Berufsschule 7, Haunstetter Str. 66, 86161 Augsburg

Dokumentation der Projektarbeit

CWO ERP

Ein Warenwirtschaftssystem für die Musikindustrie mit einer eigens entwickelten relationalen Datenbank

Luca Goldhausen
11bFI2
23.07.2021

Ausbildungsbetrieb:

TopM Software GmbH
Albert-Einstein-Straße 1-3
86399 Bobingen

INHALTSVERZEICHNIS

Die Firma TopM Software GmbH entwickelt seit 1991 Warenwirtschaftssoftware für verschiedene Branchen, wie Einzelhandel, Produktion, Reifenhandel, aber auch den Eurofighter. Alle Produkte inklusive Tools, die zur Entwicklung verwendet werden, sind von der TopM eigens entwickelt, um eine Unabhngigkeit von Drittanbietern aufrecht zu erhalten. Auch sind dadurch Änderungen jeglicher Art möglich, da die Gesamtheit aller Sourcecodes Eigentum der TopM sind und zu jeder Zeit verändert werden kann.

Ziel des Projektes war es, eine Warenwirtschaftssoftware zu haben, die spezielle auf die Branche Musik, speziell auch die Musikproduktion, zugeschnitten ist. Es soll möglich sein, Lieder, Kontakte und Rechnungen abzuspeichern, sie über eine Suchfunktion zu filtern und bearbeiten zu können. Der Bearbeitungsstatus der Lieder soll einstellbar sein, es soll Auswertungen für Liedergenres geben und es soll möglich sein, z.B. alle Lieder eines Künstlers filtern und anzeigen zu können. Benutzer müssen sich auch mit einem Benutzernamen und Passwort anmelden können, da eine Rechtestruktur inklusive Benutzermanagement nötig ist.

Eine einfache Erweiterbarkeit soll sie ebenfalls aufweisen, da ERP-Systeme (Enterprise Resource Planning, engl. Warenwirtschaftssystem) erfahrungsgemäß unter ständiger Veränderung stehen.

Da eine eigene relationale Datenbank für dieses ERP-System geschrieben wurde, war hier auch die Geschwindigkeit wichtig, um eine bestmögliche Benutzerfreundlichkeit bieten zu können. Die Datenbank soll aufgrund der Erweiterbarkeit der Software universell sein; sie soll serialisierte Objekte jeglicher Art abspeichern und abrufen können.

Im „Big Data“ Zeitalter ist es nötig, den Überblick über seine Daten zu haben und das Handhaben und Verwalten dieser einfach zu gestalten. Eine Software, die diese Arbeiten für Musikproduzenten verrichtete scheint es noch nicht in dieser Ausführung zu geben, weshalb die Entwicklung dieser sinnvoll erschien.

Vor allem Tontechniker und Produzenten, die eine vielzahl an Kunden haben, müssen den Überblick über deren Projekte behalten, was durch diese Software vereinfacht werden soll.

1.4. Verwendete Tools

Die Verwendete integrierte Entwicklungsumgebung (IDE) für dieses Projekt ist IDEA 2021.1.3 (Community Edition) von IntelliJ. Die Wahl fiel auf diese IDE, da die Programmiersprache, mit der die Software geschrieben wurde, Kotlin, ebenfalls von IntelliJ stammt. Kotlin ist eine statisch typisierte, mit Java kompatible Sprache, die auch plattformübergreifende Kapazitäten aufweist. Als Version Controlling Software (VCS) wurde Git in Verbindung mit GitHub verwendet, da sowohl der Fortschritt als auch die noch zu erledigenden Punkte sehr leicht einzusehen sind. Als Build-Management-Automatisierungs-Tool wurde Gradle gewählt, da es im Prinzip bereits in die IDE eingebaut war und das Importieren aller benötigten Libraries ermöglicht.

1.5. Verwendete Libraries

Im Folgenden werden die für das Projekt gewählten Libraries mit deren in Klammern geschrieben Gradle-Abhängigkeiten aufgezeigt.

Für die Benutzeroberfläche wurde TornadoFX *(no.tornado:tornadofx:1.7.15)*, ein JavaFX Framework für Kotlin, von Edvin Syse verwendet. Um diese Oberfläche auch bei langfristig laufenden Aufgaben ansprechbar zu halten, wurde teilweise asynchron Programmiert, sprich, es wurden Threads benötigt. Da es von IntelliJ eine eigens entwickelte Version von Threads, sogenannte Coroutines, gibt, fiel die Wahl auf diese *(org.jetbrains.kotlinx:kotlinx-coroutines-core:1.5.0)*. Da die in die Datenbank zu speichernden Objekte serialisiert werden müssen, waren Libraries für die Serialisierung notwendig. Es wurde sowohl eine JSON- *(org.jetbrains.kotlinx:kotlinx-serialization-json:1.2.1),* als auch eine Google Protocol Buffer *(org.jetbrains.kotlinx:kotlinx-serialization-protobuf:1.2.0)* Library hergenommen, um dies zu ermöglichen. Um einen CSV-Import in der Software umsetzen zu können, wurde die Kotlin CSV Library von Doyaaaaaken gewählt *(com.github.doyaaaaaken:kotlin-csv-jvm:0.15.2)*.

1.6. Projektabgrenzung

Nicht Teil des Projektes war die Datenbank selbst, da diese bereits vor Beginn des Projektes fertig geschrieben war. Zwar wurde diese im Laufe des Projektes teilweise verbessert, war aber nicht mehr Kernbestandteil des Projektes, was sich auf Inhalt, also die Hauptmodule, und die Benutzerfreundlichkeit belief.

2. Projektplanung

Da die Datenbank bereits vor diesem Projekt geschrieben wurde, waren einige Planungsschritte bereits getan, wie z.B. welche Programmiersprache und Libraries verwendet werden sollen.

Für die Planung der Hauptmodule, also Analyse wie Entwürfe, der ERP-Software sowie der Arbeitsweisen und Benutzerfreundlichkeit wurden jedoch sechs Stunden eingeplant, um später in der intensiven Implementierungsphase nicht den Überblick zu verlieren.

Im Folgenden wird eine Tabelle zur Einschätzung der Zeiteinteilungen der jeweiligene Projektphasen abgebildet.

Projektphase	**Soll-Zeit**
Analysephase	2 h
Entwurfsphase	3 h
Implementierungsphase	11 h
Erstellen der Dokumentation	4 h
Gesamt	20 h

Die Gesamtzeit des Projektes beläuft sich demnach auf 20 Stunden, wovon der größte Teil die Implementierung sein wird.

3. UMSETZUNG

3.1. PROJEKTSTRUKTUR

Bei der Umsetzung des Projektes wurde nach dem Model-View-Controller Muster vorgegangen. Das bedeutet, dass GUI und Logik getrennt geschrieben werden, um z.B. ein Austauschen eines Elementes zu begünstigen und eine ordentliche Struktur zu haben.

Die Ordnerstruktur des Projektes ist so aufgebaut, dass GUI-, Logik- und nicht kategorisierbare Elemente ihre eigenen Verzeichnisse haben.

- src\kotlin\main
  - db
  - modules
    - m1
      - gui (View)
      - logic (Controller)
      - misc (Model)
    - m2
      - gui (View)
      - logic (Controller)
      - misc (Model)
    - m3
      - gui (View)
      - logic (Controller)
      - misc (Model)
    - mx
      - gui (View)
      - logic (Controller)
      - misc (Model)

3.2. Implementierung der Datenstrukturen

Wie bereits in Kapitel *1.3 Projektabgrenzung* erwähnt wurde, war die selbst geschriebene relationale Datenbank bereits entwickelt. Diese wurde für diese Projekt verwendet. Es handelt sich hier um eine relationale Datenbank, was bedeutet, dass Datensätze miteinander verknüpft werden können. Dafür hergenommen werden sogenannte unique identifiers, also UIDs, die in einem Modul nur einmal vorkommen können. Wird ein Datensatz, z.B. ein Kontakt, in einen anderen Datensatz geladen, so wird eine Referenz zu diesem eingetragen. Sollte sich der geladene Datensatz verändern, so ändert sich dieser auch im Datensatz, in dem er geladen ist. Eine andere Form einer Datenbank wäre eine Dokumentendatenbank, in der ganze Datensätze für sich abgespeichert werden, wie z.B. eine PDF. Da hier keine Verknüpfungen bestehen kam diese Form nicht für dieses Projekt in Frage.

Für die Archivierung von Datensätzen der Hauptmodule (Lieder, Kontakte, Rechnungen) werden *Bytearrays* auf die Festplatte gespeichert, die mit der Serialisierung der Datenklassen der jeweiligen Module durch den Google Protocol Buffer, im Weiteren Protobuf gennant, generiert wurden. Indexdateien zu den Datensätzen sowie Benutzerdaten werden als serialisierte *JSON-Strings* abgespeichert, da sich Protobuf in einem experimentellen, jedoch stabilen, Zustand befand und Probleme hatte, eine *LinkedHashMap*, die ein weiteres Objekt als Wert besaß, zu serialisieren. Die Konfigurationsdatei, die beim ersten Ausführen der Software automatisch erzeugt wird, wird ebenfalls als *JSON-String* abgespeichert, um dem Benutzer beim Anpassen der Werte ein lesbares Format bereitzustellen.

Während dem Arbeiten mit dem ERP-System bleiben alle Indexdateien als deserialisierte *LinkedHashMap*s im Arbeitsspeicher, da diese selbst bei wachsenden Datenmengen eine relativ gleichbleibende Geschwindigkeit beim Lesen und Schreiben besitzen.

Indexklasse:

```
package db

import kotlinx.serialization.SerialName
import kotlinx.serialization.Serializable

@Serializable
data class Index(@SerialName("m") val module: String)
{
   @SerialName("i")
   val indexMap = mutableMapOf<Int, IndexContent>()
}
```

IndexContentklasse:

```
package db

import kotlinx.serialization.SerialName
import kotlinx.serialization.Serializable

@Serializable
data class IndexContent(
   @SerialName("u") val uID: Int,
   @SerialName("c") val content: String,
   @SerialName("p") var pos: Long,
```

```
    @SerialName("b") var byteSize: Int
)
```

Mit den oben aufgezeigten Klassen wurde die Datenstruktur der Indizes umgesetzt, wobei @SerialName verwendet wurde, um die Felderbezeichnungen im JSON-Format kurz zu halten, was in einem kleineren Speicherplatzverbrauch resultiert.

```
@ExperimentalSerializationApi
override fun indexEntry(entry: Any, posDB: Long, byteSize: Int, writeToDisk: Boolean) = runBlocking {
    entry as Contact
    buildIndex0(entry, posDB, byteSize)
    buildIndex1(entry, posDB, byteSize)
    if (writeToDisk) launch { writeIndexData() }
}

override suspend fun writeIndexData()
{
    db.getIndexFile("M2", 0).writeText(Json.encodeToString(indexList[0]))
    db.getIndexFile("M2", 1).writeText(Json.encodeToString(indexList[1]))
}

//Index 0 (Contact.uID)
override fun buildIndex0(entry: Any, posDB: Long, byteSize: Int)
{
    entry as Contact
    indexList[0]!!.indexMap[entry.uID] = IndexContent(entry.uID, "${entry.uID}", posDB, byteSize)
}
...
```

Das Indizieren erfolgt, indem das zu serialisierende Objekt der Funktion indexEntry übergeben wird. Hier werden dann die Indizes generiert, serialisiert und anschließend in die Indexdatei geschrieben.

Invoiceklasse:

```
package modules.m3

import db.CwODB
import kotlinx.serialization.Serializable
import modules.IEntry
import modules.IInvoice

@Serializable
data class Invoice(override var uID: Int) : IEntry, IInvoice
{
    var seller: String = "?"
    var sellerUID: Int = -1
    var buyer: String = "?"
    var buyerUID: Int = -1
    var date: String = "??.??.????"
    var text: String = "?"
    var price: Double = 0.0
    fun initialize() = if (uID == -1) uID = CwODB().getUniqueID("M3")
}
```

Die Datenstruktur der Klassen der Hauptmodule wurde simpel gehalten, es handelt sich hierbei um Datenklassen, die in Kotlin erzeugt werden können. Diese eignen sich hervorragend zur Speicherung größerer auf mehrere Objekte verteilte Datenmengen und stellen nützliche Funktionen wie das Kopieren oder Vergleichen von Objekten bereit.

3.3. Implementierung der Benutzeroberfläche

Die Benutzeroberfläche wurde mittels *TornadoFX* umgesetzt, was im Hintergrund eine *JavaFX* Oberfläche aufbaut. Da sich an das MVC-Muster gehalten werden soll, mussten zu den Datenklassen *Property*-Klassen und *ItemViewModel*-Klassen geschrieben werden, die sozusagen als Schnittstelle zwischen der Oberfläche und den darunterliegenden Datenklassen fungieren. Die ItemViewModel-Klassen sind diejenigen, die dann auch in die *Views* oder *Fragmente* injiziert werden und die Eingabe von Daten für den Benutzer ermöglichen.

ItemViewModel-Klasse des Invoice-Moduls:

```
class InvoiceModel : ItemViewModel<InvoiceProperty>(InvoiceProperty())
{
   val uID = bind(InvoiceProperty::uIDProperty)
   val seller = bind(InvoiceProperty::sellerProperty)
   val sellerUID = bind(InvoiceProperty::sellerUIDProperty)
   val buyer = bind(InvoiceProperty::buyerProperty)
   val buyerUID = bind(InvoiceProperty::buyerUIDProperty)
   var date = bind(InvoiceProperty::dateProperty)
   var text = bind(InvoiceProperty::textProperty)
   var price = bind(InvoiceProperty::priceProperty)
}
```

In den *ItemViewModel*-Klassen werden die *Properties* der darunterliegenden Datenklasse gebunden, was bedeutet, dass Änderungen über die Benutzeroberfläche auch den tatsächlichen Datensatz verändern, der im Anschluss serialisiert und abgespeichert wird.

```
val invoice: InvoiceModel by inject()
```

Über die inject()-Funktion wird das Objekt der *ItemViewModel*-Klasse z.B. in die *View* injiziert, was bedeutet, dass das Objekt per *lazy loading* nun über die Oberfläche ansprechbar ist und auch über den Code in der *View* modifiziert werden kann. Das *lazy loading* ist vor allem für *Controller* nützlich, da nur dann Objekte von ihnen generiert werden, wenn sie zum ersten Mal auch tatsächlich aufgerufen, also benötigt, werden. Dies kann einen unnötig hohen Arbeitsspeicherverbrauch und auch längere Ladezeiten vermeiden, was wiederum zu einer verbesserten Benutzerfreundlichkeit beiträgt.

Codeausschnitt für die Anzeige von Rechnungsdaten

```
@ExperimentalSerializationApi
class NewInvoiceMainData : Fragment("Main")
{
```

```
  private val invoice: InvoiceModel by inject()
  private val m2controller: M2Controller by inject()

  override val root = form {
    fieldset("Invoice") {
      field("Seller") {
        textfield(invoice.seller).required()
...
```

Hier ist gut zu erkennen, wie das *InvoiceModel*, also das *ItemViewModel* der Invoice-Klasse, in das *Fragment* injiziert wird, und der Verkäufter, hier Seller, in einem Textfeld angezeigt wird, was zudem auch als Pflichtfeld gekennzeichnet wurde.

Um die Benutzeroberfläche auch bei langfristig laufenden Funktionen, z.B. die Analyse der gesamten Datenbank, ansprechbar zu halten, wurde teilweise auch asynchron Programmiert, sprich, es wurden *Threads* und *Coroutines* verwendet.

Ausführen einer langfristig laufenden asynchronen Funktion in einem *Thread*:

```
action {
  maxEntries = m2controller.db.getLastUniqueID("M2")
  runAsync {
    val cityDist = m2controller.getChartDataOnCityDistribution(indexManager, numberOfCities)
    {
      progressN = it.first
      updateProgress(it.first.toDouble(), maxEntries.toDouble())
      updateMessage("${it.second} (${it.first} / $maxEntries)")
    }
    ui { showPiechart(cityDist) }
  }
}
```

Der oben abgebildete Code zeigt die ausführung der Auswertung aller gespeicherten Kontakten nach deren Wohnorten. Die Funktion `getChartDataOnCityDistribution` wird hier innerhalb der geschweiften Klammern des `runAsync`-Blocks ausgeführt. Währenddessen bleibt die Oberfläche ansprechbar und es wird vermieden, dass Windows davon ausgeht, dass das Programm nicht mehr antwortet, sollte der Benutzer mit der Maus auf den Bildschirm klicken. Sobald die Funktion durchgelaufen ist, wird der Code innerhalb der geschweiften Klammern des *ui*-Blocks ausgeführt, welcher dann das Ergebnis in einem Piechart anzeigt.

Ausführen einer Coroutine zum Schreiben von Indexdaten:

```
runBlocking { launch { indexManager.writeIndexData() } }
```

```
override suspend fun writeIndexData()
{
...
```

Hier wird eine andere Form von *Threads* verwendet: *Coroutines*. Diese sind *suspendable* (wie bei der Funktion `writeIndexData()` zu sehen ist) , was bedeutet, dass die Ausführung gestoppt und auch

wieder aufgenommen werden kann, was die CPU-Last unter Umständen besser über das Programm verteilen lässt.

3.4. Implementierung der Logik

Die Logik der Software befindet sich nach dem MVC-Musters in sogenannten *Controllern*, die sich in der Ordnerstruktur dieses Projektes in den Logic-Verzeichnissen befinden. Da der Großteil des Projektes aus der Benutzeroberfläche und den Datenstrukturen besteht, kann die Logik in diesem recht kompakten Abschnitt erklärt werden.

Ausschnitt aus dem Controller des Kontakte-Moduls:

```
@ExperimentalSerializationApi
class M2Controller : IModule, Controller()
{
...
```

Alle Controller der Software sind aufgrund der Verwendung von *TornadoFX* als UI-Frameworks mit `Controller()` gekennzeichnet. Über das Schlüsselwort `Controller()` kann diese Klasse nun auch per *lazy loading* in anderen Klassen verwendet werden. In dieser Art von Klassen befinden sich die Hauptfunktionalitäten, die über die Benutzeroberfläche ausgeführt werden, wie z.B. das Öffnen eines Datensatzes, oder das Ausführen einer Auswertung in einer Auswertungsklasse nach. Auch stellen die Controllerklassen nützliche Funktionalitäten für andere Module bereit, wie das Laden des Namen eines Kontaktes über die UID (unique Identifier, sprich eine in jedem Modul einzigartige Nummer mit der Datensätze zugeordnet werden können).

```
fun getContactName(uID: Int, default: String): String
{
   return if (uID != -1)
   {
      val contact = M2DBManager().getEntry(
         uID, db, indexManager.indexList[0]!!
      ) as Contact
      contact.name
   } else default
}
```

Aufgezeigt ist hier eine Funktion, die genau dies tut; den Namen eines Kontaktes über die Referenz per UID laden und anzeigen. Zusätzlich ist hier ein Fehlerschutz eingebaut, der verhindert, dass ein leeres Objekt zurückgegeben wird: über den Parameter `default: String` wird der bereits im Datensatz eingetragene Name zurückgegeben, falls der Kontakt z.B. nicht mehr existiert, oder keine Referenz besteht, da der Name händisch eingetragen wurde.

4. FAZIT

4.1. SOLL-/IST-VERGLEICH

Projektphase	Soll-Zeit	Ist-Zeit	Differenz
Analysephase	2 h	1 h	-1 h
Entwurfsphase	3 h	2 h	-1 h
Implementierungsphase	11 h	13 h	+2 h
Erstellen der Dokumentation	4 h	4 h	0 h
Gesamt	20 h	20 h	0 h

Das Projekt wurde mit einer Gesamtzeit von 20 Stunden geschätzt, die diesem Projekt auch vorgegeben wurden. Die Verteilung des Zeitaufwandes ist der urpsrünglichen Abschätzung abgewichen, da der Programmieraufwand etwas höher war als zuerst angedacht. Begründen lässt sich dies auf die Fehlerbehebungen und dem erhöhtem Schreibaufwand durch die extreme Anzahl an Variablen der Datenklassen, vor allem der Liederklasse. Der Mehraufwand konnte jedoch durch die verkürzten Phasen Analyse und Entwurf ausgeglichen werden, da der Entwurf der Software relativ schnell geklärt war. Die Dokumentation konnte im eingeschätzten Zeitramen fertiggestellt werden.

4.2. LERNERFOLG

Durch dieses Projekt wurden Fragen geklärt, die sich über mehrere Themen wie Datenbanken, Verschlüsselung, asynchrones Programmieren, Benutzeroberfläche und Workflow erstreckten, da es sehr vielfältig ist und einige Teilbereiche der Informatik anschneidet. Auch war dies eine gute Chance für das Umsetzen eines großen Projektes in limitierter Zeit. Der Lernerfolg ist definitiv nach Abschluss dieses Projektes gegeben.

4.3. AUSBLICK

Wie die meisten ERP-Systeme wird dieses Projekt im klassischen Sinne nie wirklich fertig, da es immer Erweiterungsbedarf gibt. Genau aus diesem Grund wurden alle Hauptbestandteile dieser Software nach dem MVC-Muster inklusive Interfaces aufgebaut, um dies zu begünstigen. Da das Programm im Moment nur lokal läuft wäre eine mögliche Erweiterung die Netzwerkfähigkeit, damit die Datenbank auf einem Server laufen kann und Abfragen per Schnittstelle möglich werden. Ebenso könnten weitere Module hinzugefügt werden, also das generelle Ausbauen des ERP-Systems.
Da es sich um eine Warenwirtschaftssoftware für die Musikindustrie handelt, wären auch Schnittstellen zu Musikstreamingdienste denkbar, um Livedaten von diesen zu beziehen, was eine neue Welt der Analysemöglichkeiten öffnet. Für die Zukunft könnte auch definitiv das Einbeziehen von künstlicher Intelligenz angezielt werden, um z.B. Benutzereingaben vorauszusehen oder komplexe Auswertungen zu ermöglichen, die den möglichen Verlauf von Prozessen einzuschätzen.