Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

DIPLOMARBEIT

Übertragen von Echtzeitdaten in eine Tabellenkalkulation durch Ansprache des Application Interface mittels C-programmierter Anwendungen.

Robin Menn
Matr.-Nr. 1828314

Abgabetermin 21.02.2012

Betreuer:	Prof. Dr.-Ing. Franz Vinnemeier
2. Betreuer:	Dipl. Ing. Andreas Theel

Kurzfassung

Der Trend zu einer immer weiter nutzerorientierten Programmierung von Anwendungen ist seit Jahren unaufhaltsam. Eine Entwicklung, die sicherlich sehr begrüßenswert ist, fordert Sie von einem Laien doch keine Tiefergreifende Beschäftigung mit der Materie der Datenerhebung und Verarbeitung, welche häufig nur ein Mittel ist, einen anderen Sachverhalt darzustellen.
Um nun eine optimale Fokussierung eines Anwenders auf ein Thema zu erreichen ist es wünschenswert, das diesem die Anwendungsumgebung, in welcher er sich zurechtfinden soll schon bekannt ist.
Der kleinste Gemeinsame Nenner für Berechnungen jedweder Art ist typischerweise ein Tabellenkalkulationsprogramm.
Das erklärte Ziel ist es also, dem Anwender Echtzeitdaten einer Messung oder Simulation z.B. in einer Tabellenkalkulation zur Verfügung zu stellen. Der Anwender kann sich nun optimal auf die eigentlich gewünschte Weiterverarbeitung oder Bewertung der Daten konzentrieren. Weitere Gedanken über die datenerhebenden und weiterführende Prozesse die vor der eigentlichen Arbeit mit den Daten stehen entfallen.
Echtzeitdaten können in einer bekannten Umgebung weiterverarbeitet werden, ohne sich länger mit Informationtechnischen Details aufhalten zu müssen.

Robin Menn
Hittfelder Landstraße 32
21218 Seevetal

Tel.: 0178 / 649 67 54
E-Mail: robin.menn@googlemail.com

Inhaltsverzeichnis

1 Einleitung 1

2 Grundlagen und Stand der Technik 3

2.1 Objektorientierte Programmierung mit C++ . . . 3

2.1.1 Vorteile gegenüber der prozeduralen Programmierung . . . 4

2.1.2 Klassen und Objekte . . . 4

2.1.3 Vererbung, Polymorphie und Interfaces . . . 9

2.2 Entwurfsmuster . . . 13

2.2.1 Erzeugungsmuster . . . 13

2.2.2 Strukturmuster . . . 13

2.2.3 Verhaltensmuster . . . 14

2.2.4 Definitionen . . . 14

2.2.5 Beispiele und Anmerkungen . . . 15

2.3 Threads . . . 18

2.3.1 POSIX Thread Grundlagen . . . 18

2.3.2 Thread-Beispiel . . . 20

2.4 Unified Modelling Language UML . . . 23

2.4.1 Zustandsdiagramm . . . 23

2.4.2 Klassendiagramm . . . 24

2.5 Multi Tier Applikation . . . 27

2.6 Openoffice/Libreoffice . . . 29

2.6.1 UNO-Schnittstelle und C++ Einbindung . . . 29

2.7 MySQL . . . 30

2.8 Doxygen . . . 31

3 Entwurf Multi Tier Applikation 33

3.1 Entwicklungsumgebung 33

3.2 Namenskonvention, Codestyle und Dokumentation 34

3.3 Applikationsübersicht 34

3.3.1 Datenbank Request Lauf 35

3.3.2 Echtzeitdaten Request Lauf 36

3.4 Presentation Tier 37

3.4.1 Controller 37

3.4.2 Views 38

3.5 Application Tier 39

3.5.1 BusinessDelegate 39

3.5.2 ServiceLocator 39

3.5.3 DAOFactory (DAO- und Factory-Pattern) 40

3.5.4 Value-Objects + Services 40

3.6 Data Tier 41

3.6.1 DAO-Factory 41

3.6.2 Data Access Object 41

3.7 Datenbank 43

3.8 Libreoffice Adapter 44

3.9 Anwendungsfälle 44

4 Implementierung 45

4.1 Vorbereitungen und Systemvoraussetzungen 45

4.2 MySQL-Connector 46

4.3 LibreOffice SDK, Libraries 46

4.4 Einrichten der intergrierten Entwicklungsumgbung Netbeans 46

4.4.1 General (1) 46

4.4.2 C++ Compiler (2) 46

4.4.3 Linker (3) 47

4.5 Presentation Tier 48

4.5.1 Abweichungen aus praktischen Überlegungen 49

4.5.2 Abweichungen aus Zeitlichen Gründen 49

4.5.3 Abweichende Funktionalitäten 49

4.5.4 Abstrakte Strukturen 49

4.5.5 Weitere Implementation 49
4.6 Application Tier 49
4.6.1 Abweichung aus praktischen Überlegungen 50
4.6.2 Abweichungen aus Zeitlichen Gründen 50
4.6.3 Abweichende Funktionalitäten 50
4.6.4 Abstrakte Strukturen 50
4.6.5 Weitere Implementation 50
4.7 Data Tier 51
4.7.1 Abweichung aus praktischen Überlegungen 51
4.7.2 Abweichungen aus Zeitlichen Gründen 51
4.7.3 Abweichende Funktionalitäten 51
4.7.4 Abstrakte Strukturen 51
4.7.5 Weitere Implementation 51
4.7.6 MySQL Datenbank MessdatenDB 51
4.8 Spezielle Umsetzungen 52
4.8.1 Threads 52
4.8.2 DAO-Factory 52
4.8.3 Log 52
4.9 Hinweise für schnelle Anpassungen 53
4.9.1 WriterDevice 53
4.9.2 PropertyStrategy 53
4.9.3 User Interface 53
4.9.4 Serviceerweiterung 53
4.9.5 Datenbank 53

5 Schlussfolgerungen, Fragen und Ausblicke 55
5.1 Fazit 55
5.2 Zukünftige Anwendungen und Anpassungen 56
5.2.1 Kurzfristig zu realisierende Anpassungen 57
5.2.2 Mittelfristig zu realisierende Anpassungen 58

Literatur 60

A Anhang **62**

A.1 IDE 62

A.2 MySQL Datenbank erstellen 64

A.3 Bekannte Bugs 65

A.3.1 pixmap warning 65

A.3.2 Ausführen des Programms außerhalb der Entwicklungsumgebung 65

1 Einleitung

Die wesentlichen Aufgabe dieser Arbeit besteht darin, ein Programm zu entwickeln, welches die Benutzerfreundlichkeit von Echtzeitdatenerfassungen erhöht. Hierzu werden sämtliche benutzte Grundlagen erläutert, so dass der Leser die Entwicklung des Programms mitverfolgen kann. Im darauffolgenden Kapitel Entwurf wird umrissen, wie das Programm ausgestalten ist. Das Kapitel Implementation stellt die Unterschiede, die sich zum Entwurf ergeben, dar.

Die Grundlagen wenden sich an einen Leser, der Grunderfahrung in der Programmierung mit bringt. Es wird der objektorientierte Ansatz in der Programmierung erläutert. Viele Aspekte dieser Programmiergrundlage werden explizit beleuchtet. Entwurfsmuster, welche eine zentrale Rolle in der Entwicklung professioneller Applikationen spielen, sind hier gesondert in ihrer Funktionsweise und Definition dargestellt. Anhand von Beispielen werden einige dieser Muster, sowie weitere Grundlagen verdeutlicht. Die in der späteren Implementierung benutzte Technik des Multithreading wird anhand eines Beispiels vorgestellt. Einige Hilfsmittel werden eingeführt und ihre Benutzung anhand von Beispielen demonstriert.

Der Entwurf dient der Darstellung der Strukturen und Architektur, die die Grundlage für das zu entwickelnde Programm sind. Die Basis des Kapitels ist die Multi-Tier-Architektur, die bereits in den Grundlagen kurz vorgestellt wird. Es finden sich Einstellungen zur integrierten Entwicklungsumgebung, sowie Informationen zur Namenskonvention und Dokumentation. Die drei Schichten werden explizit vorgestellt. Des Weiteren finden sich hier alle relevanten Vorgaben der Arbeit und deren Umsetzung. Einzelnen Problemstellungen werden Entwurfsmuster zugeordnet und erklärt.

Die Implementierung umfasst die Umsetzungen des Entwurfes. Verdeutlicht werden hier konkrete Einstellungen und Parameter. Es wird auf weiterführende, benötigte Bibliotheken eingegangen. In diesem Kapitel werden die Unterschiede zwischen Entwurf und realer Umsetzung konkret aufgezeigt. Die Grundlagen der Entscheidung für eine vom Entwurf abweichende Implementierung sind genannt. Zusätzliche Informationen zu speziellen Umsetzungen und für schnelle Anpassungen sind hinterlegt.

Schlussendlich wird ein Fazit gezogen. Hier wird auf Arbeitsweise, spezielle Problemstellungen und Ziele eingegangen. Es folgt ein Ausblick auf zukünftige Anwendungsbeispiele.

2 Grundlagen und Stand der Technik

2.1 Objektorientierte Programmierung mit C++

Die Entwicklung der Programmiersprache C++ begann in den frühen 80er Jahren. C++ ist die folgerichtige Erweiterung der Sprache C. Während C vor allem die prozedurale Programmierung unterstützt, wird dies in C++ um den objektorientierten und generischen Ansatz erweitert.
Der Begriff objektorientierte Programmierung ist seit Anfang der 90er Jahre gebräuchlich. Das Ziel dieser Herangehensweise ist, die Bedienung und Programmierung von PCs der alltäglichen Erfahrung und Aufgaben des Menschen anzunähren. Dies bedeutet für Benutzer und Entwickler, dass unter anderem Dateien, Verzeichnisse, Datenbanken, Treiber als Objekte dargestellt werden. Diese Objekte sind, angelehnt an reale Entitäten, manipulierbar. Hierdurch werden besondere Anforderungen an die Programmierung fällig.
Ein Objekt kann vereinfacht gesprochen erst einmal als eine spezielle Variable betrachtet werden. Technisch gesehen handelt es sich um einen begrenzten, mit Zugriffsrechten geschützten Speicherbereich, der Daten und Code enthält. [1]

Das erklärte Ziel der objektorientierten Programmierung ist es, die Fehleranfälligkeit zu verringern, die Wartbarkeit zu verbessern, sowie die Wiederverwendbarkeit von Programmcode zu erleichtern. Um diese Ziele zu erreichen werden unter anderem verschiedene technische Verfahren wie Datenkapselung und Vererbung eingesetzt.[4] Hilfsmittel, wie eine Integrierten Entwicklungsumgebung oder Sprachkonstruktionen wie die Unified Modelling Language (UML) (siehe 2.4) vereinfachen und fördern die Entwicklung.

Heutzutage gut geschriebene C++ Programm werden sowohl Elemente des objektorientierten, als auch des klassischen prozeduralen Programmierens enthalten. C++ ist eine erweiterbare Sprache, in der neue Typen generisch definiert werden können, die ein eigenes Verhalten implementieren, sich aber wie Standard-Datentypen benutzen lassen. Durch diese Flexibilität ist C++ somit auch für große Programmierprojekte geeignet.[5]

[1] **object**
region of data storage in the execution environment, the contents of which can represent values
Note: When referenced, an object may be interpreted as having a particular type; see 6.3.2.1
- ISO/IEC 9899:1999: 3.14

2.1.1 Vorteile gegenüber der prozeduralen Programmierung

Die objektorientierte Programmierung versucht große Softwareprojekte in den Griff zu bekommen. Ab einer gewissen Größe eines Projektes lässt sich dieses kaum noch, bzw. nur mit größter Mühe pflegen. Der Ablauf von prozeduralen Programmen, deren Programmfluss und Variablen, sind häufig nur noch schwer zu überschauen.
Objektorientierung hilft an dieser Stelle zum Beispiel dadurch, das die versehentliche Veränderung einer Variable nahezu unmöglich ist (Datenkapselung). Die Wiederverwendung von Programmteilen wird erleichtert, indem diese, einmal entwickelt, in andere Programmen integriert werden können.

Die Vorteile gegenüber der prozeduralen Programmierung sind besonders gut anhand von einigen Beispielen deutlich zu machen.

2.1.2 Klassen und Objekte

Das wichtigste Element in der objektorientierten Programmierung ist die Klasse und das von ihr instanzierte Objekt. Das folgende Beispiel dient der Verdeutlichung:

```
#include <iostream>
using namespace std;

class RaumTemperatur{
private:
  double raumtemp;
public:
  double getTempC();
  double getTempF();
  void setTempC(double tempC){raumtemp = tempC+273.16;}
  void setTempF(double tempF){raumtemp = (tempF+459.67)*0.55555556;}
};

double RaumTemperatur::getTempC(){
  double t = raumtemp-273.16;
  return t;
}

double RaumTemperatur::getTempF(){
  double t= (raumtemp *1.8)-459.67;
  return t;
}

int main(){
  RaumTemperatur wohnzimmerTemperatur;
  RaumTemperatur kellerTemperatur;
  wohnzimmerTemperatur.setTempC(22.5);
  kellerTemperatur.setTempF(43.2);

  cout<<"Im Wohnzimmer beträgt die Temperatur "<<wohnzimmerTemperatur.getTempC()<<"°C"<<endl;
  cout<<"Im Wohnzimmer beträgt die Temperatur "<<wohnzimmerTemperatur.getTempF()<<"°F"<<endl;
  cout<<"Im Keller beträgt die Temperatur "<<kellerTemperatur.getTempF()<<"°F, dass entspricht "
    <<kellerTemperatur.getTempC()<<" °C"<<endl;
}
```

Ausgabe:

```
Im Wohnzimmer beträgt die Temperatur 22.5°C
Im Wohnzimmer beträgt die Temperatur 72.518°F
Im Keller beträgt die Temperatur 43.2°F, dass entspricht 6.21222 °C
```

Die Definition der Klasse beginnt in Zeile 4 mit dem Schlüsselwort `class`. Die Klasse heißt hier RaumTemperatur und soll der Berechnung und Ausgabe von unterschiedlichen Raumtemperaturen dienen.

Zuerst wird eine private Variable `raumtemp` definiert, welche vom Typ `double` ist. Das Schlüsselwort `private` schränkt die Sichtbarkeit der Variablen in soweit ein, als dass diese von außen nicht mehr sichtbar sind. Dies dient unter anderem der Datensicherheit und schützt vor falscher Benutzung. Ein Zugriff auf die Variable `raumtemp` kann künftig nur noch durch Methoden der Klasse selbst geschehen.

Die vier implementierten Methoden (Zeile 8-11) sind durch das vorangestellte Schlüsselwort `public:` (Zeile 7) von außen sicht- und ausführbar. Die beiden Methoden in Zeile 10 und 11, sind in diesem Falle inline[2] implementiert.
Eine weitere Art der Implementation geschieht von Zeile 14-22. Dort wird der Definition entsprechend der Rückgabetyp an sich und die Signatur mittels Gültigkeitsoperator (::) (engl. scope operator) angesprochen und anschließend ausformuliert. Dabei ist auf Konsistenz und Namensgleichheit, sowie Anzahl der Parameter und deren Datentypen zu achten. Eine Klasse, die in der Definition `void` als Rückgabewert hat, kann hier nicht mit dem Rückgabetyp `int` implementiert werden.

Wie an den `set` und `get`-Methoden zu erkennen, wird die Temperatur in der Variable `raumtemp` durch Umrechnung in Kelvin gespeichert oder entsprechend ausgegeben. Es erklärt sich jetzt, warum die Kapselung von Daten so wichtig ist. Nur durch diese Methodik kann die Konsistenz der Daten Ein- und Ausgabe gewährleistet werden. Von außen ist nicht ersichtlich, nach welchen Konvertierungen die Werte in der Variablen `raumtemp` hinterlegt sind. Um von außerhalb die Temperatur zu bekommen nutzt man einzig die Methoden `getTempC()`und `getTempF()`.

Sobald ein Objekt vom Typ `RaumTemperatur` instanziert wird (Zeile 25), ist in diesem Falle der Variablen `raumtemp` noch kein Wert zugewiesen. Dies kann zu Problemen führen, wenn vor Wertzuweisung auf die Variable `raumtemp` zugegriffen wird. So ein Zustand kann mit Hilfe von Konstruktoren verhindert werden.

2.1.2.1 Konstruktoren, Destruktoren und Überladung

Der Konstruktor ist eine spezielle Methode, die beim Erzeugen von Objekten aufgerufen wird. Er versetzt dieses Objekt in einen definierten Zustand, um benötigte Ressourcen zu reservieren. Ein Konstruktor kann mit Parametern versehen werden und hat implizit eine Referenz vom eigenen Typ als Rückgabetyp (this, siehe 2.1.2.3). Sollte kein Konstruktor implementiert sein, so wird ein Standardkonstruktor implizit angewendet.

Offensichtlich kann ein Objekt einer Klasse instanziert werden, ohne dass seinen Variablen explizit Werte zugewiesen wurden. Der Konstruktor, mit Hilfe dessen dies verhindert wird, sieht in Definition und Implementierung wie folgt aus:

```
1 class RaumTemperatur{
2 ...
3   RaumTemperatur(){raumtemp=0;}
4   RaumTemperatur(const RaumTemperatur& orig);
5 ...
6 }
```

[2] Inline bedeutet, dass die Methoden schon innerhalb der Klassedefinition implementiert werden.

Der Konstruktor wird jeweils zum Zeitpunkt der Erstellung eines Objektes aufgerufen. In diesem Fall wird durch `RaumTemperatur(){raumtemp=0;}` der implementierte Standardkonstruktor aufgerufen und die Variable `raumtemp` mit dem Wert 0 initialisiert (Zeile3). Bei `RaumTemperatur(const RaumTemperatur& orig)` handelt es sich um den Kopierkonstruktor, der nicht weiter erläutert wird, jedoch von vielen IDEs[3] standardmäßig in eine neue Klasse implementiert wird.
Überladung (engl. Overloading) von Methoden beschreibt die mehrfache Implementierung einer Methode mit unterschiedlicher Signatur, die sich nur an Parametertyp und/oder Anzahl derselbigen unterscheidet, der Methodenname aber gleich bleibt.

Angenommen es soll bei Erstellung des Objektes vom Typ `RaumTemperatur` einen Wert ungleich 0 initialisiert werden. Dazu wird ein weiterer Konstruktor implementiert, der jedoch in der Definition und Implementation einen Übergabewert besitzt (Zeile 5). Zur Laufzeit entscheidet sich, je nachdem, ob bei der Instanzierung ein Übergabewert übergeben ist, oder nicht, welcher Konstruktor aufgerufen wird. Dieses nennt sich Überladung einer Methode und ist auch für andere Methoden als den Konstruktor möglich.
Nach Implementierung der gezeigten Konstruktoren kann wie folgt mit diesen gearbeitet werden.

```
1  class RaumTemperatur{
2  ...
3    RaumTemperatur(){raumtemp=0;}
4    RaumTemperatur(const RaumTemperatur& orig);
5    RaumTemperatur(double TInit){raumtemp = TInit;}
6    virtual ~RaumTemperatur(){}
7  ...
8  }
9
10
11 RaumTemperatur kuechenTemperatur;
12 RaumTemperatur badezimmerTemperatur(1000);
13
14 cout<<"In der Küche beträgt die Temperatur "<<kuechenTemperatur.getTempC()<<"°C"<<endl;
15 cout<<"Im Badezimmer beträgt die Temperatur "<<badezimmerTemperatur.getTempC()<<"°C"<<endl;
```

Ausgabe:

```
In der Küche beträgt die Temperatur -273.16°C
Im Badezimmer beträgt die Temperatur 726.84°C
```

Offensichtlich wird ab sofort bei der Erstellung eines neuen Objektes die Variable `raumtemp` standardmäßig mit 0 Kelvin initialisiert. Wenn das neue Objekt mit Übergabewert instanziert wird, übernimmt der Konstruktor mit Parameter (Zeile 5) die Initialisierung der Variable `raumtemp`.
Der Destruktor unterscheidet sich von dem Konstruktor durch eine vorangehende Tilde ˜ , besitzt niemals Parameter und gibt einen leeren Datentyp zurück. Er wird standardmäßig bei der Auflösung des Objektes ausgeführt, und ist im Normalfall dafür zuständig, vom Objekt benutzte Resource wieder freizugeben. Konstruktoren und Destruktoren können je nach Anwendungsfall sowohl `public`, `protected` als auch `private` deklariert werden.
`protected` ist ein Schlüsselwort, das Methoden oder Membervariablen eines Objektes in der Sichtbarkeit weiter einschränkt. Sie sind nur für Instanzen des Objektes, sowie seine Erben sichtbar. (Vererbung, siehe 2.1.3.1)

[3]IDE = Intergrated Development Environment

2.1.2.2 Statische Variablen und Methoden

Statische Methoden und Variablen sind nicht von der Existenz eines Objektes abhängig. Sie sind für die gesamte Klasse gültig, für alle instanzierten Objekte gleich und werden mit dem Schlüsselwort `static` definiert. Statische Methoden können ausschließlich mit statischen, also objektunabhängigen Variablen interagieren. Hiermit lässt sich zum Beispiel ein Zähler realisieren, der die Anzahl der instanzierten Objekte zählt. Eine Anpassung der bisherigen Klasse soll als Beispiel dienen:

```
1  class RaumTemperatur{
2  private:
3    double raumtemp;
4    static int zaehler;
5    static int zaehle();
6  public:
7    static int getZaehler();
8    RaumTemperatur(){raumtemp=0; zaehle();}
9
10 ...
11 }
12 ...
13 int RaumTemperatur::zaehle(){zaehler++;}
14 int RaumTemperatur::getZaehler() {
15     return zaehler;
16 }
17 ...
18
19 int RaumTemperatur::zaehler=0;
20
21 int main(){
22
23 ...
24
25 cout<<"Im Badezimmer beträgt die Temperatur "<<badezimmerTemperatur.getTempC()<<"°C"<<endl;
26
27 cout<<endl<<"Es existieren "<<RaumTemperatur::getZaehler()<<" Raumtemperatur-Objekte"<<endl;
28 }
```

Ausgabe:

```
Im Wohnzimmer beträgt die Temperatur 22.5°C
Im Wohnzimmer beträgt die Temperatur 72.518°F
Im Keller beträgt die Temperatur 43.2°F, dass entspricht 6.21222 °C
In der Küche beträgt die Temperatur -273.16°C
Im Badezimmer beträgt die Temperatur 726.84°C

Es existieren 4 Raumtemperatur Objekte
```

Mit jeder Ausführung eines der Konstruktoren wird die Variable `zaehler` um eins erhöht. Zusätzlich könnte in dem Destruktor eine Routine implementiert werden, welche bei Zerstörung eines Objektes die Variable `zaehler` um eins verringert. So wäre Aktualität gewährleistet. Auf statische Variablen und Funktionen wird mit Hilfe des scope-operators (::) zugegriffen.

2.1.2.3 Schlüsselwort this

Das Schlüsselwort `this` bezeichnet einen Zeiger auf das aktuelle Objekt, bzw. auf den Speicherbereich, in dem das Objekt hinterlegt ist. Mit Hilfe des `this`-Pointers lässt sich so zum Beispiel in Methoden ganz einfach auf Variablen oder Methoden des Objektes zugreifen. `this` wird im Kontext eines Objektes benutzt, z.B. wenn eine Membervariable durch eine lokale Variable verdeckt wird. Integrierte Entwicklungsumgebungen sind hier besonders komfortabel, indem sie nach der Eingabe des Struktur-Operators das gesamte Spektrum an Methoden und Variablen zur Auswahl stellen, siehe Bild 2.1

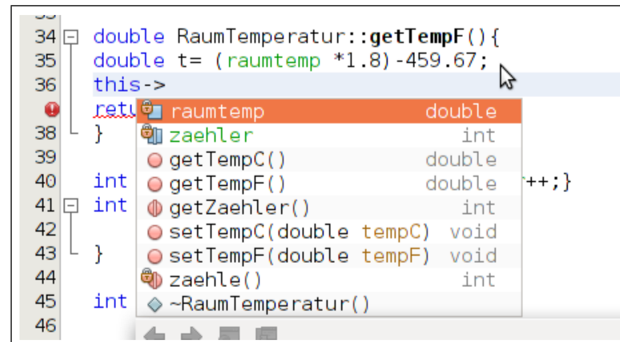

Abbildung 2.1: this-Pointer in Netbeans-Entwicklungsumgebung

2.1.2.4 new und delete-Operatoren

Die `new` und `delete`-Operatoren dienen der gegenüber C (`malloc, free`, etc) sehr vereinfachten Reservierung und Freigabe von Speicher. `new` fordert einen Speicherbereich im dynamischen Heap-Speicher an und der Konstruktor wird aufgerufen um den Speicher zu initialisieren. `delete` ruft erst alle Destruktoren auf und gibt den Heap-Speicher im Anschluss explizit wieder frei
Das Beispielprogramm wird um folgende Zeilen ergänzt:

```
1  ...
2  RaumTemperatur* Dachboden = new RaumTemperatur;
3  Dachboden->setTempC(9999);
4  //Dachboden.getTempC();
5  cout<<"Adresse:␣"<< Dachboden<<endl
6          <<"Temperatur␣unterm␣Dach:"<<Dachboden->getTempC()<<endl;
7  delete Dachboden;
8  //cout<<endl<<Dachboden->getTempC();
9  Dachboden=NULL;
10 //cout<<endl<<Dachboden->getTempC();
```

Ausgabe:
`Adresse: 0x98bc030` *(oder ähnlich)*
`Temperatur unterm Dach:9999`

In Zeile 1 wird neuer Speicher für ein neues Objekt vom Typ Raumtemperatur reserviert. Die Pointer-Variable `*Dachboden` zeigt auf die Struktur. In Zeile 2 wird mit Hilfe des Zeigeroperators (->) die Methode `setTemp(double)` aufgerufen. Die dritte, auskommentierte Zeile demonstriert, dass auf das Objekt nicht mehr mit dem .-Operator (Strukturoperator) zugegriffen werden kann. Der ->-Operator lässt sich jedoch Synonym verwenden. In Zeile 5 wird die Adresse des Pointers `Dachboden` per `cout` ausgegeben. Mit dem Pfeiloperator kann die Temperatur weiterhin wie gewohnt auslesen werden.

Jeder mit `new` reservierter Speicherbereich sollte durch ein `delete` wieder freigegeben werden (Hier in Zeile 7). Wird versucht, den in Zeile 8 auskommentierten Befehl auszuführen, so kann dies unter Umständen von Erfolg gekrönt sein, da der Speicher noch nicht anderweitig reserviert sein muss. Es ist gängige Praxis, den Pointer nach dem delete `NULL` zu setzen. Hierdurch wird ein versehentlicher erneuter Aufruf mit einem Speicherzugriffsfehler (Segfault) abgefangen.

2.1.3 Vererbung, Polymorphie und Interfaces

Dieses Kapitel orientiert sich in Teilen stark an dem ersten Kapitel des Buches *Entwurfsmuster von Kopf bis Fuß* [3]. Dieses basiert seinerseits auf dem Standardwerk [10]. Mit diesem Standardwerk führte die sogenannte Gang of Four (GoF) Entwurfsmuster ein.

2.1.3.1 Vererbung

Vererbung kommt dann zum Einsatz, wenn Zusammenhänge oder Ähnlichkeiten (in Ausführung und Verhalten) zwischen mehreren Klassen bestehen. Vererbung ist ein Ableitungsprozess, bei welchem die erbende Klasse automatisch alle sichtbaren (`protected` und `public`) Methoden, Variablen und Konstanten der vererbenden Klasse übernimmt, ohne diese noch einmal definieren und implementieren zu müssen. Die erbende Klasse kann Funktionalitäten übernehmen und bei Bedarf sogar ändern. Die vererbende Klasse wird häufig Basisklasse genannt. C++ erlaubt die Mehrfachvererbung, welche bei weitem nicht in allen Programmiersprachen zu finden ist.

Als Beispiel wird ein EntenSimulator erstellt, der vorerst eine Basisklasse `Ente` und eine erbende Klasse `Moorente` enthält. Definition und Implementation wie folgt (Hier sind Definition und Implementierung voneinander getrennt, so dass pro Klasse 2 Dateien nötig sind. Eine header-Datei mit Definitionen und eine cpp-Datei, welche die Implementierung beinhaltet.):

```
1  //Ente.h
2  #ifndef ENTE_H
3  #define ENTE_H
4  class Ente {
5  public:
6      Ente();
7      Ente(const Ente& orig);
8      virtual ~Ente();
9      void quaken();
10     void schwimmen();
11     virtual void anzeigen()=0;
12 private:
13 };
14 #endif  /* ENTE_H */
```

```
1   //Ente.cpp
2  #include "Ente.h"
3  #include <iostream>
4  using namespace std;
5  Ente::Ente() {}
6  Ente::Ente(const Ente& orig) {}
7  Ente::~Ente() {}
8  void Ente::quaken(){cout<<"Ente quakt"<<endl;}
9  void Ente::schwimmen(){ cout<<"Ente schwimmt"<<endl;}
```

```
1  //MoorEnte.h
2  #ifndef MOORENTE_H
3  #define MOORENTE_H
4  #include "Ente.h"
5  class MoorEnte :public Ente{
6  public:
7      MoorEnte();
8      MoorEnte(const MoorEnte& orig);
9      virtual ~MoorEnte();
10     void anzeigen();
11 private:
12 };
13 #endif  /* MOORENTE_H */
```

```
1  //Moorente.cpp
2  #include "MoorEnte.h"
3  #include <iostream>
4  using namespace std;
5  MoorEnte::MoorEnte() {}
6  MoorEnte::MoorEnte(const MoorEnte& orig) {}
7  MoorEnte::~MoorEnte() {}
8  void MoorEnte::anzeigen(){cout<<"Ansicht MoorEnte"<<endl;}
```

Der Aufruf erfolgt in der `main()`

```
1  //main.cpp
2  #include <cstdlib>
3  #include "MoorEnte.h"
4  using namespace std;
5  int main(int argc, char** argv) {
6      //Ente Ente1;
7      MoorEnte Ente2;
8      Ente2.quaken();
9      Ente2.schwimmen();
10     Ente2.anzeigen();
11     StockEnte Ente3;
12     return 0;
13 }
```

Ausgabe: `Ente quakt`
`Ente schwimmt`
`Ansicht MoorEnte`

Die Basisklasse `Ente` ist diejenige, von der geerbt wird. In diesem Beispiel besitzt die Basisklasse eine sog. abstrakte Methode (`anzeigen()=0;`). Abstrakte Methoden besitzen nur einen Rückgabewert und eine Signatur. Sie besitzen keinen Methodenkörper (Implementation). Durch das Abstraktsetzen einer Methoden einer Klasse wird diese Klasse in Ihrer Gesamtheit abstrakt.
Von abstrakten Klassen können keine Objekte erzeugt werden, da ihnen die abstrakte Methode keine Funktionalität bietet. Eine erbende Klasse von der ein Objekt erzeugt werden soll, muss die abstrakte Methode aus der Basisklasse implementieren, andernfalls ist auch diese Klasse abstrakt mit Folge der Nichtinstanzierbarkeit
Bezogen auf unsere Implementation folgt daraus, das wir kein Objekt Ente instanzieren können, wohl aber eines des Typen MoorEnte, da in dieser die Methode `anzeigen()` implementiert ist. Die Ausgabe offenbart, das die `Ente2`, welche instanziert wurde, quaken, schwimmen und anzeigen aufrufen kann. Obwohl in der eigentlichen Implementierung (`Moorente.cpp`) nur die Methode `anzeigen()` ausgeführt wurde. Sowohl die `schwimmen()`-, als auch `quaken()`-Methoden wurden von `Ente` geerbt. Die abstrakte Methode `anzeigen()` jedoch wurde überschrieben und durch die der MoorEnte eigenen `anzeigen()`-Methode ersetzt (Overriding).
Ähnlich diesem Beispiel lassen sich Interfaces gut erklären.

2.1.3.2 Interfaces

Interfaces, oder zu deutsch Schnittstellen, sind immer abstrakte Klassen mit ausschließlich `public`-abstrakten Methoden. Sie beschreiben Typen, die aus Methodensignaturen (und Konstanten) bestehen können, ohne ihre Implementierung anzugeben. Diese Typen werden von erbenden Klassen ausgefüllt, indem die geforderten Methodensignaturen mit Funktionalität ergänzt werden müssen.

Interfaces können als Gruppierungen von Methoden und Verhaltensmustern aufgefasst werden. Durch die Implementierung eines Interfaces ist sichergestellt, dass definierte Methoden und Verhaltensmuster konkret implementiert werden müssen. Geschieht dies nicht, ist die Klasse automatisch abstrakt. Bei Erzeugung eines Objektes dieser Klasse kommt es zur Laufzeit zu einem Fehler. Abstrakte Klassen im Gegensatz können zum Teil bereits implementierte Methoden enthalten.

Wäre `virtual void anzeigen()=0;` die einzige Methode der Basisklasse Ente, so hielte es sich bei dieser Klasse um ein Interface, welches in der abgeleiteten Klasse MoorEnte die Implementierung der Methode `anzeigen()` voraussetzt.

2.1.3.3 Polymorphie

Polymorphie bedeutet Vielgestaltigkeit und ist ein bedeutender Teil der objektorientierten Programmierung. Polymorphie tritt immer im Zusammenhang mit Vererbung oder Interfaces auf. Eine Methode ist polymorph, wenn sie in verschiedenen Klassen unter gleicher Signatur unterschiedlich implementiert ist.

Sind innerhalb der Vererbung mehrere Methoden mit gleicher Signatur in der Klassenhierarchie vorhanden, dann wird erst zur Laufzeit entschieden, welche Methode für ein instanziertes Objekt verwendet wird (Dynamic Binding). Dabei bestimmt der Datentyp zur Compilezeit, welche Basis die instanzierten Objekte zur Laufzeit haben müssen. Zur Laufzeit wird entweder die Klasse des Objektes benutzt, oder jene, die in der Vererbungshierarchie am nächsten ist.

Zur Verdeutlichung wird die bisherige SimEnte Anwendung erweitert. Eine männliche Moorenten-Klasse `MoorEnteMann` wird erzeugt, welche noch zusätzlich die Methode `balz()` erhält. Diese Klasse erbt von der Klasse MoorEnte und die Methode `anzeigen` wird überschrieben.

```
1  /*  File:    MoorEnteMann.h */
2  #ifndef MOORENTEMANN_H
3  #define MOORENTEMANN_H
4   #include "MoorEnte.h"
5  class MoorEnteMann : public MoorEnte{
6  public:
7       MoorEnteMann();
8       MoorEnteMann(const MoorEnteMann& orig);
9       virtual ~MoorEnteMann();
10      void anzeigen();
11      void balz();
12 private:
13 };
14 #endif  /* MOORENTEMANN_H */
```

```
/* File:    MoorEnteMann.cpp */
#include "MoorEnteMann.h"
#include "MoorEnte.h"
#include <iostream>

MoorEnteMann::MoorEnteMann() {}
MoorEnteMann::MoorEnteMann(const MoorEnteMann& orig) {}
MoorEnteMann::~MoorEnteMann() {}
void MoorEnteMann::anzeigen(){ std::cout <<"Ansicht männliche Moorente"<<std::endl;}
void MoorEnteMann::balz(){    std::cout<<"Moorentenmännchen zeigt Balzverhalten"<<std::endl;}
```

Die Klasse `MoorEnte` wird in der Klassendefinition (*.h) um das Schlüsselgwort `virtual` erweitert, welches auf eine polymorphe Methode hindeutet.

```
    ...
    virtual void anzeigen();
    ...
```

Der Aufruf in der `main()` wie folgt erweitert:

```
    ...
    Ente* entepoly = new MoorEnteMann;
    entepoly->anzeigen();
    entepoly->quaken();
    cout<<entepoly<<endl;
    MoorEnteMann* ep = dynamic_cast<MoorEnteMann*>(entepoly);
    cout<<ep<<endl;
    ep->balz();
```

Ausgabe:

```
Ansicht männliche Moorente
Ente quakt
0x9643030
0x9643030
Moorentenmännchen zeigt Balzverhalten
```

Der Pointer `entepoly` ist zur Compilezeit vom Typ `Ente`. Zur Laufzeit wird mit dem Schlüsselwort `new` ein `MoorEnteMann`-Objekt initialisiert. Der `Ente`-Pointer zeigt jetzt auf ein `MoorEnteMann` Objekt. Dies ist möglich, da durch Vererbung alle Elemente von Ente im MoorEnteMann enthalten sind. Die `entepoly` kann mit dem ->-Operator verwendet werden, aber die Methoden der Spezialisierung sind nicht Sichtbar. Um diesen spezialisierten Typ und seine Methoden zu erreichen muss das Objekt zu dem entsprechenden Typ konvertiert werden (typecast). Damit wird die Sichtbarkeit von Methoden und Variablen erweitert (man spricht von widening). Aus `entepoly` wird hierdurch ein echter `MoorEnteMann` (Zeile 6). Zur Überprüfung ob `entepoly` und `ep` (typecasted) auch auf dasselbe Objekt zeigen noch eine kurze Ausgabe der Adresse (Zeile 3 und 4 in der Ausgabe).
Die Typenumwandlung per `cast` ist kein expliziter Bestandteil von Polymorphie, sondern sei hier nur der Vollständigkeit halber erläutert.
Einen guten Einblick in die Möglichkeiten, die sich aus Polymorphen Klassen ergeben, liefert das erweiterte Beispiel im folgenden Abschnitt 2.2.

2.2 Entwurfsmuster

Entwurfsmuster (Design-Pattern) sind die Antwort auf immer wiederkehrende Problemstellungen in der Informatik. Das Standardwerk für Softwaretechnik heißt **Entwurfsmuster. Elemente wiederverwendbarer objektorientierter Software** [10], wird jedoch auch häufig als GoF-book (Gang of Four, bezieht sich auf die vier Autoren) bezeichnet.
Ein Entwurfsmuster hat eine bestimmte Motivation, Anforderungen und Nutzen, die zur Lösung von Software-Architekturproblemen geeignet sind. Entwurfsmuster sind eine Beschreibung oder Schablone wie ein Problem gelöst werden kann, dabei handelt es sich um abstrakte Konstrukte welche nicht einfach als Sourcecode übernommen werden können sondern an die jeweiligen Begebenheiten der Architektur angepasst werden müssen.
Die Entwurfsmuster sind dabei in drei verschiedene Kategorien unterteilt. Man unterscheidet zwischen Erzeugungsmuster (Creational Design-Pattern), Strukturmuster (Structural Design-Pattern) und Verhaltensmuster (Behavioral Design-Pattern).

2.2.1 Erzeugungsmuster

Diese Entwurfsmuster behandeln die Objektinstanzierung und nutzen dabei Vererbung, um die Klasse des zu erzeugenden Objektes zu variieren. Die Objekterzeugung wird dabei an andere Objekte delegiert.
In dieser Arbeit werden die folgenden drei Erzeugungsmuster genutzt.

- Abstract Factory-Pattern
  Kreiert ein Objekt von unterschiedlichen Familien von Klassen. (siehe 2.4.2.1)

- Factory Method-Pattern
  Kreiert ein Objekt unterschiedlicher abgeleiteter Klassen.

- Singleton-Pattern
  Beschreibt eine Klasse, von der nur eine einzige Instanz erzeugt werden kann.

2.2.2 Strukturmuster

Diese Entwurfsmuster behandeln die Zusammenfassung von Klassen und Objekten zu größeren Strukturen. Diese Strukturen nutzen Vererbung und Komposition (Aggregation) um neue Interfaces zu erstellen. Die in den Klassenmustern beschriebenen Strukturen sind zur Compilezeit festgelegt. Die durch Objektmuster beschriebenen Strukturen sind zur Laufzeit änderbar. Folgende Strukturmuster werden benutzt:

- Adapter-Pattern
  Gleicht Interfaces unterschiedlicher Klassen an.

- Facade
  Eine einzelne Klasse die ein gesamtes Subsystem repräsentiert.

2.2.3 Verhaltensmuster

Verhaltensmuster beschreiben die Interaktion zwischen Objekten und Kontrollflüssen. Klassenmuster teilen die Kontrolle auf unterschiedliche Klassen auf. Die Objektmuster nutzen Komposition anstelle von Vererbung.
Diese Verhaltensmuster werden hier benutzt:

- Strategy-Pattern
  Kapselt einen Algorithmus innerhalb einer Klasse.
- Command-Pattern
  Kapselt einen Steuerungsbefehl als ein eigenes Objekt.
- Template-Method-Pattern
  Gibt die ausführbaren Schritte eines Algorithmus an eine Unterklasse weiter.

2.2.4 Definitionen

2.2.4.1 Abstract-Factory-Entwurfsmuster

Das Abstract-Factory-Pattern *bietet eine Schnittstelle zum erstellen von Familien verwandter oder zusammenhängender Objekte an, ohne konkrete Klassen anzugeben*

2.2.4.2 Factory-Method-Entwurfsmuster

Das Factory-Method-Pattern *definiert eine Schnittstelle zur Erstellung eines Objekts, lässt aber die Unterklassen entscheiden, welche Klassen instanziert werden. Factory-Method ermöglicht einer Klasse, die Instanzierung in Unterklassen zu verlagern*

2.2.4.3 Singleton-Entwurfsmuster

Das Singleton-Pattern *sichert, dass es nur eine Instanz einer Klasse gibt, und bietet einen globalen Zugriffsppunkt für diese Instanz.*

2.2.4.4 Adapter-Entwurfsmuster

Das Adapter-Pattern *konvertiert die Schnittstelle einer Klasse in die Schnittstelle, die der Client erwartet. Adapter ermöglichen die Zusammenarbeit von Klassen, die ohne nicht zusammenarbeiten könnten, weil sie inkompatible Schnittstelen haben.*

2.2.4.5 Fassade-Entwurfsmuster

Das Facade-Pattern *bietet eine vereinheitlichte Schnitstelle für einen Satz von Schnittstellen eines Basissystems. Die Fassade definiert eine hochstufigere Schnittstelle, die die Verwendung des Basissystems vereinfacht*

2.2.4.6 Strategie-Entwurfsmuster

Das Strategy-Pattern *definiert eine Familie von Algorithmen, kapselt sie einzeln und macht sie austauschbar. Das Strategy-Pattern ermöglicht es, den Algorithmus unabhängig von den Klassen die ihn einsetzen, variieren zu lassen.*

2.2.4.7 Kommando-Entwurfsmuster

Das Command-Pattern *kapselt einen Auftrag als ein Objekt und ermöglicht es so, andere Objekte mit verschiedenen Aufträgen zu parametrisieren, Aufträge in Warteschlangen einzureihen oder zu protokollieren oder das Rückgängigmachen von Operationen zu unterstützen*

2.2.4.8 Template-Method-Entwurfsmuster

Das Template-Method-Pattern *definiert in einer Methode das Gerüst eines Algorithmus und überlässt einige Schritte den Unterklassen. Template-Method erlaubt Unterklassen, bestimmte Schritte des Algorithmus neu zu definieren, ohne die Struktur des Algorithmus zu ändern.*

2.2.5 Beispiele und Anmerkungen

Aufgrund der Anzahl und Komplexität wird im Folgenden nur ein kurzes Beispiel aus dem Gesamtumfang aller Entwurfsmuster gezeigt. Meist erklären diese sich im konkreten Zusammenhang besser. Es sei hier auf das Kapitel 4 oder den Quelltext verwiesen.

2.2.5.1 Strategy-Pattern

In dem SimEnte-Beispiel (siehe 2.1.3) wird ein veränderlicher Algorithmus `FlugVerhalten` implementiert. Dies geschieht mittels Interface, welches einzig die abstrakte Methode `fliegen` mitbringt.

```
1 //Definition:
2 class FlugVerhalten{public: virtual void fliegen()=0;};
```

Die Algorithmen leiten sich von der Klasse Flugverhalten ab. So können unter anderem Klassen wie `FliegtNicht` und `FliegtMitFlügeln`, die jeweils die `fliegen()`-Methode implementieren müssen, erstellt werden.

```
1  //Definition:
2  class FliegtMitFluegeln :public FlugVerhalten {
3  public:
4       FliegtMitFluegeln();
5       virtual ~FliegtMitFluegeln();
6       void fliegen(){cout<<"Fliege mit Flügel";}
7  private:};
8
9  class FliegtNicht :public FlugVerhalten {
10 public:
11      FliegtNicht();
12      virtual ~FliegtNicht();
13      void fliegen(cout<<"Fliegt nicht";)
14 private:};
```

Der Basisklasse `Ente` wird eine Membervariable vom Typ `Flugverhalten` hinzugefügt. Die `fliegen()`-Methode wird so modifiziert, dass sie die Methode `fliegen()` auf dem jeweiligen `FlugVerhalten`-Objekt ausführt. Um ein neues FlugVerhalten zuzuweisen wird die Methode `setFlugVerhalten(Flugverhalten f)` implementiert.

```
1  //Definition:
2  #include "FlugVerhalten.h"
3  #ifndef ENTE_H
4  #define ENTE_H
5  class Ente {
6  public:
7       Ente();
8       virtual ~Ente();
9       virtual void anzeigen()=0;
10      virtual void fliegen(){flugverhalten->fliegen();}
11      void setFlugverhalten(FlugVerhalten f){flugverhalten =f;}
12 protected:
13      FlugVerhalten* flugverhalten;
14 };
15 #endif /* ENTE_H */
```

Für die korrekte Implementierung des FlugVerhaltens wird in der Klasse MoorEnte im Konstruktor ein Standard-Flugverhalten zugewiesen. In diesem Beispiel der Algorithmus `FliegtMitFluegeln`.

```
1 #include "MoorEnte.h"
2 ...
3      MoorEnte::MoorEnte() {
4         flugverhalten= new FliegtMitFluegeln;
5      }
6 ...
```

Erzeugen wir eine neue MoorEnte, bekommt diese das Flugverhalten FliegtMitFluegeln zugewiesen. Zur Laufzeit kann der Algorithmus gegen einen anderen ausgetauscht werden.

```
MoorEnte Ente1;
Ente1.fliegen();
Ente1.setFlugVerhalten(new FliegtNicht);
Ente1.fliegen();
```

Ausgabe:
`Fliege mit Flügel`
`Fliegt nicht`

Die Ausgabe bestätigt, der Algorithmus wurde zur Laufzeit erfolgreich ausgetauscht. Jetzt lassen sich auf einfachem Wege neue Algorithmen (FlugVerhalten) entwickeln und austauschen, ohne das in der Basisklasse Veränderungen notwendig sind.

Das so realisierte Zusammenfügen von verschiedenen Verhaltensobjekten nennt sich Komposition (strikt) oder Aggregation (weniger strikt) und ist eines der immer wiederkehrenden Elemente bei der Benutzung von Entwurfsmustern.

2.3 Threads

Ein Thread ist ein programmatischer Ablauf, der einem Prozess untergeordnet ist. Er wird immer dort benötigt, wo ein Ausführungsstrang parallel zu einem anderen bearbeitet werden soll. Es existieren zwei Arten von Threads, der Kernel- und der Userthread. Die Abarbeitung und Ausführung ist Betriebssystemabhängig.

Eine Grundvoraussetzung für das Multithreading ist die Nebenläufigkeit. Nebenläufigkeit bedeutet, dass zwei Ausführungsstränge unabhängig voneinander sind und es keine Rolle spielt, welcher zuerst abgearbeitet wird. Eines der wesentlichen Ziele ist die gleichmäßige Ressourcenauslastung.

Ein Prozess beschreibt den Adressraum in dem zusammengehörige Threads parallel verarbeitet werden. Um die Verwaltung und Adressierung eines Prozesses kümmert sich das Betriebssystem. Threads hingegen müssen vom Programm die nötigen Überwachungsroutinen mitgegeben bekommen. Ein Prozess hat immer mindestens einen Thread.

Als Grundlage für die Verwendung von Threads wird der POSIX Thread Standard genutzt, dieser ist in der letzten Version von 2004 in der IEEE Std 1003.1 genormt.

Eine komplette Erläuterung von Threads, Prozessen und ihren Grundlagen würde über den Rahmen dieser Arbeit hinausgehen. Jegliche Implementation baut auf auf den Beispielen von [11, /cplusplus/multithreaded3.php] auf. Einige Grundlagen seien hier dennoch kurz erwähnt und dann in einem Beispiel mit einer Thread-Klasse verdeutlicht.

2.3.1 POSIX Thread Grundlagen

Bei dem Starten eines Programmes wird ein einzelner default-Thread (`main`-Thread) erstellt und diesem die jeweilige `int main(int argc, char** argv)`-Routine übergeben. Sobald diese beendet ist, wird der Thread und letztlich das Programm geschlossen. Dies gilt äquivalent für alle Threads, sie werden gestartet, ihnen wird die jeweilige Routine mitgegeben und sie schließen sich bei Beendigung der Routine.

Wird der `main`-Thread beendet, bevor die nebenläufigen Threads abgeschlossen sind, dann werden diese zwangsläufig mit beendet. Das beenden eines POSIX-Threads kann auch erzwungen werden. Des weiteren ist es möglich einen Thread auf die Beendigung eines anderen warten zu lassen (ein sog. `join`) bevor die Programmroutine weiterläuft. POSIX-Threads verfügen weiterhin noch über verschiedene Mechanismen zum triggern von Threadzuständen.

2.3.1.1 Kurzer Einblick pthread-Funktionen

- Thread starten
  Die Funktion um einen Thread zu starten ist `pthread_create()`

- Thread beenden
  Die Funktion um einen Thread zu Beenden, wenn dieser nicht automatisch beendet wird ist `pthread_exit()`

- Auf Beendigung eines Threads warten
  `pthread_join()` ist eine Funktion, die in einem Thread darauf wartet, das ein anderer, spezifizierter Thread beendet wird.

- Thread conditions
  `pthread_cond_init()` initialisiert eine explizite Bedingung zur Steuerung von Threads per `wait()` und `signal()`.
  `pthread_cond_wait()` setzt einen Thread anhand einer Bedingung in den wait-Zustand in welchem er verweilt, bis er aus diesem explizit wieder herausgeholt wird.
  `pthread_cond_signal()` signalisiert einem bestimmten Thread anhand einer Bedingung, das er seinen Wartezustand verlassen soll. [4]

Eine wichtige Aufgabenstellung ist die Kommunikation zwischen Threads synchron zu gestalten. Oft müssen Threads auf gemeinsame Inhalte zugreifen können(shared memory). Unter Synchronisation versteht man die Koordinierung des Zeitlichen Ablaufs mehrerer Threads. Zweck der Synchronisierung ist es z.B. gleichzeitigen Zugriff auf gemeinsame Ressourcen zu gestatten ohne inkonsistente Daten zu erhalten. Synchronisation wird mit Hilfe von Mutex-Objekten erreicht.

2.3.1.2 Mutex

Ein Mutex (Mutual exclusion lock, auf deutsch ungefähr gegenseitiges auschließendes Verschließen) ist sowohl ein Verfahren, als auch ein Objekt (Datenstruktur) das gegenseitigen Ausschluss erzwingt. In einer Multithreading-Umgebung kann das Mutex Daten enthalten, auf welche von mehreren Threads aus zugegriffen werden kann. Um sicherzustellen, das ein Thread seine Arbeit an den Daten abgeschlossen hat, bevor ein anderer Thread auf das Mutex zugreift gibt es sog. Lock-Methoden. Die Lock- und Unlock-Methoden sperren exklusiv den Zugriff auf Daten innerhalb des Mutex. Gerade bei gleichzeitigem Lese- und Schreibzugriff oder Abarbeitung von sog. Queues und Listen, kann so der inkonsistente Zustand verhindert werde, Neben dem Mutex gibt es noch andere Methoden, exklusiven Zugriff auf Speicher zu sichern, hier sei z.b. der *Semaphor* genannt.

Bei unsachgemäßer Implementierung kann es zu einem Deadlock genannten Zustand kommen, in dem alle Threads auf dem Mutex gelockt sind!

[4] Für weitere Informationen zu Übergabewerten, Benutzung, etc. siehe pthread-Standard

2.3.2 Thread-Beispiel

Dieses Beispiel liefert eine vereinfachte Darstellung des in dem Projekt verwendeten Thread-Modells. Es soll schon an dieser Stelle den grundsätzlichen Programmablauf erläutern, so dass die Implementation vollständig zu verstehen ist. Nach der Erläuterung von Threads und Mutex werden hier nun deren Implementationen vorgestellt.

2.3.2.1 Klasse Thread und IRunnable

Es wurden bereits alle nötigen Funktionen für die Nutzung von POSIX-Threads in einer Klasse `Thread` gekapselt und für die Verwendung vorbereitet. Diese Klasse kann als Wrapper-Klasse bezeichnet werden. Der Wrapper ist dem Adapter-Pattern ähnlich (siehe 2.2.4.4). Um Objekten, die einem eigenen Thread laufen sollen die benötigte `run`-Methode zu liefern wird das Interface `IRunnable` bereitgestellt. Alle Thread-Objekte müssen dieses Interface implementieren. Die `run()`-Methode ist diejenige, welche vom Thread abgearbeitet wird.

2.3.2.2 QueueMutex-Klasse

Die Mutex-Klasse stellt das Verbindungsglied zwischen mehreren Threads dar. Konkret implementiert sind hier die `lock()` und `unlock()`-Methoden sowie `wait()` und `signal()`. Die QueueMutex-Klasse erbt alle Mutex-Elemente und stellt eine FIFO-`Queue`[5] mit Lese- und Schreibzugriff zur Verfügung. Der exklusive Lese und Schreibzugriff wird durch die Methoden `lock()`, `unlock()`, `wait()` und `signal()` realisiert. In dieser Konstruktion kann auf der zur Verfügung gestellten Queue nur ein Lese- oder Schreibzugriff zur selben Zeit erfolgen. Der Schreibthread fügt neue Elemente immer am Ende der Queue ein. Der Lesethread seinerseits liest immer das erste Element aus und verwirft es.
Quelltextauszug der QueueMutex-Klasse:

```
1  ...
2  class QueueMutex : public Mutex {
3  public:
4      QueueMutex() {}
5      float read() {
6          lock();
7          if (isEmpty()) { // wait for work
8              cout << "No Reading Work, Gone Sleeping, waiting for wakeup signal";
9              wait(); // auto unlock des Mutex
10         }
11         float f = v.front();
12         v.pop();
13         unlock();
14         return f;
15     }
16     void write(float f) {
17         lock();
18         v.push(f);
19         unlock();
20         if (getSize() > 0) {
21             signal();
22             cout<<"wakeup call by writer"<<endl;
23         }
24     }
25     std::queue<float> getQueue() {
26         return v;    }
27     int getSize() {
28         return v.size();    }
29     bool isEmpty() {
30         return v.empty();    }
31 protected:
32     std::queue<float> v;
33 };
```

[5] First In First Out

2.3.2.3 MyThread-Klasse

Dieses Listing zeigt den Aufbau der Klasse MyThread (header und cpp).

```
...
class MyThread :public IRunnable {
public:
    MyThread(QueueMutex* q);
    virtual ~MyThread();
    virtual void* run();
    bool getsignalstop();
private:
    bool signalstop;
    QueueMutex* queue;
};
```

```
...
MyThread::MyThread(QueueMutex* q) {
    signalstop = false;
    this->queue = q;
}
MyThread::~MyThread() {}
void* MyThread::run() {
    cout << "starte Thread" << endl;
    for (int i = 1; i <= 80; i++) {
        cout << "Schreibe im Thread auf queue, ";
        if (i > 70) {
            sleep(1);
            cout << "Langsam ";
        }
        cout << ": " << i << endl;
        queue->write(i);
    }

  while(true){
    if(queue->getSize()==0){break;}
  }
    signalstop = true;
    std::cout << "Thread1 fertig, terminating Thread now! " << endl;
}

bool MyThread::getsignalstop() {
    return signalstop;
}
```

Bei Instanzierung der Klasse MyThread wird ein QueueMutex Objekt übergeben. Im Konstruktor wird die `signalstop`-Variable auf `false` gesetzt. Die `run`-Methode beinhaltet den Code, der bei Instanzierung von MyThread abgearbeitet wird. Es werden die Zahlen 1-80 in das `QueueMutex`-Objekt geschrieben. Die eigentliche Schreibaktion auf die Queue geschieht in der Basisklasse Mutex. Vor Beendigung der `run()`-Methode wird `signalstop` gesetzt um zu informieren, das die Arbeit abgeschlossen ist.

2.3.2.4 Programmausführung

Mit den zuvor entwickelten Klassen kann jetzt ein kleiner Programmablauf gestaltet werden.

```
int main(int argc, char** argv) {
    QueueMutex* meineQueue = new QueueMutex;
    MyThread* einThread = new MyThread(meineQueue);

    auto_ptr<IRunnable> t(einThread);
    auto_ptr<Thread> thread1(new Thread(t));
    thread1->start();

    usleep(250);
    cout << "Starte auslesen der Queue im Mainthread jetzt!" << endl;
    do {

        cout << "ausgelesen im Mainthread: " << meineQueue->read() << endl;
    } while (!einThread->getsignalstop());

    cout << "Mainthread work done, waiting for join" << endl;
    thread1->join(); //
    cout << "Join successful, terminating mainthread now!" << endl;
}
```

Ein QueueMutex-Objekt für den Lese- und Schreibzugriff wird instanziert. Ein neues Thread-Objekt wird initialisiert. Die `auto_ptr` stellen dynamisch sichere Adressbereiche zur Verfügung. Der Thread wird anschließend an diesen Adressbereich übergeben und gestartet. Er springt in seine eigene `run()`-Methode (siehe 2.3.2.3) und beginnt mit der Abarbeitung (schreiben ins QueueMutex). Die `main()`-Routine läuft "parallel" dazu weiter und beginnt Ihrerseits mit der "Lesearbeit" auf dem QueueMutex (siehe 2.3.2.2) bis der nebenläufige Thread via `signalstop=true` Arbeitsende signalisiert. Per `join()` werden beide Threads zusammengeführt und der nebenläufige Thread endet.

Das Arbeiten mit Threads erfordert immer besondere Aufmerksamkeit des Programmierers, es kann niemals garantiert werden, welcher Thread seine Arbeit wie schnell abarbeitet. Dies kann man sehr gut beobachten, indem man dieses Beispielprogramm mehrmals ausführt und die Ausgaben genauer betrachtet. Es wird selten vorkommen, das ein Lauf des Programms dem anderen gleicht. Zu beachten ist, das ein Thread so schnell arbeiten kann, dass es zu Überschneidungen bei der Ausgabe in der Konsole kommt.

2.4 Unified Modelling Language UML

Die Unified Modelling Language ist ein nützliches Werkzeug für die Erstellung von objektorientierten Software-Architekturen. Es handelt sich um eine graphische Modellierungssprache die sowohl zur Spezifikation als auch zur Dokumentation von Software benutzt wird. Durch verschiedene Aspekte der UML können diverse Ansichten unterschiedlicher Aufgaben dargestellt werden. UML beinhaltet 14 unterschiedliche, standardisierte Diagrammtypen, wie zum Beispiel Sequenzdiagramm, Aktivitätsdiagramm, Klassendiagramm und Zustandsdiagramm. Klassen- und Zustandsdiagramm werden im Anschluss anhand kurzer Beispiele erklärt. Für Umfangreichere Informationen sei auf das Buch *Analyse und Design mit UML 2.3*[2] verwiesen.

2.4.1 Zustandsdiagramm

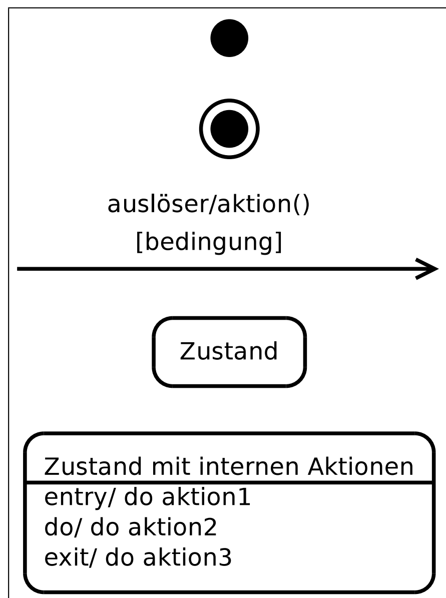

Abbildung 2.2: Grundlegende Zustandsdiagramm-Elemente

Das Zustandsdiagramm besteht aus den namengebenden Zuständen und Events/Triggern. Ein Zustand wird durch ein Rechteck mit abgerundeten Ecken repräsentiert. Bei Eintritt und Beendigung eines Zustandes können verschiedene Funktionen ausgelöst werden. Diese internen Aktionen werden zu einem definierte Zeitpunkt (entry/, do/, exit/) aufgerufen. Der Start in einem Zustandsdiagramm wird durch einen schwarzen Punkt symbolisiert. Das Ende durch einen schwarzen Punkt, der von einem Kreis umgeben ist. Ein Zustand kann sich nur durch einen Auslöser verändern. Ein solcher Auslöser (auch Event oder Transition) wird durch einen Pfeil dargestellt. Zusätzlich kann ein solcher Trigger auch noch mit dem Auslöser, auszuführender Funktion und zu erfüllender Bedingung versehen werden.

2.4.1.1 Beispiel

Am Beispiel der in Kapitel 2.3 entwickelten Thread-Applikation wird ein Zustandsdiagramm den Programmablauf veranschaulichen. Zwei Threads durchlaufen hier nacheinander verschiedene Zustände und versetzen ein Mutex Objekt in den Zustand locked/unlocked.

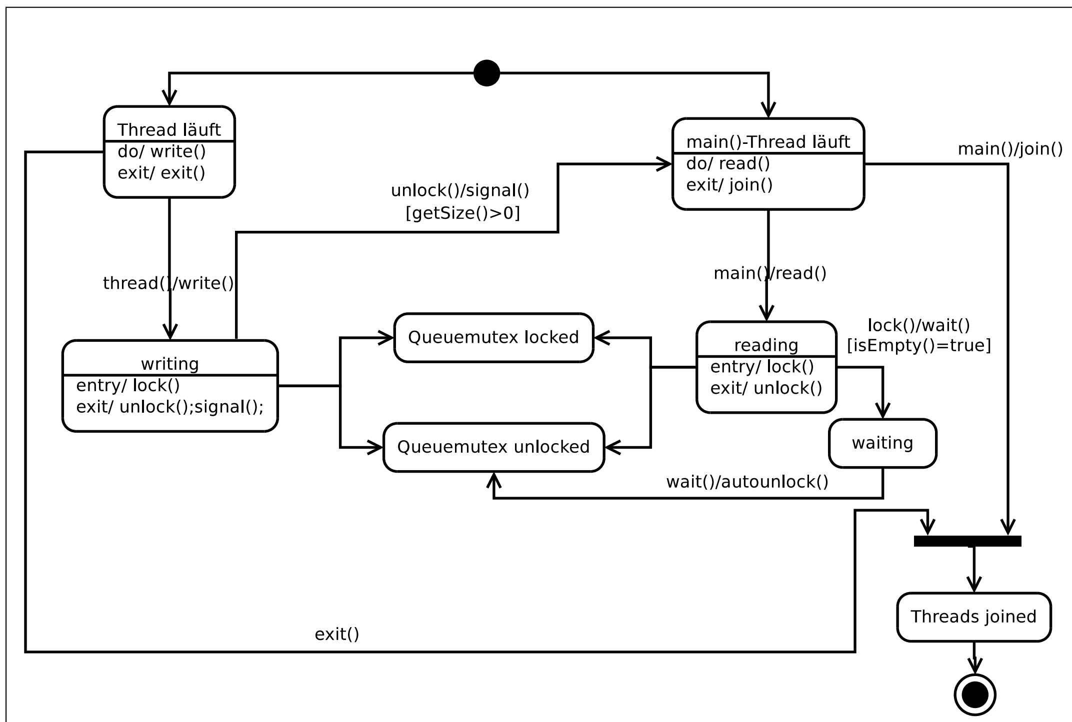

Abbildung 2.3: Zustandsdiagramm Threadapplikation mit Mutex

Die beiden Threads (`main()`-Thread und nebenläufiger Thread) werden nach dem Start in den Zustand "läuft" versetzt. Jeder dieser beiden Threads startet jetzt seine ihm eigene Methode `read()`, bzw. `write()`. `read()` und `write()` versetzen den jeweiligen Thread in den Zustand reading oder writing. Angenommen der `write()`-Thread ist schneller, so versetzt er das Mutex-Objekt in den Zustand locked, schreibt und unlocked wieder. Anschließend sendet er ein Signal an den `main()`-Thread, das dieser seine Arbeit aufnehmen kann. Der `main()`-Thread geht in den Zustand reading über, lockt das Mutex-Objekt, liest und unlockt. Da nicht garantiert ist, welcher Thread nach dem unlock der schnellere ist, der den Zugriff übernimmt, fragt der `main()`-Thread die Datenverfügbarkeit im Mutex ab. Im Falle, dass keine Daten Verfügbar sind, geht der `main()`-Thread in den Zustand waiting über und gibt das Mutex automatisch frei. Durch das `signal()` des nebenläufigen Threads wird der `main()`-Thread wieder aktiviert. Wird der nebenläufige Thread von außen beendet, wird der `main()`-Thread seine Arbeit noch zu Ende führen und dann das Programm beenden.

2.4.2 Klassendiagramm

Klassendiagramme werden in der OOP benutzt, um zu beschreiben, welche Klassen existieren und in welchen Beziehungen sie zueinander stehen. Unter anderem lassen sich Vererbungshierarchien abbilden, Verhaltensmuster darstellen und komplette Strukturen beschreiben. Klassendiagramme dienen im Kapitel Entwurf dazu, mittels einer grafischen Darstellung die Strukturen und Verhalten zu verdeutlichen.

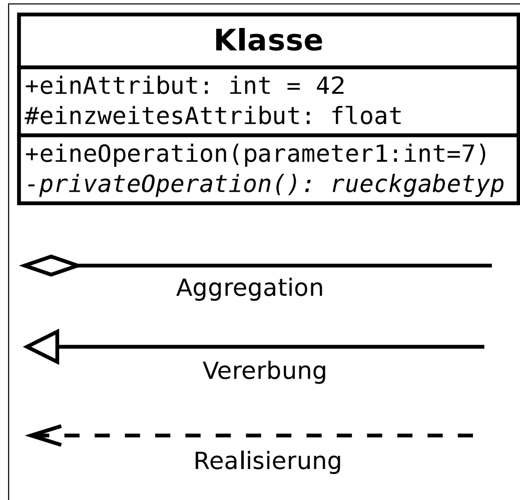

Abbildung 2.4: Grundlegende Klassendiagramm-Elemente

Klassen sind in Klassendiagrammen durch Rechtecke repräsentiert, die mindestens den Namen der Klasse beinhalten müssen. Zusätzlich können dort noch Attribute und Methodendefinitionen enthalten sein, diese werden getrennt von einander angegeben. Jeweils vor Methode oder Attribut wird die Sichtbarkeit derselbigen gekennzeichnet, "+" steht für `public` "-" für `private` und "#" für das `protected`-Privileg.

Für Attribute gilt folgende Namenskonvention:
Attributname: *Attributtyp*=*Initialwert*
Die allgemeine Bezeichnungskonvention für Methoden ist:
Methodenname(*Argument*:*Argumenttyp*=*Standardwert*):*Rückgabetyp*
Abstrakte Methoden werden kursiv gesetzt, Klassenmethoden (`static`) durch Unterstreichung gekennzeichnet.

Um die verschiedenen Beziehungen zwischen Klassen abzubilden existieren unterschiedliche Verbindungselemente.
Eine **Aggregation** beschreibt die Zusammensetzung einer Klasse (Aggregat) aus anderen Klassen. Kennzeichnend ist, dass das Aggregat Aufgaben stellvertretend für seine Teile wahrnimmt. In einer Aggregation zwischen zwei Klassen muss genau ein Ende das Aggregat sein und das andere für die Einzelteile stehen. Eine Aggregation wird als Linie zwischen zwei Klasse dargestellt und mit einer kleinen Raute versehen. Die Raute steht auf der Seite des Aggregats. Sie symbolisiert das Behälterobjekt, in dem die Einzelteile gesammelt sind.[2, S. 314]
Hierarchien, die durch **Vererbung** entstehen (siehe 2.1.3) werden durch einen Pfeil zwischen den Klassen verdeutlicht. Die Pfeilrichtung zeigt dabei immer auf die eigene Basisklasse.
Eine Realisierungsbeziehung ist eine Abhängigkeit von einem, oder mehreren Quellelementen zu einem, oder mehreren Zielelementen. Die Zielelemente sind für die Spezifikation oder die Implementierung der Quellelemente erforderlich.[2, S.318] Die Darstellung einer **Realisierung** erfolgt durch einen gestrichelten Pfeil. Dieser zeigt jeweils von der abhängigen auf die unabhängige Klasse.

Neben den Genannten gibt es noch weitere Beziehungselemente, die hier unerwähnt bleiben.

2.4.2.1 Beispiel Abstract Factory Pattern

Das Beispiel Abstract Factory Pattern zeigt einige der zuvor genannten UML-Elemente. Die Abbildung verdeutlicht das in 2.2.4.1 definierte Factory-Pattern. Dieses wird später für die Implementierung der Persistenten Datenspeicherung genutzt. Im Klassendia-

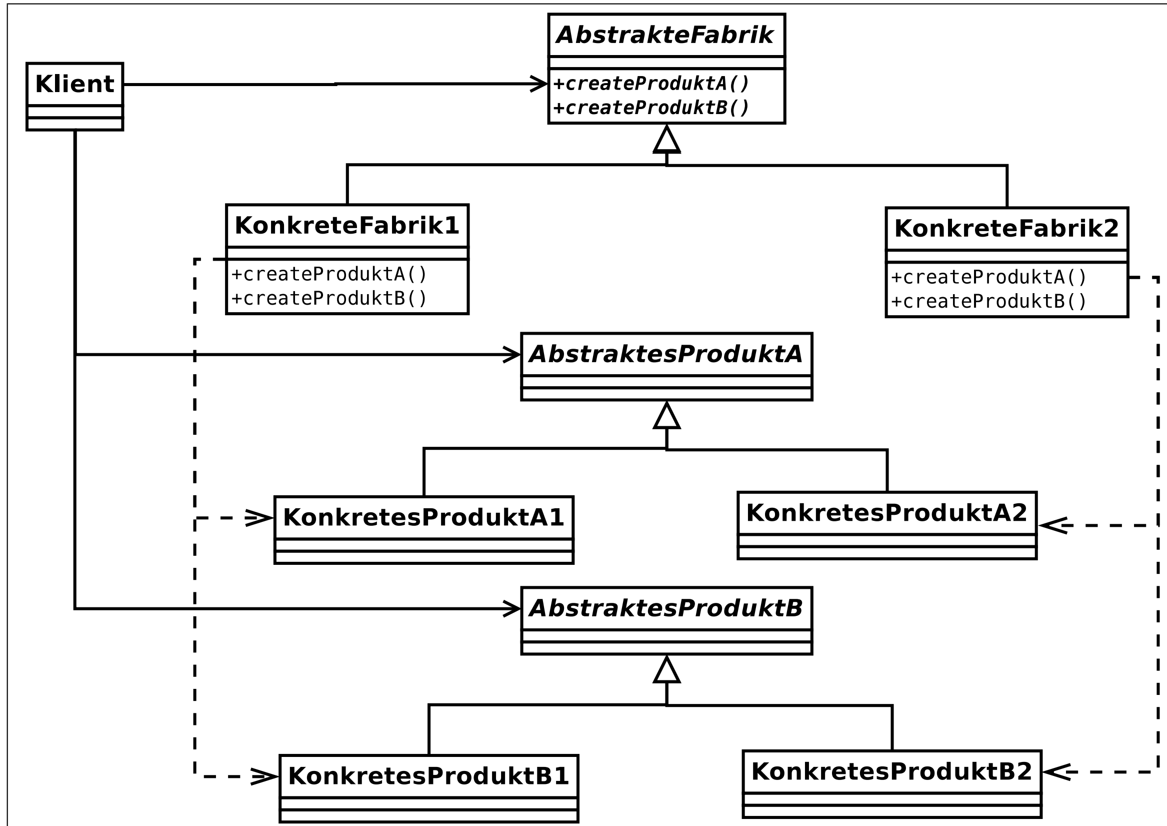

Abbildung 2.5: Klassendiagramm des Abstract Factory Entwurfsmusters

gramm lassen sich die Eigenheiten des Entwurfsmusters leicht darstellen. Die konkreten Fabriken erben und implementieren die Methoden der abstrakten Fabrik. Durch die Implementation werden sie zur Produktion von Objekten befähigt. Die konkreten Fabriken implementieren die verschiedenen Produktfamilien um ein Produkt herzustellen. Die konkreten Produkte erben und implementieren die Eigenschaften der ihr übergeordneten Produktfamilie.
Die Klient-Klasse nutzt eine der konkreten Fabriken um einen vollständigen Satz an Produkten zu produzieren, muss aber niemals selbst ein Produktobjekt instanzieren.

2.5 Multi Tier Applikation

Die Multi-Tier-Architektur ist das Prinzip einer Struktur von Softwaresystemen. Es handelt sich dabei um Eine Client-Server Architektur, bei der Präsentation, Applikationsablauf und Datenmanagment von einander logisch getrennte Prozesse sind. Dabei gibt es verschiedene Schichtmodelle, die weitverbreitetste ist die 3-Schicht-Architektur (3 Tier model). Dieses Modell ist zusätzlich auch ein Designkonzept nach dem eine Software aufgebaut wird.
Die 3-Schicht-Architektur hat die folgenden 3 Schichten:

- Präsentationsschicht (Presentation Tier)
  Die Präsentationsschicht ist verantwortlich für die Repräsentation der Daten, für das Annehmen von User-Eingaben und das Bereitstellen von bestimmten Kontrollmechanismen für die Benutzer-Schnittstelle. Die Präsentationsschicht wird häufig auch als Front-End bezeichnet, Sie kommuniziert ausschließlich mit der Applikationsschicht.

- Applikations-Schicht (Business Logic)
  Die Applikations-Schicht
  Die Applikations-Schicht stellt die gesamte Geschäftslogik, beinhaltet alle Verarbeitungsprozesse und kontrolliert die Applikationsfunktionalität. Hier ist das eigentliche Geschäftsmodell mit allen beteiligten Geschäftsobjekten mit deren Logik implementiert. Sie bildet ein konsistentes Konstrukt, welches unabhängig von Präsentations- und Datenhaltungs-Schicht funktional ist.

- Datenhaltuns-Schicht (Data Tier)
  Die Datenhaltungs-Schicht kapselt den Zugriff auf persistent gehaltene Daten. Die persistente Speicherung geschieht beispielsweise in Datenbanken und Dateien. Dabei sorgt die Datenhaltungs-Schicht für den korrekten Datenaustausch und kommuniziert ausschließlich mit der Applikations-Schicht.

Diese Architektur lässt sich sowohl systemübergreifend als auch innerhalb eines Softwaresystems umsetzen. Systemübergreifend spricht man von verteilten Systemen. Die einzelnen Schichten sind nicht nur logisch, sondern auch physikalisch voneinander getrennt. Dabei lassen sich zum Beispiel die Präsentationsschicht als Client-Applikation abbilden, die Applikationsschicht auf mehrere Servern verteilen (Skalierbarkeit) und die Datenhaltungsschicht greift auf eine zentral verwaltete Datenbank zu.
Innerhalb eines Softwaresystems repräsentieren die einzelnen Schichten jeweils ein Softwaremodul und sind gemäß der Schichteneinteilung voneinander entkoppelt. Die Kommunikation zwischen den Modulen folgt dabei dem vorgegebenen Ablauf.
Um die Kommunikation zwischen und innerhalb der Schichten definiert sowie kontrolliert ablaufen zu lassen, kann sich sogenannter Enterprise-Design-Pattern [6] bedient werden. Diese sind zum größten Teil von Martin Fowler und Grady Booch entwickelt worden und entstammen der "Java-Welt". Diese Entwurfsvorlagen sind aber größtenteils universell einsetzbar. Es existieren für jede Schicht mehrere Design Pattern, die für diese Arbeit bedeutendsten seien hier kurz erwähnt.

[6] http://corej2eepatterns.com/Patterns2ndEd/

- Front-Controller
  Der Frontcontroller ist ein zentralisierter Eingangspunkt für die Präsentations-Schicht. Dieser nimmt alle Anfragen an die Schicht entgegen und leitet diese an die Applikation weiter und/oder generiert eine Benutzerfreundliche Ausgabe.

- Business Delegate
  Der Business Delegate ist der zentralisierte Eingangspunkt für die Applikations-Schicht. Alle Anfrage an die Logik werden an den Business Delegate gestellt. Die Applikationslogik wird "versteckt", bei einem Zugrifff muss keine weitere Kenntnis der Business-Logik vorhanden sein. Der BD delegiert den Aufruf an einen entsprechenden Service. Zusätzlich fängt er alle Logik-Fehler ab und entscheidet dabei ob ein Fehler z.B. dem User angezeigt wird.

- Data Access Object (DAO) Das Data Access Object abstrahiert und kapselt jeden Zugriff auf persistente Daten. Es verwaltet die Verbindung zur Datenquelle um Daten zu erhalten und zu speichern.

2.6 Openoffice/Libreoffice

LibreOffice ist ein freies Softwarepaket welches unter der GNU LPGL Lizenz veröffentlicht wird. LibreOffice wird Plattformübergreifend für Linux, Windows und Mac angeboten, aufgrund des freien Quelltextes kann es sogar auf weitere Systeme wie Solaris o.ä. portiert werden.
Libreoffice ist eine eigenständige Abspaltung des OpenOffice-Projektes. Libreoffice 3.4 wird hier als Teil des Frontends des Projektes genutzt. Um weitere Konfusionen zu vermeiden: Openoffice und Libreoffice werden im Folgenden Synonym gebraucht, bezeichnen jedoch immer die benutzte Libreoffice 3.4 Distribution. Dies ist nötig, da im Laufe der Implementation nur allzu häufig auf die Dokumentation von Openoffice zurückgegriffen werden musste (siehe Stand der Dokumentation).

Die Verwendung von LibreOffice war Teil der Aufgabenstellung dieser Arbeit. Die Nutzung vor allem als Teil des User-Interface ergab sich erst im Laufe der Zeit. Der Vorteil der durch die Übertragung von Daten in ein Libreoffice-Calc Arbeitsblatt entsteht, ist das der Benutzer seine gewohnte Arbeitsumgebung nicht verlassen muss (Usability). Daten können z.B. mit Calc eigenen Funktionen sortiert, als Graph dargestellt oder als Datenblatt gespeichert werden.
Eine ursprünglich angedachte Funktionalität, welche aus den empfangenen Daten automatisch Graphen erstellt würde die erreichte Flexibilität nur wieder einschränken.

2.6.1 UNO-Schnittstelle und C++ Einbindung

Die UNO-Schnittstelle ist die von Libreoffice zur Verfügung gestellte API. Der Name kommt von **U**niversal **N**etwork **O**bjects. Es handelt sich hierbei um eine universelle Schnittstelle, welche neben C++ sowohl Java und C# bedienen kann. Dieses geschieht jeweils Syntaxspezifisch. Eine ursprünglich für diese Arbeit vorgesehene Einbindung in C wurde aufgrund der Tatsache verworfen, dass OpenOffice keine reine C-Schnittstelle zur Verfügung stellt. Eine Einbindung in C-Code wäre möglich, ließe sich aber wegen der Vermischung von prozeduralen und objektorientierten Ansätzen nur sehr schwer umsetzen.[7]

2.6.1.1 Stand der Dokumentation

Die Dokumentation der UNO-Schnittstelle für die C++ Einbindung ist in einem desolaten Zustand. Dies ist wohl vor allem an dem Umstand geschuldet, dass Openoffice alias Libreoffice komplett in Java entwickelt worden ist und somit auch die Dokumentierung der Java API mehr Aufmerksamkeit bekam. Diese API unterscheidet sich jedoch in vielem Elementar von der C++-Syntax und Funktionalität, so dass keine Assoziationen möglich sind.
Implementationen in dieser Arbeit beziehen sich teilweise auf die Dokumentationen des SDK 1.1, wobei mit dem SDK 3.4 gearbeitet wurde. Hier lagen jedoch keine oder nur unzureichende Dokumentationen vor. [8]

[7] Eine Sprachanbindung für C ist angedacht, und es werden hierfür Entwickler gesucht. Siehe http://www.openoffice.org/de/doc/faq/api/

[8] Inzwischen wurde Version 3.5 des Software Development Kits released, dieses ist in weiten Teilen besser dokumentiert. http://api.libreoffice.org/docs/tools.html

2.7 MySQL

Bei MySQL handelt es sich um ein Datenbankpaket, welches unter der GPL-Lizenz verteilt wird. MySQL hat sich in den vergangenen Jahren vor allem im Zusammenhang mit dynamischen Webseiten etabliert. Dabei steht es für nahezu alle Betriebssysteme zur Verfügung. Es handelt sich jedoch nicht ausschließlich um eine Datenbank für Web Applikationen. MySQL kann z.B. per ODBC-Schnittstelle unter Windows für Aufgaben rund um das Einsatzgebiet relationaler Datenbanken eingesetzt werden. [1, Vorwort]
MySQL zeichnet sich durch eine Client/Server-Architektur aus, welche vor allem für eine hohe Benutzerzahl von Vorteil ist. MySQL unterstützt die **S**tructured **Q**uery **L**anguage und folgt dabei dem ANSI-SQL /92 Standard mit einigen Einschränkungen. Es stehen APIs für die verschiedensten Programmiersprachen zur Verfügung. MySQL gilt allgemein als sehr schnelles Datenbanksystem. Daten werden in Relation zueinander über einzigartige Schlüsselbeziehungen verknüpft und in Tabellen hinterlegt und aus diesen abgefragt.

Beispiele

Ein einfaces SQL-Query sieht folgendermaßen aus:

```
SELECT *
FROM TestTabelle;
```

Dieses Listet alle Daten der Tabelle `TestTabelle` auf. Zum besseren Verständnis sollten alle Schlüsselwörter in der SQL-Syntax groß geschrieben werden.

Ein weiteres SQL-Query zur Abfrage aus verknüpften Tabellen könnte wie folgt aussehen:

```
SELECT Nebentabelle.Text, Haupttabelle.Text
FROM Haupttabelle, Nebentabelle
WHERE Haupttabelle.ID = Nebentabelle.ID
```

Die Ausgabe gibt alle Texte der Nebentabelle sowie den zugehörigen Text der Haupttabelle aus. Die Verknüpfung erfolgt per ID. Datensätze mit fehlerhafter oder ohne Verknüpfung werden nicht ausgegeben.

Im Kapitel 3 wird noch einmal auf die Strukturen und Tabellen eingegangen.

2.8 Doxygen

Doxygen ist ein Dokumentierungswerkzeug für Objektorientierte Softwareentwicklung, welche es dem Entwickler schon zur Entwicklungszeit erlaubt, Methoden, Rückgabewerte, Parameter und allgemeine Anmerkungen zu schreiben. Diese sollten regulär dem Verständnis des Quellcodes dienen. Der Vorteil dieser Art der Dokumentierung ist, dass eine Dokumentation des Quelltextes immer aus dem tatsächlich aktuellsten Quellcode generiert wird, und somit die Fehleranfälligkeit bei der Dokumentationsübertragung in ein anderes System umgangen wird.
Doxygen wurde während der Entwicklung benutzt um den kompletten Quelltext zu dokumentieren. Eine komplette Dokumentation des Quelltextes ist der Arbeit beigefügt.

Beispiel

```
1  /**
2   * @class         Berechne
3   * @brief         Exemplarische Klasse, die eine Berechnungsmethode zur Verfügung stellt.
4   */
5  class Berechne{
6
7      /**
8       * @brief   Exemplarische Funktion
9       *          Diese Funktion gibt den übergebenen Parameter auf der
10      *          Konsole aus, multipliziert ihn mit 2 und gibt ihn zurück
11      *
12      * @param        faktor    Auszugebender Parameter
13      * @return       int       Gibt faktor als integer zurück, nachdem dieser verdoppelt worden ist
14      *
15      */
16    int malZwei(int faktor)
17    {
18        cout<<"Faktor: "<<faktor;
19        faktor *=2;
20        return faktor;
21    }
22 }
```

Die Kommentare und Beschreibungen im Quelltext werden in HTML-Code umgesetzt. Auch Vererbungen und weitere Beziehungen können dargestellt werden.

3 Entwurf Multi Tier Applikation

Der ursprüngliche Ansatz zur Realisierung dieses Projektes war eine Implementierung in prozeduralem C. Aufgrund der Tatsache, das LibreOffice für reines C keine API bereitstellt, sondern nur für C++, wurde dieser Ansatz rasch verworfen. Weitere Vorteile, die für C++ sprachen sind unter anderem die Hardwarenähe und die objektorientierte Programmierung.
Der komplett objektorientierte Ansatz wurde gewählt, da sich dieses Projekt somit für zukünftige Anwendungsfälle optimal vorbereiten ließ. Hierdurch soll maximale Flexibilität, Wartungs- und Erweiterbarkeit erreicht werde. Objektorientiert lassen sich Entwürfe von Software sehr gut mit Hilfsmitteln wie UML und Entwurfsmustern erstellen.

Sowohl die Datenbank MySQL als auch LibreOffice sind vorgegeben, lassen sich durch ein objektorientiertes Design im Nachhinein leicht austauschen. All diese Faktoren wurden in dem folgenden Entwurf berücksichtigt.

Weitere Spezifikationen, z.B. bzgl. konkreter Hardware sind zur Zeit nicht vorhanden. Durch flexible Gestaltung ist jedoch auch hier eine schnelle Anpassung möglich. Um die Lauffähigkeit zu gewährleisten wird auf simulierte Daten zugegriffen. Aus diesem Grund sind viele der im Entwurf und in der Implementation erstellten Strukturen abstrakt gehalten.

3.1 Entwicklungsumgebung

Nach der Entscheidung, die Umsetzung in C++ zu realisieren, standen diverse Entwicklungsumgebungen in der näheren Auswahl. Da die Entwicklung auf einem Unix-ähnlichen System geschehen sollte, verblieben Netbeans, Eclipse und vim in der engeren Auswahl. Nach dem Testen der Funktionsweise, Benutzerfreundlichkeit und im Zusammenhang mit C++ erwies sich Netbeans als die scheinbar beste Wahl zur Umsetzung. Beispiel Codecompletion, siehe 2.1.

3.2 Namenskonvention, Codestyle und Dokumentation

Namenskonventionen sind in praxisorientierten Entwicklungen von großer Wichtigkeit. Sie dienen dem Verständnis von Quellcode, machen diesen leichter lesbar und erleichtern die Dokumentation. Neben einer Namenskonvention ist es ebenso unerlässlich einen Codestyle d.h. eine syntaktische Anordnung, die vorab spezifiziert und eingehalten werden muss. Hierbei ist eine integrierte Entwicklungsumgebung ein großer Vorteil, da sie diese Spezifikation leicht umsetzen kann.

Der benutzte Codestyle basiert auf [15].
Zur Dokumentation soll das in Kapitel 2.8 vorgestellte Doxygen-System verwendet werden. Hierzu sind alle Kommentare des Quelltextes einheitlich zu gestallten. Bei der Kommentarsyntax wird, in diesem Fall den Regeln von Javadoc gefolgt. [16]

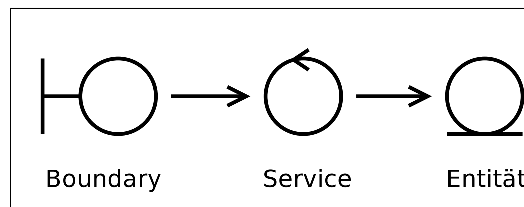

Abbildung 3.1: Vereinfachte Applikationsübersicht

In dieser Abbildung ist das Programm auf seine abstrakteste Ebene heruntergebrochen. Die "Grenze" symbolisiert das Frontend. Die Businesslogik wird durch den Service repräsentiert und eine Entität stellt ein persistentes Datenobjekt dar.

3.3 Applikationsübersicht

Der Entwurf der Applikation soll mit der in Kapitel 2.5 dargestellten Architektur realisiert werden. Der Entwurf der 3 Schichten mit Ihren Funktionen wird in diesem Kapitel mittels Klassendiagrammen (siehe 2.4.2) dargestellt. Aus den entwickelten Klassen lässt sich dann im nachfolgenden Kapitel 4 die Implementationen sehr leicht verwirklichen.
Als Grundlage wird das Enterprise-Design-Pattern in einer vereinfachten Darstellung herangezogen. In der auf das Wesentliche reduzierten Darstellung wird die Funktionsweise grob skizziert.

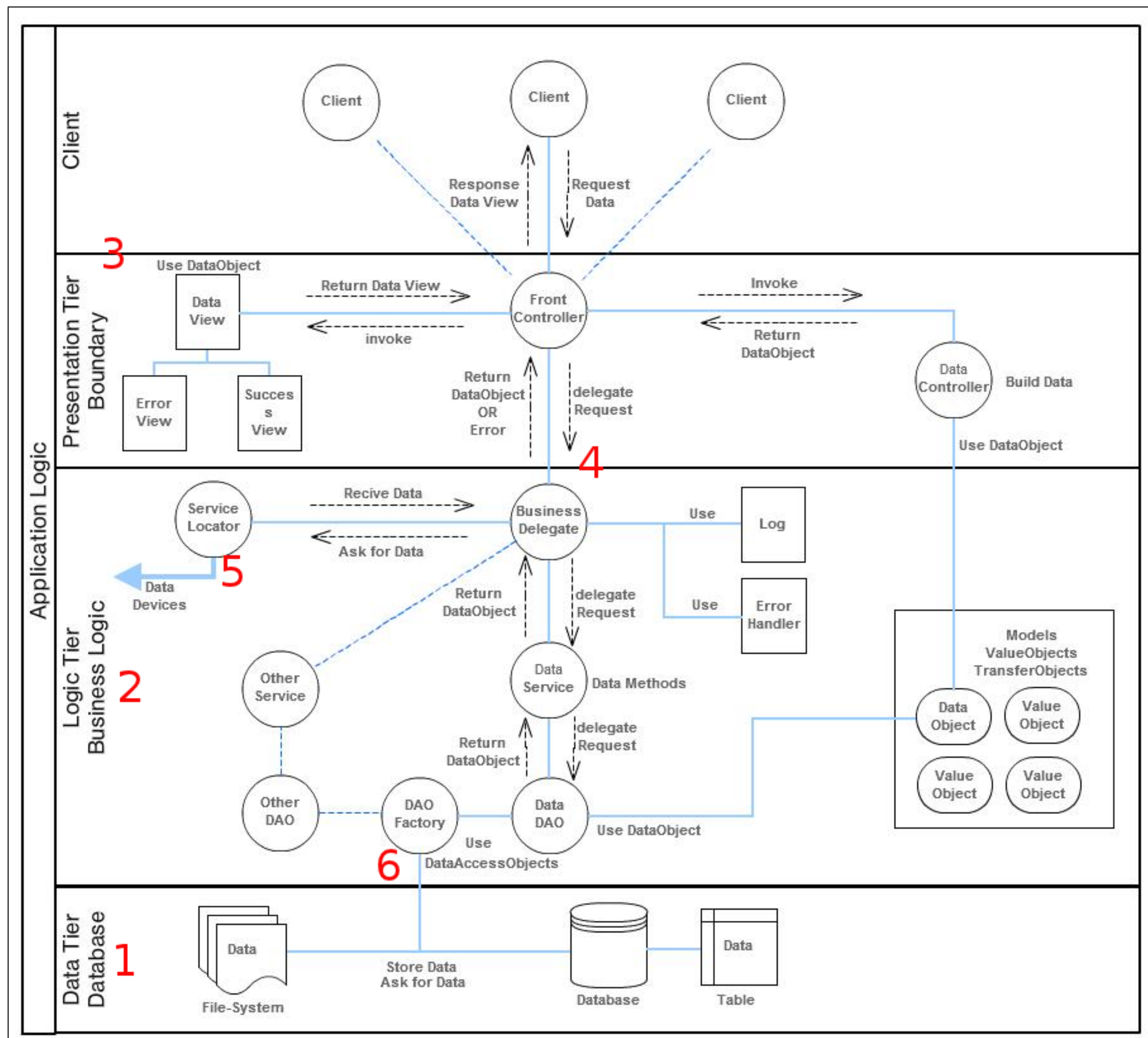

Abbildung 3.2: Multi Tier Application Übersicht

3.3.1 Datenbank Request Lauf

Eine Anfrage (request) des Benutzers (Client) wird an den Frontcontroller gestellt. Der Froncontroller ruft den für die Anfrage zuständigen Controller auf und leitet die Anfrage weiter an diesen weiter. Der Controller erzeugt ein für die Anfrage benötigtes Value-Objekt (Data-Object). Die Anfrage wird mit diesem Objekt an die Business-Logik geschickt.
Die Businesslogik nimmt die Anfrage am BusinessDelegate entgegen und entscheidet anhand der aufgerufenen Methode, wie weiter verfahren werden soll. Wird für die Weiterverarbeitung des übergebenen Anfrage-Objekt ein Service (Datenservice) benötigt, wird dieser aufgerufen und die Anfrage weitergegeben. Der Datenservice entscheidet nun seinerseits, was mit dem Anfrage-Objekt geschehen soll. In diesem Beispiel wird ein Objekt zur persistenten Speicherung an das zuständige DAO-Objekt weitergegeben. Die zuständige DAO-Fabrik-Klasse kann jetzt die Verbindung zur Datenbank herstellen (DAO-Factory-spezifisch) und das Objekt wird gespeichert.

Bei erfolgreicher Speicherung wird eine Erfolgsmeldung durch dir zuvor durchlaufene Delegationskette an den Frontcontroller zurückgegeben. Dieser erstellt einen Success-View (Erfolgsmeldung) und gibt ihn dem Client zurück. Der Request-Lauf für Datenbankzugriff ist beendet.

3.3.2 Echtzeitdaten Request Lauf

Für Echtzeitdatenerfassung mit Datenumleitung in Libreoffice wird über den Business Delegate der Service Locator entsprechend der Anfrage gestartet. Die angegebenen Daten-Devices werden instanziert. Die Daten werden jetzt automatisch von einem Hardware-Device aufgenommen und an den Libreoffice-Adapter gegeben.

3.4 Presentation Tier

Die Presentation Tier ist für die User-Eingabe und User-Ausgabe Steuerung zuständig. Alle Befehle sollten über Commands getriggert werden. Zusätzlich sollen Eingabe und Ausgabe eingeschränkt und von einander getrennt werden. So soll das Absetzen eines Commands erst einmal über Eingabe in der Console erfolgen, diese könnte zusätzlich noch Systemnachrichten anzeigen (Debug Nachrichten). Die Ausgabe soll über ein zu entwickelndes Device direkt nach LibreOffice erfolgen. Alternativ soll ausgewählt werden können: direkt in die Datenbank schreiben oder von der Datenbank nach Libreoffice umleiten. Auch weitere, zu entwickelnde Optionen sollen gesetzt werden können.

Die Presentation Tier verfügt über verschieden Controller zur Befehlsaufnahme und Weitergabe. Views werden für die Ausgabe von Systemnachrichten oder Daten herangezogen.

3.4.1 Controller

Ein Controller soll Benutzereingaben entgegen nehmen und diese auswerten. Für jeden Anwendungsfall muss ein eigener Controller (und ein sich ergebender View, siehe 3.4.2) zur Verfügung gestellt werden. Dieser bereitet je nach Useranfrage die Daten für die Applikationslogik vor. Der Controller wird Anfragen in die Applikationslogik delegieren. Für Ausgearbeitete Controller kann ein Interface entwickelt werden.

3.4.1.1 FrontController

Der FrontController soll als Hauptcontroller alle Anfragen annehmen und diese innerhalb der Presentation-Tier an den zuständigen Controller weiter delegieren. Der Front-Controller ist also das Zentrale Element der Präsentationsschicht. Es soll nur ein Front-Controller in der Applikation vorhanden sein (Singleton, siehe 2.2.4.3)

3.4.1.2 DatenController

Der DatenController wird für Anfragen ein entsprechendes Datenobjekt erstellen und dieses an den im zugehörigen View oder an die Businesslogik weitergeben. Er ist somit verantwortlich für das Erstellen einer Ansicht der Daten (DatenView) und für das Erhalten von Daten aus der Logik.

3.4.1.3 LogController

Der LogController wird für das Logging und die Anzeige (LogView) noch zu definierender Ereignisse zuständig sein (Fehler etc.).

3.4.1.4 AdapterController

Der AdapterController übernimmt die Verwaltung der Anbindung zur Anzeige und Bearbeitung der Daten in Libreoffice. Er hat dabei keinen eigenen View, sondern benutzt

Libreoffice als Anzeige.

3.4.2 Views

Views werden für die Darstellung und die Entgegennahme von Benutzeraktionen zuständig sein. Ein Interface für die Mindestimplementation wird entwickelt.

3.4.2.1 DatenView

Der DatenView wird für die Präsentation von Daten-Objekten zuständig sein. Er wird dabei von dem DatenController herangezogen. Er erhält eine error und eine success, sowie u.U. noch weitere Meldungen.

3.4.2.2 AdapterView

Der AdapterView wird durch Libreoffice verwirklicht.

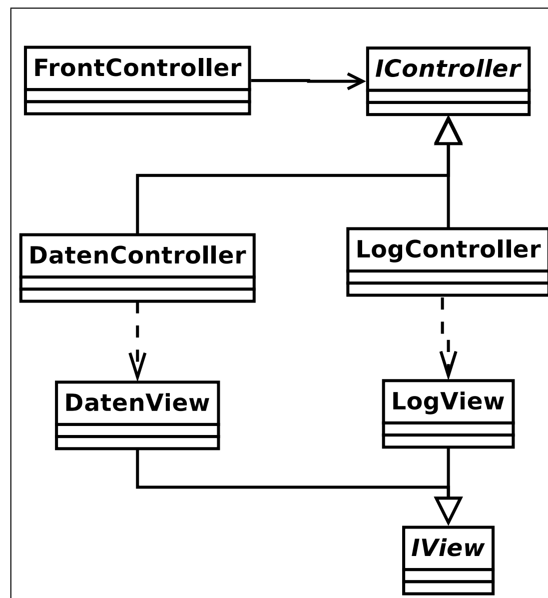

Abbildung 3.3: Entwurf Frontend: Controller und Views

3.5 Application Tier

Die Application Tier soll die Geschäftslogik enthalten. Diese wird so erstelle, dass sowohl Frontend, als auch Datenbank ausgetauscht werden können. Die Logik soll jederzeit um weitere Geschäftsprozesse erweiterbar sein. Enthalten sein werden folgende Designelemente (nach [13]):

- BusinessDelegate-Pattern
- ServiceLocator-Pattern
- DAOFactory (DAO- und Factory-Pattern)
- Value-Objects & Services
- Entity-Objects

3.5.1 BusinessDelegate

Der BusinessDelegate wird der Zentrale Eingangspunkt für alle Anfragen (requests) und Antworten (response) aus der Businesslogik. Die entsprechenden Services sollen hier instanziert werden. Methoden im BusinessDelegate werden im Laufe der Implementation hinzugefügt. Errorhandling und Logging der Logik sollen hier übernommen werden. Der BusinessDelegate wird als Singleton umgesetzt (siehe 2.2.4.3)

3.5.2 ServiceLocator

Der ServiceLocator wird für das Handling der Devices zum Lesen und Schreiben der Echtzeitdaten verantwortlich sein. Hierzu wird er als Thread realisiert und instanziert die Devices ebenfalls in eigenen Threads. Er übernimmt das Threadhandling der Devices sowie deren Mutex-Objekt.

3.5.2.1 Threading

Für die Threads werden, entspreched der an Sie gestellten Anforderungen, diverse Interfaces und Klassen entworfen. (Ähnlich Beispiel 2.3)

3.5.2.2 Devices

Es wird zwei Arten von Devices geben: Ein Reader- und ein WriterDevice. Die Bezeichnung derselbigen ist dem Zugriff entsprechend, den sie auf das Mutex haben, definiert. Das WriterDevice ist somit dasjenige, welches die Hardware auslesen soll. Für die Devices wird ein Standard-Device erstellt, das vordefinierte Funktionalitäten mitbringen sollen, welche erweitert werden können.
In dem WriterDevice werden zur Veranschaulichung Methoden zur Simulation eingebaut.
Die Devices werden eine Strategie implementieren (Strategy-Pattern, siehe 2.2.4.6), mit Hilfe derer sie die erhaltenen Hardwaredaten konvertieren können. Dies garantiert nach Implementierung konkreter Messgeräte Flexibilität beim Auslesen.

3.5.3 DAOFactory (DAO- und Factory-Pattern)

Die DAOFactory wird als Anbindung zum Data-Tier entwickelt. Sie wird deshalb, obwohl Teil der Applikationslogik, im entsprechenden Kapitel (3.6)erläutert.

3.5.4 Value-Objects + Services

Value-Objects sind für die temporäre Datenhaltung zu entwickeln. Für die eigentlichen Messdaten entsprechend dem Datenbankentwurf. Weiterhin für die Log-Funktionalität. Zu jedem Value-Object gehört ein entsprechender Service, der für dessen Handhabung zuständig ist. Der Service selbst soll vom BusinessDelegate instanziert werden.

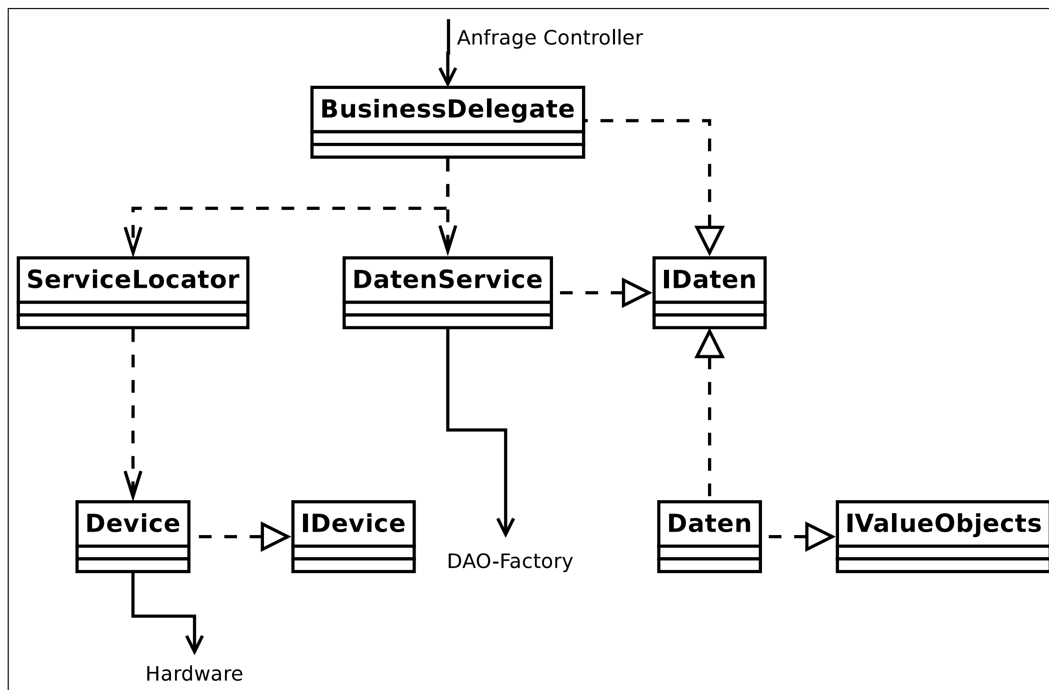

Abbildung 3.4: Abstrakter Entwurf Application Tier

3.6 Data Tier

Hier sollen Daten persistent gespeichert werden. Die Data-Tier enthält die Treiber für die Kommunikation mit der MySQL-Datenbank. Außerdem ist eine Umschaltung zwischen verschiedenen Möglichkeiten gewünscht, diese erfolgt via DAO-Factory. Für die Übertragung der Daten werden DAO-Objekte benötigt die über eine CRUDS Funktionalität verfügen.

3.6.1 DAO-Factory

Die DAO-Factory bestimmt welche Art von persistenter Speicherung benutzt wird. Sie wird nach dem Abstract-Factory-Pattern entworfen (siehe 2.2.4.1) Zur Speicherung stellt diese die Entsprechende Schnittstellen, Treiber und die DAO-Objekte bereit. Die DAO-Factory soll zu Programmbeginn einmal instanziert werden, um Defaultwerte zu erhalten. Standard wird hier der Typ MySQL eingestellt. Zur Laufzeit könnte auf einen anderen Speichertyp umgestellt werden. Hierzu wird die File-Funktionalität in rudimentären Ansetzen hizugefügt.

3.6.2 Data Access Object

Ein Data Access Object soll über die konkreten Methoden (CRUDS) zur Speicherung von Daten verfügen. Es wird von der DAO-Fabrik bereitgestellt.

3.6.2.1 CRUDS (Create Read Update Delete Search)

CRUDS steht für die Methoden create() read() update() delete() und search(). Diese sind die Mindestimplementation für DAO Objekte. Die Funktionalität wird entsprechend Speichertyp von der Factory in den DAO-Objekten bereitgestellt.

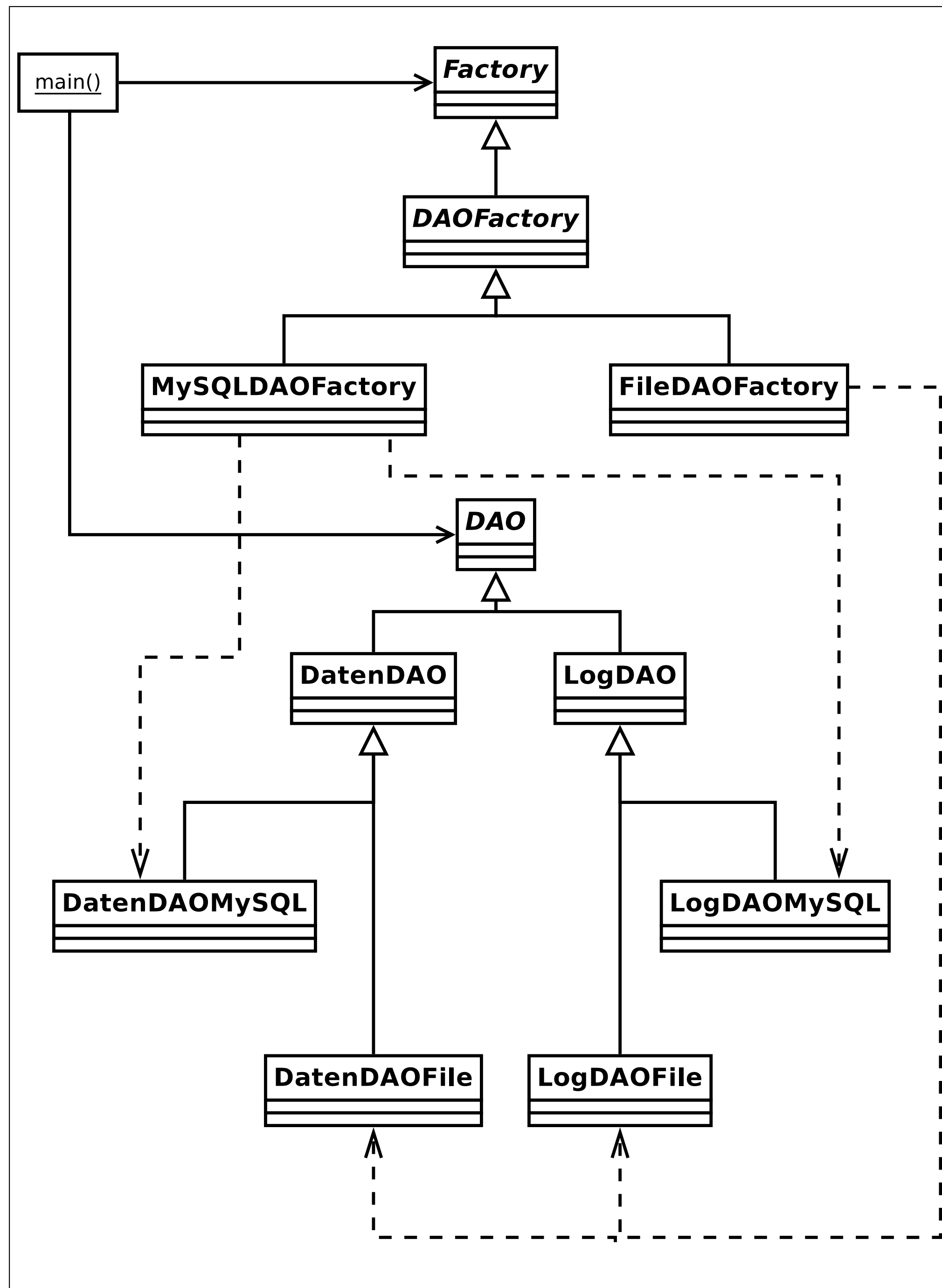

Abbildung 3.5: DAO-Factory Entwurf

3.7 Datenbank

Die Datenbank ist wie folgt entworfen worden:

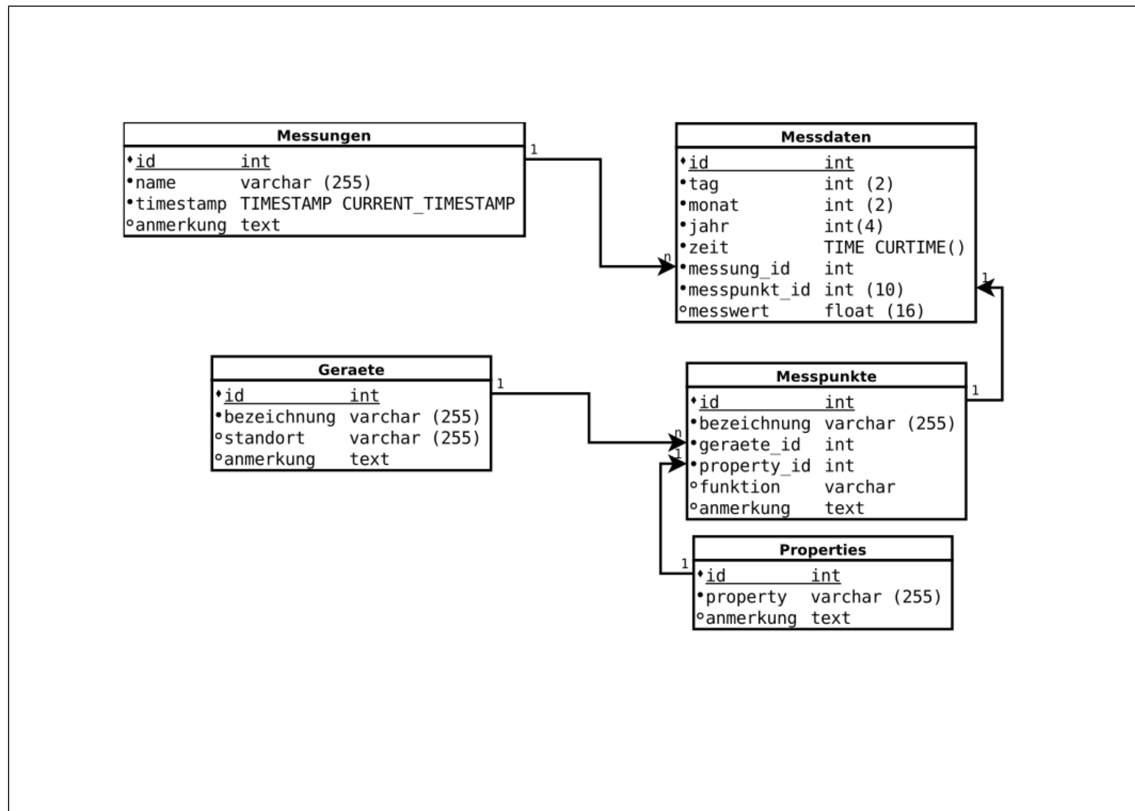

Abbildung 3.6: Datenbank Entwurf, Entity Relationship Model

Der Fokus liegt auf den Messdaten. Daten können einfach **einer** individuellen Messung zuordnet werden. Dazu wird den Messdaten eine `messung_id` hinzugefügt über welche einer Messung ein `name` referenziert werden kann. Weiterhin wird ein Zeitstempel hinzugefügt, welcher aus `tag`, `monat`, `jahr` und der aktuellen `zeit` besteht. Der eigentliche Messwert wird unter `messwert` gespeichert. Die Eigenschaft des Messwertes wird über die `messpunkt_id` implementiert.
Messpunkte bezeichnen ein Messgerät (`bezeichnung`), das an einem wie auch immer gearteten Gerät (`geraet_id`) zur Messung verwendet wird. Die Maßeinheit in der am Messpunkt gemessen wird, kann über `property_id` eingefügt werden. Des Weiteren besitzt Messpunkte noch `funktion`, in welchem die Funktion des Messpunktes hinterlegt werden kann. Auch eine `anmerkung` kann hinzugefügt werden.
`Properties` besitzt einzig `property` für die Maßeinheit, sowie ein `anmerkung` Feld und wird über die ID referenziert.
`Geraete` schlussendlich stellt neben der `bezeichnung` und der `anmerkung` noch einen `standort` zur Verfügung.

3.8 Libreoffice Adapter

Der LibreOfficeAdapter soll die API von Openoffice in einer Klasse bündeln. Hierbei sollen nur Methoden, die für die Darstellung von Daten in LibreOffice relevant sind, implementiert werden. Es handelt sich um ein Adapter-Entwurfsmuster (siehe 2.2.4.4). Für die Benutzung von LibreOffice innerhalb der Applikation wird dieser Adapter herangezogen.

3.9 Anwendungsfälle

Es sind im Voraus verschiedene Anwendungsfälle denkbar, die eine unterschiedliche Herangehensweise in der Datenerhebung sinnvoll erscheinen lassen.
Ein Unterschied findet sich hauptsächlich in den Abständen, mit welcher die Daten erhoben werden sollen. Einerseits eine Anwendung, die von der hardwarenäheren Programmierung und Einbindung der Sensorik in C++ profitiert (Hier werden die Messabstände nur durch die Laufzeit begrenzt). Andererseits Anwendungsfälle, die eine langfristige Datenerhebung voraussetzen, bei denen die Abstände zwischen den Messpunkten nicht der kritische Parameter ist.
Diese unterschiedlichen Anwendungsfälle machen einen Unterschied mit dem Umgang der Datenbankeinbindung erforderlich. Bei echtzeitkritischen Anwendungen würde ein Zugriff auf die Datenbank das Programm erheblich verlangsamen. Bei einer Langfristigen Erhebung ist die Laufzeit nicht kritisch und Daten könnten u.U. direkt in die Datenbank geschrieben werden.

Die Datenbankanwendungsfälle werden sich je nach Datenerhebungsstrategie ein wenig unterscheiden. Der Ablauf an sich bleibt jedoch immer gleich. Außerdem sollen gespeicherte Datensätze aufgerufen und ins User Interface übergeben werden können.

4 Implementierung

Das Kapitel Implementation beschreibt die Umsetzung des Entwurfs. Systemvoraussetzungen werden beschrieben, um Reproduzierbarkeit zu gewährleisten. Vorbereitungen für einen Reibungslosen Ablauf werden getroffen. Es wird auf Differenzen zwischen Entwurf und Implementation hingewiesen. Dies ist für eine eventuelle Erweiterung des Projektes unerlässlich, da der Entwickler Kenntnis über diverse Problemlösungen, Abweichungen vom Entwurf, sowie deren Grundlage erhält. Abschließend wird noch einmal explizit auf die, für eine schnelle Anpassung wichtigsten Punkte hingewiesen. Hierdurch kann das Programm in seiner Grundfunktionalität uneingeschränkt, schnell an neue Bedürfnisse angepasst werden.

Eine vollständige Beschreibung sämtlicher in diesem Projekt befindlichen Klassen, Interfaces, Methoden und Variablen inklusive Vererbungshierarchie und Abhängigkeitsbeziehungen ist in der mit Doxygen erstellten Dokumentation in HTML verfügbar und liegt dieser Arbeit auf dem Datenträger bei.

4.1 Vorbereitungen und Systemvoraussetzungen

Das System auf welchem entwickelt wurde ist Ubuntu-Linux 11.10 32bit.
Folgende Systempakete haben sich dabei als relevant ergeben und **müssen** vorhanden sein, oder installiert werden:

1. g++
2. cmake
3. libc6-dev
4. libboost-dev
5. libstdc++.5

Diese Voraussetzungen sind weiterhin für das Projekt relevant:

- MySQL-Connector und eine laufende MySQL-Datenbank, sowie Programmanpassungen (siehe 4.7.6)
- LibreOffice SDK, Libraries
- Einrichtung Integrierte Entwicklungsumgebung

Nach Installation der Pakete kann der MySQL-Treiber kompiliert werden.

4.2 MySQL-Connector

Der MySQL-Connector ist der für die Anbindung von MySQL zuständige Systemtreiber. Der Treiber, der binär bei http://mysql.com herunterzuladen ist, ist unter Umständen nicht mit den aktuellen C++ Bibliotheken zu verwenden (aktuell glib 3.2.3). In diesem Falle ist der Source-Code des MySQL-Connectors von [6] herunterzuladen und selbst zu kompilieren. Eine komplette Anleitung findet sich unter [14]

4.3 LibreOffice SDK, Libraries

Das Systempaket libreoffice-dev muss neben LibreOffice installiert sein. Sollte zur Compilezeit ein Fehler mit den LibreOffice Libraries entstehen, so lassen sich diese noch einmal explizit mit dem Befehl:

```
1 cppumaker -Gc -BUCR -O<includes_path> <ooffice_path>\URE\misc\types.rdb
2             <ooffice_path>\Basis\program\offapi.rdb
```

erstellen, das Einbinden in eine intergrierte Entwicklungsumgebung wird im nächsten Abschnitt erklärt.

4.4 Einrichten der intergrierten Entwicklungsumgbung Netbeans

Als Entwicklungsumgebung kommt Netbeans 6.9 zu Einsatz. Die Einrichtung der Entwicklungsumgebung wird im Folgenden geschildert, da sie existentiell für jegliche weiterführende Implementierung ist.

4.4.1 General (1)

Zuerst muss darauf geachtet werden, dass immer mit der *Release* Konfiguration gearbeitet wird, dies ist erforderlich für die MySQL-Treiber. Weitere Maßnahmen unter General sind vorerst nicht nötig.

4.4.2 C++ Compiler (2)

Unter C++ Compiler müssen sowohl include directories, als auch additional commandline options hinzugefügt werden.

4.4.2.1 Include directories

- `/opt/libreoffice3.4/basis3.4/sdk/includecpp`

  u.U. Erstellt per cppumaker, siehe vorheriger Abschnitt.

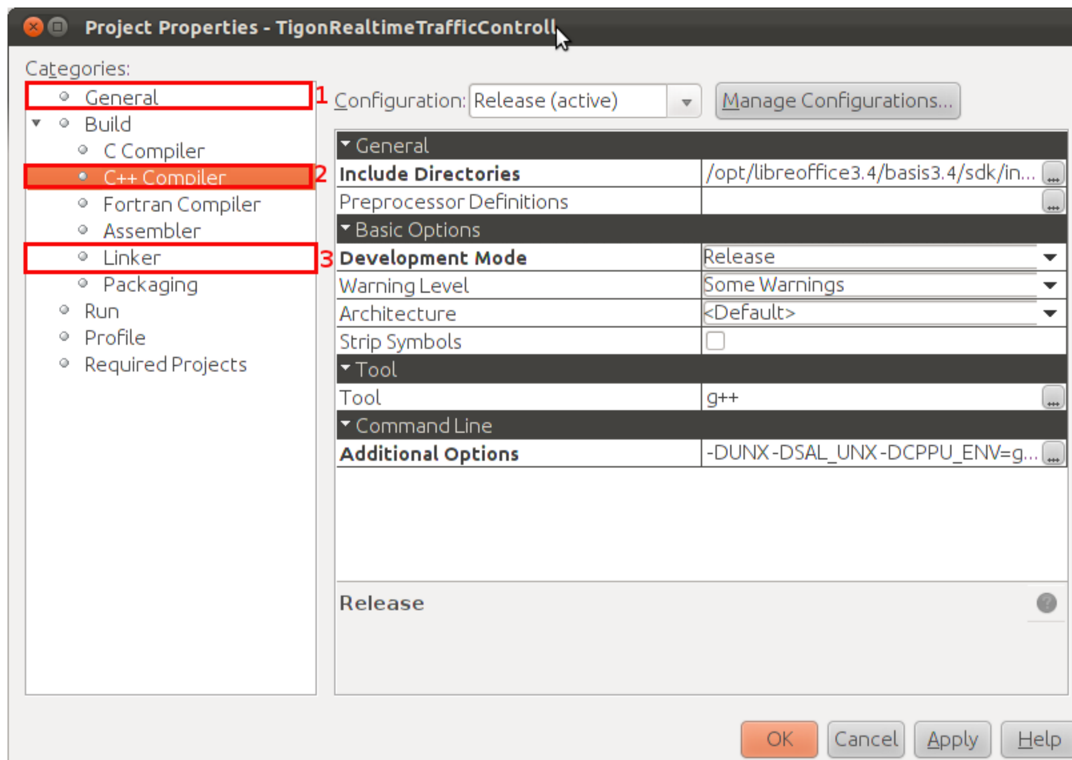

Abbildung 4.1: Netbeans Projekt Einstellungen

- `/usr/local/lib`

  Selbstkompilierte MySQL-Connector Dateien.

- `/opt/libreoffice3.4/basis3.4/sdk/include`

  Weitere Libreoffice SDK include Dateien.

4.4.2.2 Commandline Options

- `-DUNX -DSAL_UNX`

  Die UNIX-Umgebung bekanntzumachen.

- `-DCPPU_ENV=gcc3 -D__cplusplus`

  Die Entwicklungsumgebung (Hier C++ mit gcc3 Compiler) bekanntmachen

4.4.3 Linker (3)

In den Linker-Optionen müssen Additional Library Directories und Libraries hinzugefügt werden

4.4.3.1 Additional Library Diretories

- `/usr/local/lib`

  Selbsterstellte MySQL-Libraries

- `/opt/libreoffice3.4/basis3.4/sdk/lib`

  Libreoffice Software Development Kit Libraries

- `/usr/lib/i386-linux-gnu`

  Umgeht einen Fehler in der libstd5 der bei der kompilierung auftritt, wenn diese mit Commandline-Option

  `-DCPPU_ENV=gcc3`

  gestartet wird.

4.4.3.2 Libraries

- `/usr/local/lib/libmysqlcppconn.so`
  `/usr/local/lib/libmysqlcppconn-static.a`

  Selbsterstellte MySQL-Libraries

- `/opt/libreoffice3.4/basis3.4/sdk/lib/libuno_cppu.so`
  `/opt/libreoffice3.4/basis3.4/sdk/lib/libuno_cppuhelpergcc3.so`
  `/opt/libreoffice3.4/basis3.4/sdk/lib/libuno_sal.so`
  `/opt/libreoffice3.4/basis3.4/sdk/lib/libuno_salhelpergcc3.so`

  Libreoffice Software Development Kit Libraries

Eine ausführliche Auflistung aller Befehle ist im Anhang zu finden. Sie beziehen sich auf ein frisch installiertes ubuntu-System. (siehe A.1)

4.5 Presentation Tier

Die Implementation im Presentation Tier weicht in einigen Punkten von dem Entwurf ab. Diese Abweichungen sind zum Einen aus praktischen Überlegungen heraus entstanden. Zum Anderen ließ sich eine Implementation auch aus Zeitgründen nicht realisieren. Des weiteren existieren einige Funktionsbereiche, die der Verdeutlichung dienen, aber absichtlich abstrakt gehalten sind.

4.5.1 Abweichungen aus praktischen Überlegungen

Aus praktischer Überlegung heraus wurde im Frontend auf den Libreoffice-Controller und den LibreOffice-View verzichtet. Der Verzicht auf den View ergibt sich, da LibreOffice selbst den View darstellt. Außerdem ergibt sich durch die direkte Verwendung von LibreOfficeAdapter ein nicht zu unterschätzender Geschwindigkeitsvorteil. Der Controller entfällt aus eben diesen Gründen.

4.5.2 Abweichungen aus Zeitlichen Gründen

Aus zeitlichen Gründen wurde auf die Implementation eines kompletten GUI-Views verzichtet. Die Idee Libreoffice als Eingabemaske für Datenpflege zu benutzen ist zum Teil vorbereitet. Konnte aufgrund des Zustandes der API-Dokumentation aber nicht zeitgerecht ausgeführt werden.

4.5.3 Abweichende Funktionalitäten

Der Frontcontroller kann derzeit nur Kommandos aufnehmen, die vom Typ `DatenController` oder `LogController` sind. Damit ist es vorerst nicht möglich ein Command an einen anderen Controller zu senden. Eine sinnvolle Typisierung wäre nach dem Interface IController.

4.5.4 Abstrakte Strukturen

Eine Logging-Funktionalität ist komplett vorbereitet. Es müssen in den Klassen noch entsprechenden Methodenimplementationen erfolgen

4.5.5 Weitere Implementation

Eine globale CommandQueue für das Absetzen und Ausführen von Kommandos an die zuständige Controller wurde implementiert. Die Queue ist dabei nach den Entwurfsmustern Singleton (siehe 2.2.4.3) und Command (siehe 2.2.4.7) aufgebaut. Eine Callback-Methode gibt es derzeit nicht.
Eine Klasse QueryHelper ist notwendig, um aus verketteten Strings eine `map` mit Wertepaaren zu machen (key-value). Diese wird eingesetzt um einen String-Command aufzulösen. Die übergebenen Wertepaare werden dann in ein Datenobjekt umgewandelt, damit die Businesslogik die Anfrage wie gewünscht annehmen kann.

4.6 Application Tier

Die Application Tier besteht zum Großteil aus 1:1 Umsetzung des Entwurfs. Dennoch ergaben sich bei der Realisierung einige Unterschiede. Auch in diesem Abschnitt sind einige Elemente wieder bewusst abstrakt gehalten und können somit später implementiert werden.

4.6.1 Abweichung aus praktischen Überlegungen

Im Application Tier wird der LibreOfficeAdapter entgegen der strikten Trennung der Schichten bewusst im Device verwendet. Dennoch ist der Geschwindigkeitsvorteil, welcher sich hieraus ergibt unverzichtbar.
Der BusinessDelegate wurde entgegen der Spezifikationen als Singleton (siehe 2.2.4.3) definiert. Er enthält keine Logging- und Fehlerbehandlungs-Methoden.

4.6.2 Abweichungen aus Zeitlichen Gründen

Das Exceptionhandling wurde aus zeitlichen Gründen nicht vollständig implementiert, Fehler werden derzeit nicht abgefangen oder korrigiert. Die eng verwandte Logging-Struktur ist abstrakt vorbereitet.

4.6.3 Abweichende Funktionalitäten

Der ServiceLocator erfüllt nicht alle Spezifikationsanforderungen, die an ihn nach der Definition [13] gestellt werden.

4.6.4 Abstrakte Strukturen

Eine Logging-Funktionalität ist komplett vorbereitet. Es müssen in den Klassen noch entsprechenden Methodenimplementationen erfolgen

4.6.5 Weitere Implementation

Es wurden aufgrund der Tatsache, das Libreoffice in der Businesslogik angesprochen wird ein ReaderDeviceAdapter implementiert um direkt in LibreOffice zu schreiben. Ein Implementiertes ReaderDeviceMySQL nimmt den, im Entwurf vorhergesehenen Weg, über den Frontcontroller.
Sowohl Reader- als auch WriterDevice sind als Standarddevice vorhanden. Von ihnen müssen mögliche Unterklassen abgeleitet werden.
Der ServiceLocator wurde neben den Devices ebenfalls als Threadobjekt ausgeführt.
Eine Klasse TimeStampHelper wird benötigt, um alle Datumsformate, sowohl für Datenbank, als auch für Adapter vorzubereiten.
Das Interface Synchronizable ist unnötig, wird aber in einigen Klassen noch verwendet, das Interface hat keine Funktion. Die Klasse VektorMutex ist entgegen ihrer Namensgebung kein Vektor, sondern eine Queue, richtige Bezeichnung wäre QueueMutex, siehe 2.3.

4.7 Data Tier

Die Datenhaltungsschicht wurde aus der entworfenen Datenbank (siehe 3.6) aufgebaut.

4.7.1 Abweichung aus praktischen Überlegungen

Das Zeit-Feld wurde von `TIME` in einen `String` umgewandelt, um der Erfassung inklusive Millisekunden Rechnung zu tragen. Das normale SQL-TIME-Feld kann nur in Sekunden auflösen.

4.7.2 Abweichungen aus Zeitlichen Gründen

Eine komplette Implementierung mit sämtlichen Tabellen ist vorhanden, wird jedoch noch nicht benutzt. Die Umsetzung inklusive aller Tabellen ist jedoch möglich. Wird jedoch erst sinnvoll zu benutzen sein, wenn tatsächlich Messpunkte/Geräte vorhanden sind, um diese eindeutig zuzuordnen.

4.7.3 Abweichende Funktionalitäten

Derzeit keine.

4.7.4 Abstrakte Strukturen

Abstrakt ist das Speichern in Datendateien vorbereitet. Hier müssen nurnoch die entsprechenden Methoden für das Dateihandling implementiert werden.

4.7.5 Weitere Implementation

Keine.

4.7.6 MySQL Datenbank MessdatenDB

Die Implementierung der Datenbank MessdatenDB nach Abbildung 3.6 erfolgt mit dem im Anhang befindlichen SQL-File. Dieses erzeugt die MessdatenDB und alle relevanten Tabellen für die persistente Speicherung der Projektdaten. Nach dem ausführen des SQL-Files in der MySQL-Konsole steht die Datenbank zur Verfügung und kann sofort benutzt werden.
Die Definitionen der für die Datenbankverbindung benötigten Variable erfolgt in der Datei MySQLDAOFactory.h und müssen an die zu benutzende Datenbank angepast werden.
Abschnitt der geändert werden muss:

```
1 #define DB_HOST "tcp://127.0.0.1:3306"  //IP und PORT für SQL connection
2 #define DB_USER "root"          //Benutzername
3 #define DB_PASS "password"          //passwort
4 #define DB_DATABASE "MessdatenDB"   //Datenbankname
```

4.8 Spezielle Umsetzungen

Die hier aufgezeigten Umsetzungen bedürfen einer besonderen Erwähnung, da ihre Implementation sich als kompliziert herausstellte.

4.8.1 Threads

Die Implementation von POSIX-Threads erwies sich als schwierig. Der Ausbruch aus dem klassischen "Nacheinanderdenken" ist ein Herausforderung für den Programmierer. Es muss bedacht werden, dass keine Garantie auf Laufzeiten gegeben werden kann. Es sollte immer sichergestellt sein, dass Threads vernünftig synchronisiert werden. In diesem Fall wurden mit der Funktion `usleep` Verzögerungen in den Programmablauf eingebracht. Ohne Verzögerungen wurden die Threads zu schnell und es war kein vernünftiges Debugging mehr möglich. In der Konsole verwischten sich die Ausgaben derart, das keine logische Schlussfolgerung aus dem Verhalten gezogen werden konnte. Durch die Verlangsamung war es jetzt möglich, das "Arbeiten" der Threads genau zu beobachten.
Auch in Zukunft ist es wichtig, die implementierten Threads bzgl. Ihrer Tätigkeit aufeinander abzustimmen. Es kann an allen Schnittstellen (MySQL, Libreoffice und Hardwaredevice) zu noch nicht bekannten Laufzeiten kommen. Diese sind System- und Hardwarespezifisch.

4.8.2 DAO-Factory

Die Vorteile der Implementation einer DAO-Factory sind nicht auf Anhieb zu erkennen. Scheint diese das Ansprechen der MySQL-Datenbank nur unnötig zu verkomplizieren. Außerdem ergibt sich beim Implementieren einer neuen Fabrik (bsplw. für XML) ein Mehraufwand durch das Erstellen der vielen Klassen.
Vorteilhaft ist jedoch dass sich gerade durch diese Konstruktion die Implementation der konkreten Typen "verstecken" lässt. Der Entwickler benötigt keinerlei Kenntnis über die Struktur. Ein weiterer Vorteil ist die Tatsache, dass sich zur Laufzeit diese Komplexe Struktur komplett auf einen anderen Typen spezifizieren lässt. Der Mehraufwand relativiert sich mit der Größe des Projektes und mag für dieses Projekt einen überdimensionierten Eindruck machen.

4.8.3 Log

Die Implementation von Log-Methoden ist durch das ganze Projekt hindurch zu entdecken. Aufgrund zeitlicher Begrenzung konnte diese nicht komplett realisiert werden. Sie hätte der Aufzeichnung und Darstellung von Fehlern in einem Logobjekt gedient. Auch das Starten oder Beenden von Messungen kann geloggt werden. Für eine weiterführende Arbeit ist dies sicherlich einer der ersten sinnvollen Ergänzungen.

4.9 Hinweise für schnelle Anpassungen

In diesem Abschnitt sollen in Kürze die Programmabschnitte erwähnt werden, die für eine unkomplizierte Anpassung vorteilhaft sind.

4.9.1 WriterDevice

Das WriterDevice implementiert zur Zeit nur Methoden zur Simulation von Daten, nicht aber die Anbindung an vorhandene Hardware (Messpunkte). Für eine schnelle Anpassung kann von diesem WriterDevice eine Unterklasse erstellt werden, welche die Entsprechenden Methoden überschreibt. (WriterDevice.cpp)

4.9.2 PropertyStrategy

Die PropertyStrategy bietet derzeit nur eine Konvertierungsmöglichkeit von Messdaten. Es kann schnell ein neuer Algorithmus erstellt werden um Messdaten zu konvertieren. (PropertyStrategy.h)

4.9.3 User Interface

Das User Interface wird derzeit von der `main()`-Methode erstellt. Eine Anpassung ist nach dem vorhandenen Code ohne weiteres möglich, indem die gewünschten Ausgaben hinzugefügt werden. Benutzereingaben werden in dem `switch-case` Block bearbeitet. (main.cpp)

4.9.4 Serviceerweiterung

Für den DatenService kann eine gezielte Datenabfrage schnell implementiert werden. Denkbar wären Methoden wie `getByName()`, `getByID()`, `getByDate()`, die einzig ein overloading der `read()`-Methode im Service und im DAO bedeuten.

4.9.5 Datenbank

Die Variablen zum Benutzen der Datenbank werden derzeit im Quelltext definiert. Dies bedeutet, das bei Änderung von Username oder Passwort das Programm neu kompiliert werden muss.

5 Schlussfolgerungen, Fragen und Ausblicke

5.1 Fazit

Die Aufgabenstellung in Ihrer ursprünglichen Form schien zunächst einfach zu lösen. Hätten doch mit prozeduraler Programmierung einzig die Anbindung an Libreoffice und MySQL realisiert werden müssen. Eine gewisse Logik wäre hinterlegt worden, welche die eine oder die andere Anbindung anspricht. Im Nachhinein jedoch ergibt sich, Dank der objektorientierten Herangehensweise ein anderes, differenzierteres Bild.
Die Einarbeitung in die Objektorientierung und die daraus resultierende Umstellung in der Herangehensweise fiel nicht leicht. Muss doch eine Menge an eingeschliffenen Denkmustern über Bord geworfen werden. Nach einer gewissen Einarbeitungszeit und durch die Benutzung einer integrierten Entwicklungsumgebung wurden die Vorteile jedoch nach und nach sichtbar.
Die Trennung logischer Bereiche, die Kapselung von Daten, das Anwenden von Entwurfsmustern zum Lösen von Problemen erwiesen sich mehr und mehr als sinnvoll. Gerade die Notwendigkeit des Anwendens von Entwurfsmustern blieb lange verborgen. Während der Implementierung der Muster wurde die Flexibilität und Wiederverwendbarkeit deutlich. Dies spiegelt sich unter anderem darin wieder, das in dem folgenden Abschnitt als Ausblick diverse Möglichkeiten genannt werden. Diese Flexibilität wäre einer klassisch prozedural programmierten Anwendung nicht gegeben gewesen.
Nicht zu verschweigen jedoch ist der Mehraufwand, der sich anfangs durch die Klassenprogrammierung ergibt. Einfache Problemstellungen werden gefühlt "aufgebläht". Dies gilt hier vor allem für die Anbindung der Datenbank. Aber auch hier ist erkennbar, das sich mit Vergrößerung des Projektumfangs der anfängliche Mehraufwand rasch relativiert.
Auch eine verbesserte Übersicht im Quelltext ergibt sich allein schon aus der Tatsache, das sich jede Klasse in eigenen Dateien befindet. Die Dokumentation mit Doxygen tut hier ihr übriges und gewährt einen sehr guten Überblick über alle Funktionalitäten.
LibreOffice und die zugehörige API (sowie vor allem deren Dokumentation) ließen zeitweise Zweifel an der Realisierbarkeit der Anforderung aufkommen. Die Recherche erwies sich als langwierig und teilweise sehr frustrierend.
Auch die eigenständige Kompilierung des benötigten MySQL-Connectors (siehe 4.2) erwies sich trotz guter Dokumentation als alles andere als einfach. Der durch diese und ähnliche Begebenheiten entstandene Zeitverlust ließ zum Ende des Projektes die Implementation einiger wünschenswerter, interessanter Features leider nicht zu.
Eine durchgängige Dokumentationserstellung in Form von Kommentaren für Doxygen war ebenfalls sehr Zeitaufwändig. Diese ließ zuerst keinen Mehrwert erkennen und

schränkte manchesmal die schnelle, "kreative" Arbeit im Quelltext ein, da Veränderungen stets dokumentiert werden mussten.

Im letzten Drittel der Arbeit zeigte sich Vorteil nach dem Anderen. Die objektorientierte Programmierung gewährte große Flexibilität, die Dokumentation mit Doxygen lieferte überzeugende Ergebnisse und jede Komponente funktionierte gemäß der gewünschten Spezifikation.
Das Projekt erwies sich mehr und mehr als eine Basis, auf der nachfolgend aufgebaut werden kann und sollte. Alle nötigen Informationen und Grundlagen für weitere Umsetzungen sind aufgeführt. Eine rasche Einarbeitung sollte somit möglich sein und für die Zukunft eine großes Spektrum an Anwendungsmöglichkeiten bieten.
Einige mögliche Erweiterungen seien nachfolgend als Denkanstoß beigefügt. Der Kreativität, was die Verarbeitung der Daten angeht, sind kaum Grenzen gesetzt.

Die im Voraus gesetzten Ziele wurden allesamt erreicht:

- Der Endanwender wird von dem tieferen Einblick in den informationstechnischen Prozess befreit.

- Der Benutzer kann in seiner gewohnten Tabellenkalkulation mit den erfassten Daten umgehen.

- Echtzeitdaten einer Simulation oder Messung werden direkt Übertragen.

5.2 Zukünftige Anwendungen und Anpassungen

Wie eigentlich jede Software, ist auch die hier entwickelte Applikation noch mit Makeln behaftet, die Verbesserungen oder Erweiterungen wünschenswert machen. Wie bereits im vorigen Kapitel angedeutet, wurde manches Features schon angedacht, jedoch z.B. wegen Zeitmangel nicht, oder nur abstrakt ausgeführt. Für eine Weiterführung des Projektes sind hier mögliche Anpassungen oder Erweiterungen genannt. Eine Aufteilung in kurz- und mittelfristige Realisierbarkeit wurde vorgenommen, um erste Ansatzpunkte zu liefern.

5.2.1 Kurzfristig zu realisierende Anpassungen

Kurzfristig realisierbar sind vor allem solche Anpassungen, die bereits angedacht wurden, aber aus Zeitmangel nicht komplett implementiert sind. Siehe ergänzend hierzu Kapitel 4.

5.2.1.1 Logging und Exceptionhandling

Das Logging und Exceptionhandling wurde während der Entwicklung bereits bedacht und abstrakt ausgeführt. Eine Implementierung ist schnell zu realisieren. Hierzu müssten Exceptions hinzugefügt und das Logging mit seinen Methoden implementiert werden. Diese Aufgabe ließe sich unter Umständen gut mit der nächsten Aufgabe verbinden, um einen Log-File zu erstellen, der per Texteditor ausgelesen werden kann. Auch vorbereitet wurde das Schreiben von Logdaten in die Datenbank.

5.2.1.2 Dateibasierte Speicherung

Das Arbeiten mit Dateien, anstatt mit der MySQL-Datenbank wurde bereits in der DAO-Factory vorbereitet. Methoden in der DAOFile-Klasse müssten um die CRUDS-Funktionalitäten (siehe 3.6.2.1) für Dateien ergänzt werden. Somit lässt sich eine persistente Datenspeicherung in Dateien schnell realisieren.

5.2.1.3 Datenbankanpassung

Die Datenbank kann in Ihrer Funktionalität erweitert werden. Zur Zeit ist es nur möglich, die Datenbank zu wechseln, indem die entsprechenden `#define`s anpasst, und das Programm neu kompiliert (siehe 4.7.6) wird. Hier ist eine Anpassung sehr schnell und unkompliziert möglich, muss doch nur eine Userabfrage realisiert werden.
Eine andere Anpassung der Datenbank ist dergestalt möglich, als dass zur Zeit nur die Haupttabelle der Datenbank benutzt wird. Eine Anpassung würde dir verknüpften Tabellen komplett verwenden. Dies erscheint jedoch erst dann sinnvoll, sobald konkrete Messpunkte vorliegen. Diesen werden dann Messwerte zugeordnet. Die Datenbank ist hierfür komplett vorbereitet, nur die Speicherung müsste erweitert werden.

5.2.2 Mittelfristig zu realisierende Anpassungen

Als mittelfristig zu realisieren werden hier Anpassungen vorgeschlagen, deren Umfang eine einfache Anpassung überschreitet. Viele dieser Ideen ergaben sich erst im Laufe der Entwicklung und wurden deshalb nicht abstrakt implementiert.

5.2.2.1 Mehrere Devices

Zur Zeit können einzig von einem einzigen WriterDevice Daten empfangen werden. Eine Anpassung würde mehrere Devices implementieren und könnte unter Umständen eine Auswahl anbieten, von welchen Devices Daten empfangen werden sollen. Auch eine parallele Abfrage von Devices ist denkbar.

5.2.2.2 Webapplikation

Im Zuge der Arbeit ergaben sich, nicht zuletzt wegen der unzureichenden Dokumentation der Libreoffice-API, alternative Ideen bezüglich der Datendarstellung. Eine sehr komfortable Einbindung von Datendarstellung im Browser scheint das Webtoolkit zu bieten (http://www.webtoolkit.eu/wt). Hiermit könnte die Datenerfassung realisiert werden. Jegliche Funktionalität, auch zur Steuerung der Businesslogik scheint enthalten zu sein. Ein näherer Blick scheint hier in jedem Falle sinnvoll. Daten könnten in Echtzeit an eine Webschnittstelle geliefert werden. Dies würde eine physikalische Trennung von Businesslogik und Frontend ermöglichen.

5.2.2.3 Graphical User Interface

Ein grafisches Userinterface könnte programmiert werden. Entweder könnten Addins für LibreOffice Calc erstellt werden, die in Calc selber verschiedene Buttons zur Messdatenerfassung bereitstellen. Andererseits kann ein Qt-Fenstersystem benutzt werden, das die Aufgaben des Frontends übernimmt. Aufgrund der Erfahrungen mit der LibreOffice-API würde sich eher der Weg über Qt, oder die im vorherigen Abschnitt genannte Webschnittstelle anbieten.

Literaturverzeichnis

[1] Michael Kofler: MySQL: Einführung, Programmierung, Referenz. Addison-Wesley, 2001, ISBN 3-8273-1762-2.

[2] Oestereich, Bernd: Analyse und Design mit UML 2.3: Objektorientierte Softwareentwicklung. Oldenbourg, 2009, ISBN 978-3-486-58855-2.

[3] Freeman, Eric; Freeman, Elisabeth: Entwurfsmuster von Kopf bis Fuß. Oreily, Köln, 2008, ISBN 3-89721-421-0.

[4] Erlenkötter, Helmut: C++: Objektorientiertes Programmieren von Anfang an. Rowohlt,2005, ISBN 3-499-60077-3.

[5] Tschabitsche, Heinz :Einführung in C++. http://www.ladedu.com/, 2005, [05.01.2012]

[6] Oracle Corporation and its affiliates: http://www.mysql.com/downloads/connector/cpp/#downloads, 2011, [11.01.2012]

[7] http://www.c-plusplus.de [15.11.2012]

[8] Sun Microsystems, Inc. : StarOffice Programmer's Tutorial , Part Number 806-5845 , May 2000 []

[9] http://www.oooforum.org/forum/viewtopic.phtml?t=78063

[10] Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides: Entwurfsmuster. Elemente wiederverwendbarer objektorientierter Software: ISBN 3-8273-2199-9, 1994

[11] http://www.bogotobogo.com [11.01.2012]

[12] http://www.cplusplus.com [30.01.2012]

[13] http://corej2eepatterns.com/Patterns2ndEd/ [02.02.2012]

[14] http://dev.mysql.com/tech-resources/articles/building-mysql-connector-cpp.html [04.02.2012]

[15] http://geosoft.no/development/cppstyle.html [06.02.2012]

[16] http://www.stack.nl/ dimitri/doxygen/index.html [08.02.2012]

Abbildungsverzeichnis

2.1 this-Pointer in Netbeans-Entwicklungsumgebung 8
2.2 Grundlegende Zustandsdiagramm-Elemente 23
2.3 Zustandsdiagramm Threadapplikation mit Mutex 24
2.4 Grundlegende Klassendiagramm-Elemente 25
2.5 Klassendiagramm des Abstract Factory Entwurfsmusters 26

3.1 Vereinfachte Applikationsübersicht 34
3.2 Multi Tier Application Übersicht 35
3.3 Entwurf Frontend: Controller und Views 38
3.4 Abstrakter Entwurf Application Tier 40
3.5 DAO-Factory Entwurf 42
3.6 Datenbank Entwurf, Entity Relationship Model 43

4.1 Netbeans Projekt Einstellungen 47

A Anhang

A.1 IDE

```
sudo apt-get install g++
sudo apt-get install cmake
sudo apt-get install libc6-dev

download netbeans
install netbeans /usr/local/netbenas directory
sudo sh /usr/local/netbeans/bin/netbeans
start netbenas as root
sudo /usr/local/netbeans-7.1/bin/netbeans &
Set/check c++ options - build tools

basedir /usr/bin
c compiler /usr/bin/gcc
c++ compiler /usr/bin/g++
make command  /usr/bin/make
Debugger Command /usr/bin/gdb
cmake command /usr/bin/cmake

All includes stores in /usr/local/include
all libs stores in /usr/local/lib


sudo apt-get install libreoffice-dev
BoostLib  sudo apt-get install libboost-dev libboost-doc
mysqlclient sudo apt-get install libmysqlclient16-dev
sudo apt-get install libstdc++.5

http://www.mysql.de/downloads/connector/cpp/#downloads
download mysql c++ connector source code
unpack mysql-connector-c++-1.1.0.tar.gz


http://dev.mysql.com/tech-resources/articles/building-mysql-connector-cpp.html
# which cc
/usr/bin/cc

# which CC
/usr/bin/CC

# export CC=cc
# export CXX=CC
```

```
cmake -DCMAKE_REQUIRED_FLAGS=-xO4 -DCMAKE_INSTALL_PREFIX=/export/expts/MySQLConnectorC++
      -DMYSQL_CONFIG_EXECUTABLE=/usr/bin/mysql_config -DCMAKE_BUILD_TYPE=Release
      -DMYSQLCPPCONN_TRACE_ENABLE:BOOL=1 -DHAVE_FUNCTION_STRTOL=1 -DHAVE_FUNCTION_STRTOUL=1
      -DHAVE_STDINT_H=1 -DHAVE_INTTYPES_H=1

cmake -L

make

sudo make install

output:
...
[ 98%] Built target bug123
[100%] Built target bug456
Install the project...
-- Install configuration: "Release"
-- Installing: /export/expts/MySQLConnectorC++/./README
-- Installing: /export/expts/MySQLConnectorC++/./COPYING
-- Installing: /export/expts/MySQLConnectorC++/./ANNOUNCEMENT
-- Installing: /export/expts/MySQLConnectorC++/include/cppconn/build_config.h
-- Installing: /export/expts/MySQLConnectorC++/include/cppconn/config.h
-- Installing: /export/expts/MySQLConnectorC++/include/cppconn/connection.h
-- Installing: /export/expts/MySQLConnectorC++/include/cppconn/datatype.h
-- Installing: /export/expts/MySQLConnectorC++/include/cppconn/driver.h
-- Installing: /export/expts/MySQLConnectorC++/include/cppconn/exception.h
-- Installing: /export/expts/MySQLConnectorC++/include/cppconn/metadata.h
-- Installing: /export/expts/MySQLConnectorC++/include/cppconn/parameter_metadata.h
-- Installing: /export/expts/MySQLConnectorC++/include/cppconn/prepared_statement.h
-- Installing: /export/expts/MySQLConnectorC++/include/cppconn/resultset.h
-- Installing: /export/expts/MySQLConnectorC++/include/cppconn/resultset_metadata.h
-- Installing: /export/expts/MySQLConnectorC++/include/cppconn/statement.h
-- Installing: /export/expts/MySQLConnectorC++/include/cppconn/sqlstring.h
-- Installing: /export/expts/MySQLConnectorC++/include/cppconn/warning.h
-- Installing: /export/expts/MySQLConnectorC++/lib/libmysqlcppconn.so.5.1.1.0
-- Installing: /export/expts/MySQLConnectorC++/lib/libmysqlcppconn.so.5
-- Installing: /export/expts/MySQLConnectorC++/lib/libmysqlcppconn.so
-- Installing: /export/expts/MySQLConnectorC++/lib/libmysqlcppconn-static.a
-- Installing: /export/expts/MySQLConnectorC++/include/mysql_connection.h
-- Installing: /export/expts/MySQLConnectorC++/include/mysql_driver.h

copy lib to /usr/local/lib
copy include to /usr/local/include
```

A.2 MySQL Datenbank erstellen

```
1  CREATE DATABASE MessdatenDB;
2
3  USE MessdatenDB;
4  CREATE TABLE `MessdatenDB`.`Geraete` (
5  `id` INT( 10 ) UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
6  `bezeichnung` VARCHAR( 255 ) NOT NULL ,
7  `standort` VARCHAR( 255 ) NULL ,
8  `anmerkungen` TEXT NULL ,
9  UNIQUE (
10        `id`
11        )
12 ) ENGINE = MYISAM ;
13
14 CREATE TABLE `MessdatenDB`.`Properties` (
15 `id` INT( 10 ) UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY ,
16 `property` VARCHAR( 255 ) NOT NULL ,
17 `anmerkung` TEXT NULL ,
18 UNIQUE (
19        `id`MessdatenDBver
20        )
21 ) ENGINE = MYISAM ;
22
23 CREATE TABLE `MessdatenDB`.`Messpunkte` (
24 `id` INT( 10 ) UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY ,
25 `bezeichnung` VARCHAR( 255 ) NOT NULL ,
26 `geraete_id` INT( 10 ) NOT NULL ,
27 `property_id` INT( 10 ) NOT NULL ,
28 `funktion` VARCHAR( 255 ) NULL ,
29 `anmerkung` TEXT NULL ,
30
31 UNIQUE (
32        `id`
33        )
34 ) ENGINE = MYISAM ;
35
36 CREATE TABLE `MessdatenDB`.`Messungen` (
37 `id` INT( 10 ) UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY ,
38 `name` VARCHAR( 255 ) NOT NULL ,
39 `timestamp` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ,
40 `anmerkung` TEXT NULL ,
41
42 UNIQUE (
43        `id`
44        )
45 ) ENGINE = MYISAM ;
46
47 CREATE TABLE `MessdatenDB`.`Messdaten` (
48 `id` INT( 10 ) UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY ,
49 `tag` INT( 2 ) NOT NULL ,
50 `monat` INT( 2 ) NOT NULL ,
51 `jahr` INT( 4 ) NOT NULL ,
52 `zeit` TIME NOT NULL DEFAULT '00:00:00',
53 `messungen_id` INT( 10 ) NOT NULL,
54 `messpunkt_id` INT( 10 ) NOT NULL ,
55 `messwert` FLOAT( 16 ) NOT NULL,
56
57 UNIQUE (
58        `id`
59        )
60 ) ENGINE = MYISAM ;
```

A.3 Bekannte Bugs

A.3.1 pixmap warning

Es kann wärend des Startens der im Hintergrund benötigten soffice-Instanz zu einer dem Folgenden ähnlichen Meldung kommen:
`(soffice:8111): Gtk-WARNING **: Im Modulpfad »pixmap« konnte keine Themen-Engine gefunden werden`

Workaraound
Diese Warnung kann durch das Installieren des `gtk2-engines-pixbuf`-Paketes aus den Repositories behoben werden.

A.3.2 Ausführen des Programms außerhalb der Entwicklungsumgebung

A.3.2.1 Library Fehler

Es kann während des Versuches, das Programm außerhalb der Entwicklungsumgebung zu Starten zu einem Fehler kommen, bei dem zur Laufzeit gegen die falsche Library gelinkt wird.
Die Anzeige beinhaltet:
`libstdc++.so.6: version 'GLIBCXX_3.4.15' not found`
zusammen mit `/opt/libreoffice3.4/basis3.4/ure-link/lib`
oder ähnlich je nach Installation von Libreoffice und dem zugehörigen SDK.

Dieser Fehler entsteht, da die Library im SDK veraltet ist und nicht der entspricht, mit der Kompiliert wurde.

Workaround
Die Library `libstdc++.so.6` in
`<LibreOfficInstallationsVerzeichnis>/basis3.4/ure-link/lib` durch diejenige ersetzen, mit der kompiliert wurde (ggf. Namen anpassen), oder eine statische Verknüpfung mit dem Namen `libstdc++.so.6` auf die Library mit der kompiliert wurde.