listing 4.1: Exemplarischer Lizenztext aus der Datei server.cpp

4.2 Vorgehensweise

Die Programmierung soll möglichst strukturiert und stets übersichtlich erfolgen, sodass auftretende Fehler unter Kontrolle und Veränderungen nachvollziehbar bleiben. Vor allem aber soll von vorne herein mehreren Entwicklern die Möglichkeit gegebenen werden, am Projekt teilzuhaben.

4.2.1 Versionskontrolle

Versionskontrolle[2] bietet die elementare Grundlage um Herr über dem Projekt zu bleiben. Mehreren Entwicklern wird es ermöglicht Quellcode beizusteuern und es wird eine konsistente Code-Basis garantiert. Eine Lösung über USB-Sticks und Kopieren der Dateien wäre undenkbar.

Abbildung 4.2: Bazaar Logo

Als Versionskontrollsystem soll »bazaar« verwendet werden. »bzr«, die Abkürzung für das Kommandozeilentool, stellt dabei mit

- checkout
- update
- commit

alle für dieses Projekt notwendigen Funktionalitäten bereit.

4.2.2 Code Hosting

Als Projekt-Plattform, auf die der Quellcode, Versionen, Meilensteine usw. bereitgestellt werden, soll der kostenlose Service von Launchpad in Anspruch genommen werden. Dieser stellt ein »bazaar«-*Repository* bereit und damit den geeigneten Ort für den Quellcode im Einklang mit dem Versionskontrollsystem.

Abbildung 4.3: Launchpad Logo

Die erstellte Projektseite ist unter `launchpad.net/fos` erreichbar. Neben den einzelnen Versionszweigen sind Meilensteine und sog. Blueprints, zu dt. etwa Pläne, verfügbar.

4.2.3 Entwicklungsumgebung

Als Entwicklungsumgebung kommt naheliegender Weise der »QtCreator« zum Einsatz. Dieser bietet mit der integrierten Dokumentation von Qt[3], dem Debugger und der hervorragenden Projektverwaltung das ideale Ambiente für eine entspannte Entwicklung. Dieses kommt auf einem archlinux-Rolling-Release-System zum Einsatz. Dabei handelt es sich um eine Linux-Distribution, die im Gegensatz zu herkömmlichen Derivaten ohne Ausnahmen stets auf aktuelle Software-Pakete setzt. Diese Lösung erweist sich insofern als geeignet für dieses Projekt, als dass mit den neuesten Qt-Versionen großer Wert auf Aktualität gesetzt wird: Ständig erscheinen sog. Bug-Fix-Releases[4], die bei der Entwicklung ins Gewicht fallen.[5]

[2] im Englischen oft als *versioning* bezeichnet
[3] siehe [QtDoc]
[4] zu dt. etwa »fehlerkorrigierende Versionen«
[5] Im Laufe der Entwicklung erschien ein Bug-Fix-Release, welches effektiv einen auftretenden Fehler in der Applikation ausmerzte. Ohne diese Korrekturen sind einem schnell die Hände gebunden.

4.3 Module

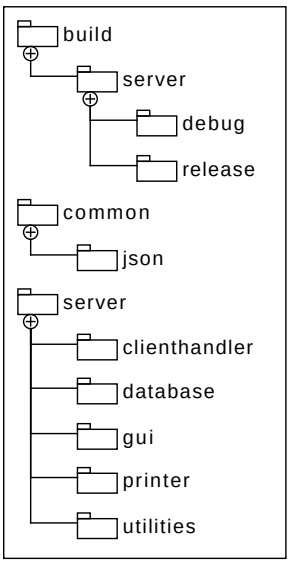


Abbildung 4.4: Auszug der Orderstruktur

In Abbildung 4.4 ist die den Server betreffende Orderstruktur im Launchpad-Repository und damit die grundlegende Modularisierung abgebildet. Erwähnenswert hierbei ist das »common«-Verzeichnis, in welches gemeinsam genutzter Quellcode (Server und Client) abgelegt wird. Darunter fällt u. a. auch das JSON-Handling.

4.4 JSON-Handling

Alle Objekte, welche von und in JSON konvertiert werden können, erben von der Basisklasse »JsonConvertable<T>« (Listing 4.2). Der Typ T spezifiziert dabei die erbende Klasse und abstrahiert dadurch den Rückgabewert der Konvertierungsmethode »fromJson«.

```
32 template <class T>
33 class JsonConvertable
34 {
35 public:
36     virtual QJsonObject toJson() const = 0;
37
38     inline QByteArray toUtf8Json() const {
39         return QJsonDocument(toJson()).toJson() + ",";
40     }
41
42     static bool check(const QJsonObject&);
43     static T fromJson(const QJsonObject&);
44
45     virtual ~JsonConvertable() {}
46 };
```

Listing 4.2: Json-Basisklasse

```
23 #ifndef JSONOBJECT_H
24 #define JSONOBJECT_H
25
26 #include <QSharedPointer>
```

```
27 #include "jsonconvertable.h"
28
29 namespace fos {
30 namespace json {
31
32
33 /**
34  * All the json objects that are intended to be send (as "root" object) will be
35  * derived from this base class. The advantage of such a deriving-structure is,
36  * that we can return a pointer to a JsonObject, determine it's type and convert
37  * it to the actual type. So each JsonObject has a type member, which is
38  * represented by the enum JsonObject::Type which has to be extended when new
39  * jsonobjects are added. So we can use switch-statements to handle json objects
40  * and otherwise than by QVariant, we don't have to check if a JsonObject is a
41  * certain type, we know exactly which type it is.
42  * @brief JsonObject is the base class of all objects that can be send and
43  *        received.
44  */
45 class JsonObject : public JsonConvertable<JsonObject>
46 {
47 public:
48     /**
49      * Represents the different types of a JsonObject. Each object that can be
50      * send and received through the json stream has it's own type listed here.
51      * These types have exactly the same name as the actual class which
52      * represents the json object.
53      * @brief The Type enum represents the type of a JsonObject
54      */
55     enum class Type : quint8 {
56         InitialData,        //0
57         CategoryUpdate,     //1
58         ProductUpdate,      //2
59         LocationUpdate,     //3
60         TableUpdate,        //4
61         OrderitemUpdate,    //5
62         Login,              //6
63         Logout,             //7
64         Checkout,           //8
65         CheckoutConfirmation,//9
66         Countermand,        //10
67         Order,              //11
68         Error               //12
69     };
70
71     /**
72      * Returns the type of the JsonObject. It's guaranteed that you can cast a
73      * pointer to JsonObject to a pointer to the actual object indicated by this
74      * type.
75      * @brief type indicates the real type of the jsonobject
76      * @return the actual type of the jsonobject.
77      */
78     Type type() const { return m_type; }
79
80     /**
81      * Creates a QJsonObject with a type property as an int which represents the
82      * type of the json object (numeric enum value).
83      * @brief toJson
84      * @return QJsonObject which represents this object
85      */
86     virtual QJsonObject toJson() const;
87
88     /**
89      * Returns true if the passed jsonObject contains an integer "type" property,
90      * false otherwise.
91      * @brief Checks if the passed json object is a fos::json::JsonObject
92      * @return true if object contains a "type" property.
93      */
94     static bool check(const QJsonObject &);
95
96     virtual ~JsonObject() {}
```

```
97
98  protected:
99      JsonObject(Type type) :
100         m_type(type) {}
101 private:
102     Type m_type;
103 };
104
105 }
106 }
107
108 #endif // JSONOBJECT_H
```

Listing 4.3: Json-Objekt Klasse

Listing 4.3 zeigt hingegen die Basisklasse aller JSON-Objekte, die als Nachricht[6] versendet werden können und somit ein »type«-Attribut besitzen. Der Aufzählungstyp »Type« enthält die möglichen Werte dieses Attributes.

Diese Typ-parametrisierten Klassen ermöglichen die in der Protokolldefinition verwendete *typunabhängige* Realisierung des Model-View-Konzeptes. In Listing 4.4, wo stellvertretend für alle anderen JSON-Objekte die Definition der »Orderitem«-Typen abgedruckt ist, zeigen sich die Vorzüge der Template-Klassen[7]: Durch »typdefs« wird dem Compiler mitgeteilt, dass aus den »Orderitem«-Typen die speziellen »Update«- und »KeyValuePair«-Typen erstellt werden müssen (Zeile 93–95).

```
23 #ifndef ORDERITEM_H
24 #define ORDERITEM_H
25
26 #include "jsonconvertable.h"
27 #include "table.h"
28 #include "product.h"
29 #include "idvaluepair.h"
30 #include "jsonmap.h"
31 #include "jsonobject.h"
32 #include "update.h"
33
34 namespace fos {
35 namespace json {
36
37 class OrderitemId : public JsonConvertable<OrderitemId>
38 {
39 public:
40     OrderitemId() {}
41     OrderitemId(const TableId& e_tableId,
42                 const ProductId& e_productId,
43                 const QString& e_extras) :
44         tableId(e_tableId),
45         productId(e_productId),
46         extras(e_extras) {}
47
48     TableId tableId;
49     ProductId productId;
50     QString extras;
51
52     virtual QJsonObject toJson() const;
53
54     static bool check(const QJsonObject &);
55     static OrderitemId fromJson(const QJsonObject &);
56 };
57
58 inline bool operator <(const OrderitemId& id1, const OrderitemId& id2) {
59     if (id1.tableId == id2.tableId) {
```

[6] siehe 3.7.2

[7] Templates bezeichnen die Definition Typ-parametrisierter Klassen in C++

```
60          if (id1.productId == id2.productId) {
61              return id1.extras < id2.extras;
62          }
63          return id1.productId < id2.productId;
64      }
65      return id1.tableId < id2.tableId;
66  }
67
68  inline bool operator ==(const OrderitemId& id1, const OrderitemId&id2) {
69      return id1.tableId == id2.tableId &&
70              id1.productId == id2.productId &&
71              id1.extras == id2.extras;
72  }
73
74
75  class OrderitemValue : public JsonConvertable<OrderitemValue>
76  {
77  public:
78      OrderitemValue() {}
79      OrderitemValue(quint32 e_count, const QString& e_price,
80                     const QString& e_extraprice) :
81          count(e_count), price(e_price), extraprice(e_extraprice) {}
82
83      quint32 count;
84      QString price;
85      QString extraprice;
86
87      virtual QJsonObject toJson() const;
88
89      static bool check(const QJsonObject &);
90      static OrderitemValue fromJson(const QJsonObject &);
91  };
92
93  typedef IdValuePair<OrderitemId, OrderitemValue> Orderitem;
94  typedef JsonMap<OrderitemId, OrderitemValue> OrderitemMap;
95  typedef Update<OrderitemId, OrderitemValue, JsonObject::Type::OrderitemUpdate> ↩
        OrderitemUpdate;
96
97  }
98  }
99
100 Q_DECLARE_METATYPE(fos::json::OrderitemId)
101 Q_DECLARE_METATYPE(fos::json::OrderitemUpdate)
102
103 #endif // ORDERITEM_H
```

Listing 4.4: Orderitem Implementierung

Die Vergleichsoperatoren der »OrderitemId« sowie die Macros in den Zeilen 101 und 102 werden hingegen für die Ablage der Daten in die Qt-Container-Klasse QMap<Key,Value> benötigt.

```
26  QJsonObject fos::json::OrderitemId::toJson() const
27  {
28      QJsonObject o;
29      o.insert("tableId", tableId.toJson());
30      o.insert("productId", productId.toJson());
31      o.insert("extras", QJsonValue(extras));
32      return o;
33  }
34
35  bool fos::json::OrderitemId::check(const QJsonObject &o)
36  {
37      return TableId::check(o.value("tableId").toObject()) &&
38              ProductId::check(o.value("productId").toObject()) &&
39              o.value("extras").isString();
40  }
41
```

```
42 fos::json::OrderitemId fos::json::OrderitemId::fromJson(const QJsonObject &o)
43 {
44     return OrderitemId(TableId::fromJson(o.value("tableId").toObject()),
45                        ProductId::fromJson(o.value("productId").toObject()),
46                        o.value("extras").toString());
47 }
```

Listing 4.5: JSON-Konvertierungsmethoden

Listing 4.5 zeigt repräsentativ für andere JSON-Objekte die Implementierung der Konvertierungsmethoden für den »OrderitemId«-Typen.

4.5 Client-Handling

Der Kern des asynchronen Client-Handlings liegt wohl im »LoginHandler«. Dieser garantiert, dass nur für autorisierte Benutzer ein neuer Thread angelegt wird. Essenziell in der Implementierung dieses Designs ist die Abhandlung der eingehenden Verbindungen am TCP-Server (Listing 4.6).

```
75 void Server::incomingConnection(qintptr socketDescriptor)
76 {
77     LoginHandler* loginHandler = new LoginHandler(this, socketDescriptor);
78     loginHandler->moveToThread(&m_loginThread);
79     connect(loginHandler, SIGNAL(loginSuccessfull(ClientHandler*)),
80             this, SLOT(clientLoggedIn(ClientHandler*)));
81     QMetaObject::invokeMethod(loginHandler, "handle");
82 }
```

Listing 4.6: Eingehende Verbindung am Server

Die Verarbeitung der Verbindungsanfragen wird also in einen eigenen Thread – dem »Login«-Thread – verschoben. Der entscheidende Abschnitt ist dem Listing 4.7 zu entnehmen.

```
62 void LoginHandler::read()
63 {
64     m_pTimer->stop();
65     QByteArray data = m_pSocket->readAll();
66     // before he's logged in, we only handle the first object
67     const QList<QSharedPointer<JsonObject>>& list = parseJson(data);
68     QSharedPointer<JsonObject> json = list.size() > 0 ?
69                                       list.first() : QSharedPointer<JsonObject>();
70     if (json.isNull()) {
71         qWarning() << "received crap: json parse error";
72     } else if (json->type() == JsonObject::Type::Login) {
73         QSharedPointer<fos::json::Login> login = json.dynamicCast<Login>();
74         if (m_pDatabaseConnection->checkCredentials(login->username,
75                                                     login->password)) {
76             if (m_pServer->isWaiterLoggedIn(login->username)) {
77                 writeError(Error(Error::ALREADY_LOGGED_IN));
78             } else {
79                 // todo check if waiter is already logged in
80                 QString connectionName = QString("%1:%2").arg(login->username).↩
                        arg(
81                         m_pSocket->socketDescriptor());
82                 ClientHandler* clientHandler = new ClientHandler(m_pSocket,
83                         login->username, m_pServer->databaseFactory()->↩
                            getConnection(connectionName));
84                 emit loginSuccessfull(clientHandler);
85             }
86         } else {
87             writeError(Error(Error::WRONG_CREDENTIALS));
88         }
```

```
89      } else {
90          qDebug() << "received missplaced json object: " << static_cast<int>(json↩
                ->type());
91      }
92      deleteLater();
93  }
```

Listing 4.7: Loginhandler

Listing 4.8 beinhaltet zum einen die achtenswerte Implementation des Konstruktors des »Clienthandler« und zum anderen die allgemeine Abarbeitung von eingehenden JSON-Objekten.

```
34  ClientHandler::ClientHandler(QTcpSocket *socket, QString loginName,
35                               QSharedPointer<DatabaseConnection> connection) :
36      m_loginName(loginName),
37      m_pSocket(socket),
38      m_connection(connection)
39  {
40      // you can't move an object with a parent to another thread
41      m_pSocket->setParent(NULL);
42      moveToThread(&m_thread);
43      m_pSocket->moveToThread(&m_thread);
44      m_pSocket->setParent(this);
45
46      connect(&m_thread, SIGNAL(started()), this, SLOT(sendFirstObjects()));
47      connect(m_pSocket, SIGNAL(disconnected()), this, SIGNAL(loggedOut()));
48      connect(m_pSocket, SIGNAL(readyRead()), this, SLOT(receiveJson()));
49      m_thread.setObjectName(QString("%1:%2").arg(m_loginName).arg(m_pSocket->↩
            socketDescriptor()));
50      m_thread.start();
51  }
52
53  ClientHandler::~ClientHandler()
54  {
55      m_thread.quit();
56      m_thread.wait();
57  }
58
59  void ClientHandler::sendFirstObjects()
60  {
61      m_pDataManager = m_connection->getDataManager(this);
62      connect(m_pDataManager, SIGNAL(productUpdate(fos::json::ProductUpdate)),
63              this, SLOT(updateProduct(fos::json::ProductUpdate)));
64      connect(m_pDataManager, SIGNAL(categoryUpdate(fos::json::CategoryUpdate)),
65              this, SLOT(updateCategory(fos::json::CategoryUpdate)));
66      connect(m_pDataManager, SIGNAL(locationUpdate(fos::json::LocationUpdate)),
67              this, SLOT(updateLocation(fos::json::LocationUpdate)));
68      connect(m_pDataManager, SIGNAL(tableUpdate(fos::json::TableUpdate)),
69              this, SLOT(updateTable(fos::json::TableUpdate)));
70      connect(m_pDataManager, SIGNAL(orderitemUpdate(QString,fos::json::↩
            OrderitemUpdate)),
71              this, SLOT(updateOrderitem(QString,fos::json::OrderitemUpdate)));
72
73      writeJsonObject(*m_pDataManager->getInitialData(m_loginName));
74  }
75
76  void ClientHandler::handleJsonObject(QSharedPointer<JsonObject> jsonObject)
77  {
78      if (jsonObject.isNull()) {
79          qWarning() << "received crap: json parse error";
80      } else {
81          switch (jsonObject->type()) {
82          case JsonObject::Type::Logout:
83              if (jsonObject.dynamicCast<Logout>()->printClearance) {
84                  emit clearance();
85              }
86              m_pSocket->disconnectFromHost(); // emits disconnected --> loggedOut
```

```
87              break;
88          case JsonObject::Type::Order: {
89              auto object = jsonObject.dynamicCast<Order>();
90              emit order(*object);
91              m_pDataManager->order(m_loginName, *object);
92              break;
93          }
94          case JsonObject::Type::Checkout: {
95              auto object = jsonObject.dynamicCast<Checkout>();
96              QString price = m_pDataManager->checkout(m_loginName, *object);
97              if (price.isEmpty()) {
98                  qDebug() << "checkout failed, writing error...";
99                  writeJsonObject(Error(Error::CHECKOUT_OUT_OF_SYNC));
100             } else {
101                 writeJsonObject(json::CheckoutConfirmation(price));
102             }
103             break;
104         }
105         case JsonObject::Type::Countermand: {
106             auto object = jsonObject.dynamicCast<Countermand>();
107             qDebug() << "received countermand...";
108             if (!m_pDataManager->counter(m_loginName, *object)) {
109                 qWarning() << "countermand failed, writing error...";
110                 writeJsonObject(Error(Error::COUNTERMAND_OUT_OF_SYNC));
111             } else {
112                 emit counter(*object);
113             }
114             break;
115         }
116         case JsonObject::Type::CategoryUpdate:
117         case JsonObject::Type::ProductUpdate:
118         case JsonObject::Type::LocationUpdate:
119         case JsonObject::Type::TableUpdate:
120         case JsonObject::Type::OrderitemUpdate:
121         case JsonObject::Type::InitialData:
122         case JsonObject::Type::CheckoutConfirmation:
123         case JsonObject::Type::Error:
124         case JsonObject::Type::Login:
125             qWarning() << "received misplaced json object, total nonsense";
126             break;
127         }
128     }
129 }
```

Listing 4.8: Clienthandler

4.6 Datenverwaltung

Das Listing 4.9 zeigt die CREATE-TABLE-Befehle wie sie in Qt und PostgreSQL umgesetzt wurden und damit die Implementierung im Feinst-Modell des Datenbankdesigns.

```
126     result = q.exec("CREATE TABLE waiters("
127                     "  id SERIAL PRIMARY KEY,"
128                     "  login VARCHAR(30) UNIQUE NOT NULL,"
129                     "  password BYTEA NOT NULL,"
130                     "  firstname VARCHAR(50),"
131                     "  lastname VARCHAR(50),"
132                     "  lastlogin TIMESTAMP"
133                     ")");
134
135     if (!result) goto error;
136
137     result = q.exec("CREATE TABLE locations("
138                     "  id SERIAL PRIMARY KEY,"
139                     "  name VARCHAR(30),"
140                     "  abbreviation VARCHAR(6) NOT NULL"
```

```
141                     ")");
142
143     if (!result) goto error;
144
145     result = q.exec("CREATE TABLE tables("
146                     "  id INTEGER NOT NULL CHECK(id > 0),"
147                     "  instances SMALLINT NOT NULL DEFAULT 1,"
148                     "  location INTEGER NOT NULL REFERENCES locations"
149                     "     ON DELETE CASCADE,"
150                     "  PRIMARY KEY(id, location)"
151                     ")");
152
153     if (!result) goto error;
154
155
156     result = q.exec("CREATE TABLE categories("
157                     "  id SERIAL PRIMARY KEY,"
158                     "  name VARCHAR(30) NOT NULL UNIQUE,"
159                     "  abbreviation VARCHAR(6) NOT NULL"
160                     ")");
161
162     if (!result) goto error;
163
164     result = q.exec("CREATE TABLE printers("
165                     "  id SERIAL PRIMARY KEY,"
166                     "  name VARCHAR(30),"
167                     "  ip CIDR NOT NULL"
168                     ")");
169
170     if (!result) goto error;
171
172     result = q.exec("CREATE TABLE products("
173                     "  id SERIAL PRIMARY KEY,"
174                     "  name VARCHAR(30),"
175                     "  abbreviation VARCHAR(6) NOT NULL,"
176                     "  description VARCHAR(100),"
177                     "  price DECIMAL NOT NULL CHECK(price >= 0),"
178                     "  category INTEGER REFERENCES categories"
179                     "    ON DELETE CASCADE,"
180                     "  printer INTEGER REFERENCES printers"
181                     ")");
182
183     if (!result) goto error;
184
185     result = q.exec("CREATE TABLE orderitems("
186                     "  \"table\" INTEGER NOT NULL,"
187                     "  location INTEGER NOT NULL,"
188                     "  instance SMALLINT NOT NULL DEFAULT 1,"
189                     "  product INTEGER NOT NULL REFERENCES products,"
190                     "  waiter INTEGER NOT NULL REFERENCES waiters,"
191                     "  extras VARCHAR(200) NOT NULL DEFAULT '',"
192                     "  count SMALLINT NOT NULL,"
193                     "  extraprice DECIMAL NOT NULL DEFAULT 0,"
194                     "  firstorder TIMESTAMP,"
195                     "  lastorder TIMESTAMP,"
196                     "  PRIMARY KEY (\"table\", location, instance, product, ↩
                            waiter, extras),"
197                     "  FOREIGN KEY (\"table\", location) REFERENCES tables (id, ↩
                            location)"
198                     ")");
```

Listing 4.9: CREATE-Table Befehle in PSQL

Das Listing 4.10 beherbergt mit der Methode »order« zum Bestellen eines einzelnen Elementes der Bestellliste (wird in einer Schleife aufgerufen) und der Methode »recalculateTablePrice«, welche den Preis eines Tisches nach jeglichen Veränderungen (Bestellen, Stornieren, getrenntes Bezahlen) neu berechnet und ein »TableUpdate« entsendet, einen Auszug der Implementierung vom »DataManager«.

```
760 void PSQLDataManager::order(const fos::json::TableId &tableId,
761                               const fos::json::OrderRequest& orderRequest,
762                               const QString &waiter)
763 {
764     QSqlQuery query(m_database);
765     query.prepare("SELECT waiters.id FROM orderitems JOIN waiters ON waiter=id "
766                   "WHERE \"table\" = :t AND location = :location "
767                   "AND instance = :i AND product = :p AND extras = :e "
768                   "AND login = :waiter");
769
770     query.bindValue(":t", tableId.number);
771     query.bindValue(":location", tableId.locationId.id);
772     query.bindValue(":i", tableId.instance);
773     query.bindValue(":p", orderRequest.productId.id);
774     query.bindValue(":e", orderRequest.extras);
775     query.bindValue(":waiter", waiter);
776
777     Q_ASSERT(query.exec());
778     Q_ASSERT(query.size() < 2); // WHERE clause contains complete primary key
779
780     if (query.size() == 0) {
781         // not ordered yet from this waiter
782         query.prepare("INSERT INTO orderitems(\"table\", location, instance, "
783                       "product, waiter, extras, count, extraprice, firstorder,"
784                       "lastorder)"
785                       "VALUES(:number, :location, :instance, :product, "
786                       "(SELECT id FROM waiters WHERE login = :login), :extras, "
787                       ":count, :extraprice , NOW(), NOW())");
788         query.bindValue(":number", tableId.number);
789         query.bindValue(":location", tableId.locationId.id);
790         query.bindValue(":instance", tableId.instance);
791         query.bindValue(":product", orderRequest.productId.id);
792         query.bindValue(":login", waiter);
793         query.bindValue(":extras", orderRequest.extras);
794         query.bindValue(":count", orderRequest.count);
795         query.bindValue(":extraprice", orderRequest.extraprice.toDouble());
796         Q_ASSERT(query.exec());
797     } else {
798         // waiter id: query.value(0);
799         // already orderded from this waiter
800         Q_ASSERT(query.first());
801         quint32 waiterId = query.value(0).value<quint32>();
802         query.prepare("UPDATE orderitems SET count = count + :addCount, "
803                       "                      lastorder = NOW()"
804                       "WHERE \"table\" = :number AND location = :location AND "
805                       "instance = :instance AND product = :product AND "
806                       "waiter = :waiter AND extras = :extras");
807         query.bindValue(":number", tableId.number);
808         query.bindValue(":location", tableId.locationId.id);
809         query.bindValue(":instance", tableId.instance);
810         query.bindValue(":product", orderRequest.productId.id);
811         query.bindValue(":waiter", waiterId);
812         query.bindValue(":extras", orderRequest.extras);
813         query.bindValue(":addCount", orderRequest.count);
814         Q_ASSERT(query.exec());
815     }
816
817     query.prepare("SELECT SUM(count), SUM(count*price) + SUM(count*extraprice) ↩
        FROM orderitems "
818                   " JOIN products ON product=id "
819                   " WHERE \"table\" = :table AND location = :location "
820                   " AND instance = :instance AND product = :product "
821                   " AND extras = :extras");
822     query.bindValue(":table", tableId.number);
823     query.bindValue(":location", tableId.locationId.id);
824     query.bindValue(":instance", tableId.instance);
825     query.bindValue(":product", orderRequest.productId.id);
826     query.bindValue(":extras", orderRequest.extras);
827     Q_ASSERT(query.exec());
```

```
828     Q_ASSERT(query.size() == 1);
829     Q_ASSERT(query.first());
830
831     quint32 count = query.value(0).value<quint32>();
832     json::OrderitemId id(tableId, orderRequest.productId, orderRequest.extras);
833     json::OrderitemValue value(count, query.value(1).toString(),
834                                orderRequest.extraprice);//hallo schotzi :D
835     json::OrderitemUpdate::Command command = json::OrderitemUpdate::Command::↩
            Update;
836     if (count == orderRequest.count)
837         command = json::OrderitemUpdate::Command::Insert;
838
839     m_orderitems->operator [](id) = value;
840     emit m_pRootManager->orderitemUpdate(waiter, json::OrderitemUpdate(
841                                              command, id, value));
842 }
843
844 void PSQLDataManager::recalculateTablePrice(const fos::json::TableId &tableId)
845 {
846     QSqlQuery query(m_database);
847     query.prepare("SELECT SUM(count*price) FROM orderitems "
848                   " JOIN products ON product=id "
849                       " WHERE \"table\" = :table AND location = :location "
850                   " AND instance = :instance");
851     query.bindValue(":table", tableId.number);
852     query.bindValue(":location", tableId.locationId.id);
853     query.bindValue(":instance", tableId.instance);
854
855     Q_ASSERT(query.exec());
856     Q_ASSERT(query.size() == 1);
857     Q_ASSERT(query.first());
858
859     QString newPrice = query.value(0).toString();
860
861     m_clientTables->operator [](tableId) = newPrice;
862     emit m_pRootManager->tableUpdate(json::TableUpdate(json::TableUpdate::Command↩
            ::Update,
863                                      tableId, json::TableValue(newPrice)));
864 }
```

Listing 4.10: DataManager Klasse

4.7 Benutzeroberfläche

Als letzten Teil der Implementierung seien noch einige Bildschirmfotos der Benutzeroberfläche des Servers angehängt.

Die Abbildungen 4.6 und 4.7 zeigen das Datenverwaltungsfenster, welches Änderungen am Datenbestand direkt an die verbundenen Clients weiterleitet.

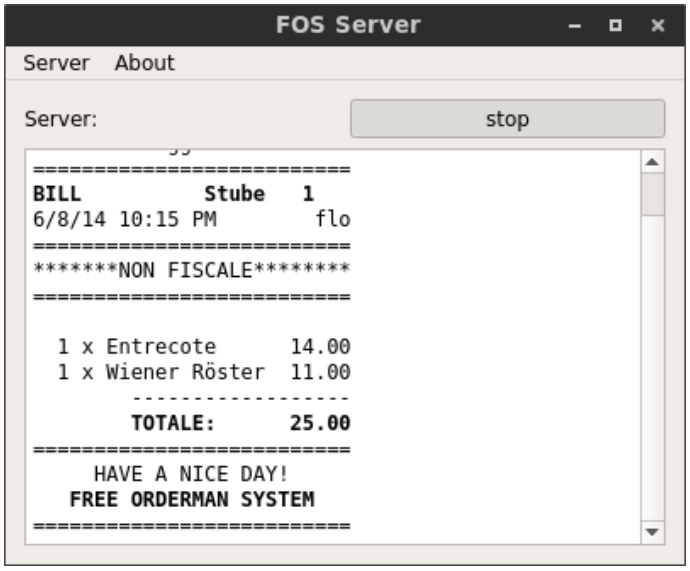


Abbildung 4.5: Server Fenster mit den letzten Ereignissen

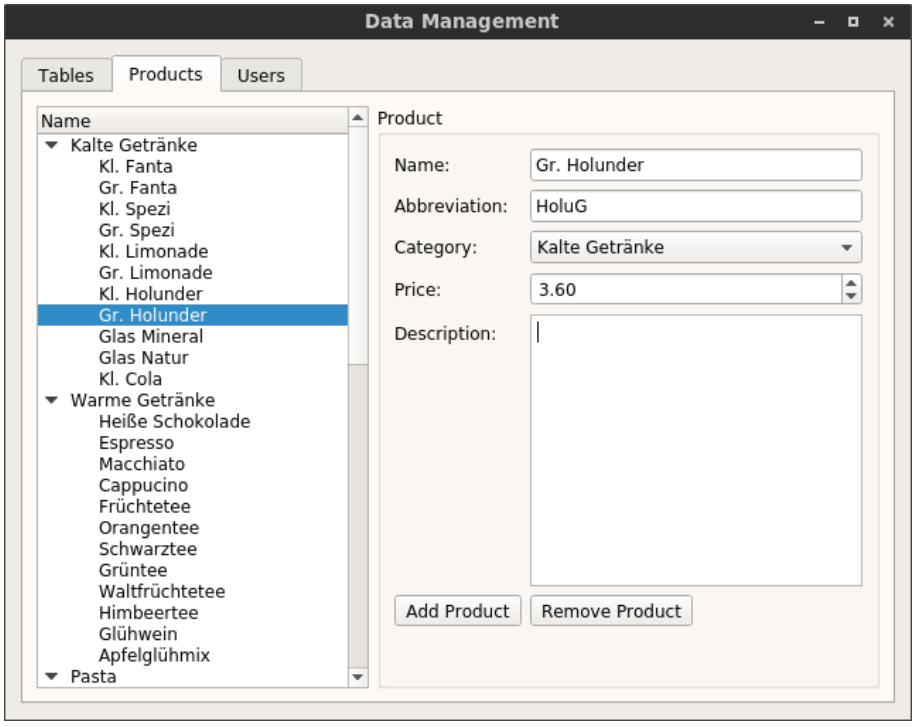


Abbildung 4.6: Server Datenverwaltungsoberfläche Produkte

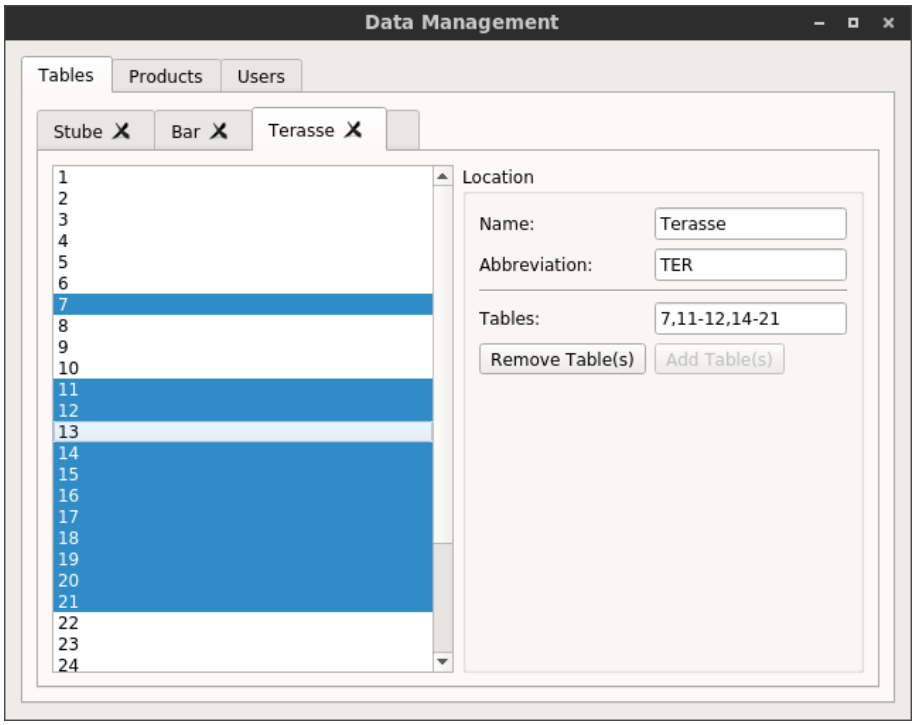


Abbildung 4.7: Server Datenverwaltungsoberfläche Tische

Kapitel 5

Evaluierung

Wie bereits in der Einleitung als Ziel definiert[1], soll das System im Produktiveinsatz in einem Gastronomiebetrieb getestet werden. Die dafür geeignete Lokalität fand sich in Salurn, wo Alex Mazzon's Onkel »Heini« eine Bar und Eisdiele betreibt.

5.1 Bar Salurn

Wie viele Kleinbetriebe in Südtirol scheut Bar Salurn die Anschaffungskosten eines professionellen Kassensystems, da die finanziellen Nachteile überwiegen. Damit zählt diese Bar als Musterbeispiel zur Zielgruppe des Free Orderman Systems und kann durch dieses Projekt in den Genuss der Vorzüge eines automatisierten Kassensystems kommen.

5.1.1 Status quo

Das Service-Personal verwendet bis dato klassisch Block und Feder zur Aufnahme der Bestellungen und behält durch eine Korkplatte hinter der Theke den Überblick über die laufenden Bestellungen. Abgerechnet wird über eine herkömmliche Registrierkasse, welche durch das Eintippen der Mengen und Preise den Kassenbon druckt.

5.2 Testbedingungen

Am späten Nachmittag des 14. Mai 2014 herrschte die passend ruhige Atmosphäre für den ersten Test vor.

5.2.1 Hardware

Zu Entwicklungs– und Testzwecken wurde im Vorfeld der verbreitete Bondrucker TM-T20 der Firma Epson mit Netzwerkschnittstelle angekauft. Zusammen mit zwei Smartphones, einem Laptop und dem schon vorhandenen Router erfüllte dieser die Hardwareanforderungen für den Praxistest.

5.2.2 Vorbereitung

Das WLAN-Netzwerk wurde mit dem Router eingerichtet, der Drucker durch einen Twisted-Pair-Kabel mit der Ethernet-Schnittstelle am Router verbunden. Als DHCP-Client konfiguriert, bekam der Drucker eine IP-Adresse zugewiesen. Daraufhin wurde der Netzwerkdrucker am Laptop mit der entsprechenden Adresse eingerichtet und konnte nun Druckaufträge verarbeiten.

[1] siehe Abschnitt 1.3

Am Laptop[2] startete der Server und eine neu angelegte Datenbank wurde mit dem Sortiment und den Tischen der Bar Salurn gefüllt. Der Laptop und die zwei Smartphones meldeten sich im WLAN-Netz an und waren dadurch wie gewünscht im selben Netzwerk. Auf den Smartphones wurde die Android-Version des Clients installiert und die IP-Adresse des Laptops bekanntgegeben.

5.2.3 Beteiligte Kellner

Alex Mazzon und sein etwas jüngerer Cousin fungierten als Kellner mit ihrem Smartphone.

5.3 Verlauf

Mit der Anmeldung der beiden Kellner über das Smartphone startete der Test reibungslos und es klappte alles wie geplant. Die ersten Kunden wurden mit dem Smartphone bedient und der Drucker brachte die erste echte Bestellung hervor, was sich in diesem unscheinbaren und doch so bedeutsamen Bild (Abbildung 5.1) manifestiert.

Die Kassenbons, welche im Laufe des Tests gedruckt worden sind, mussten alle noch per Hand in die Registrierkasse eingetippt werden, damit sie finanztechnisch registriert und *gültige* Bons ausgehändigt werden konnten.

Ansonsten erwies sich das System als intuitiv und schnell in der Bedienung. Neue Bestellungen erschienen in nicht erkennbarer Verzögerung auf dem jeweils anderen Smartphone und die Applikation wurde als ruckelfrei und gutmütig empfunden.

Abbildung 5.1: Erster Produktiveinsatz vom Free Order System

5.4 Bewertung

Der Chef der Bar, »Heini«, äußerte sich überaus positiv zum Free Orderman System. Beeindruckt von der unerwarteten Praxistauglichkeit und begeistert von der intuitiven Bedienung und Geschwindigkeit des Systems beschloss er, das System sobald als möglich im Alltagsgeschäft einzusetzen.

Den Worten ließ er nur 1 Woche später Taten folgen und investierte in einen neuen Router und in ein Smartphone, die dem Einsatz von FOS dienen.

[2]Betriebssystem Ubuntu 14.04

Kapitel 6

Schlusswort

Das Projekt erwies sich hinsichtlich Umfang, Praxistauglichkeit und erschlossener Themenfelder in den letzten Schulmonaten als voller Erfolg. Nicht nur Klassenkameraden, Bekannte und Verwandte, sondern auch Fachlehrer und unvoreingenommene Besucher des Projekttages hinterließen ein rundum positives Feedback. Der erfolgreiche Praxistests und der nun effektive Einsatz des Free Orderman Systems in der Bar in Salurn verwirklichen die kühnsten Träume.

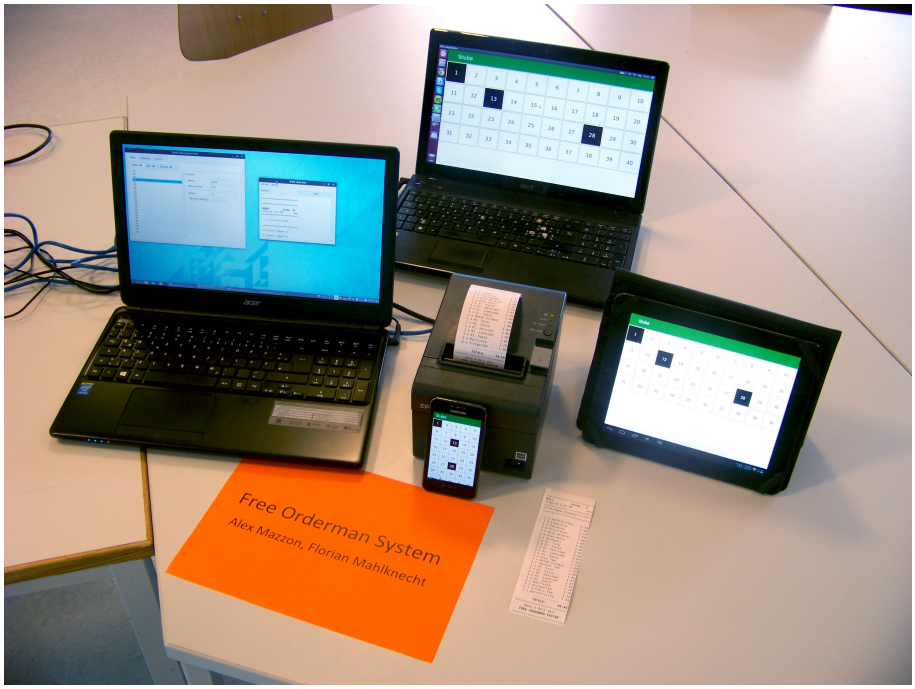


Abbildung 6.1: Vorstellung am Projekttag der Schule

6.1 Zukunftsperspektiven

Zweifelsohne wird ein lebendiges Fortbestehen von *FOS* angestrebt. Mit gebündelter Kraft soll der Übergang in ein eigenständiges Projekt gelingen.

Die erste große Hürde, die es dabei zu meistern gilt, ist die Veröffentlichung der Software für die diversen Betriebssysteme in Form einfacher Pakete mit benutzerfreundlichen Installationsroutinen. Verständlicherweise braucht es dafür recht spezielle Kenntnisse von den jeweiligen Systemen und deren Installationspraktiken, sodass es doch eine nicht zu

unterschätzende Schwierigkeit darstellt. Ist diese geschafft kann die erste Publikation erfolgen, welche vom „Otto-Normal-Verbraucher" ohne Compiler und Entwicklungsumgebung genutzt werden kann.

Die nächste große Hürde stellen die Kassenbons dar, die nach den geltenden Gesetzten *registriert* werden müssen. Um diese zu überwinden wird eine Kooperation mit vergleichbaren Projekten angestrebt, wodurch größeren Schwierigkeiten von Beginn an aus dem Weg gegangen werden kann.

6.1.1 Monetarisierung

Am Ende stellt sich noch die Frage nach einer möglichen Wertschöpfung des Projekts. Vielen Menschen erscheint die Möglichkeit mit *freier* Software Profit zu erzielen kontrovers. In der Praxis haben sich jedoch mehrere Ansätze dafür entwickelt. Einer davon wäre auf Support und Service zu setzen und dementsprechend durch kostenpflichtige *Dienstleistungen*, wie sie bei der Installation des Systems, bei der Auswahl der Geräte[1] und wunschgemäßen Weiterentwicklungen sowie Fehlerkorrekturen gebraucht werden, Gewinne zu erzielen.

Die Entwicklungen in dieser Hinsicht stehen jedoch noch in den Sternen, sodass das Projekt zunächst im Rahmen der Maturaprüfung sicherlich als positiv abgeschlossen erachtet werden kann. Danach wird sich zeigen was die Zukunft mit sich bringt.

[1] Folien zur Entspiegelung von Displays, Stifte zur Bedienung des Touchscreens, staub– und wasserfeste Smartphones, ... um nur einige der möglichen Beratungsfelder zu nennen

Anhang A

CD Inhalt

Auf der beiliegenden CD finden Sie folgendes:

- Quellcode
- Ausführbare Dateien für Client und Server (ELF) für 64-Bit Linux
- die vorliegende Arbeit als PDF-Dokument und den dazugehörigen LaTeX-Quellcode
- alle angeführten Quellen als PDF Dokument[1]
- alle verwendeten Grafiken

Der Quellcode entspricht dabei dem beim Erscheinen der Arbeit (v0.3.55) aktuellen Inhalt des Launchpad-Repositories[2]

[1] ausgenommen dem Buch [BREY]
[2] siehe Abschnitt 4.2.2, Seite 28

Glossar

Aktivitätsdiagramm

In UML standardisiertes Diagramm zur Darstellung von sowohl softwaretechnischen als auch geschäftlich-industriellen Abläufen. 5

Betriebssystem

Ansammlung von Software welche das System verwaltet und die Ausführung von Anwendungsprogrammen ermöglicht. 3, 22, 42, 43

Bibliothek

Datei, die eine Sammlung von Funktionen beinhaltet und von einem Programm *verwendet* werden kann. 49

C++

Umfangreiche standardisierte Programmiersprache, welche u. a. objektorientierte, prozuderale und generische Programmierung ermöglicht. 22, 23, 31

Community

Allgemeine Bezeichnung einer Gemeinschaft, welche das Internet als Kommunikations- und Austauschplattform nutzt. 27

Compiler

Programm, welches den Quellcode in Maschinencode übersetzt und damit die Grundlage ausführbarer Dateien bildet. 44

DBMS

Datenbankmanagementsystem, in der Informationsverwaltung eingesetzt ist es für Speicherung, Sicherheit und Konsistenz von Daten verantwortlich. 15, 24

Entität

Sammelbegriff für existierende konkrete oder abstrakte Gegenstände; in der Informatik als Realisierung eines Entitätstypen zu verstehen, bspw. »Herr Müller« als Realisierung von »Kunde«. 47

Entwicklungsumgebung

Programm, das meist Editor, Linker und Debugger vereint und somit das Entwicklungswerkzeug für einen Programmierer darstellt. 44

ER-Diagramm

Entity-Relation-Ship Diagramm, stellt Entitäten, ihre Eigenschaften und Beziehungen dar und ist meist integraler Bestandteil des Datenbankdesigns. 16, 51

Ethernet

Spezifikation von Soft– und Hardware, welche eine kabelgebundene Vernetzung von Rechnern ermöglicht. 41

Framework

Softwaregerüst, welches die nötige Rahmensoftware (oft mit einer Bibliothek) bietet, um Projekte realisieren zu können. 22

IP

Internet Protcol, häufig verwendetes Netzwerkprotokoll, welches die Vermittlung und Wegsuche für Daten übernimmt. 17

ISO

Internationale Organisation für Normung mit Sitz in Genf. 49

Klasse

Abstraktes Modell, welches in der objektorientierten Programmierung zur Definition und Beschreibung von Objekten dient. 19, 20, 24, 29, 31, 32

Konsistenz

Korrektheit und Integrität von Daten in einer Datenbank oder Gleichheit von replizierten Daten auf verteilten Systemen. 15

LAN

Local Area Network, in Ausdehnung beschränktes Netzwerk, welches für Heimnetzwerke und kleinere Unternehmen eingesetzt wird. 4

Objektorientierte Programmierung

Programmierparadigma, welches versucht (reale) Objekte und deren Eigenschaften durch die Programmiersprache abzubilden. 23, 48

Objekt-Sequenz-Diagramm

In UML standardisiertes Diagramm zur Darstellung des Zusammenspiels von Objekten bei der Erledigung einer Aufgabe. 23

Peer to Peer

Verbindung *gleichberechtigter* Kommunikationspartnern in Rechnernetzen. 12

Persistenz

Bereithaltung von Daten oder logischen Verbindungen über einen längeren Zeitraum. 4, 53

Prozess

stellt die laufende Abarbeitung und Ausführung eines Programms dar. 49

Prozesshierarchiediagramm

Diagramm zur Darstellung von Geschäftsprozessen und Gliederung eines Unternehmens in Geschäftsbereiche. 6

Quelltext

Text nach wohldefinierter Syntax und Semantik einer Programmiersprache, welcher den Ursprung von ausführbaren Programmen darstellt. 12, 27

Router

Netzwerkgerät, welches Pakete zwischen mehreren Rechnernetzen weiterleiten kann und so häufig zur Internetanbindung genutzt wird. 14, 41, 42

Software

Oberbegriff für Programme und deren Bibliotheken, die auf Hardware ausgeführt und eingesetzt werden, um die verschiedensten Aufgaben zu erledigen. iii, 2, 3, 8, 10, 16, 18, 27, 43, 44

TCP

Transmission Control Protcol, verbindungs– und paketorientiertes Transportprotokoll, welches die Art und Weise *wie* Daten übermittelt werden spezifiziert. 17, 33, 49

Thread

Ausführungsstrang in der Abarbeitung eines Programms – ist somit *Teil* eines Prozesses. 23, 33

TLS

Transmission Layer Security, verschlüsselte zu TCP konforme Transportschicht. 17

UML

Unified Modeling Language, von der ISO genormte grafische Modellierungssprache, welche eine Sammlung von Notationen und Diagrammtypen bietet. 47, 48

Use Case Diagramm

Diagramm zur Darstellung der Funktionalitäten eines Systems aus Benutzersicht. 8

Verteilte Systeme

Mehrere autonome Systeme (Computer), die durch Vernetzung zu einer Einheit verschmolzen werden, um gemeinsam Dienste zu erledigen. iii, 10, 12

WLAN

Wireless Local Area Network, in Ausdehnung beschränktes lokales Funknetzwerk, oft auch als WiFi bezeichnet. iii, 2, 13, 14, 17, 41, 42

Stichwortverzeichnis

Abbildungsverzeichnis

Listings

Literaturverzeichnis

[BREY] Ulrich Breymann *Der C++ Programmierer* 2. aktualisierte Auflage 2011, ISBN 978-3-446-42691-7.

[DrKS] Bundesamt für Sicherheit in der Informationstechnik *Drahtlose Kommunikationssysteme und ihre Sicherheilt* `http://www.netzmafia.de/skripten/netze/drahtkom_sicherheit.pdf`, abgerufen am 6. Juni 2014.

[GaSt] Ploner Hospitality Consulting *Gastrostudie*. `http://www.orderman.com/gastrostudie.html`, abgerufen am 2. Juni 2014.

[GiaMuz] Giacomuzzi *Webauftritt*. `www.giacomuzzi.it`, abgerufen am 29. Mai 2014.

[KasSys] Wikipedia *Kassensystem*. Persistenter Link: `http://de.wikipedia.org/w/index.php?oldid=130001418` [1]

[LeProg] LerneProgrammieren.com, Dipl.-Ing. Christian Moser, *Muss man Englisch können um Programmieren zu lernen?*, `http://www.lerneprogrammieren.com/blog/motivation/muss-man-englisch-koennen-um-programmieren-zu-lernen`, abgerufen am 7. Juni 2014

[MoGe] Wikipedia *Mooresches Gesetz*. Persistenter Link: `http://de.wikipedia.org/w/index.php?oldid=130747563` [1]

[OrdUG] Orderman® *Unternehmengeschichte*. `http://www.orderman.com/ueber-uns/unternehmensgeschichte.html`, abgerufen am 8. Mai 2014.

[QtDoc] Qt *Dokumentation*. `doc.qt-project.org`, abgerufen am 8. Juni 2014.

[RS485] Wikipedia *RS485*, englische Version. Persistenter Link: `http://en.wikipedia.org/w/index.php?oldid=595190791` [1]

[1] Eine Liste der Wikipedia Autoren ist in den PDF-Versionen der Quellen enthalten (siehe Anhang A, Seite 45)