

Softwareprojekt

Entwicklung eines
Verwaltungsprogramms für ein
Kreuzfahrtschiff mit
JavaFX und Microstream



Entwickelt von:

David Elsner

M-ITMO

Herr Albers

Hans-Böckler-Berufskolleg Marl / Haltern am See

Abgabe: 14.08.2020



*Anker Kreuzfahrten GmbH
Seit 1962 in Flensburg*

Inhaltsverzeichnis

Projekt: Entwicklung eines Verwaltungsprogramms für ein großes Kreuzfahrtschiff mit JavaFX	1
Hintergrund des Projektes: Die Reederei Anker Kreuzfahrten GmbH.....	1
Unternehmensgeschichte	1
Die Reederei im Kurzüberblick	2
Die Schiffe	3
Problemstellung.....	4
Pflichtenheft zum Verwaltungsprogramm	5
1. Visionen und Ziele	5
2. Rahmenbedingungen.....	5
3. Kontext und Überblick	5
4. Funktionelle Anforderungen.....	5
5. Produktleistungen und Wünsche	6
Projektplanung	7
Entwicklungsumgebung, Plugins, Zusatzprogramme.....	7
Zeit- und Ressourcenplanung	7
Meilensteine des Projektes	8
MySQL-Datenbank	8
Probleme mit der MySQL-Datenbank und deren Behebung	10
Projektdurchführung	11
Die Entwicklung der grafischen Benutzeroberflächen für das Programm	11
Der Aufbau einer grafischen Benutzeroberfläche.....	12
Probleme mit der Entwicklung der grafischen Benutzeroberflächen und deren Behebung	14
Zugriff auf die MySQL-Datenbank.....	14
Installation des MySQL-Connectors	14
Verbindung mit der Datenbank herstellen	15
Daten aus der Datenbank lesen	16
Daten in der Datenbank bearbeiten	20
Daten in der Datenbank einfügen.....	22
Daten aus der Datenbank löschen	23
Zugriff auf die Microstream Datenbank	24
Aufbau einer Microstream Datenbank	25
MicrostreamDB installieren.....	26
Daten aus MicrostreamDB lesen	27
Daten in MicrostreamDB bearbeiten	29
Daten in MicrostreamDB einfügen	32

Daten in MicrostreamDB löschen	34
Daten in MicrostreamDB sortieren	34
Probleme bei der Implementierung von MicrostreamDB.....	36
Die Fachklasse ErrorHandler	36

Projekt: Entwicklung eines Verwaltungsprogramms für ein großes Kreuzfahrtschiff mit JavaFX

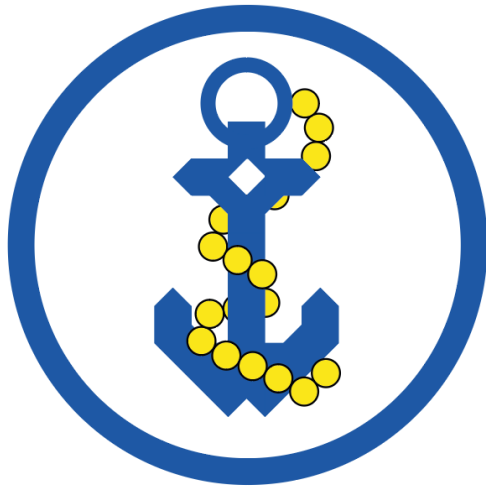
Hintergrund des Projektes: Die Reederei Anker Kreuzfahrten GmbH

Unternehmensgeschichte

Die Flensburger Reederei Anker Kreuzfahrten GmbH wurde im Jahre 1962 von Edward Anker gegründet und bot den Kunden in der Anfangszeit Flusskreuzfahrten auf Dampfschiffen an. Nach und nach wurde die Flotte erweitert und die Schiffe immer nobler, sodass sie eher an schwimmende Luxushotels erinnerten als an gewöhnliche Flusskreuzer. Das Unternehmen verwendete für seine Flusskreuzfahrten ausschließlich historische Dampfschiffe, welche für ihren alten Charm berühmt waren und in der wachsenden Touristikbranche ein Unikat blieben.

Allerdings stiegen die Unterhaltskosten für Dampfschiffe im Laufe der Jahre rapide an, sodass die Geschäftsleitung gezwungen war, sie nach und nach zu verkaufen. Außerdem gestaltete sich die Ersatzteilbeschaffung für Wartungsarbeiten als äußerst schwierig, wodurch die Kosten ebenfalls in die Höhe getrieben wurden. Nur durch zusätzliche Hilfskredite konnte das Überleben des Unternehmens während den Wirtschaftskrisen der 70er Jahre gesichert werden, doch sie belasteten die Geschäftskonten erheblich.

Nachdem Edward Anker im Jahre 2006 verstarb, wurde das Unternehmen von seinem Sohn Aaron Anker weitergeführt. Dabei sind zwei der historischen Dampfschiffe stets im Besitz der Familie geblieben und fungieren als schwimmendes Hotel mit Sternegastronomie sowie als Firmensitz. Mit dem Erlös der alten Flotte wurden fünf Schiffe erworben, welche jeweils einen anderen Teil der Welt bereisen. Das neueste Schiff Green Pioneer wurde erst 2013 in den Dienst gestellt und wird mit Windkraft angetrieben, um die Umwelt und das Meer besser zu schützen.



Anker Kreuzfahrten GmbH
Seit 1962 in Flensburg

Name: Anker Kreuzfahrten GmbH

Rechtsform: Gesellschaft mit beschränkter Haftung (GmbH)

Gründung: 24. August 1962

Sitz: An den Anfurten 12
24944 Flensburg

Telefon: 0461 / 21420915-1

Telefax: 0461 / 21420910

Geschäftsführer: Aaron Anker

Mitarbeiterzahl: ca. 1400

Umsatz: 940 Millionen Euro

Branche: Touristik, Kreuzfahrten

Website: www.ankerkreuzfahrten.de

E-Mail: info@ankerkreuzfahrten.de

Die Schiffe

Wie bereits oben in der Geschichte des Unternehmens erwähnt, wurden die ersten Schiffe noch mit Dampf angetrieben und aufgrund der teuren Wartungskosten außer Dienst gestellt und verkauft. Lediglich zwei von ihnen befinden sich noch im Besitz des Unternehmens: Die *Flensburg* dient heute als schwimmender Firmensitz, während die *Adler* zu einem schwimmenden Sternerestaurant umgebaut wurde. Mit dem Erlös des Verkaufs der restlichen Flotte und den Umsatz konnten im Laufe der Jahre weitere Schiffe erworben werden; derzeitig sind die in der Tabelle abgebildeten Schiffe noch im Dienst.

SchiffsNr	Name	Flagge	Länge in m	Breite in m	Leistung in PS	Passagierkapa- zität	Anzahl Zim- mer	Bauwerft	Indienststel- lung	Reisekonti- nent
1	Rising Sun	Deutsch- land	165,30	41	24.705	1220	330	Meyer Werft Pa- penburg	21. Februar 1989	Asien
2	Mayflower	Deutsch- land	178,90	38	27.217	1380	395	Meyer Werft Pa- penburg	18. August 1994	Nordame- rika
3	European Jewel	Deutsch- land	181,51	39	28.312	1449	404	Meyer Werft Pa- penburg	06. Oktober 2005	Europa
4	Magnolia	Deutsch- land	174,77	45	26.935	1338	363	Meyer Werft Pa- penburg	22. Juni 2011	Karibik
5	The Green Pioneer	Deutsch- land	130	16	5.085	241	110	Meyer Werft Pa- penburg	30. Mai 2013	Weltweit

Problemstellung

Die derzeitig eingesetzte Software zur Verwaltung von Stammdaten wie Passagiere, Kabinen und Bordpersonal ist völlig veraltet. So wird beispielsweise ein altes Datenbanksystem aus den 70er Jahren eingesetzt, welches noch mit einer Konsolenanwendung bedient werden muss. Gleichzeitig gibt es immer weniger Informatiker, die sich mit dem alten System auskennen und es warten können, sodass es von Jahr zu Jahr immer schwierig wird, das System am Laufen zu halten. Moderne Programme können ebenfalls nicht genutzt werden, da das Verwaltungssystem keine Schnittstellen besitzt, mit dem moderne Programme an das System angebunden werden können. Deswegen müssen die Daten erst aufwendig exportiert und für das jeweilige aufgearbeitet werden, um sie bequem bearbeiten zu können. Dies kostet nicht nur Zeit, sondern auch Geld, denn die Mitarbeiter, welche die Daten aufbereiten, müssen schließlich auch bezahlt werden.

Ein weiterer wichtiger Faktor beim Einsatz alter Programme ist der Support. Das Softwareunternehmen, welches das Verwaltungssystem auf dem Markt gebracht hat, ist längst insolvent, obwohl der Herr Anker erst vor fünf Jahren einen weiteren Support- bzw. Wartungsvertrag mit dem Unternehmen abschloss. Schon damals deuteten sich in den Bilanzen und den immer langsam werdenden Entwicklungszyklus der Software an, dass das Unternehmen eines Tages Insolvenz anmelden wird und die Software seitens des Urhebers nicht mehr gewartet werden kann.

Neue Mitarbeiter an Bord müssen mit viel Mühe in die Verwaltungssoftware eingearbeitet werden, wodurch die Kosten ebenfalls in die Höhe getrieben werden, denn auch hier müssen schließlich Gehälter gezahlt werden. Die Mitarbeiter beschwerten sich häufig über die Benutzerunfreundlichkeit des Programmes; die Kapitäne fanden sogar mehrere Fehler in den Stammdaten, weil das Programm nicht richtig mit Sonderzeichen und Umlauten umgehen kann.

Weil sich die Beschwerden der Fehler immer mehr häuften, beauftragte der Geschäftsführer Herr Anker die Entwicklung eines komplett neuen Programms, welches nicht nur mit modernen Schnittstellen arbeitet, sondern auch viel benutzerfreundlicher ist als die derzeitig eingesetzte Verwaltungssoftware. Nachdem die mit dem Auftrag betreuten Entwickler JavaFX als Framework vorschlugen und die Vorteile dieses neuen Frameworks ausgiebig auf einer Konferenz mit dem Geschäftsführer und den Kapitänen der fünf Schiffe erläuterten, wurde dem grünes Licht gegeben, sodass mit der Entwicklung schnellstmöglich begonnen werden kann. Dazu machten sie sich während der Konferenz Notizen zu den Kundenwünschen und Anforderungen des neuen Programms und fassten sie in einem Pflichtenheft zusammen.

Pflichtenheft zum Verwaltungsprogramm

1. Visionen und Ziele

/V10/ Die Mitarbeiter auf den Schiffen sollen in der Lage sein, die Stammdaten effektiv zu verwalten.

/Z10/ Die alte Software soll komplett durch das neue System ersetzt werden.

/Z20/ Die Produktivität soll durch das neue System enorm gesteigert werden.

/Z30/ Die Einarbeitung in das neue System soll deutlich einfacher sein als bei der alten Software.

/Z40/ Das System soll wartungsfreundlicher sein.

/Z50/ Die *The Green Pioneer* soll als erste Testumgebung für das neue System dienen

2. Rahmenbedingungen

/R10/ Das neue Verwaltungssystem ist eine nautische Anwendung zur Erfassung und Verwaltung von Schiffsstammdaten wie Passagiere, Kabinen und Bordpersonal.

/R20/ Die Zielgruppe des Systems sind die Mitarbeiter im Firmensitz und die Führungskräfte an Bord der Schiffe (Kapitäne und Offiziere).

/R30/ Das System wird sowohl in einer Büroumgebung als auch an Bord von Schiffen eingesetzt.

/R40/ Das System muss in der Lage sein, mindestens 18 Stunden arbeiten zu können.

/R50/ Die eingesetzte Software ist MySQL-Server und Java mit dem Framework JavaFX und Maven.

/R60/ Jeder Arbeitsrechner (egal ob an Bord oder an Land) ist via VPN mit dem Server verbunden, welcher darüber hinaus über einen Internetanschluss verfügt.

3. Kontext und Überblick

/K10/ Das System besitzt eine Schnittstelle zu Logblog 4, einem Verwaltungssystem von Logbüchern.

4. Funktionelle Anforderungen

/F10/ Das System soll auf einer MySQL-Datenbank zugreifen können.

/F11/ Nicht jedes Attribut muss mit Werten gefüllt werden.

/F12/ Das Primärattribut zur Identifizierung von Datensätzen muss einmalig sein und darf keine Dopplungen enthalten (z.B. PersonalNr, KabinenNr, PassagierNr, ...).

/F20/ Die Datensätze sollen durch die Suche nach der entsprechenden Identifikationsnummer vollständig im Programm angezeigt werden.

/F30/ Die Verwaltung der Stammdaten (Passagiere, Bordpersonal, Kabinen) erfolgt in eigenen Benutzeroberflächen.

/F31/ Die Benutzeroberflächen sollen mit FXML und CSS generiert und formatiert werden.

/F40/ Standardmäßig werden die Daten aufsteigend nach der Identifikationsnummer sortiert. Dennoch soll es noch weitere Sortierungen und Gruppierungen geben:

1. Für die Passagierkabinen:
 - a. Absteigend nach der Größe in m²
 - b. Aufsteigend nach Preis der Kabine
2. Für die Passagiere:
 - a. Aufsteigend nach Nachnamen des Passagiers
 - b. Gruppiert nach der HafenNr, wohin der Passagier reisen will
3. Für das Bordpersonal:
 - a. Aufsteigend nach Nachnamen des Crewmitglieds
 - b. Gruppiert nach der Nationalität des Crewmitglieds

/F50/ Für eine bessere Wartungsfreundlichkeit des Programms soll der Quelltext in Englisch verfasst werden.

5. Produktleistungen und Wünsche

/L10/ Das System soll für ca. 40 Benutzer ausgelegt sein.

/L20/ Das System soll unterschiedliche Benutzerberechtigungen unterstützen.

/L30/ Die Stammdaten sollen in einem Datenbanksystem gespeichert werden, damit auch andere Anwendungen wie Logblog 4 darauf zugreifen können.

/L40/ Die Anwendung soll so aufgebaut sein, dass das Datenbanksystem möglichst ohne große Konsequenzen ausgetauscht werden kann.

/W10/ Da das Bordpersonal der Schiffe aus verschiedenen Ländern kommt, soll das System zudem auch in Englisch verfügbar sein.

/W20/ Das System sollte möglichst benutzerfreundlich sein.

/W30/ Die Benutzeroberfläche des Systems soll so gestaltet werden, dass es später auch auf Tablets genutzt werden kann.

/W40/ Das System darf nicht die Kabinennummer 13 vergeben können!

Projektplanung

Die Planung eines Softwareprojektes spielt eine wichtige Rolle, denn mit ihr können sowohl die Zeit als auch die benötigten Ressourcen genau abgestimmt werden. Gleichzeitig gibt sie dem Projektleiter und dem Team einen Überblick über die Etappen, welche nach und nach abgearbeitet werden sollen.

Entwicklungsumgebung, Plugins, Zusatzprogramme

Für die Entwicklung dieser Anwendung wird die Entwicklungsumgebung Eclipse mit der Version 2020-06 verwendet. Da die Anwendung jedoch mithilfe von JavaFX geschrieben wird und das Framework nicht mehr im eigentlichen Java Development Kit (kurz: *JDK*) von Oracle enthalten ist, muss es extern eingebunden werden. Weil die Entwickler von JavaFX verstärkt auf Open Source setzen, wird anstelle des offiziellen JDKs von Oracle das Open JDK verwendet.

Damit Eclipse mit JavaFX optimal zusammenarbeiten kann, wird das Plugin *e(fx)clipse* verwendet. Es fügt der Entwicklungsumgebung hilfreiche Menüs hinzu, mit der sich beispielsweiseFXML-Dateien direkt ohne großen Aufwand erstellen lassen. Das Tool Maven dient als Build-Management-Tool, um das Programm standardisiert zu erstellen und zu verwalten.

Für die Gestaltung der Benutzeroberflächen wird der Scene Builder von Gluon verwendet, da der offizielle Scene Builder von Oracle nicht mehr weiterentwickelt wird. Er ist ebenfalls kostenlos erhältlich. Die Speicherung der Daten wird mithilfe von MySQL realisiert, welches zusammen mit XAMPP, einer Sammlung nützlicher Programme für die Softwareentwicklung (Webserver, Datenbankserver, Javawebserver, Mailserver, FTP-Server), ausgeliefert wird.

Zeit- und Ressourcenplanung

Die nachfolgende Tabelle zeigt den für das Projekt vorgegebenen Zeitplan auf und stellt dar, wann welche Phase erreicht wird. Dabei stehen die violetten Felder in der Spalte Datum für die Praxisphase des Praktikums, während die grünen Felder den Beginn der Sommerferien stehen.

Zeitplan für das Projekt			
Entwicklung eines Verwaltungsprogramms für ein großes Kreuzfahrtschiff mithilfe von JavaFX			
Woche	Datum	Phase des Projektes	Was ist zu erledigen?
1	11.05. - 15.05.	Problemaufriss / Projektplanung	Ideenfindung zum Projekt, Anregungen durch Gespräch in der Schule
2	18.05. - 22.05.		Modellierung der fiktiven Reederei & des Schiffes, welches das Programm benötigt; Dokumentation jener Reederei in der Mappe
3	25.05. - 29.05.	Projektdurchführung	Entwicklung und Füllung der MySQL-Datenbank für das Programm
4	01.06. - 05.06.		Entwicklung der GUIs für das Programm mit dem SceneBuilder
5	08.06. - 12.06.		Entwicklung der Controllerklassen für die Programmlogik in JavaFX
6	15.06. - 19.06.		Entwicklung der Controllerklassen für die Programmlogik in JavaFX
7	22.06. - 26.06.		Fertigstellung und Test des Programms
8	29.06. - 03.07.		Einarbeitung in Microstream, um komplett auf die MySQL-Datenbank zu verzichten
9	06.07. - 10.07.		Versuch der Implementierung von Microstream in das Programm
10	13.07. - 17.07.	Projektabschluss	Versuch der Implementierung von Microstream in das Programm
11	20.07. - 24.07.		Abschließende Reflexion des Projektes
12	27.07. - 31.07.		Letzte Korrekturen an der Mappe und ggf. an dem Programm
13	03.08. - 07.08.		Abgabe der Mappe
	14.08.		Späteste Abgabe der Mappe und des Programms!

Meilensteine des Projektes

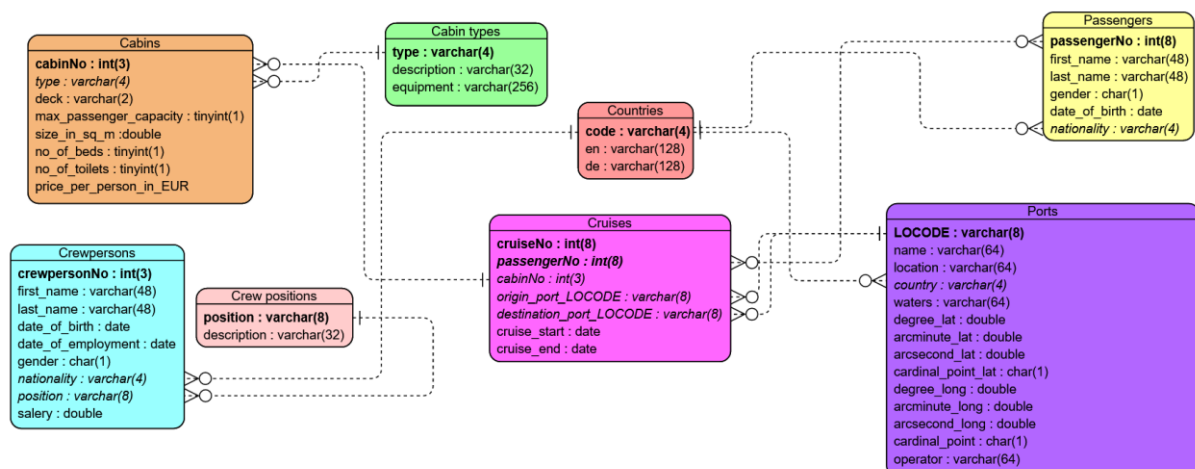
Jedes Softwareprojekt wird in kleineren Etappen zerlegt; eine Kombination bestimmter Etappen wird zu einem Meilenstein, auf dem die nächsten Etappen aufbauen. Für das Schiffsverwaltungsprogramm sind die folgenden Meilensteine vorgesehen:

- Modellierung der Reederei und des Schiffes, auf dem das Programm später laufen soll.
- Entwicklung der MySQL-Datenbank und ihre anschließende Füllung mit Stammdaten.
- Entwicklung der Benutzeroberflächen mit dem Scene Builder von Gluon.
- Entwicklung der Controller-Klassen der Benutzeroberflächen, worin die eigentliche Programmlogik gespeichert wird. Jede Controller-Klasse erbt die wichtigsten Eigenschaften vom Master-Controller.
- Fertigstellung des Programms auf Basis von MySQL.
- Erstellung eines Prototyps, welcher nicht mit MySQL, sondern mit Microstream arbeitet.
- Fertigstellung des Prototyps.

MySQL-Datenbank

Wie im Pflichtenheft erwähnt, sollen die Stammdaten des Schiffes in einer MySQL-Datenbank gespeichert werden. Dazu wird das kostenlose Softwarepaket XAMPP verwendet, worin die erforderliche Software zum Testen einer vollwertigen MySQL-Datenbank enthalten ist.












































































Zunächst wurde die Datenbank im ERM modelliert, um zu sehen, welche Daten überhaupt gespeichert werden sollen und welchen Beziehungstyp die Tabellen untereinander besitzen. Zunächst waren nur drei große Tabellen in Planung, doch mithilfe der Normalisierung ergaben sich schließlich acht Tabellen, wie das vorliegende ERM zeigt.



Anmerkung

Beim Betrachten der Attribute fällt auf, dass sich durchaus noch mehr Daten normalisieren lassen (z.B. der Betreiber des Hafens (Operator) oder die Gewässer (Waters)). Dies würde jedoch das Programm vergrößern und es womöglich sogar aus dem Zeitplan schieben, weswegen aus Vereinfachungsgründen darauf verzichtet wurde.

Eine Besonderheit findet sich in der Tabelle *Cruises*, worin sich der Primärschlüssel aus den Spalten *cruiseNo* und *passengerNo* zusammensetzt. Wäre nur die Kreuzfahrtnummer der Primärschlüssel, so könnten keine weitere Datensätze bezüglich einer gleichen Kreuzfahrt mehr eingefügt werden. Bei dem zusammengesetzten Primärschlüssel wird dieses Problem gelöst, indem nur eine identische Kombination gleicher Datensätze aus den zwei Spalten nicht zulässig ist. Das nachfolgende Bild veranschaulicht dies:

					cruiseNo	passengerNo	cabinNo	origin_port_LOCODE	destination_port_LOCODE	cruise_start	cruise_end		
<input type="checkbox"/>		Bearbeiten		Kopieren		Löschen	1	34	108	DEHAM	USBOS	2013-06-01	2013-06-08
<input type="checkbox"/>		Bearbeiten		Kopieren		Löschen	1	37	29	DEHAM	USBOS	2013-06-01	2013-06-08
<input type="checkbox"/>		Bearbeiten		Kopieren		Löschen	1	50	60	DEHAM	USBOS	2013-06-01	2013-06-08
<input type="checkbox"/>		Bearbeiten		Kopieren		Löschen	1	55	33	DEHAM	USBOS	2013-06-01	2013-06-08
<input type="checkbox"/>		Bearbeiten		Kopieren		Löschen	1	57	44	DEHAM	USBOS	2013-06-01	2013-06-08
<input type="checkbox"/>		Bearbeiten		Kopieren		Löschen	1	64	84	DEHAM	USBOS	2013-06-01	2013-06-08
<input type="checkbox"/>		Bearbeiten		Kopieren		Löschen	1	72	49	DEHAM	USBOS	2013-06-01	2013-06-08
<input type="checkbox"/>		Bearbeiten		Kopieren		Löschen	2	2	86	DEHAM	MAAGA	2013-06-19	2013-06-26
<input type="checkbox"/>		Bearbeiten		Kopieren		Löschen	2	2	89	DEHAM	MAAGA	2013-06-19	2013-06-26
<input type="checkbox"/>		Bearbeiten		Kopieren		Löschen	2	10	90	DEHAM	MAAGA	2013-06-19	2013-06-26
<input type="checkbox"/>		Bearbeiten		Kopieren		Löschen	2	23	69	DEHAM	MAAGA	2013-06-19	2013-06-26
<input type="checkbox"/>		Bearbeiten		Kopieren		Löschen	2	30	48	DEHAM	MAAGA	2013-06-19	2013-06-26
<input type="checkbox"/>		Bearbeiten		Kopieren		Löschen	2	44	72	DEHAM	MAAGA	2013-06-19	2013-06-26
<input type="checkbox"/>		Bearbeiten		Kopieren		Löschen	2	52	98	DEHAM	MAAGA	2013-06-19	2013-06-26
<input type="checkbox"/>		Bearbeiten		Kopieren		Löschen	2	52	99	DEHAM	MAAGA	2013-06-19	2013-06-26
<input type="checkbox"/>		Bearbeiten		Kopieren		Löschen	2	56	48	DEHAM	MAAGA	2013-06-19	2013-06-26
<input type="checkbox"/>		Bearbeiten		Kopieren		Löschen	2	65	50	DEHAM	MAAGA	2013-06-19	2013-06-26
<input type="checkbox"/>		Bearbeiten		Kopieren		Löschen	2	76	93	DEHAM	MAAGA	2013-06-19	2013-06-26
<input type="checkbox"/>		Bearbeiten		Kopieren		Löschen	3	4	25	DEHAM	GBLON	2013-07-07	2013-07-10
<input type="checkbox"/>		Bearbeiten		Kopieren		Löschen	3	11	20	DEHAM	GBLON	2013-07-07	2013-07-10
<input type="checkbox"/>		Bearbeiten		Kopieren		Löschen	3	11	38	DEHAM	GBLON	2013-07-07	2013-07-10
<input type="checkbox"/>		Bearbeiten		Kopieren		Löschen	3	19	55	DEHAM	GBLON	2013-07-07	2013-07-10
<input type="checkbox"/>		Bearbeiten		Kopieren		Löschen	3	19	84	DEHAM	GBLON	2013-07-07	2013-07-10
<input type="checkbox"/>		Bearbeiten		Kopieren		Löschen	3	21	44	DEHAM	GBLON	2013-07-07	2013-07-10
<input type="checkbox"/>		Bearbeiten		Kopieren		Löschen	3	23	26	DEHAM	GBLON	2013-07-07	2013-07-10

Hier wurde der Primärschlüssel irrtümlich noch auf die dritte Spalte erweitert, was jedoch nicht sinnvoll ist. Die Datensätze drücken aus, dass der zweite Passagier im System auf der zweiten Kreuzfahrt sowohl in Kabine 86 als auch in 89 schläft, was jedoch nicht möglich sein kann.

Die Tabellen wurden in Microsoft Excel erstellt und mit den entsprechenden Stammdaten gefüllt. Da der MySQL-Server jedoch keine Excel-Arbeitsblätter importieren kann, müssen sie zunächst in das CSV-Format übertragen werden, um sie anschließend in der entsprechenden Tabelle zu importieren. Dabei ist es wichtig, die Spaltenüberschriften aus der Excel-Tabelle zu entfernen, weil sonst Konflikte mit den entsprechenden Datentypen auftreten können (z.B. ein Text in einer Spalte vom Datentyp int).

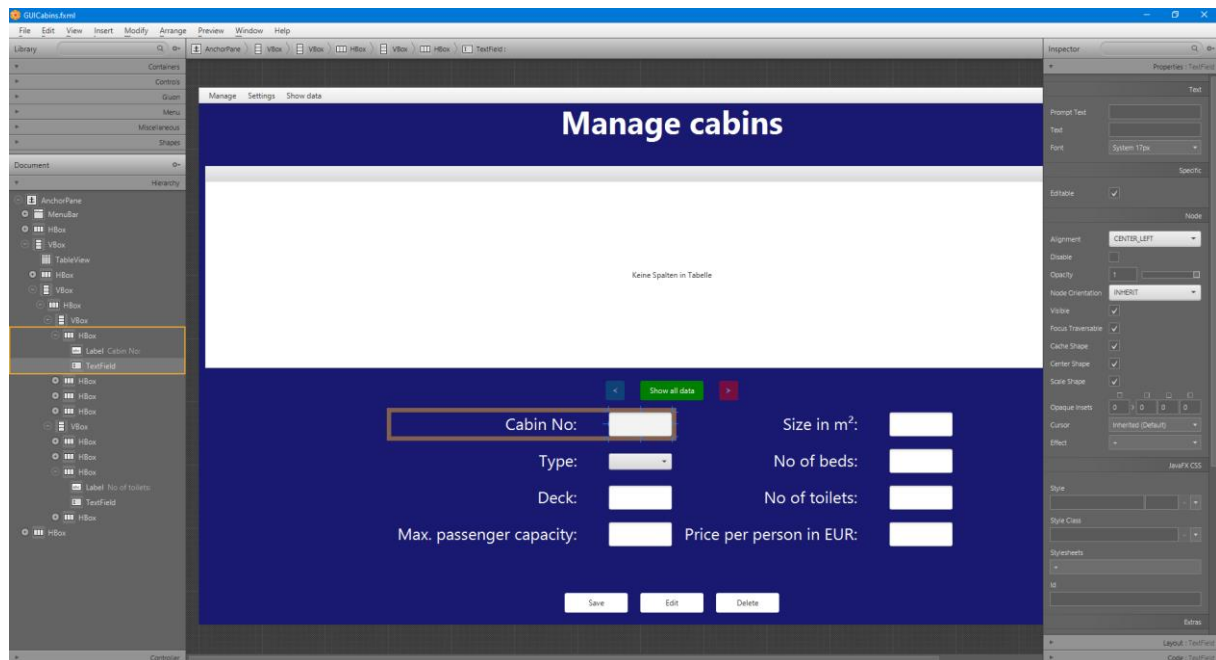
Probleme mit der MySQL-Datenbank und deren Behebung

Während die Modellierung der Datenbank nahezu ohne nennenswerte Zwischenfälle durchgeführt werden konnte, erwies sich die Implementierung der Stammdaten als gar nicht so einfach wie bisher angenommen. Dies lag daran, dass es bei großen Datensätzen häufig zu Dopplungen kommen kann, die dem Entwickler gar nicht bewusst sind, denn es ist unmöglich, jeden einzelnen Wert aus jeder Zelle im Kopf zu behalten. Bemerkbar machte sich das Datenbanksystem mit einer Fehlermeldung, wenn es einen doppelten Wert beim Primärschlüssel gab. Aber auch die Beziehungen zwischen den Tabellen lösten Fehler aus, wenn in der Kind-Tabelle Werte gespeichert wurden, die in der Eltern-Tabelle gar nicht existieren. Gerade an dieser Stelle zeigt sich, dass die referentielle Integrität eine wichtige Rolle spielt, denn sie verhindert mit der Überprüfung der Kind- und Eltern-Tabellen inkonsistente Datensätze.

Glücklicherweise verrät das System, bei welchem Datensatz die Überprüfung auf die referentielle Integrität fehlschlug. Hier muss der Entwickler entscheiden, ob der entsprechende Datensatz geändert, ergänzt oder gelöscht werden soll. Wenn in der Eltern-Tabelle der Datensatz nicht existierte, musste er lediglich dort ergänzt werden. Gab es eine Dopplung in einer Primärschlüsselspalte, so musste diese gelöscht werden.

Projektdurchführung

Die Entwicklung der grafischen Benutzeroberflächen für das Programm

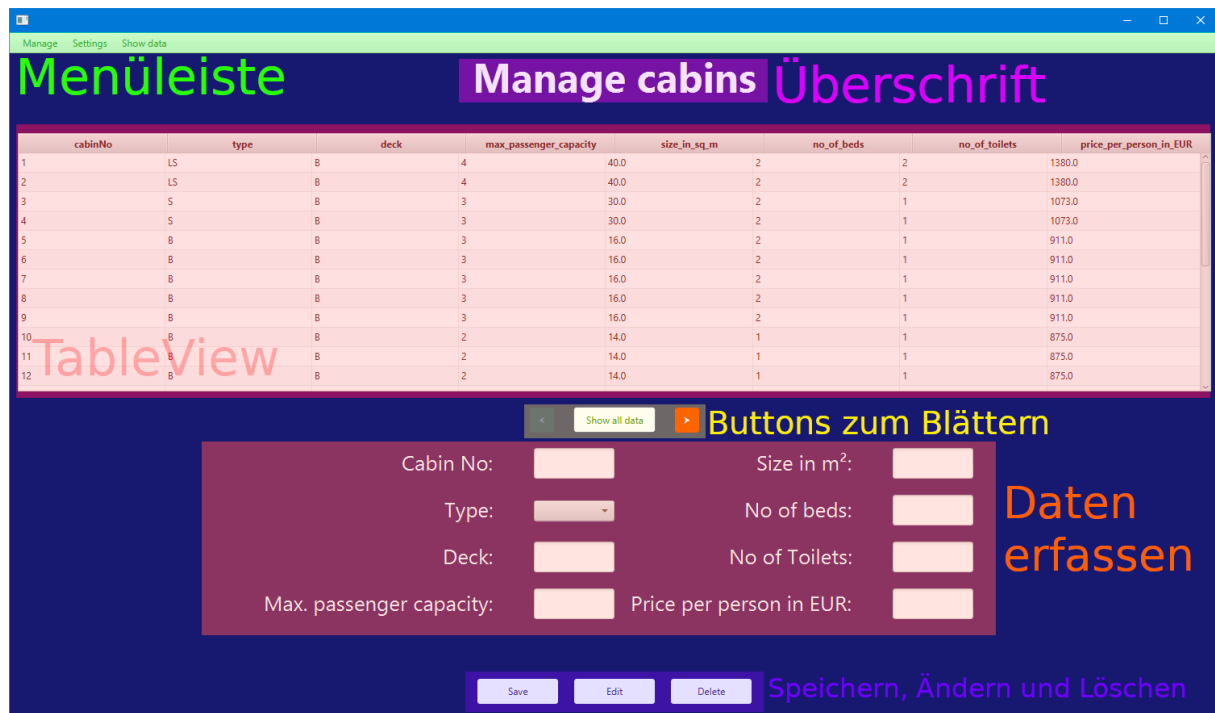


Eine grafische Benutzeroberfläche, kurz GUI genannt, ist das wichtigste Element eines Programms, denn hiermit interagieren die Benutzer und greifen auf die restlichen Programmbestandteile, wie etwa der Datenbank, zu. Aus diesem Grund ist ein benutzerfreundliches GUI für den ordnungsgemäßen Betrieb eines Programmes unerlässlich, zumal ein übersichtliches GUI die Produktivität steigert als ein unübersichtliches GUI mit zu vielen Buttons und Menüs. Bei den alten Versionen von SAP ERP beklagten sich einige Benutzer aufgrund der zu komplexen Menüstruktur der Anwendung. Die Führungskräfte bemängelten die lange Einarbeitungszeit in dem Programm.

Die GUIs in dem Schiffsverwaltungsprogramm sind für eine benutzerfreundliche Bedienung ausgelegt und besitzen dieselbe Struktur. Dadurch muss sich der Benutzer nicht laufend in neue Benutzeroberflächen einarbeiten, wie es bei anderen Programmen der Fall ist. Gerade ERP-Programme sind dafür bekannt, oftmals zu komplex und anspruchsvoll zu sein, obwohl die anfallenden Aufgaben durchaus einfacher und komfortabler gelöst werden können. Das liegt oftmals daran, dass diese Anwendungen in älteren Frameworks bzw. Programmiersprachen geschrieben wurden und eine Migration in ein neues Framework erheblich viel Zeit und Ressourcen benötigt. Wenn das Programm jedoch weiter gepflegt werden soll, ist ein Update oftmals unerlässlich, wie es beispielsweise bei Betriebssystemen der Fall ist.

Der Aufbau einer grafischen Benutzeroberfläche

Jedes GUI in der Anwendung gliedert sich nach demselben vorliegenden Schema:



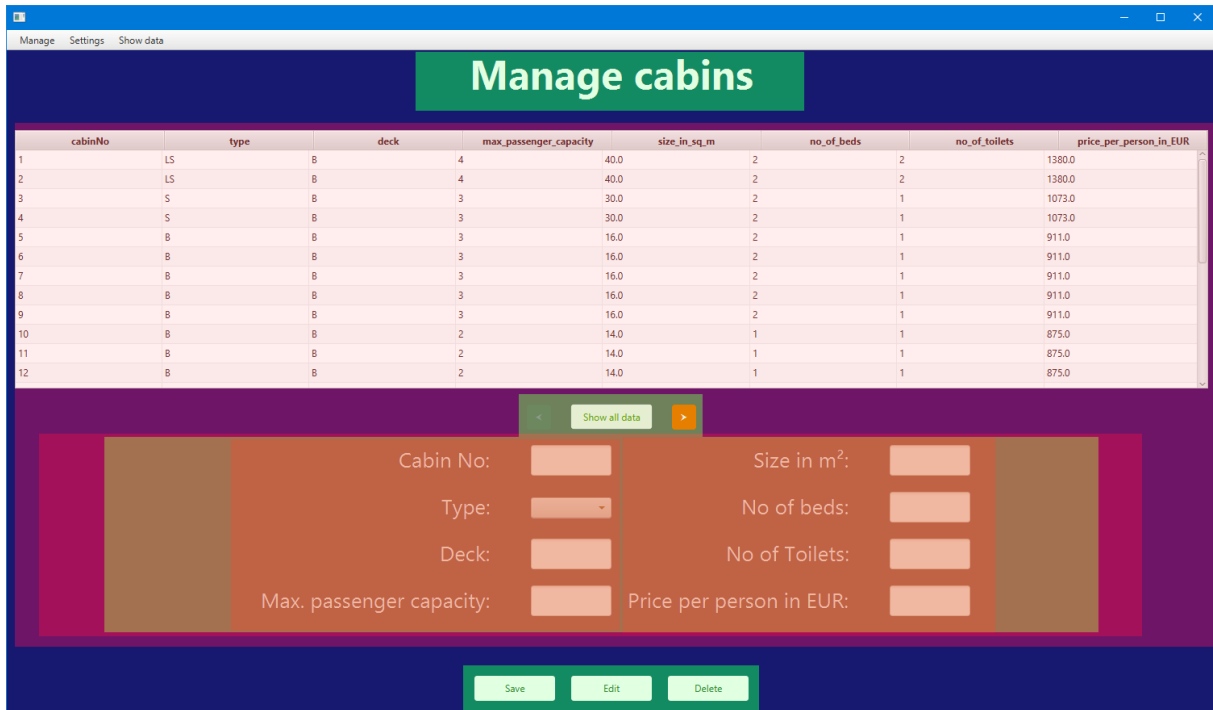
Dabei liegen die Elemente des GUIs wie die Tabelle, die Label mit den entsprechenden Textfeldern oder die Navigationsbutton nicht einfach lose im GUI herum, sondern sind in so genannten H-Boxen und V-Boxen verankert. In den H-Boxen gehören die Elemente im GUI, welche horizontal nebeneinander liegen. Ein gutes Beispiel dafür sind die Labels mit den darauf folgenden Textfeldern bzw. Comboboxen. In den V-Boxen sind wiederum die Elemente gespeichert, welche vertikal wie in einer Liste angeordnet sind.

Der Vorteil an diesen Boxen liegt darin, dass sich die Elemente viel besser im GUI positionieren und formatieren lassen. So lässt sich beispielsweise der Abstand zwischen den Textfeldern und den Labels gleichmäßig einstellen, genauso wie die Anordnung der jeweiligen Elemente.

Erwähnenswert ist außerdem die Formatierung der Elemente mittels CSS. Jedem Element kann dabei entweder eine ID oder eine Klasse zugewiesen werden, wie es bei CSS der Fall ist. Anhand der Formatierungseinstellungen in der CSS-Datei kann festgelegt werden, wie das Element aussehen und welche Effekte es beinhalten soll. Dabei stehen dem Entwickler dieselben Befehle wie beim normalen CSS zur Verfügung; der einzige Unterschied liegt darin, dass jeder Befehl den Präfix `-fx-` trägt.

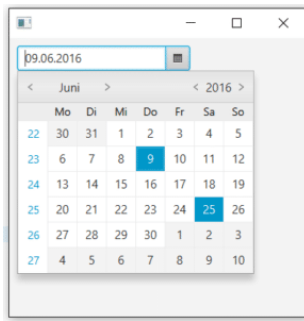
Nachfolgend ist noch ein Bild mit dem Aufbau der H-Boxen und V-Boxen. Die grün hinterlegten Elemente befinden sich in einer H-Box, während sich die rot hinterlegten Elemente in einer V-Box befinden. Die Darstellung der Boxen ist schwierig, da sie häufig verschachtelt und kombiniert werden, um die besten Ergebnisse zu erzielen. Dennoch soll die nachfolgende Abbildung

eine kleine Orientierung geben, wie das so genannte Boxing funktioniert. Es ist vergleichbar mit der Webentwicklung, worin die Elemente einer Website nach ihrer Funktion in Div-Container geordnet werden.



Ein ebenso wichtiger Punkt in der Softwareentwicklung sind Namenskonventionen. Diese stellen eine einheitliche Benennung von Variablen und Elementen im Quellcode bereit und sind daher unverzichtbar. Die Namen der so genannten FXIDs der Elemente, mit dessen Hilfe sie im Quellcode angesprochen werden können, werden mithilfe des Kürzels und ihrem Zweck gebildet. Neue Wörter werden dabei wieder mit einem Großbuchstaben begonnen. Die unten abgebildete Tabelle zeigt das Prinzip der Benennung auf:

Element im GUI	Kürzel	Beispiel
MenuBar	mb	mbManage
Menu	m	mSettings
MenuItem	mi	miCrewpersons
CheckMenuItem	cmi	cmiEnableEditMode
Label	lbl	lblFirstName
TextField	txt	txtLastName
ComboBox	cb	cbGender
DatePicker	dp	dpDateOfBirth
TableView	tbl	tblPorts
TableColumn	tc	tcCabinNo
Button	btn	btnSave



Eine wesentliche Neuerung in JavaFX ist der DatePicker, welcher ab JavaFX 8 unterstützt wird. Mit ihm ist es möglich, ein Datum aus einem ausklappenden Kalender auszuwählen und direkt in eine entsprechende Variable zu speichern. Die Darstellungsform des Datums wird aus dem lokalen System ermittelt. Ein auf Englisch eingestelltes Betriebssystem besitzt die amerikanische Darstellungsform, während ein auf Deutsch genutztes Betriebssystem die klassische deutsche Darstellungsform verwendet. Dabei kann das Datum auch beliebig formatiert werden.

Probleme mit der Entwicklung der grafischen Benutzeroberflächen und deren Behebung

Bei der Entwicklung der GUIs gab es keine großen Probleme, welche die Entwicklung massiv störten. Allerdings mussten einige Fehler in der Benennung in jedem GUI umbenannt werden, was sich etwas negativ auf die Zeit auswirkte. In einem anderen Fall glich sich die Formatierung eines Elementes nicht an, wodurch sich das ganze Aussehen der GUI verschob. Dieser Fehler konnte jedoch durch ein einfaches Drag & Drop in die richtige HBOX behoben werden.

Zugriff auf die MySQL-Datenbank

Es soll mit dem Programm möglich sein, auf die in der MySQL-Datenbank gespeicherten Stammdaten zuzugreifen und sie zu verändern. Dazu gehören die Erfassung von neuen Daten sowie die Bearbeitung und Löschung bisherigen Daten aus der Datenbank. Die Implementierung in dem Programm erfolgt mithilfe des so genannten MySQL-Connectors, einem kleinen Plugin, der den Zugriff auf die Datenbank erlaubt. Ist das Plugin nicht installiert, kommt es zu einem Fehler.

Installation des MySQL-Connectors

Die Installation erfolgt wie bei Maven üblich in der so genannten pom.xml, worin die wichtigsten Einstellungen des Projektes gespeichert werden. Die Installation des MySQL-Connectors kann dabei entweder über die grafische Benutzeroberfläche des XML-Dokumentes erfolgen oder in der Datei. In der Datei müssen dazu lediglich die folgenden Zeilen in dem Dependencies-Baum ergänzt werden:

pom.xml

```
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>8.0.20</version>
</dependency>
```

Dadurch ist der MySQL-Connector installiert und das Programm kann eine Verbindung zu einer MySQL-Datenbank aufbauen.

Verbindung mit der Datenbank herstellen

Um eine Verbindung mit der Datenbank herzustellen, werden spezielle Variablen und Methoden benötigt, welche aus der Bibliothek des MySQL-Connectors stammen. Sie werden in einer separaten Fachklasse angelegt, um den Überblick zu behalten und um das Projekt zu strukturieren.

MasterAccessor.java

```
package org.green_pioneer.cruiseship_management.dbaccess;
//Pakete für den DBZugriff importieren
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

//Fachklasse für das Erzeugen von Fehlermeldungen importieren
import org.green_pioneer.cruiseship_management.errorhelper.ErrorHelper;

public abstract class DBMasterAccessor {

    protected PreparedStatement stmt;
    protected Connection con;
    protected ResultSet rs = null;
    //Variablen für den DBZugriff deklarieren

    public void openDB() {

        try {
            String url = "jdbc:mysql://localhost/greenpioneer";
            con = DriverManager.getConnection(url, "root", "");
        } catch (SQLException sqllex) {
            ErrorHelper.initErrorDialogWithoutHeader("Unable to connect
                to the database!");
        }

    }

    public void closeDB() {
        try {
            this.stmt.close();
            this.con.close();
            rs = null;
        } catch (SQLException sqllex) {
            ErrorHelper.initErrorDialogWithoutHeader("Unable to close the
                connection to the database!");
        }

    }

}
```

Zunächst werden die drei Variablen `stmt` vom Typ `PreparedStatement`, `con` vom Typ `Connection` und `rs` vom Typ `ResultSet` zu Beginn der Fachklasse deklariert. Da sich die anderen Fachklassen für den Datenbankzugriff von dieser Fachklasse ableiten, werden sie mit dem Zugriffsmodifizierer `protected` versehen, während die Fachklasse selber als `abstract` deklariert wird. Dadurch kann sie keine Objekte erzeugen.

Die beiden Methoden `openDB()` und `closeDB()` sind für den Zugriff erforderlich. In `openDB()` wird zunächst die URL der Datenbank in einer String-Variable gespeichert. Anschließend wird mithilfe der Variable `con` vom Typ `Connection` versucht, eine Verbindung zur Datenbank herzustellen. Die Methode `getConnection()` aus der Bibliothek `DriverManager` verlangt drei String-Parameter: Die URL zur Datenbank, den Benutzernamen und das Passwort. Da jedoch zunächst nur XAMPP verwendet wird und diese darin enthaltene MySQL-Datenbank zu Testzwecken dient, wird der Benutzer `root` ohne Passwort verwendet. Bei großen Anwendungen können sich Benutzer mithilfe dieser Methode in die Datenbank einloggen. Sollte es zu einem Fehler kommen, wird der Benutzer durch ein Fehlerfenster darauf hingewiesen.

Exkurs: Ausnahmebehandlung mit try-catch

Die Behandlung von Ausnahmen (engl. Exceptions) wird mithilfe von try-catch ausgeführt. In dem try-Block kommen die Befehle, in denen es sicherlich zu Fehlern kommen kann (z.B. durch eine falsche Benutzereingabe). Java versucht dann, diesen Befehlsblock auszuführen. Sollte es zu einem Fehler kommen, wird dieser abgefangen und im catch-Block behandelt. In dem Kopf steht die jeweilige Ausnahme, die behandelt werden soll.
--

Werden Ausnahmen nicht behandelt, so besteht die Gefahr, dass es zu Schäden innerhalb des Programmablaufs kommen kann und das Programm abstürzt. Aus diesem Grund müssen Ausnahmen immer dort behandelt werden, wo sie auftauchen können. Dies ist häufig bei Verbindungsversuchen zu anderen Schnittstellen wie etwa MySQL oder Benutzereingaben der Fall.

Die Methode `closeDB()` schließt die Verbindung zur Datenbank wieder. Dies ist notwendig, da es sonst zu unerwarteten Fehlern kommen kann, wenn die Verbindung nicht geschlossen wird. Was es genau mit den Variablen `stmt` und `rs` vom Typ `PreparedStatement` und `ResultSet` auf sich hat, erkläre ich im späteren Verlauf. Ebenso gehe ich später auf den so genannten *ErrorHelper* zur Fehlerbehandlung ein.

Daten aus der Datenbank lesen

Um Daten aus der Datenbank zu lesen, wird eine Fachklasse benötigt, welche die Methoden zum Lesen einer Datenbank enthält. Um eine Verbindung zur Datenbank herzustellen, muss die Fachklasse sich von dem `MasterAccessor` ableiten, damit sie eine Verbindung zur Datenbank aufbauen, aber auch wieder abbauen kann. Aus diesem Grund befindet sich nach dem

Klassenkopf das Schlüsselwort `extends DBMasterAccessor`. Ferner werden in dieser Fachklasse auch weitere Methoden für Operationen an der Datenbank gespeichert. Nachfolgend findet sich der Quellcode, um Daten aus der Datenbank in eine `ObservableList` zu speichern und zurückzugeben.

Exkurs: Die `ObservableList` in JavaFX

Erzeugte Objekte oder primitive Datentypen lassen sich in Java mit vielen Wegen speichern: Entweder als Variable, in einem Array oder in einer `ArrayList`. JavaFX stellt hierfür eine eigene Listenstruktur zur Verfügung, die so genannte `ObservableList`. Mit ihr können die Werte von GUI-Elementen wie Tabellen oder Comboboxen gesetzt werden, in dem der Entwickler einfach schreibt:
`einElement.setItems(eineObservableList)`.

DBAccessCabinTypes.java

```
public class DBAccessCabinTypes extends DBMasterAccessor {
    //Methode vom Typ ObservableList<CabinType> deklarieren
    public ObservableList<CabinType> getCabinTypesInObservableList(int
        intFirstDataSet, final int INT_DATA_LIMIT) {
        //ObservableList vom Typ CabinType deklarieren
        ObservableList<CabinType> oblCabinTypes = FXCollections.observableArrayList();
        oblCabinTypes.clear(); //Liste leeren

        //Variablen anlegen, mit denen das Objekt vom Typ CabinType erzeugt werden soll
        String strType;
        String strDescription;
        String strEquipment;

        try {
            openDB(); //Datenbankverbindung öffnen
            //SQL-Statement speichern
            String strSQLStatement = "SELECT * FROM tblcabinTypes ORDER BY tblcabinTypes.type ASC LIMIT ?,? ;";
            //Statement mithilfe eines PreparedStatements vorbereiten.
            stmt = con.prepareStatement(strSQLStatement);
            //Die mit einem ? unbekannten Paramter mit den Werten der übergebenen Variablen verbinden.
            stmt.setInt(1, intFirstDataSet);
            stmt.setInt(2, INT_DATA_LIMIT);
            //Statement ausführen und das Ergebnis in der Variable rs speichern.
            rs = stmt.executeQuery();
            /*Solange es noch eine nachfolgende Zeile aus dem Ergebnis der Abfrage gibt, erzeuge aus der jeweiligen Elementen der */Zeile ein Objekt und speichere es in oblCabinTypes.
            while(rs.next()) {
                strType = rs.getString(1);
                strDescription = rs.getString(2);
                strEquipment = rs.getString(3);
                oblCabinTypes.add(new CabinType(strType, strDescription, strEquipment));
            }
        }
```

```

        //Datenbankverbindung schließen.
        closeDB();
    } catch(SQLException sqlex) {
        //Infofenster erzeugen, falls es zu einem Fehler kommen sollte.
        ErrorHandler.initErrorDialogWithoutHeader("Unable to get the cabin
        types from the database!");
    }
    //Die zuvor gefüllte Liste zurückgeben.
    return oblCabinTypes;
}
...
}

```

Beim Betrachten des SQL-Statements fällt auf, dass Platzhalter in dem LIMIT-Befehl verwendet wurden. In diesen Platzhaltern werden im Laufe der Methode die übergebenen Parameter festgelegt und gespeichert¹, bis das SQL-Statement schließlich ausgeführt wird. Diese so genannten PreparedStatements schützen das System vor Angriffen, wie etwa SQL-Injections. Der ResultSet dient lediglich dazu, das Ergebnis der Abfrage zu speichern.

Diese Methode wird immer dann von dem entsprechenden GUI aufgerufen, wenn eine Liste mit allen Kabinentypen benötigt wird. Darüber hinaus ist es mit ihr sogar möglich, nur eine bestimmte Anzahl an Kabinentypen aus der Datenbank zu erhalten. Dies geschieht mithilfe der Parameter im Methodenkopf. Der erste Parameter gibt dabei den ersten Datensatz an, während der andere Parameter angibt, wie viele Datensätze ausgehend vom ersten Datensatz angezeigt werden sollen.

Beispiele

Es sollen nur die ersten beiden Kabinentypen angezeigt werden. Der Methode werden dabei die Werte 0 und 2 übergeben. Das SQL-Statement sieht nach dem Preparing und Binding so aus:

```
SELECT * FROM tblcabinotypes LIMIT 0,2;
```

Es sollen nur die letzten beiden Kabinentypen angezeigt werden. Durch die Funktion COUNT ist bekannt, dass es insgesamt nur fünf Kabinentypen gibt. Der gesamten Anzahl an Kabinentypen wird das Limit (hier: zwei) abgezogen, sodass der Methode die Werte 3 und 2 übergeben werden. Das SQL-Statement sieht nach dem Preparing und Binding so aus:

```
SELECT * FROM tblcabinotypes LIMIT 3,2;
```

Um die Daten schließlich in der Tabelle anzuzeigen, wird die Methode in der Fachklasse GUI-CabinTypesController.java aufgerufen, wenn der Benutzer auf dem Button *showAllData* klickt. Das Datenlimit ist hierbei auf 25 Datensätze eingestellt, um ein bequemes Durchblättern zu ermöglichen. Da der Benutzer sich beim Blättern am Anfang der Tabelle befindet, ist der erste Datensatz 0. Der Methode werden somit die Werte 0 und 25 beim Aufruf übergeben. Nachfolgend befindet sich der Code aus der entsprechenden Fachklasse:

¹ Auch als Binding bezeichnet.

GUICabinTypesController.java

```
@FXML //Notation, dass sich der Button in der FXML-Datei (GUI) befindet.
private void showAllData() {
    /*Button zum Weiterblättern aktivieren, falls er zuvor deaktiviert
    */wurde
    btnNextData.setDisable(false);
    /*Prüfen, ob es noch möglich ist, durch die Daten vor oder zurück zu
    */blättern.
    if(!(intFirstDataSet == 0)) {
        btnPreviousData.setDisable(true);
    } else if(INT_NUMBER_OF_CABIN_TYPES < INT_DATA_LIMIT) {
        btnPreviousData.setDisable(true);
        btnNextData.setDisable(true);
    }
    intFirstDataSet = 0;
    oblCabinTypes.clear();
    /*Kabinentypen aus der Datenbank lesen und in die ObservableList
    */oblCabinTypes speichern.
    oblCabinTypes = aDBCabinTypesAccessor.getCabinTypesInObservable-
        List(intFirstDataSet, INT_DATA_LIMIT);
    //Tabelle mit den Objekten füllen
    tblCabinTypes.setItems(oblCabinTypes);
}
```

Die Variable `INT_NUMBER_OF_CABIN_TYPES` wird beim Start des Programmes initialisiert und speichert die Anzahl der Kabinen, welche mithilfe der Funktion `COUNT` in einer Datenbankoperation ermittelt und zurückgegeben wurde. Ihre Definition sieht aus wie folgt:

```
private final int INT_NUMBER_OF_CABIN_TYPES = aDBCabinTypesAcces-
sor.getNumberOfCabinTypes();
```

Da sich die Anzahl der Kabinen im Laufe der Anwendung nur selten ändert, wurde diese Variable mit dem Schlüsselwort `final` versehen. Dadurch kann sie in der gesamten Anwendung nicht mehr verändert werden. Die Namenskonvention für konstante Variablen besagt, dass diese in Großbuchstaben benannt werden sollten.

Bei `aDBCabinTypesAccessor` handelt es sich um ein vererbtes Zugriffsobjekt aus der Fachklasse `MasterController.java`. In ihr sind alle Zugriffsobjekte deklariert.

Exkurs: Atomare Werte aus einer Zeile zurückgeben

Gibt ein SQL-Statement genau eine Zeile mit n-Spalten zurück, können die Werte der Spalten dennoch ausgelesen und in eine Variable gespeichert werden. In dem Rumpf der While-Schleife wird der Wert aus Spalte n in der Variable gespeichert. Der nachfolgende Code veranschaulicht dies:

DBAccessCabinTypes.java

```
public int getNumberOfCabinTypes() {
    int intNumberOfCabinTypes = 0;

    try {
        openDB();
        String strSQLStatement = "SELECT COUNT(*) FROM tblcabinTypes;";
        stmt = con.prepareStatement(strSQLStatement);
        rs = stmt.executeQuery();
    }
```

```

        while(rs.next()) {
            /* Wert aus der Spalte mithilfe des ColumnIndex in die
            */ Variable speichern.
            intNumberOfCabinTypes = rs.getInt(1);
        }

        closeDB();
    } catch(SQLException sqlex) {
        ErrorHandler.initErrorDialogWithoutHeader("Unable to count the
        number of cabin types from the database!");
    }
    //Anzahl der Kabinentypen zurückgeben
    return intNumberOfCabinTypes;
}

```

Daten in der Datenbank bearbeiten

Um bereits bestehende Daten in der Datenbank zu bearbeiten, ist die Implementierung einer weiteren Methode notwendig. Als Parameter enthält sie alle Attribute des Objektes, die sich auch als Spalten in der Datenbank wiederfinden. Für die Kabinentypen wären dies der Typ, die Beschreibung und die Ausstattung. Da es jedoch auch möglich sein soll, den Typen zu ändern, muss der neue Typ ebenfalls übergeben werden, damit der alte Typ, der geändert werden soll, im SQL-Statement mit WHERE gefunden werden kann.

DBAccessCabinTypes.java

```

public void updateCabinType(String strType, String strNewType, String
    strDescription, String strEquipment) {
    try {
        openDB();
        String strSQLStatement = "UPDATE tblcabin types SET tblcabin
            types.type = ?, tblcabin types.description = ?, tblcabin-
            types.equipment = ? WHERE tblcabin types.type = ?";
        stmt = con.prepareStatement(strSQLStatement);
        stmt.setString(1, strNewType);
        stmt.setString(2, strDescription);
        stmt.setString(3, strEquipment);
        stmt.setString(4, strType);
        stmt.execute();
        closeDB();
        ErrorHandler.initInformationDialogWithoutHeader("The cabin type " +
            strType + " was updated successfully!");
    } catch(SQLException sqlex) {
        ErrorHandler.initErrorDialogWithoutHeader("Unable to update the
            cabin type from the database!");
    }
}

```

Die Vorgehensweise ähnelt hierbei sehr der Methode zum Lesen der Daten aus der Datenbank. Die größten Unterschiede liegen in der Abwandlung des SQL-Statements und die Tatsache, dass die Methode nichts zurückgibt.

Die Methode zum Ändern des Kabinentypen wird dann aufgerufen, wenn der Benutzer auf den Button *Edit* klickt. Dieser muss jedoch erst mithilfe des Bearbeitungsmodus aktiviert

werden. Um in den Bearbeitungsmodus zu gelangen, muss der Menüpunkt *Settings* → *Enable edit mode* aufgerufen werden. Anschließend erscheint neben dem Kabinentypen ein gelbes Textfeld, worin der neue Kabinentyp eingegeben werden kann.

Zuvor wird jedoch geprüft, ob der zu ändernde Kabinentyp im System existiert. Dazu wird die boolesche Methode `isCabinTypeExisting` aufgerufen, welche die Kabinentypen erneut in einer Liste speichert und sie anschließend mit einer erweiterten For-Schleife durchläuft. Hier befindet sich eine If-Anweisung, welche genau dann `true` zurückgibt, wenn der zu bearbeitende Kabinentyp in der Liste existiert. Der Code der Methode lautet wie folgt:

GUICabinTypesController.java

```
private boolean isCabinTypeExisting(String strCabinType) {
    ObservableList<CabinType> oblExistingCabinTypes = FXCollections.observableArrayList();
    oblExistingCabinTypes = aDBCabinTypesAccessor.getCabinTypesInObservableList(0, INT_NUMBER_OF_CABIN_TYPES);

    for(CabinType aCabinType : oblExistingCabinTypes) {
        if(strCabinType.equals(aCabinType.getStrType())) {
            return true;
        }
    }
    return false;
}
```

Die Methode zum Ändern eines Kabinentypen wird in der Fachklasse `GUICabinTypesController.java` wie folgt implementiert:

GUICabinTypesController.java

```
@FXML
private void editCabinType() {
    //Benutzereingaben aus den Textfeldern in Variablen speichern
    String strCabinType = txtCabinType.getText();
    String strNewCabinType = txtNewCabinType.getText();
    String strDescription = txtDescription.getText();
    String strEquipment = txtEquipment.getText();
    //Prüfen, ob eines der Textfelder leer ist
    if(strCabinType.equals("") || strNewCabinType.equals("") || strDescription.equals("") || strEquipment.equals("")) {
        ErrorHandler.initErrorDialog("Unable to save a new cabin type!", "One or more textfields were empty!");
    } else {
        //Prüfen, ob die zu bearbeitende Kabine existiert
        if(isCabinTypeExisting(strCabinType)) {
            /*Infofenster erzeugen, ob der Benutzer sicher ist, den Kabinentypen zu ändern
            Optional<ButtonType> confMessage = ErrorHandler.initConfirmationDialog("Are you sure that you want to update the cabin type?", "");
            /*Wenn OK gedrückt, ändere den Kabinentypen mit den Paramtern, die zuvor in den Variablen gespeichert wurden und zeige die Änderung sofort an
            if(confMessage.get() == ButtonType.OK) {
                aDBCabinTypesAccessor.updateCabinType(strCabinType, strNew-
```



```

        CabinType, strDescription, strEquipment);
        oblCabinTypes = aDBCabinTypesAccessor.getCabinTypesInObservableList(intFirstDataSet, INT_DATA_LIMIT);
        tblCabinTypes.setItems(oblCabinTypes);
    } else {
        return;
    }
} else {
    //Benutzer informieren, falls die Kabine nicht gefunden wurde
    ErrorHandler.initErrorDialog("Unable to edit cabin type!", "The cabin type was not found!");
}
}
}

```

Daten in der Datenbank einfügen

Wie beim Lesen und Ändern auch, muss auch eine Methode für das Einfügen von Daten in die Datenbank implementiert werden. Dabei unterscheiden sich die Methoden nur kaum voneinander. Es muss lediglich kein neuer Kabinentyp mehr übergeben werden und das SQL-Statement ändert sich von UPDATE zu INSERT. Aus diesem Grund ändert sich auch der Code zu der Methode kaum ab:

DBAccessCabinTypes.java

```

public void insertNewCabinType(String strType, String strDescription, String strEquipment) {
    try {
        openDB();
        String strSQLStatement = "INSERT INTO tblcabin-types (tblcabin-types.type, tblcabin-types.description, "
            + "tblcabin-types.equipment) VALUES (?, ?, ?);";
        stmt = con.prepareStatement(strSQLStatement);
        stmt.setString(1, strType);
        stmt.setString(2, strDescription);
        stmt.setString(3, strEquipment);
        stmt.execute();
        closeDB();
        ErrorHandler.initInformationDialogWithoutHeader("A new cabin type has been added to the database successfully!");
    } catch (SQLException sqlex) {
        ErrorHandler.initErrorDialogWithoutHeader("Unable to add a new cabin type to the database!");
    }
}
}

```

Der Aufruf dieser Methode erfolgt wieder in der Fachklasse `GUICabinTypesController.java`, wenn der Benutzer auf den Button *Save* klickt. Die Implementierung dieser Methode ist jedoch nicht schwer:

GUICabinTypesController.java

```

@FXML
private void saveCabinType() {
    String strCabinType = txtCabinType.getText();
    String strDescription = txtDescription.getText();
    String strEquipment = txtEquipment.getText();
}

```

```

        if(strCabinType.equals("") || strDescription.equals("") || strEquipment.equals("")) {
            ErrorHandler.initErrorDialog("Unable to save a new cabin type!",
                "One or more textfields were empty!");
        } else {
            aDBCabinTypesAccessor.insertNewCabinType(strCabinType, strDescription, strEquipment);
            oblCabinTypes = aDBCabinTypesAccessor.getCabinTypesInObservableList(intFirstDataSet, INT_DATA_LIMIT);
            tblCabinTypes.setItems(oblCabinTypes);
        }
    }
}

```

Der Vorteil bei dieser Implementierung liegt darin, dass der Benutzer sofort den neuen Kabinentypen in der Tabelle sehen kann.

Daten aus der Datenbank löschen

Um einen Datensatz aus der Datenbank zu entfernen, wurde ebenfalls eine eigene Methode für diesen Zweck implementiert. Dabei wird zur Identifizierung des Datensatzes der Primärschlüssel verwendet, in diesem Beispiel handelt es sich um das Typkürzel der Kabine.

In der Fachklasse DBAccessCabinTypes.java wird die Methode zum Löschen eines Kabinentyps wie folgt implementiert:

DBAccessCabinTypes.java

```

public void deleteCabinType(String strType) {
    try {
        openDB();
        String strSQLStatement = "DELETE FROM tblcabinTypes WHERE tblcabinTypes.type = ?;";
        stmt = con.prepareStatement(strSQLStatement);
        stmt.setString(1, strType);
        stmt.execute();
        closeDB();
        ErrorHandler.initInformationDialogWithoutHeader("The cabin type " + strType + " has been deleted from the database!");
    } catch (SQLException sqlex) {
        ErrorHandler.initErrorDialogWithoutHeader("Unable to delete the cabin type from the database!");
    }
}

```

Da der Primärschlüssel der Tabelle benötigt wird, um den Datensatz eindeutig zu identifizieren, muss dieser der Methode auch übergeben werden, um die Löschung des Datensatzes durchführen zu können. Der Benutzer erhält wie beim Einfügen und Ändern entweder eine Erfolgs- oder Misserfolgsmeldung, ob das Löschen auch funktionierte.

Im Controller erfolgt die Implementierung der Methode genauso einfach wie die Methode zum Ändern neuer Datensätze. Zunächst wird nur das Textfeld mit dem Primärschlüssel in eine Variable gespeichert. Anschließend wird überprüft, ob der Kabinentyp im System überhaupt existiert. Wenn ja, wird ein Infowindow erzeugt, worin der Benutzer die Löschung des

Kabinentyps bestätigen muss. Tut er dies, wird der Kabinentyp aus der Datenbank gelöscht, die Kabinentypen werden neu eingelesen und in der Tabelle angezeigt. Somit sieht der Benutzer das Ergebnis sofort.

Gibt es jedoch in Kindtabellen (hier: `tblcabins`) noch Kabinen mit dem zu löschenden Typen, kann er wegen der referentiellen Integrität nicht gelöscht werden.

GUICabinTypesController.java

```
@FXML
private void deleteCabinType() {
    oblCabinTypes = aDBCabinTypesAccessor.getCabinTypesInObservable-
        List(intFirstDataSet, INT_NUMBER_OF_CABIN_TYPES);
    String strCabinType = txtCabinType.getText();

    if(isCabinTypeExisting(strCabinType)) {
        Optional<ButtonType> confMessage = ErrorHandler.initConfirmationDia-
            log("Are you sure that you want to delete the cabin type?", "This
                cannot be undone!");
        if(confMessage.get() == ButtonType.OK) {
            aDBCabinTypesAccessor.deleteCabinType(strCabinType);
            oblCabinTypes = aDBCabinTypesAccessor.getCabinTypesInObservable-
                List(intFirstDataSet, INT_DATA_LIMIT);
            tblCabinTypes.setItems(oblCabinTypes);
        } else {
            return;
        }
    } else {
        ErrorHandler.initErrorDialog("Unable to delete cabin type!", "The
            cabin type was not found!");
    }
}
```

Zugriff auf die Microstream Datenbank

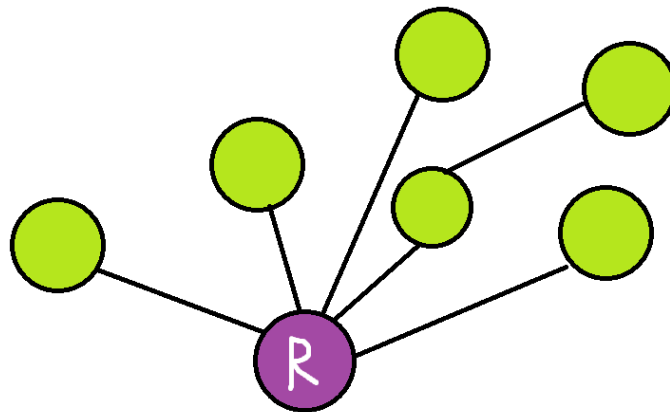


Mithilfe von MicrostreamDB ist es möglich, eine so genannte In-Memory-Datenbank zu erstellen. Dabei befindet sich der Inhalt der Datenbank nicht mehr auf der Festplatte, sondern wird in den Arbeitsspeicher des Rechners / Servers geladen, wodurch die Zugriffszeit enorm beschleunigt wird. Dabei wird die Datenbank selber nicht mehr in Tabellen, sondern in einem Objektgraphen gespeichert, wie es auch bei Java bei der Erzeugung von Objekten der Fall ist. Die SQL-Befehle werden mit so genannten Streams realisiert, welche in Java 8 hinzugefügt wurden. MicrostreamDB gehört deswegen zu den NoSQL-Datenbanken (NoSQL steht hierbei für *Not only SQL*).

In diesem Beispiel wird die Fachklasse `Cabin.java` ihre Daten mit MicrostreamDB speichern.

Aufbau einer Microstream Datenbank

Wie bereits oben erwähnt, arbeiten Microstream Datenbanken mit Objektgraphen und stellen die Daten auch in dieser Form dar. Die nachfolgende Abbildung zeigt eine vereinfachte Darstellung eines Objektgraphen:



Dabei verwendet Microstream eine Wurzel (hier mit dem R für root dargestellt), welche die Wurzel aller anderen Objekte ist. Die Wurzel des Knotenpunktes wird dabei als eigene Fachklasse implementiert, die eine Liste enthält, worin alle Objekte gespeichert werden sollen. Dieses Listenobjekt kann dabei ein Array, eine ArrayList oder eine ObservableList sein. Da es sich bei dieser Anwendung um eine JavaFX-Anwendung handelt, wird die ObservableList verwendet. Der folgende Codeausschnitt zeigt die einfache Implementierung der Root-Klasse:

RootCabins.java

```
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import org.green_pioneer.cruiseship_management.objectclasses.Cabin;

public class RootCabins {

    public ObservableList<Cabin> oblCabins = FXCollections.observableArrayList();

}
```

MicrostreamDB installieren

Um MicrostreamDB zu installieren, müssen in der pom.xml nur wenige Einträge hinzugefügt werden, um diese Bibliothek zu nutzen. In den *Repositories* muss folgender Eintrag eingefügt werden:

Pom.xml

```
<repository>
  <id>microstream-releases</id>
  <url>https://repo.microstream.one/repository/maven-public/</url>
</repository>
```

Anschließend müssen nur noch zwei Dependencies hinzugefügt werden, damit MicrostreamDB im Programm verwendet werden kann:

Pom.xml

```
<dependency>
  <groupId>one.microstream</groupId>
  <artifactId>storage.embedded</artifactId>
  <version>03.00.00-MS-GA</version>
</dependency>
<dependency>
  <groupId>one.microstream</groupId>
  <artifactId>storage.embedded.configuration</artifactId>
  <version>03.00.00-MS-GA</version>
</dependency>
```

Nach dem Einfügen dieser Einträge in die Pom.xml ist MicrostreamDB betriebsbereit und kann im gesamten Programm verwendet werden.

Daten aus MicrostreamDB lesen

Um Daten aus MicrostreamDB zu lesen, müssen diese vorher in den Objektgraphen importiert werden. Hierzu wurde eine Methode geschrieben, mit der es möglich ist, die Daten aus der MySQL-Datenbank zu importieren. Dazu wurde eine Methode implementiert, welche eine Microstream-Datenbank erzeugt.

Zunächst müssen aber in der Fachklasse selber Zugriffsobjekte erzeugt werden. Ein Zugriffsobjekt dient dem Zugriff auf der Root-Klasse und das andere dient dem Zugriff auf der DB-Fachklasse, worin die Methoden für den Zugriff auf die MySQL-Datenbank gespeichert sind. Sie werden wie folgt implementiert:

```
private static RootCabins aCabinRoot = new RootCabins();  
private DBAccessCabins aDBCabinsAccessor = new DBAccessCabins();
```

Eine Microstream-Datenbank wird wie folgt erzeugt:

MSDBCabinsAccessor.java

```
public void createDatabase() {  
  
    /* Eingebetten Storage Manager erzeugen und eine Sitzung starten. Mit  
    * ihm ist es möglich, Operationen wie Speichern, Löschen und Ändern  
    */ in der Datenbank vorzunehmen.  
    final EmbeddedStorageManager storageManager = EmbeddedStorageManager.start();  
  
    // Liste aus der Root-Klasse leeren, um vorherige Objekte zu löschen.  
    aCabinRoot.oblCabins.clear();  
    /* Liste aus der Root-Klasse mit Objekten aus der MySQL-Datenbank  
    */ füllen.  
    aCabinRoot.oblCabins = aDBCabinsAccessor.getCabinsInObservableList(0,  
        1000);  
    /* Die aus der MySQL-Datenbank importierten Objekte in Microstream  
    */ als Wurzelobjekt festlegen.  
    storageManager.setRoot(aCabinRoot);  
    // Jedes Objekt in der Konsole ausgeben.  
    aCabinRoot.oblCabins.forEach(System.out::println);  
    // Die zuvor festgelegten Objekte speichern.  
    storageManager.storeRoot();  
    // Die aktive Sitzung beenden.  
    storageManager.shutdown();  
}
```

Sofern keine Fehler auftreten, kann mit der Datenbank sofort gearbeitet werden.

Um die zuvor erzeugten Daten zu lesen und in die Tabelle aus dem GUI anzuzeigen, wurde eine weitere Methode implementiert, die das Auslesen der Daten ermöglicht. Da Microstream eine nicht-relationale Datenbank ist und es somit auch keine SQL-Queries gibt, müssen sie händischer ausgelesen werden. Dies geschieht mit einer for-Schleife, die die gesamte Liste aus der Root-Klasse durchläuft und die Daten nacheinander in eine separate Liste hinzufügt. Da JavaFX-Tabellen eine ObservableList erwarten, worin die Objekte gespeichert sind, wird diese auch zurückgegeben.

MSDBCabinsAccessor.java

```
public ObservableList<Cabin> getCabinRootList(int intFirstIndex, int
intLastIndex) {
    // Separate Liste anlegen, worin die Daten gespeichert werden sollen.
    ObservableList<Cabin> limitedOblCabins = FXCollections.observable-
        bleArrayList();
    /* Liste leeren, um zuvor hinzugefügte Objekte zu löschen (wichtig
    /* beim Durchblättern der Liste mit den beiden Buttons im GUI).
    limitedOblCabins.clear();

    if(aCabinRoot == null) {
        System.out.println("Keine Objekte enthalten!");
    } else {
        /* Prüfen, ob der letzte Index zum Blättern größer ist als die
        /* Liste, um ein herausblättern zu verhindern.
        if(intLastIndex > aCabinRoot.oblCabins.size()) {
            intLastIndex = aCabinRoot.oblCabins.size();
        /* Prüfen, ob der erste Index kleiner ist als 0, um ein
        /* versehentliches herausblättern zu verhindern.
        } else if(intFirstIndex < 0) {
            intFirstIndex = 0;
            intLastIndex = 24;
        }
        /* Die gesamte Liste durchlaufen und dabei die Objekte der
        /* separaten Liste hinzufügen. Der erste und letzte Index wird
        /* zum Durchblättern der Liste verwendet.
        for(int intIndexcounter = intFirstIndex; intIndexcounter < int-
            LastIndex; intIndexcounter++) {
            limitedOblCabins.add(aCabinRoot.oblCabins.get(intIndexcoun-
                ter));
        }
    }
    // Die gefüllte Liste zurückgeben.
    return limitedOblCabins;
}
```

Nicht vergessen!

Ein Array mit 25 Objekten beginnt bei 0 und endet mit 24!

Der Aufruf der Methode in der Controller-Klasse ist nahezu mit der vorherigen Herangehensweise mit klassischen Datenbanken identisch, weil auch diese Methode letztendlich eine ObservableList zurückgibt, mit der die Tabelle mit Daten gefüllt wird.

GUICabinsController.java

```
@FXML
private void showAllData() {
    //Sortiermodus deaktivieren
    boolSortByPrice = false;
    boolSortBySizeInSqm = false;
    btnNextData.setDisable(false);
    if(!(intFirstIndex == 0)) {
        btnPreviousData.setDisable(true);
    } else if(INT_NUMBER_OF_CABINS < INT_DATA_LIMIT) {
        btnPreviousData.setDisable(true);
        btnNextData.setDisable(true);
    }
    intFirstIndex = 0;
    intLastIndex = 24;
    oblCabins.clear();
    oblCabins = aMSDBCabinsAccessor.getCabinRootList(intFirstIndex, intLastIndex);
    tblCabins.setItems(oblCabins);
}
```

Daten in MicrostreamDB bearbeiten

Um Daten in einer MicrostreamDB zu bearbeiten, müssen die Attribute des zu bearbeitenden Objektes verändert werden. Dies geschieht mithilfe der set-Methoden, da es in Microstream keine SQL-Queries via Update gibt, welche diese Aufgabe erledigen können. Da auch die WHERE-Klausel nicht vorhanden ist, muss das zu bearbeitende Objekt mit einer anderen Methode gefunden werden. Eine Möglichkeit sind die in Java 8 eingeführten Streams, mit denen sich Listenstrukturen sehr effizient und übersichtlich durchsuchen lassen. Ansonsten ähnelt die Methode sehr der Methode zur Bearbeitung der Daten mithilfe einer Datenbank.

Die Schreibweise

```
.filter(aCabin -> intCabinNo == aCabin.getIntCabinNo())
```



gehört ebenfalls zu den neuen Features aus Java 8 und wird als Lambda-Ausdruck bezeichnet. Mit ihnen ist es möglich, weniger Code zu schreiben und ihn übersichtlicher zu halten. Dieser Lambda-Ausdruck ist am ehesten mit einer If-Anweisung vergleichbar. Ein Skript der Technischen Universität Kaiserslautern definiert Lambda-Ausdrücke wie folgt:

Lambda-Ausdrücke in Java sind quasi Methoden ohne Namen. Sie bestehen aus einer Liste von formalen Parametern, einem Pfeil -> und einem Funktionsrumpf. Im Gegensatz zu Methoden werden der Rückgabotyp und Exceptions nicht spezifiziert, sondern vom Compiler inferiert.

Annette Bieniusa, Mathias Weber, Peter Zeller – [Lambda Ausdrücke in Java \(Software Entwicklung 1\)](#)

Da Lambda-Ausdrücke in ihrer Thematik sehr komplex werden können, belasse ich es an dieser Stelle mit dieser einfachen Definition.

MSDBCabinsAccessor.java

```
public void updateCabin(int intCabinNo, int intNewCabinNo, String str-
Type, String strDeck, int intMaxPassengerCapacity, double dblSize-
InSqm, int intNoOfBeds, int intNoOfToilets, double dblPricePerPerson){

    EmbeddedStorageManager storageManager = EmbeddedStorage.start();

    /* Einen Stream in der Liste oblCabins aus der Root-Klasse starten
    */ und ein ggf. gefundenes Objekt vom Typ Kabine speichern.
    Cabin cabinToUpdate = aCabinRoot.oblCabins.stream()
        /* In jedem Kabinenobjekt nach der übergebenen KabinenNr
        * suchen und mit der bestehenden KabinenNr via der
        */ get-Methode vergleichen.
        .filter(aCabin -> intCabinNo == aCabin.getIntCabinNo())
        /* Wenn eine Kabine mit der übergebenen KabinenNr gefunden
        * wurde, soll der gesamte Stream nach dem Objekt durchsucht
        */ werden.
        .findAny()
        /* Wenn keine Kabine mit der übergebenen KabinenNr gefunden
        */ wurde, wird null zurückgegeben.
        .orElse(null);

    /* Die Werte des zu verändernden Objektes mit den übergebenen Werten
    */ verändern bzw. überschreiben.
    cabinToUpdate.setIntCabinNo(intNewCabinNo);
    cabinToUpdate.setStrType(strType);
    cabinToUpdate.setStrDeck(strDeck);
    cabinToUpdate.setIntMaxPassengerCapacity(intMaxPassengerCapacity);
    cabinToUpdate.setDblSizeInSqm(dblSizeInSqm);
    cabinToUpdate.setIntNoOfBeds(intNoOfBeds);
    cabinToUpdate.setIntNoOfToilets(intNoOfToilets);
    cabinToUpdate.setDblPricePerPersonInEUR(dblPricePerPerson);
    // Das veränderte Objekt abspeichern.
    storageManager.store(cabinToUpdate);
    storageManager.shutdown();
}
```

Die Methode im Controller unterscheidet sich im Gegensatz zur Datenbanknutzung ebenfalls nur geringfügig. Die vielen If-Abfragen dienen dazu, eventuelle Eingabefehler des Benutzers abzufangen.

Info: Änderungen in beiden Systemen

Die Daten werden sowohl in MicrostreamDB als auch in der MySQL-Datenbank geändert (dies betrifft auch das Einfügen und Löschen von Daten), weil andere Fachklassen die Daten aus der MySQL-Datenbank benötigen, weil das Programm nicht vollständig auf MicrostreamDB umgerüstet wurde.

GUICabinsController.java

```
@FXML
private void editCabin() {
    try {

        /* Benutzereingaben aus den Textfeldern speichern. Aus Platzgründen
        */ in dieser Dokumentation weggelassen.

        /* Umsetzung des Kundenwunsches: Das System darf nicht eine Kabine
        */ mit der Nummer 13 speichern!
        if(intNewCabinNo == 13) {
            ErrorHandler.initErrorDialog("Unable to save cabin!", "A cabin
            with the no 13 mustn't be board!");
            // Textfeld fokussieren, damit es blau umrandet wird.
            txtNewCabinNo.requestFocus();
            // Text im Textfeld markieren.
            txtNewCabinNo.selectAll();
            // Prüfen, ob die Benutzereingaben in den Feldern gültig sind.
        } else if(isPassengerCapacityValid(intMaxPassengerCapacity) ==
        false || isSizeInSqmValid(dblSizeInSqm) == false ||
        isNoOfBedsValid(intNoOfBeds) == false ||
        isNoOfToiletsValid(intNoOfToilets) == false) {
            /* Der Code zum interagieren mit dem Benutzer befindet sich
            */ in den Methoden zur Überprüfung der Benutzereingaben.
        } else if(isCabinExisting(intCabinNo)) {
            Optional<ButtonType> confMessage = ErrorHandler.initConfirmation-
            Dialog("Are you sure that you want to update the cabin?", "");
            if(confMessage.get() == ButtonType.OK) {
                // Kabine in der MySQL-Datenbank verändern.
                aDBCabinsAccessor.updateCabin(intCabinNo, intNewCabinNo, str-
                Type, strDeck, intMaxPassengerCapacity, dblSizeInSqm, int-
                NoOfBeds, intNoOfToilets, dblPricePerPerson);
                // Kabine in MicrostreamDB verändern.
                aMSDBCabinsAccessor.updateCabin(intCabinNo, intNewCabinNo,
                strType, strDeck, intMaxPassengerCapacity, dblSizeInSqm,
                intNoOfBeds, intNoOfToilets, dblPricePerPerson);
                oblCabins.clear();
                // Liste erneut einlesen, um die Änderungen zu sehen.
                oblCabins = aMSDBCabinsAccessor.getCabinRootList(intFirstIn-
                dex, intLastIndex);
                // Tabelle mit den neuen Daten füllen.
                tblCabins.setItems(oblCabins);
            } else {
                return;
            }
        } else {
            ErrorHandler.initErrorDialog("Unable to edit cabin!", "The cabin
            was not found!");
        }

    } catch(NumberFormatException nfex) {
        ErrorHandler.initErrorDialog("Invalid number format or empty text-
        fields!", "Please check your entered values!");
    }
}
```

Daten in MicrostreamDB einfügen

Das Einfügen von Daten in MicrostreamDB ist im Gegensatz zu MySQL wesentlich leichter und kürzer, da das Preparing und Binding vollständig entfällt. Stattdessen wird dem Objektgraphen einfach ein neues Objekt hinzugefügt.

MSDBCabinsAccessor.java

```
public void insertNewCabin(int intCabinNo, String strType, String
    strDeck, int intMaxPassengerCapacity, double dblSizeInSqm, int intNoOf-
    Beds, int intNoOfToilets, double dblPricePerPerson) {
    EmbeddedStorageManager storageManager = EmbeddedStorage.start();
    // Ein neues Objekt vom Typ Kabine erzeugen.
    Cabin aNewCabin = new Cabin(intCabinNo, strType, strDeck, intMaxPas-
        sengerCapacity, dblSizeInSqm, intNoOfBeds, intNoOfToilets, dblPrice
        PerPerson);
    // Das neue Objekt der Liste aus der Root-Klasse hinzufügen.
    aCabinRoot.oblCabins.add(aNewCabin);
    // Die Änderungen der Liste speichern und übernehmen.
    storageManager.store(aCabinRoot.oblCabins);
    storageManager.shutdown();
}
```

Die entsprechende Methode in der Controller-Klasse ähnelt dagegen den anderen Methoden zum Speichern von Daten. Da es in Microstream jedoch kein Autoincrement gibt, muss dieses aus Datenbanken bekannte Feature manuell implementiert werden. Dazu muss lediglich ermittelt werden, wie groß die größte Kabinennummer aller Kabinen in der Liste ist. Eine Methode durchläuft dabei die Liste mit einem Stream, vergleicht die Werte der Kabinennummer via `getIntCabinNo()` miteinander und speichert das Objekt mit der größten Kabinennummer in die Objektvariable `cabinWithMaxPassengerCapacity`. Anschließend wird eine Variable vom Typ `int` angelegt, in der die größte Kabinennummer aus dem zuvor ermittelten Objekt extrahiert und gespeichert wird. Diese wird am Ende der Methode zurückgegeben.

MSDBCabinsAccessor.java

```
public int getHighestCabinNo() {
    Cabin cabinWithHighestCabinNo = aCabinRoot.oblCabins.stream()
        /* Die Kabine mit dem größten Wert aller gespeicherten
        * Kabinen ermitteln, in dem das Attribut intCabinNo mit
        */ jeder anderen Kabine aus der Liste verglichen wird.
        .max(Comparator.comparingInt(Cabin::getIntCabinNo))
        // Die ermittelte Kabine per get bekommen und speichern.
        .get();
    /* Die Kabinennummer der zuvor ermittelten Kabine extrahieren und in
    * eine separate Variable speichern, die anschließend zurückgegeben
    * wird.
    int intHighestCabinNo = cabinWithHighestCabinNo.getIntCabinNo();
    return intHighestCabinNo;
}
```

In der Controller-Klasse wird zu Beginn eine Variable deklariert, die einen Aufruf der Methode speichert. Sie enthält anschließend den Wert der höchsten Kabinennummer.

```
private int intHighestCabinNo = aMSDBCabinsAccessor.getHighestCabinNo();
```

Mit ihr lässt sich Autoinkrement in der Methode `saveCabin()` implementieren:

MSDBCabinsAccessor.java

```
@FXML
private void saveCabin() {
    try {
        /* Prüfen, ob die Kapazität des Schiffes bzgl. der Kabinen beim
        */ Erfassen einer neuen Kabine überschritten wird.
        if(aMSDBCabinsAccessor.getNumberOfCabins() >= INT_MAX_NUMBER_OF_
            CABINS) {
            ErrorHandler.initErrorDialog("Unable to save the cabin!", "The
                max cabin capacity of this ship has been reached!");
        } else {
            //Kabinennummer um 1 inkrementieren.
            intHighestCabinNo++;

            /* Benutzereingaben aus den Textfeldern speichern. Aus Platz-
            */ Gründen in dieser Dokumentation weggelassen.

            /* Nimmt eine Kabine die Nummer 13 an, wird der Wert ein
            if(intHighestCabinNo == 13) {
                intHighestCabinNo++;
                aDBCabinsAccessor.insertNewCabin(strType, strDeck, intMaxPas-
                    sengerCapacity, dblSizeInSqm, intNoOfBeds, intNoOfToilets,
                    dblPricePerPerson);
                aMSDBCabinsAccessor.insertNewCabin(intHighestCabinNo, str-
                    Type, strDeck, intMaxPassengerCapacity, dblSizeInSqm, int-
                    NoOfBeds, intNoOfToilets, dblPricePerPerson);
                oblCabins.clear();
                oblCabins = aMSDBCabinsAccessor.getCabinRootList(intFirstIn-
                    dex, intLastIndex);
                tblCabins.setItems(oblCabins);
            } else if(isPassengerCapacityValid(intMaxPassengerCapacity) ==
                false || isSizeInSqmValid(dblSizeInSqm) == false || isNoOf-
                BedsValid(intNoOfBeds) == false || isNoOfToilets-
                Valid(intNoOfToilets) == false) {
                /* Der Code zum interagieren mit dem Benutzer befindet sich
                */ in den Methoden zur Überprüfung der Benutzereingaben.
            } else {
                /* Selbiges Einfügen wie oben, jedoch wird die Variable int-
                */ HighestCabinNo nicht inkrementiert.
            }
        }
    } catch (NumberFormatException nfex) {
        ErrorHandler.initErrorDialog("Invalid number format or empty text-
            fields!", "Please check your entered values!");
    }
}
```

Info: Mehrere Konstruktoren in der Fachklasse Cabin.java

Die Fachklasse besitzt aufgrund der Verwendung von MicrostreamDB als auch MySQL neben dem leeren Standardkonstruktor zwei weitere Konstruktoren. Einer benötigt zum Erstellen einer Kabine nicht die Kabinennummer, der andere wiederum nicht. Ersterer wird von MicrostreamDB verwendet, der andere von MySQL. Die Kabinennummer ist beim MySQL-Konstruktor nicht nötig, da MySQL mit Autoincrement arbeitet.

Daten in MicrostreamDB löschen

Objekte lassen sich ganz einfach aus dem Objektgraphen von MicrostreamDB löschen. Dazu wird in der jeweiligen Methode wieder eine Sitzung und anschließend ein Stream in der Liste der Objekte gestartet. Mit diesem wird das Objekt bzw. die zu löschende Kabine anhand der übergebenen Kabinennummer ermittelt. Dieses Objekt wird anschließend gelöscht und die Änderungen an der Liste werden übernommen. Zu guter Letzt wird die Sitzung geschlossen.

MSDBCabinsAccessor.java

```
public void deleteCabin(int intCabinNo) {
    EmbeddedStorageManager storageManager = EmbeddedStorage.start();
    Cabin cabinToDelete = aCabinRoot.oblCabins.stream()
        .filter(aCabin -> intCabinNo == aCabin.getIntCabinNo())
        .findAny()
        .orElse(null);
    // Kabine aus der Datenbank löschen.
    aCabinRoot.oblCabins.remove(cabinToDelete);
    // Änderungen an der Datenbank übernehmen.
    storageManager.store(aCabinRoot.oblCabins);
    storageManager.shutdown();
}
```

Daten in MicrostreamDB sortieren

Die Sortierung von Daten in MicrostreamDB funktioniert nicht ganz so trivial, wie es in MySQL der Fall ist. Während man dort lediglich die Klausel ORDER BY mit der zugehörigen Spalte und dem Sortiermodus (aufsteigend oder absteigend) anhängt und die Objekte wieder neu erzeugt und anschließend in eine ObservableList speichert und zurückgibt, muss bei Microstream auf native Sortiermethoden zurückgegriffen werden. Dazu wird die zu sortierende Liste in eine andere Liste kopiert, sortiert und anschließend zurückgegeben. Mithilfe der booleschen Sortiervariablen in der Controller-Klasse wird der Sortiermodus mittels If-Abfrage automatisch aufgerufen, sobald er über das Menü aktiviert wurde. Im folgenden Listing wird gezeigt, wie die Sortierung als Methode implementiert wurde:

MSDBCabinsAccessor.java

```
public ObservableList<Cabin> sortBySizeInSqm(int intFirstIndex, int int-
LastIndex) {
    // Listen für die Sortierung deklarieren.
    ObservableList<Cabin> oblCabinsToSortBySizeInSqm = FXCollections.ob-
servableArrayList();
    ObservableList<Cabin> oblSortedCabinsBySizeInSqm = FXCollections.ob-
servableArrayList();
    // MicrostreamDB mit Kabinen in die Liste kopieren.
    oblCabinsToSortBySizeInSqm.addAll((aCabinRoot.oblCabins));
    // Liste mittels eines Comparators sortieren.
    oblCabinsToSortBySizeInSqm.sort(Comparator.comparingDouble
(Cabin::getDbSizeInSqm));
    // Indizes prüfen, um eine OutOfBounds-Exception zu verhindern.
    if(intLastIndex > oblCabinsToSortBySizeInSqm.size()) {
        intLastIndex = oblCabinsToSortBySizeInSqm.size();
    } else if(intFirstIndex < 0) {
        intFirstIndex = 0;
    }
}
```

```

        intLastIndex = 24;
    }

    // Liste leeren, um alte Daten beim Wiederaufruf zu löschen.
    oblSortedCabinsBySizeInSqm.clear();

    /* Objekte aus der sortierten Liste der anderen Liste hinzufügen, um
    */ sie mittels der übergebenen Indizes zu steuern.
    for(int intIndexcounter = intFirstIndex; intIndexcounter < intLastIndex; intIndexcounter++) {
        oblSortedCabinsBySizeInSqm.add(oblCabinsToSortBySizeInSqm.get(intIndexcounter));
    }
    // Die sortierte Liste zurückgeben
    return oblSortedCabinsBySizeInSqm;
}

```

Wie zuvor erwähnt, wird die Methode in der Controller-Klasse aufgerufen, wenn der Benutzer den entsprechenden Menüpunkt anklickt. Diese wird zuvor über eine andere Methode aufgerufen, welche die boolesche Variable mit dem Sortiermodus aktiviert.

GUICabinsController.java

```

@FXML
private void sortBySizeInSqm() {
    boolSortBySizeInSqm = true;
    oblCabins.clear();
    // Aufruf der Methode zum Sortieren der Daten
    oblCabins = aMSDBCabinsAccessor.sortBySizeInSqm(intFirstIndex, intLastIndex);
    tblCabins.setItems(oblCabins);
}

```

Klickt der Benutzer anschließend auf den Button *Weiter* (➤), so wird überprüft, ob ein Sortiermodus aktiviert wurde. Ist dem so, wird die Methode mit den neuen Indizes aufgerufen und die Liste wird in der Methode mit den nächsten Daten aus der sortierten Liste gefüllt. Die Vorgehensweise bei dem Button *Zurück* (◀) funktioniert ähnlich, sodass diese nicht weiter erläutert wird.

GUICabinsController.java

```

@FXML
private void showNextData() {
    // Werte der Indizes aktualisieren
    intFirstIndex = intLastIndex;
    intLastIndex = intLastIndex + INT_DATA_LIMIT;
    // Buttons deaktivieren, um ein Herausblättern zu verhindern.
    if ((intFirstIndex + INT_DATA_LIMIT) >= INT_NUMBER_OF_CABINS) {
        btnNextData.setDisable(true);
    } else if(intFirstIndex >= 0) {
        btnPreviousData.setDisable(false);
    }
    /* Wenn ein Sortiermodus aktiviert ist, die Liste mit den nächsten
    */ Daten füllen. Ansonsten einfach weiter.
    if(boolSortByPrice == true || boolSortBySizeInSqm == true) {
        enableSortmode(boolSortByPrice, boolSortBySizeInSqm);
    }
}

```

```
} else {  
    oblCabins.clear();  
    oblCabins = aMSDBCabinsAccessor.getCabinRootList(intFirstIndex,  
        intLastIndex);  
    tblCabins.setItems(oblCabins);  
}  
}
```

Die Methode

`enableSortmode(boolSortByPrice, boolSortBySizeInSqm)`

überprüft, welcher Sortiermodus aktiv ist und ruft die jeweilige Methode mit einer einfachen If-Abfrage auf, weswegen auch hier auch auf das Listing verzichtet wurde, da dieses äußerst trivial ist. Der Sortiermodus wird verlassen, in dem der Benutzer auf den Button *Show all data* klickt, denn dadurch werde alle booleschen Variablen für den Sortiermodus automatisch deaktiviert bzw. auf den Wert *false* gesetzt.

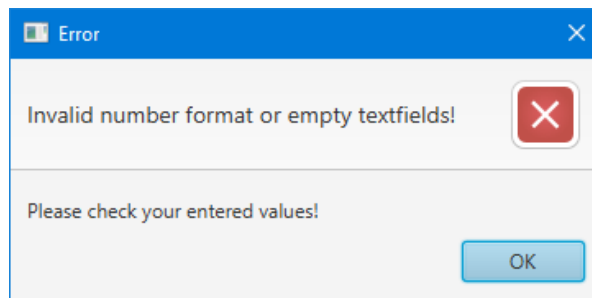
Probleme bei der Implementierung von MicrostreamDB

Die Implementierung von MicrostreamDB war etwas ungewohnt, weil es sich hier um eine NoSQL-Datenbank handelt, welche überhaupt keine SQL-Queries mehr benötigt. Gleichzeitig werden die Daten nicht mehr in Tabellen gespeichert, sondern in Objektgraphen, wodurch sich auch das Verfahren zum Anlegen der Datenbank änderte. Dies führte beim Starten von mehreren Sitzungen dazu, dass die Datenbank im Zustand *locked* war, da bereits eine andere Sitzung noch aktiv war und nicht beendet wurde. Dieser Fehler tritt dann auf, wenn das Programm nicht startet, aber dennoch z.T. kompiliert wurde. In meinem Betrieb erklärte man mir, dass dies auch bei SQL-Datenbanken durchaus der Fall ist, wenn mehrere Leute Operationen wie Ändern oder Löschen genau gleichzeitig durchführen und das System zum Schutz auf den Zustand *locked* wechselt, wodurch keine Änderungen mehr möglich sind. Auch traten einige Exceptions aus diesem Grund auf, wenn Sitzungen nicht geschlossen wurden und eine weitere Sitzung gestartet wurde. Daher müssen Sitzungen immer geschlossen werden.

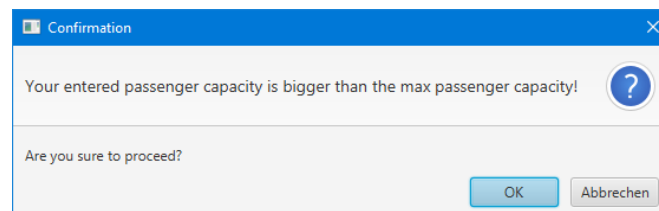
Letztendlich waren jedoch alle Probleme lösbar und konnten durch gezielte Fehlersuche aufgespürt und behoben werden.

Die Fachklasse ErrorHandler

Die Fachklasse ErrorHandler ist eine Hilfsklasse, in der Fehlermeldungen erzeugt werden können. Diese können an jeder beliebigen Stelle des Programms erzeugt werden, wodurch es dem Entwickler möglich ist, den Benutzer bei eventuellen Falscheingaben zu informieren. Bei der nachfolgenden Fehlermeldung wird der Benutzer beispielsweise darüber informiert, dass er entweder nicht alle Textfelder oder diese mit einem ungültigen Zahlenformat ausfüllte. Dieses tritt beispielsweise dann auf, wenn der Benutzer ein Komma statt eines Punktes verwendet.



Eine weitere Kategorie sind so genannten Confirmation-Dialoges, mit denen der Benutzer eine Aktion bestätigen kann. In diesem Beispiel gab der Benutzer eine größere Passagierkapazität der Kabine ein als die Kapazität der Kabine mit der höchsten Passagierkapazität auf dem Schiff. Klickt der Benutzer auf *OK*, so wird die Eingabe bestätigt und die Kabine wird in der Datenbank erfasst. Klickt der Benutzer hingegen auf *Abbrechen*, so wird das Einfügen der Kabine abgebrochen und das entsprechende Feld wird im GUI markiert, sodass der Benutzer seine Eingabe noch einmal überprüfen kann.



Info: Deutsche Buttons trotz englisches Programm

JavaFX bezieht die Namen der Buttons aus den Spracheinstellungen des jeweiligen Betriebssystems. Ein auf Englisch eingestelltes Windows wird die Buttons in Englisch anzeigen, während ein auf Deutsch eingestelltes Betriebssystem die deutschen Namen der Buttons verwendet.

Die Fenster wurden mithilfe von Methoden realisiert und ähneln sich dem Aufbau sehr stark. Lediglich die Methode für den Confirmation-Dialog gibt das Ergebnis des Benutzers zurück. Nachfolgend ist sowohl die Implementierung für ein Error-Dialog und ein Confirmation-Dialog zu sehen:

ErrorHelper.java

```
public static void initErrorDialog(String strHeaderText, String strContentText) {
    // Titel des Fensters festlegen
    anErrorAlert.setTitle("Error");
    // Header und Text mit den übergebenen Strings festlegen
    anErrorAlert.setHeaderText(strHeaderText);
    anErrorAlert.setContentText(strContentText);
    /* Das Programm anhalten und abwarten, bis der Benutzer die Fehler-
    * meldung bestätigt.
    */
    anErrorAlert.showAndWait();
}
```


ErrorHelper.java

```
public static Optional<ButtonType> initConfirmationDialog(String strHeaderText, String strContentText) {
    aConfirmationAlert.setTitle("Confirmation");
    aConfirmationAlert.setHeaderText(strHeaderText);
    aConfirmationAlert.setContentText(strContentText);
    /* Ergebnis des Benutzers (OK oder Abbrechen) in der Variable result
    */ vom Typ Optional<ButtonType> speichern.
    Optional<ButtonType> result = aConfirmationAlert.showAndWait();
    // Das Ergebnis zurückgeben
    return result;
}
```

Damit die Methoden im gesamten Programm aufgerufen werden können, müssen sie zusätzlich mit dem Parameter `static` deklariert werden. Ebenso müssen jegliche Arten von Dialogs vorher im Programm mittels eines Alert-Objektes deklariert werden, damit auf sie zugegriffen werden kann. Sie besitzen ebenfalls den Parameter `static`, da sie nicht von Objekten abhängig sind.

```
private static Alert aConfirmationAlert = new Alert(AlertType.CONFIRMATION);
private static Alert anInformationAlert = new Alert(AlertType.INFORMATION);
private static Alert aWarningAlert = new Alert(AlertType.WARNING);
private static Alert anErrorAlert = new Alert(AlertType.ERROR);
```

Exkurs: Optional<Datentyp>

Mit dem Datentyp `Optional` kann ein Objekt deklariert werden, dass entweder einen Wert enthält oder leer ist (Achtung! Leer ist hier nicht `null`!). Dadurch ist es möglich, Fehler effektiver abzufangen. Der hier verwendete Datentyp `ButtonType` stammt aus JavaFX und gibt, wie der Name schon andeutet an, um welchen Buttontypen es sich genau handelt. Optionals werden erst ab Java 8 unterstützt.