FH - Studiengang für Informationstechnik und System-Management Salzburg

ITS

Übungen in Spezielle Softwaretechnologien

Protokoll

Gegenstand der Übung gemäß Anleitung:

Softwarekomponenten – Dynamic Link Libraries

Version: 1

Datum der Übung: 06.10.2016

Datum der Abgabe: 20.10.2016

Autoren: Christopher Wieland, Martin Wieser, Stephanie Kaschnitz

Unterschrift des Autors / der Autorin:

Historie

Änderung	Datum:	Autor:	Version:
Kunden&Konten Komponente: Kundenerstellung/Änderung/Löschung, Konto anlegen/löschen wurde hinzugefügt, Funktionen wurden separiert.	09.10.16	CW	0.1
Kunden&Konten Komponente: Kontonummer, Kontostand wurden hinzugefügt, Namenskonventionen wurden geändert, Logger wurde implementiert, Textausgabe wurde durch Logger ersetzt, Funktionen wurden auf Parameterübergabe umgeschrieben, Sicherungen und Typspezifizierer wurden hinzugefügt	10.10.16	CW	0.2
Kunden&Konten Komponente: Sicherungen wurden ergänzt, Funktion Kundendatenabfrage wurde hinzugefügt	11.10.16	CW	0.3
Kontofunktionen Komponente: Funktionen Überweisung, Abheben, Einzahlen wurden hinzugefügt.	11.10.16	SK	0.4
Datenspeicherungs Komponente: externe Library für Datenspeicherung via JSON wurde hinzugefügt.	13.10.16	MW	0.5
Hauptprogramm: Komponenten wurden zusammengefügt zu einem Bank.cpp	13.10.16	CW	0.5
Kontofunktionen Komponente + Hauptprogramm: Funktion Kontoauszug wurde hinzugefügt und Pfade im Programm wurden relativiert	14.10.16	SK	0.6
Datenspeicherungs Komponente: Funktionen für Customer wurden hinzugefügt	15.10.16	MW	0.7
Kunden&Konten Komponente + Hauptprogramm: Programm wurde umstrukturiert, Funktionsbeschreibungen wurden hinzugefügt und Kundendatenänderung, Kundenerstellung und Kunden löschen Funktion wurde ergänzt mit persistenter Datenspeicherung.	15.10.16	CW	0.8
Datenspeicherungs Komponente: Funktionen für Kundenabspeicherung wurde korrigert, Funktion zur Abspeicherung von Sparkonten wurde hinzugefügt.	16.10.16	MW	0.8.1
Kunden&Konten Komponente + Hauptprogramm: Implementierung von überarbeiteten Code, Funktionsbeschreibungen wurden hinzugefügt.	16.10.16	CW	0.9

Kontofunktionen Komponente: Währungsmodul, welches die Umrechnung und das Kursverhalten beinhaltet, wurde hinzugefügt. Fehlerbehebung in der Überweisungsfunktion.	16.10.16	SK	0.9.1
Datenspeicherungs Komponente: Die Funktionen wurden überarbeitet und das persistente Speichern von Konten und Kunden funktioniert nun.	18.10.16	MW	0.9.2
Kunden&Konten Komponente + Hauptprogramm: Die Funktionen NeuesSparKonto2-4 und NeuesKreditKonto2-4 wurden entfernt. Überarbeitung der Programmstruktur und Fehlerausbesserung an einzelnen Funktionen. Sämtliche Konsolenausgaben wurden entfernt. Die Funktionen doSparen() und Sparnachweis() wurden erstellt.	18.10.16	CW	1.0

Inhaltsverzeichnis

Tabellenverzeichnis

1 Aufgabenstellung

Die Aufgabenstellung bestand darin, grundlegende Funktionen einer Bank mithilfe von nachladbaren Dynamic Link Libraries (DLL) in C/C++ zu entwickeln.
Die Anzahl der Komponenten sowie die Umsetzung ist selbst zu gestalten.
Um die Interoperabilität mit verschiedenen Programmiersprachen sicherzustellen, sind die Komponenten in C zu entwerfen.

D.h. alle Funktionen der DLLs müssen als C-Funktionen aus der DLL exportiert werden, um sie aus beliebigen Programmiersprachen aufrufen zu können.

Die folgenden Basisfunktionalitäten sollten beinhaltet sein:

- Kunden anlegen, löschen, umbenennen, ändern (Adressdaten, ...)
- Konten für Spar- und Kreditgeschäfte anlegen, schließen, verwalten
    - Ein Kunde kann mehrere Konten haben
    - Ein Konto kann mehrere Kontoverfüger haben
- Überweisungen von einem Konto zum anderen, Abhebung/Einzahlung, Kontoauszüge, Kontoabschlüsse
- Währungsmodul (Umrechnung, Kursverwaltung, etc.)
- Persistente Datenhaltung
- Sonstige Basis-/Hilfsfunktionen nach eigenem Ermessen

2 Klassenübersicht

Für die Umsetzung der Aufgabenstellung wurden die Aufgaben in drei große Bereiche gespalten: Kunden & Konten, Kontofunktionalitäten und Persistenz. Jeder der Übungsteilnehmer hat sich einem Thema gewidmet.

2.1 Kunden & Konten

Der Teil der Kunden&Konten beinhaltet die Klassen des Customers, des Sparkontos sowie des Kreditkontos. Mithilfe dieser Klassen ist das Erstellen von Konten sowie von Kunden gewährleistet. Weiters kann auf die Klassen für andere Funktionen zugegriffen werden wie beispielsweise „Kontofunktionalitäten“ in Kapitel 2.2.

2.1.1 Klasse „Customer“

Die Klasse Customer wird für die Erstellung von neuen Kunden benötigt. Mithilfe dieser Klasse kann einem Konto ein Verfüger zugewiesen werden.

2.1.2 Klasse „Sparkonto“

Mit der Klasse Sparkonto kann ein neues Sparkonto erstellt werden. Auf das Sparkonto kann nur eingezahlt werden. Abhebungen sind nicht möglich.

2.1.3 Klasse „Kreditkonto“

Mit der Klasse Kreditkonto kann ein neues Kreditkonto erstellt werden. Auf das Kreditkonto kann eingezahlt, abgehoben und überwiesen werden.

2.2 Kontofunktionalitäten

Die Kontofunktionalitäten Komponente übernimmt die Anwendungen, welche ein Benutzer auf sein Konto ausführen kann. Sie beinhaltet insgesamt zwei Klassen: Ueberweisung und Waehrungsmodul. Mithilfe dieser Klassen und deren Funktionen wird dem Benutzer die Verwaltung des Kontos sichergestellt.

2.2.1 Klasse „Ueberweisung“

Bei Verwendung dieser Klasse wird eine neue Überweisung vom Quellkonto, welches dem Konto des Benutzers, der eine Transaktion betätigen will, entspricht, auf ein Zielkonto betätigt. Diese Klasse benötigt Informationen über das Konto des Benutzers sowie eines Betrages und eines Verwendungszweckes. Nach jeder Überweisung werden Daten bezüglich des Datums, Verwendungszweckes und Betrags in ein Kontoauszug-File im Projektordner gesichert.

2.2.2 Klasse „Waehrungsmodul“

Diese Klasse dient der Umrechnung des Kontostands in vier Währungen (USD, CHF, GBP, JPY) sowie der Kursverwaltung. Bei Verwendung dieser Klasse werden Informationen über das Konto des Benutzers benötigt. Die Ausgabe der Umrechnung und der Kursverwaltung findet jeweils in einem eigenen Text-File, welches im Projektordner gespeichert wird, statt.

2.3 Persistente Datenspeicherung

Das Modul „Persistente Datenspeicherung“ hat die Aufgabe die Kunden und die Konten zu Persistieren. Für dieses Beispiel werden die Kunden- und Kontendaten im JSON Format abgespeichert und diese können dann einzeln ausgelesen, verarbeitet und wieder festgeschrieben werden. Mit weiteren Funktionen kann ein Konto oder ein Kunde hinzugefügt oder entfernt werden. Für das lesen und schreiben der Daten im JSON Format wurde die externe Bibliothek:“cJSON“ eingebunden, so ist es möglich die einzelnen JSON-Objects zu Parsen und damit zu arbeiten.

3 Dokumentation der Funktionalität der DLL

Dieses Kapitel umfasst die Funktionalitäten der gesamten Dynamic Link Library. Diese sind gegliedert in drei Gruppen (Kunden&Konten, Kontofunktionalitäten, persistente Datenspeicherung). Weiters beinhaltet die Dokumentation zu jedem Beriech die internen Funktionen, die Funktionen der Schnittstelle, die Hilfsfunktionen und die Aufrufs Beispiele.

3.1 Kunden & Konten

Die Komponente besteht aus insgesamt drei Klassen („Customer“, siehe Kapitel 2.1.1, „Sparkonto“, siehe Kapitel 2.1.2 und „Kreditkonto“, siehe Kapitel 2.1.3) und deren Methoden. In den nachfolgenden Kapiteln wird genauer auf die jeweiligen Funktionen der Schnittstellen sowie Hilfsfunktionen und Aufrufbeispiele eingegangen.

3.1.1 Funktionen der Schnittstelle

Alle Funktionen der Schnittstelle müssen als C-Funktionen aus der DLL exportiert werden, um sie aus beliebigen Programmiersprachen aufrufen zu können. Zu diesen Funktionen zählen:

- `CUSTOMER* NeuerKunde(char* _Vorname, char* _Nachname, char* _Geburtsdatum, char* _adresse, char* _Wohnort, char* _Telefon);`

  Diese Funktion wird verwendet um eine neue Instanz der Klasse „Customer“ zu erzeugen. Informationen über den Vornamen, den Nachnamen, das Geburtsdatum, die Adresse, den Wohnort und der Telefonnummer werden benötigt. Mit dieser Funktion wird eine neue Instanz erzeugt mit den übergebenen Parametern. Weiters wird mit der Funktion addUser() die erstellte Instanz in ein JSON File kopiert für eine persistente Datensicherung. Die Funktion übergibt zum Schluss eine Instanz des neuen Objektes.

- `void Kundenvornamenänderung(CUSTOMER *Kunde, char* Vorname);`

  Bei Kundenvornamenänderung wird einem vorhandenen Kunden der Vorname geändert. Dazu wird der Parameter _Vorname mitübergeben. In dieser Funktion wird außerdem die Funktion writeUser() ausgeführt, die den bestehenden Kunden mit den neuen Informationen überschreibt.

- `void Kundennachnamenänderung(CUSTOMER *Kunde, char* _Nachname);`

  Mit Kundennachnamenänderung wird einem vorhandenen Kunden der Nachname geändert. Dazu wird der Parameter _Nachname mitübergeben. In dieser Funktion wird außerdem die Funktion writeUser() ausgeführt, die den bestehenden Kunden mit den neuen Informationen überschreibt.

- `void Kundenadressänderung(CUSTOMER *Kunde, char* _Adresse);`

  Mit der Funktion Kundenadressänderung wird einem vorhandenen Kunden die Adresse geändert. Dazu wird der Parameter _Adresse mitübergeben. In dieser Funktion wird außerdem die Funktion writeUser() ausgeführt, die den bestehenden Kunden mit den neuen Informationen überschreibt.

- `void Kundenwohnortsänderung(CUSTOMER *Kunde, char* _Wohnort);`

  Bei Kundenwohnortsänderung wird einem vorhandenen Kunden der Wohnort geändert. Dazu wird der Parameter _Wohnort mitübergeben. In dieser Funktion wird außerdem die Funktion writeUser() ausgeführt, die den bestehenden Kunden mit den neuen Informationen überschreibt.

- `void Kundentelefonänderung(CUSTOMER *Kunde, char* _Telefon);`

  Mit dieser Funktion wird einem vorhandenen Kunden die Telefonnummer geändert. Dazu wird der Parameter _Telefon mitübergeben. In dieser Funktion wird außerdem die Funktion writeUser() ausgeführt, die den bestehenden Kunden mit den neuen Informationen überschreibt.

- `void Kundendatenabfrage(CUSTOMER * Kunde);`

  Mit dieser Funktion werden alle Kundendaten des übergebenen Kontos an den Logger ausgeschrieben.

- `void Kundeentfernen(CUSTOMER* Kunde);`

  Bei Kundeentfernen wird ein bestehender Kunde gelöscht. Zusätzlich wird mit der Funktion removeUser() der übergebene Kunde auch aus dem Datensicherungsfile gelöscht.

- `SPARKONTO* NeuesSparkonto(CUSTOMER* Kunde);`

  Diese Funktion wird verwendet um eine neue Instanz der Klasse „Sparkonto“ zu erzeugen. Ein Kunde muss als Kontoverfüger übergeben werden. Mit dieser Funktion wird eine neue Instanz erzeugt mit dem Kunden als Kontoverfüger mithilfe

der Setter Funktion setVerfüger(). Die Funktion addKontoverfüger() wurde zusätzlich ausgeführt, um den übergebenen Verfüger dem Konto zuzuweisen. Weiters wird mit der Funktion writeUser() die Kontonummer des neu erstellten Sparkontos in den übergebenen Kunden eingetragen, um später vom Kunden auf das Konto zugreifen zu können.

- `KREDITKONTO* NeuesKreditkonto(CUSTOMER* Kunde);`

  Diese Funktion wird verwendet um eine neue Instanz der Klasse „Kreditkonto“ zu erzeugen. Ein Kunde muss als Kontoverfüger übergeben werden. Mit dieser Funktion wird eine neue Instanz erzeugt mit dem Kunden als Kontoverfüger mithilfe der Setter Funktion setVerfüger(). Die Funktion addKontoverfüger() wurde zusätzlich ausgeführt, um den übergebenen Verfüger dem Konto zuzuweisen. Weiters wird mit der Funktion writeUser() die Kontonummer des neu erstellten Sparkontos in den übergebenen Kunden eingetragen, um später vom Kunden auf das Konto zugreifen zu können.

- `int addSparKontoverfüger(SPARKONTO* sk, CUSTOMER* cust);`

  Bei addSparKontoverfüger wird die ID des übergebenen Customers ausgelesen und die id wir in die Funktion addKontoverfüger übergeben. Weiters wird writeSparKonto() das übergeben Sparkonto aktualisiert.

- `int addKreditKontoverfüger(SPARKONTO* sk, CUSTOMER* cust);`

  Bei addKreditKontoverfüger wird die ID des übergebenen Customers ausgelesen und die id wir in die Funktion addKontoverfüger übergeben. Weiters wird writeKreditKonto() das übergeben Kreditkonto aktualisiert.

- `void Sparkontoentfernen(SPARKONTO* Konto, CUSTOMER* Verfüger);`

  Bei Sparkontoentfernen wird ein bestehendes Konto gelöscht. Mit der Funktion getClassId() wird überprüft, ob es sich bei dem übergebenen Konto um ein Sparkonto handelt. Trifft das zu, so wird das Konto von dem übergebenen Verfüger entfernt und das Konto wird mit delete entfernt. Weiters wird mithilfe von removeSparKonto() das Konto aus dem Datensicherungsfile gelöscht.

- `void Kreditkontoentfernen(KREDITKONTO* Konto, CUSTOMER* Verfüger);`

  Bei Kreditkontoentfernen wird ein bestehendes Konto gelöscht. Mit der Funktion getClassId() wird überprüft, ob es sich bei dem übergebenen Konto um ein Kreditkonto handelt. Trifft das zu, so wird das Konto von dem übergebenen

Verfüger entfernt und das Konto wird mit delete entfernt. Weiters wird mithilfe von removeKreditKonto() das Konto aus dem Datensicherungsfile gelöscht.

3.1.2 Funktionen der Klasse „Customer“

Die Funktionen der Klasse bestehen aus sogenannte „Setter“ und „Getter“. Diese dienen als Zugriffsfunktionen, um Daten zu ändern oder abzufragen. Zu diesen Funktionen zählen:

Funktion	Erklärung
Char* getVorname()	Abruf des Vornamens
Char* getNachname()	Abruf des Nachnamens
Char* getGeburtsdatum()	Abruf des Geburtsdatums
Char* getWohnort()	Abruf des Wohnortes
Char* getAdresse()	Abruf der Adresse
Char* getTelefon()	Abruf der Telefonnummer
int getID()	Abruf der ID
int getKtnr1()	Abruf der ersten Kontonummer
int getKtnr2()	Abruf der zweiten Kontonummer
int getKtnr3()	Abruf der dritten Kontonummer
int getKtnr4()	Abruf der vierten Kontonummer
Void setVorname(char*)	Setzt den Vornamen
Void setNachname(char*)	Setzt den Nachnamen
Void setGeburtsdatum(char*)	Setzt das Geburtsdatum
Void setWohnort(char*)	Setzt den Wohnort
Void setAdresse(char*)	Setzt die Adresse
Void setTelefon(char*)	Setzt die Telefonnummer
Void setID(int)	Setzt die ID

Void setKtnr1(int)	Setzt die erste Kontonummer
Void setKtnr2(int)	Setzt die zweite Kontonummer
Void setKtnr3(int)	Setzt die dritte Kontonummer
Void setKtnr4(int).	Setzt die vierte Kontonummer

Tabelle 3.1: Funktionen der Klasse „Customer“

Bei den in der Tabelle 3.1 beschriebenen Funktionen handelt es sich um public Funktionen. Diese werden vor allem von den Hilfsfunktionen (Kapitel 3.1.5) benötigt.

Weiters ist eine Funktion genannt getClassId () implementiert. Diese Funktion dient zur Überprüfung des Objektes im Programm und gibt in diesem Fall den char* „customer“ zurück.

3.1.3 Funktionen der Klasse „Sparkonto“

Die Funktionen der Klasse bestehen aus sogenannte „Setter“ und „Getter“. Diese dienen als Zugriffsfunktionen, um Daten zu ändern oder abzufragen. Zu diesen Funktionen zählen:

Funktion	**Erklärung**
double getKontostand()	Abruf des Kontostandes
CUSTOMER getVerfüger()	Abruf des Kontoverfügers
int getKontonummer()	Abruf der Kontonummer
int getKontoverfüger()	Gibt die ID des Verfügers zurück
Void setKontoverfüger(int)	Setzt die ID des übergebenen Verfügers
Void setKontostand(double)	Setzt den Kontostand
Void setVerfüger(CUSTOMER*)	Setzt den Verfüger
Void setKontonummer(int)	Setzt die Kontonummer

Tabelle 3.2: Funktionen der Klasse „Sparkonto“

Bei den in der Tabelle 3.2 beschriebenen Funktionen handelt es sich um public Funktionen. Diese werden vor allem von den Hilfsfunktionen (Kapitel 3.1.6) benötigt.

Weiters ist eine Funktion genannt getClassId() implementiert. Diese Funktion dient zur Überprüfung des Objektes im Programm und gibt in diesem Fall den char* „sparkonto“ zurück. Eine weitere Funktion dieser Klasse ist die addKontoverfüger() Funktion. Diese gibt die übergebene id in die benötigte Kontoverfügervariable.

3.1.4 Funktionen der Klasse „Kreditkonto“

Die Funktionen der Klasse bestehen aus sogenannte „Setter“ und „Getter“. Diese dienen als Zugriffsfunktionen, um Daten zu ändern oder abzufragen. Zu diesen Funktionen zählen:

Funktion	Erklärung
double getKontostand()	Abruf des Kontostandes
CUSTOMER getVerfüger()	Abruf des Kontoverfügers
int getKontonummer()	Abruf der Kontonummer
int getKontoverfüger()	Gibt die ID des Verfügers zurück
Void setKontoverfüger(int)	Setzt die ID des übergebenen Verfügers
Void setKontostand(double)	Setzt den Kontostand
Void setVerfüger(CUSTOMER*)	Setzt den Verfüger
Void setKontonummer(int)	Setzt die Kontonummer

Tabelle 3.3: Funktionen der Klasse „Kreditkonto“

Bei den in der Tabelle 3.3 beschriebenen Funktionen handelt es sich um public Funktionen. Diese werden vor allem von den Hilfsfunktionen (Kapitel 3.1.7) benötigt.

Weiters ist eine Funktion genannt getClassId() implementiert. Diese Funktion dient zur Überprüfung des Objektes im Programm und gibt in diesem Fall den char* „kreditkonto“ zurück. Eine weitere Funktion dieser Klasse ist die addKontoverfüger() Funktion. Diese gibt die übergebene id in die benötigte Kontoverfügervariable.

3.1.5 Aufrufbeispiele

- CUSTOMER* Kunde1 = NeuerKunde(„Vorname“,“Nachname“,“Geburtsdatum“,“Adresse“,“Wohnort“,“Telefonnummer“);
- Kundennachnamenänderung (Kunde1, „Nachname2“);

- SPARKONTO* Sparmeister = NeuesSparkonto(Kunde1);
- Sparkontoentfernen(Sparmeister, Kunde1);

Selbiges, was für das Sparkonto gilt, gilt beim Aufruf auch für das Kreditkonto.

3.2 Kontofunktionalitäten

Die Kontofunktionalitäten bestehen aus insgesamt zwei Klassen (Ueberweisung, siehe Kapitel 2.2.1, und Waehrungsmodul, siehe Kapitel 2.2.2) und deren Methoden. In den nachfolgenden Kapiteln wird genauer auf die jeweiligen Funktionen der Schnittstellen sowie Hilfsfunktionen und Aufrufbeispiele eingegangen.

3.2.1 Funktionen der Schnittstelle

Alle Funktionen der Schnittstelle müssen als C-Funktionen aus der DLL exportiert werden, um sie aus beliebigen Programmiersprachen aufrufen zu können. Zu diesen Funktionen zählen:

- `UEBERWEISUNG* NeueUeberweisung(KREDITKONTO* quellkonto, KREDITKONTO* zielkonto, double betrag, char* verwendungszweck);`

  Diese Funktion wird verwendet um eine neue Instanz der Klasse „Ueberweisung“ zu erzeugen. Informationen über das Quell-, sowie Zielkonto der Klasse „Kreditkonto“ werden benötigt. Weiters muss der Funktion der Betrag und Verwendungszweck der Überweisung als Parameter mitgegeben werden. In dieser

Funktion wird eine neue Instanz erstellt. Weiters wird dem Quellkonto der Betrag mit Hilfe der „setKontostand“-Funktion abgezogen und dem Zielkonto mittels der Funktion „doEinzahlen“ dazugezählt. Zuletzt wird die „doAbheben“-Funktion ausgeführt, um die persistente Sicherung der Daten und ein Eintrag in den Kontoauszug zu gewährleisten. Als Rückgabewert erhält man einen Verweis auf das Objekt „Ueberweisung“.

- `void doAbheben(KREDITKONTO* zielkonto, double betrag);`

  Die „doAbheben“-Funktion dient dazu um Geld von einem Konto abzuheben. Sie benötigt ein Zielkonto der Klasse „Kreditkonto“, von welchem der Betrag abgebucht wird. Ein weiterer Parameter stellt der Betrag dar. Der Kontostand des mitgegebenen Kontos wird mit der „getKontostand“-Funktion abgerufen. Die Funktion setzt den aktuellen Kontostand des Zielkontos mittels „setKontostand“ neu und führt die Funktion „Buchen“ aus. Mit Hilfe der „writeKreditKonto“-Funktion werden die geänderten Daten des Kontos gespeichert.

- `void doEinzahlen(KREDITKONTO* zielkonto, char* verwendungszweck, double betrag);`

  Diese Funktion beschreibt das Einzahlen auf ein Konto. Hierfür wird wieder das Zielkonto der Klasse „Kreditkonto“ benötigt. Weitere Parameter stellen der Betrag und der Verwendungszweck dar. Mithilfe der „getKontostand“ wird der aktuelle Kontostand abgerufen, der im Parameter eingegebene Betrag dazugezählt und mit „setKontostand“ erneut gesetzt. Zu letzt wird die „Buchen“ Funktion aufgerufen. Mit Hilfe der „writeKreditKonto“-Funktion werden die geänderten Daten des Kontos gespeichert.

- `void doSparen(SPARKONTO* zielkonto, char* verwendungszweck, double betrag);`

  Diese Funktion beschreibt das Einzahlen auf ein Konto. Hierfür wird wieder das Zielkonto der Klasse „Sparkonto“ benötigt. Weitere Parameter stellen der Betrag und der Verwendungszweck dar. Mithilfe der „getKontostand“ wird der aktuelle Kontostand abgerufen, der im Parameter eingegebene Betrag dazugezählt und mit „setKontostand“ erneut gesetzt. Zuletzt wird die „Buchen“ Funktion aufgerufen. Mit Hilfe der „writeSparKonto“-Funktion werden die geänderten Daten des Kontos gespeichert.

- `WAEHRUNGSMODUL* NeuesWaehrungsmodul(KREDITKONTO* konto);`

Die „NeuesWaehrungsmodul“-Funktion wird dazu verwendet um eine neue Instanz der Klasse „Waehrung“ zu erzeugen. Als Parameter wird ein Konto der Klasse „Kreditkonto“ mitgegeben, mit welchen Daten eine neue Instanz erstellt wird. Der Rückgabewert ist ein Verweis auf die soeben erstellte Instanz.

- `void doUmrechnung(WAEHRUNGSMODUL* waehrungsmmodul, char* waehrung);`

  Die „doUmrechnung“-Funktion wird benötigt um den aktuellen Kontostand in eine der vier Währungen umzurechnen. Um Umzurechnen wird die Funktion „umrechnung“ aufgerufen. Als Parameter wird ein Verweis auf das Währungsmodul mitgegeben. In welchem sich Informationen über das Konto befindet.

- `void doKursverwaltung(WAEHRUNGSMODUL* waehrungsmodul);`

  Diese Funktion dient der Kursverwaltung, indem die Funktion „kursverwaltung“ aufgerufen wird. Als Parameter wird ein Verweis auf das Währungsmodul mitgegeben, um auf die Daten des Kontos zugreifen zu können.

3.2.2 Funktionen der Klasse „Ueberweisung“

Die Funktionen der Klasse bestehen aus sogenannte „Setter“ und „Getter“. Diese dienen als Zugriffsfunktionen, um Daten zu ändern oder abzufragen. Zu diesen Funktionen zählen:

Funktion	Erklärung
Char* getempfaengername()	Abruf des Empfängernamens
Char* getVerwendungszweck()	Abruf des Verwendungszwecks
Double getBetrag()	Abruf des Betrags
Double getKontostand()	Abruf des Kontostandes
Int getKontonummer()	Abruf der Kontonummer
Void setempfaengername(char*)	Setzt den Empfängernamen
Void setVerwendungszweck(char*)	Setzt den Verwendungszweck
Void setBetrag(double)	Setzt den Betrag
Void setKontostand(double)	Setzt den Kontostand
Void setKontonummer(int)	Setzt die Kontonummer

Tabelle 3.4: Funktionen der Klasse „Ueberweisung“

Bei den in der Tabelle 3.4 beschriebenen Funktionen handelt es sich um „public“ Funktionen. Diese werden vor allem von den internen Funktionen (Kapitel 3.2.4) benötigt.

3.2.3 Funktionen der Klasse „Waehrungsmodul“

Die Funktionen dieser Klasse bestehen ebenfalls aus public „Setter“ und „Getter“. Zu diesen Funktionen zählen:

Funktion	Erklärung
Double getkontostand()	Abruf des Kontostands
Int getKontonummer()	Abruf der Kontonummer
Char* getWaehrung	Abruf der Währung
Void setkontostand(double)	Setzt den Kontostand
Void setKontonummer(int)	Setzt die Kontonummer
Void setWaehrung(char*)	Setzt die Währung

Tabelle 3.5: Funktionen der Klasse "Waehrungsmodul"

3.2.4 Interne Funktionen der Klasse „Ueberweisung“

Die internen Funktionen der Klasse werden benötigt um einen Ablauf der Buchungen weitgehend in der Programmiersprache „C++“ zu gewährleisten. Diese Funktionen werden von den Schnittstellen-Funktionen aufgerufen.

- `void Buchen(KREDITKONTO* zielkonto, char* verwendungszweck, double betrag, int art)`

  Die Funktion „Buchen“ dient der Aufbereitung des Kontoauszugs. Hierfür wird ein Zielkonto der Klasse „Kreditkonto“, Verwendungszweck, Betrag und Art als Parameter übergeben. Die Art gibt wieder welche Aktion ausgeführt werden soll. Wenn Art gleich 1 ist, handelt es sich um eine Überweisung, wenn sie 2 ist um eine Abhebung und wenn sie 3 ist handelt es sich um eine Einzahlung. Um den Kontoauszug übersichtlich zu gestalten wird der Betrag in einen String

umgewandelt und je nach Art ein „+“ oder „-„vorangestellt. Weiters wird die Funktion „BUCHUNGEN“ aufgerufen.

- `void Sparnachweis(SPARKONTO* zielkonto, char* verwendungszweck, double betrag, int art)`

  Die Funktion „Sparnachweis“ dient der Aufbereitung des Kontoauszugs. Hierfür wird ein Zielkonto der Klasse „Sparkonto“, Verwendungszweck, Betrag und Art als Parameter übergeben. Die Art gibt wieder welche Aktion ausgeführt werden soll. Wenn Art gleich „3“ ist, handelt es sich um eine Einzahlung. Um den Kontoauszug übersichtlich zu gestalten, wird der Betrag in einen String umgewandelt und ein „+“ vorangestellt. Weiters wird die Funktion „BUCHUNGEN“ aufgerufen.

- `void BUCHUNGEN(char* verwendungszweck, char* betrag, string kontonummer)`

  Diese Funktion erzeugt den Namen des Text-Files, welches für den Kontoauszug verwendet wird. Hierbei dient der Parameter „kontonummer“ mit der Endung „_Buchungen.txt“ als Name dieser Datei. In dieser Funktion wird mit Hilfe der Hilfsfunktion „fileExist“ (siehe Kapitel 3.2.6) überprüft ob ein Dokument bereits vorhanden ist, wenn nicht wird ein Neues mit Hilfe „initializeBuchungen“ initialisiert und mit der Funktion „insertBuchungToFile“ Daten hinzugefügt.

- `void initializeBuchungen(int kontonummer, string textFileName)`

  „initializeBuchungen“ wird verwendet um bei nicht vorhandenen Text-Files ein neues File mit spezieller Kopfzeile zu erstellen. Der „textFileName“ Parameter wird verwendet um ein neues File mit dessen Namen zu erstellen. Die Kontonummer wird im Kopf dieses Dokumentes hinzugefügt.

- `void insertBuchungToFile(string textFileName, char* verwendungszweck, char* betrag)`

  Diese Funktion wird verwendet um die Daten in das Text-File zu schreiben. Hinzu kommt neben den Parametern auch das Datum. Der „textFileName“ wird benutzt um das richtige File zu öffnen.

3.2.5 Interne Funktionen der Klasse „Waehrungsmodul“

- `void umrechnung(WAEHRUNGSMODUL* waehrungsmodul, char* waehrung)`

  In dieser Funktion werden die Kontoinformationen, wie zum Beispiel der Kontostand und die Kontonummer, mit Hilfe des Parameters „waehrungsmodul“

abgelesen. Der zweite Parameter gibt wieder in welcher Währung der Kontostand umgerechnet werden sollte. Die „umrechnung“-Funktion erzeugt für jede Kontonummer ein eigenes Text-File mit der Endung „_Umrechnung.txt“. Es wird wiederum überprüft mit Hilfe der Funktion „fileExist“ überprüft ob ein solches File bereits existiert. Wenn es existiert wird es gelöscht, sodass es mit den neuen Daten gefüllt werden kann. Je nach dem Parameter „waehrung“ wird ein neues File mittels „createUmrechnungsFile“ erzeugt und beschrieben.

- `void createUmrechnungsFile(string textFileName, char* waehrung, double kontostand, double waehrungsKontostand)`

  Der Parameter „textFileName“ gibt den Namen des Text-Files wieder. Die restlichen Parameter werden in dieses File geschrieben. Bei „waehrungsKontostand“ handelt es sich um den umgerechneten Kontostand.

- `void kursverwaltung(WAEHRUNGSMODUL* waehrungsmodul)`

  Diese Funktion benötigt einen Verweis von der Klasse „Waehrungsmodul“ um auf die Kontodaten zugreifen zu können. Es wird erneut ein neuer Name für das Text-File erstellt mit der Endung „_Kursverwaltung.txt“. Auf dessen Existenz wird wieder mit der Funktion „fileExist“ überprüft. Wenn ein solches File bereits existiert wird es zunächst gelöscht und danach erneut mittels der Funktion „createKursverwaltungsFile“ befüllt.

- `void createKursverwaltungsFile(string textFileName)`

  Mit Hilfe dieser Funktion wird ein neues Text-File, welches nach dem Parameter „textFileName“ benannt ist, erstellt und mit den Kursen der vier Währungen (USD, CHF, GBP, JPY) beschrieben.

3.2.6 Hilfsfunktionen

- `int fileExist(string name)`

  Diese Funktion testet ob ein File mit den Namen des übergebenen Parameters bereits existiert. Wenn ein solches vorhanden ist, ist der Rückgabewert „1“, ansonst „2“.

3.2.7 Aufrufbeispiele

Aufrufbeispiel für die Kontofunktionen (Überweisung, Abheben, Einzahlen):

- UEBERWEISUNG* ueberweisung = NeueUeberweisung(giro, spare, 500, „Test Überweisung“);
- doEinzahlen(giro, „Einzahlung“, 100.50);

- doSparen(sparen, „Sparen“, 100);
- doAbheben(giro, 200);

Aufrufbeispiel für ein Währungsmodul (Umrechnen, Kursverwaltung):

- WAEHRUNGSMODUL* waehrungsmodul = NeuesWaehrungsmodul(giro);
- doUmrechnung(waehrungsmodul, „USD“);
- doKursverwaltung(waehrungsmodul);

3.3 Persistente Datenspeicherung

Die Komponente für die Persistenz übernimmt das Speichern von Kunden und Konten und es gibt Zugriff auf einzelne Einträge davon. Diese Einträge können mit read* gelesen und mit write* geschrieben werden. Zudem gibt es noch Funktionen zum Hinzufügen und Entfernen der Kunden oder Konten. Um mit der ID die Kundendaten auszulesen gibt es noch eine Suchfunktion die bei einem vorhandenen Kunden deren ID zurückgibt.

Es gibt für jede Klasse 4 Funktionen für die Persistenz:

- **read*** gibt ausgelesenes Objekt zurück
- **write*** überschreibt vorhandenes Objekt
- **add*** fügt ein Objekt hinzu
- **remove*** löscht einen Eintrag

Beispiel Funktion CUSTOMER* **read**User(int);

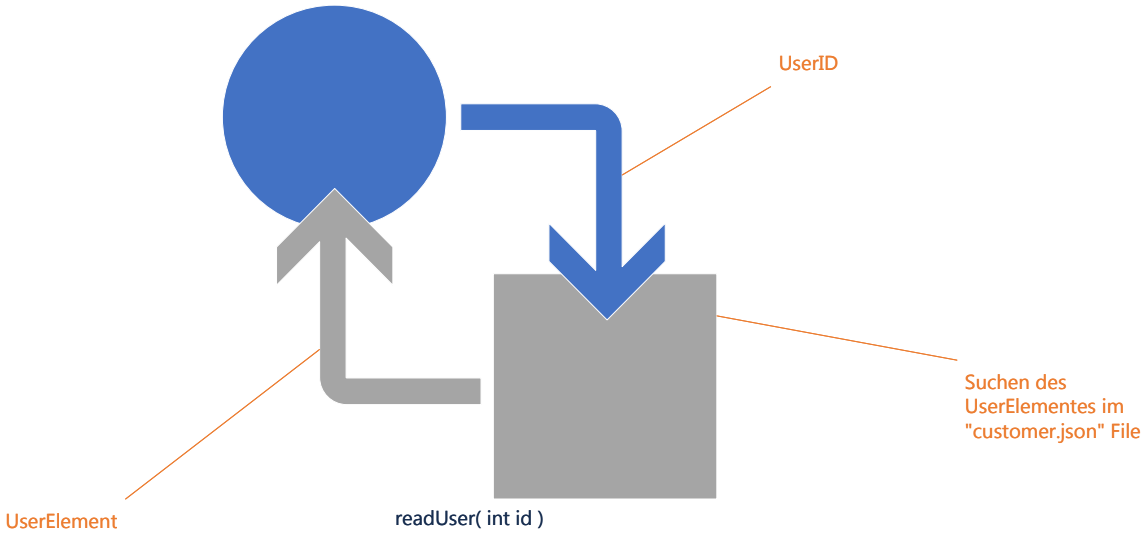

Abbildung 1 : readUser Funktion

Bei diesem Beispiel wird der Kunde nach der ID gesucht, das CUSTOMER Element aus dem „customer.json“ File herausgesucht. Schließlich wird dieses JSON-Object auf ein CUSTOMER Objekt gemappt und zurückgegeben. Nun kann mit diesem Objekt gearbeitet werden. Anschließend wird der Kunde mit **write**User(CUSTOMER*) festgeschrieben.

3.3.1 Funktionen der Schnittstelle

Alle Funktionen der Schnittstelle müssen als C-Funktionen aus der DLL exportiert werden, um sie aus beliebigen Programmiersprachen aufrufen zu können. Zu diesen Funktionen zählen:

- int searchUser(char* vorname, char* nachname, char* geb);
- SPARKONTO* readSparKonto(int ktnr);
- KREDITKONTO* readKreditKonto(int ktnr);
- CUSTOMER* readUser(int id);

Um nach außen eine Sicherheit zu gewährleisten werden hier nur die **read*** Funktionen nach außen bereitgestellt. Die **write***, **add*** **und remove*** Funktionen sind in der Komponente „Kontofunktionalitäten“ verwendet.

Die gesamten Funktionen werden in den weiteren Kapiteln kurz beschrieben.

3.3.2 Filefunktionen

Mit diesen Funktionen können die Files in cJSON* oder als char* eingelesen oder geschrieben werden.

- cJSON* readJsonFile_cJson(char *filename);
- char* readJsonFile_char(char *filename);
- bool writeJsonFile(char *filename, cJSON * jobj);
- bool writeJsonFile(char *filename, char* jobj);

3.3.3 File Hilfsfunktionen

Diese Funktionen sind zum Anlegen der benötigten JSON-Files. Sie geben TRUE zurück wenn das File angelegt wurde.

- bool createUserCountFile();
- bool createCreditNumberCountFile();
- bool createUserFile();
- bool createSparKontoFile();
- bool createKreditKontoFile();

Die folgenden 2 Funktionen sind dazu da um bei Anlegen eines neuen Kontos die nächste Kontonummer auszulesen. Dies wird im „Kontencounter.json“ File abgelegt und mit writeCount(int) erhöht.

- int readCount();
- void writeCount(int);

Die folgenden 2 Funktionen sind dazu da um bei Anlegen eines neuen Kunden die nächste ID auszulesen. Dies wird im „Usercounter.json“ File abgelegt und mit writeUserCount(int) erhöht.

- int readUserCount();
- void writeUserCount(int);

3.3.4 Mapping Funktionen

Folgende Funktionen sind zum „Mappen“ der Klassen auf JSON-Objects vice versa.

- CUSTOMER* cJSONToCustomer(cJSON* customerItem);
- cJSON* customerTocJSON(CUSTOMER* cust);
- cJSON* sparkontoTocJSON(SPARKONTO* sk);
- SPARKONTO* cJSONToSparkonto(cJSON* skItem);
- cJSON* kreditkontoTocJSON(KREDITKONTO* sk);
- KREDITKONTO* cJSONToKreditkonto(cJSON* skItem);

3.3.5 UserFunktionen

- CUSTOMER* readUser(int id); // return Customer wenn ID vorhanden
- bool writeUser(CUSTOMER* cust); // true wenn erfolgreich
- bool addUser(CUSTOMER* cu); // true wenn erfolgreich
- bool removeUser(int id); // true wenn erfolgreich
- bool userExist(int id); // true wenn user vorhanden
- bool checkUserItem(cJSON* item, char* vorname, char* nachname, char* geb);
  // suchen nach dem User (hilfsfunktion für searchUser Funktion)

- int searchUser(char* vorname, char* nachname, char* geb);
  // gibt UserID zurück wenn parameter mit ausgelesenem JSON Object matchen

3.3.6 SparKonto Funktionen

- SPARKONTO* readSparKonto(int ktnr);
  // return Sparkonto wenn Kontonummer vorhanden

- bool writeSparKonto(SPARKONTO* kt); // true wenn erfolgreich
- bool addSparKonto(SPARKONTO* kt); // true wenn erfolgreich
- bool removeSparKonto(int ktnr); // true wenn erfolgreich
- bool sparkontoExist(int ktnr); // true wenn erfolgreich
-
- int addSparKontoverfüger(SPARKONTO* sk, CUSTOMER* cust);
  // Fügt CustomerID zum Sparkonto hinzu. Funktion sollte in eine andere
  // Komponente?

3.3.7 KreditKonto Funktionen

- KREDITKONTO* readKreditKonto(int ktnr);
  // return Kreditkonto wenn Kontonummer vorhanden

- bool writeKreditKonto(KREDITKONTO* kt); // true wenn erfolgreich
- bool addKreditKonto(KREDITKONTO* kt); // true wenn erfolgreich

- bool removeKreditKonto(int ktnr); // true wenn erfolgreich
- bool kreditkontoExist(int ktnr); // true wenn erfolgreich

- int addKreditKontoverfüger(KREDITKONTO* kk, CUSTOMER* cust);
  // Fügt CustomerID zum Kreditkonto hinzu. Funktion sollte in eine andere
  // Komponente?

3.3.8 HilfsFunktionen

- bool checkItem(cJSON * item, int id);
  // Hilfsfunktion zum Suchen von Kunden mit der ID und einem JSON Object.
  // Return true wenn gefunden

- bool checkKtItem(cJSON * item, int id);
  // Hilfsfunktion zum Suchen von Konten mit der Kontonr. und einem JSON
  // Object. Return true wenn gefunden

3.4 Allgemeine Hilfsfunktionen

Neben den internen Funktionen mancher Klassen, wurden ebenso auch allgemeine Hilfsfunktionen benötigt.

- `string time_to_string();`

In dieser Funktion wird die momentan aktuelle Zeit in string ausgegeben. Dies ist für die nächste Funktion Logging interessant.

- `void LOGGING(char* Errortext, char* LEVEL);`

Mit der Funktion LOGGING ist es möglich, eine Text Datei zu erstellen, die Je nach Parameterübergabe einen ERROR oder einen OK Log in dem erzeugten File abspeichert. Im ersten Parameter wird die Meldung niedergeschrieben und LEVEL Parameter wird angegeben, ob es sich bei dem Log um einen ERROR handelt, oder ob der schritt OK war. Die Funktion time_to_string() in dieser Funktion gibt neben dem Errortxt auch die aktuelle Zeit aus, in welcher der Log entstanden ist.

4 Zusätzliche externe Komponenten

Für die Komponente „Persistente Datenhaltung“ wurde eine externe Libary „cJSON“ verwendet.

Informationen:

Libary „cJSON“: https://github.com/DaveGamble/cJSON (10.10.2016)

Informationen zu JSON: http://www.json.org/

5 Zusammenfassung und Ausblick

Die DLL konnte erfolgreich in der angegebenen Zeit programmiert werden. In der aktuellen Version ist es möglich, dass ein Kunde bis zu 5 Konten besitzen kann. Hierzu können Sparkonten sowie Kreditkonten hinzugefügt werden. Bei Sparkonten ist nur das Einzahlen erlaubt und Kreditkonten bieten die Möglichkeit zur Einzahlung, Abhebung oder auch Überweisung. Dementsprechend funktionieren auch die bereits genannten Funktionen. Ein Währungsmodul ist ebenfalls implementiert, welche die Währung in vier Währungen umrechnen kann. Alle erzeugten Objekte sowie Auszüge und Buchungen werden mithilfe von JSON Dateien persistent gespeichert.

5.1 Derzeit nicht implementiert

Ein Teilpunkt wurden jedoch nicht erfüllt: Kontoabschlüsse wurden aus Zeitgründen nicht mehr implementiert.

5.2 Ausblick

Das Projekt lässt noch vieles offen für weitere Implementationen. Beispielsweise könnten noch Kundenspezifische Zugriffe implementiert werden.