Diplomarbeit

Implementierung eines Multiprojektmanagement-Tools in C++

Ausgeführt am

TECHNIKUM WIEN

FACHHOCHSCHULSTUDIENGANG ELEKTRONIK

Ausgeführt von

Arthur Zaczek

Wagramerstr. 4/802

9910011131

Betreuer

DI Dr. Robert Pucher

Firmenbetreuer

Dr Dietmar Wehinger

Datum

19.5.2002

Problemstellung

Bedingt durch den ständig steigenden Termin- und Kostendruck wird die konsequente Planung und Abwicklung von Projekten immer wichtiger.

Im Hinblick auf die Implementierung einer Softwarelösung sind insbesondere folgende Anforderungen zu berücksichtigen:

1. Projektleiter/Mitarbeiter und Management erhalten einen schnellen Überblick über die Auslastung einzelner Ressourcen.
2. Erleichterung der Planung von Ressourcen.
3. Terminkonflikte im Vorfeld erkennen und beseitigen.
4. Die Kontrollaufgaben (Kosten/Termine) eines Projektleiters erleichtern.
5. Die Software soll helfen, eine "ganzheitliche" Sicht auf Daten und Informationen zu bekommen.
6. Optimierung der Planung und Abwicklung neuer Projekte aufgrund der Erfahrung aus vergangenen Projekten.

Zusammenfassung

Im heutigen wirtschaftlichen Umfeld gibt es einen ständig steigenden Bedarf an Lösungen, die die Planung und Abwicklung von Projekten erleichtern, vor allem im Hinblick auf die Faktoren "Budget" und "Termine". Die mit Abstand am weitesten verbreitete IT-Lösung ist *MS Project* von Microsoft.

Der Nachteil von *MS Project* ist jedoch, dass in brauchbarer Weise stets nur *ein* Projekt bearbeitet bzw. ausgewertet werden kann. Damit ist es beispielsweise nicht möglich, die Auslastung aller Ressourcen (Personen / Gruppen / Maschinen) über alle Projekte hinweg zu bestimmen. Diese Anforderung, die häufig gestellt wird, war der Ausgangspunkt für die Implementierung der vorliegenden Lösung (*aceProject*).

In dieser Diplomarbeit werden zunächst die wichtigsten programmtechnischen Voraussetzungen geklärt und mögliche Gefahrenquellen analysiert. Dabei wird nur auf die Entwicklung der Software Bezug genommen.

Anschließend wird der technische Aufbau des Servers beschrieben. Da die gesamte Lösung mittlerweile an die 140.000 Lines of Code hat, können in der vorliegenden Arbeit nur die wichtigsten Aspekte beleuchtet werden. Dabei wird auf die Vermittlung der Ideen bei der technischen Umsetzung fokussiert.

Zum Abschluss wird noch auf einige wirtschaftliche Aspekte eingegangen, wie den kommerziellen Nutzen durch die Verwendung der Software bei ausgewählten Kunden.

Abstract

In our modern business environment there are increasing needs for Tools to help project managers to perform their project planning and processing tasks. There are special needs to watch deadlines and costs. The IT-Tool most used for such tasks is *MS Poject*.

The disadvantage of *MS Project* is that it is only possible to plan and watch one project at the same time. Therefore it is not possible to see if resources (employees, groups, machines) are used to capacity in all projects. Especially this is a requirement that is often asked for.

This tool is implemented as 3-tier-architecture. A Database holds the data, the middle tier is responsible for managing the data and generating reports. The client tier is responible for building views for the user.

The middle-tier is implemented as a DCOM-Server. It uses transaction orientated transmissions to guarantee consistence of the data.

This solution is very complex, it has about 140.000 lines of code. That makes it very hard to keep the overview. With object orientated programming methods this problem can be solved.

Inhaltsverzeichnis

1 Einleitung **7**

2 Grundlagen **9**

2.1 Objektorientierung . . . 9
2.1.1 Klassen . . . 9
2.1.2 Vererbung . . . 10
2.1.3 Interfaces . . . 14
2.1.4 Virtuelle Funktionen . . . 15
2.1.5 Templates . . . 17
2.1.6 Collections . . . 19
2.2 COM/DCOM . . . 21
2.2.1 Die Idee von COM . . . 21
2.2.2 Aufbau/Funktionsweise . . . 22
2.2.3 Instanzierung . . . 25
2.2.4 Lebenszyklus . . . 27
2.2.5 Arten von COM Objekten . . . 27
2.3 Zeiger . . . 28
2.3.1 Grundlagen . . . 28
2.3.2 Verwendung/Probleme . . . 31
2.4 Smart Pointer . . . 37
2.4.1 Definition . . . 37
2.4.2 Beispiel eines Smart Pointers . . . 39
2.4.3 Verwendung . . . 40
2.5 Smart Objects . . . 41
2.6 Observer . . . 44

3 Ausführung **46**

3.1 Zielsetzung . . . 46
3.2 Integration in eine Gesamtlösung . . . 47
3.3 Vorteile der Lösung . . . 48
3.4 Funktionsbeschreibung . . . 49
3.4.1 Projektansicht . . . 49
3.4.2 Ressourcenansicht . . . 50
3.4.3 Versionen . . . 51
3.4.4 Filter . . . 52

3.4.5 Kapazitätsdiagramm 52
3.5 Das Basiskonzept 53
3.6 Server 54
3.6.1 Datenmodell 54
3.6.2 Datenobjekte 55
3.6.3 Versionen 59
3.6.4 Transaktionen 59
3.6.5 Datenbank 62
3.6.6 Security 65
3.6.7 Berechnungen 65
3.6.8 Schnittstellen 68
3.6.9 Anbindung MSProject 71
3.7 Client 71

4 Wirtschaftliche Aspekte **72**
4.1 Einsatz beim Kunden 72
4.2 Fallbeispiel Magna Steyr Fahrzeugtechnik (SFT) Graz 72
4.3 Fallbeispiel VA-TECH SAT Wien 73
4.4 Fallbeispiel Immobilientocher einer deutschen Bank 73
4.5 Bedeutung für die ace 74

5 Diskussion **75**
5.1 Client/Server Architektur 75
5.2 Transaktionen 75
5.3 Performance 76

6 Ausblick **78**

A Anhang **83**

Kapitel 1

Einleitung

Ausgangspunkt dieses Projektes war, dass mit *MS Project* eine Kapazitätsplanung der Ressourcen über alle Projekte hinweg nicht in der erforderlichen Weise möglich ist.

Deshalb war in Planung, eine Softwarelösung zu implementieren, die auf *MS Project* aufsetzt und dessen funktionelle Schwächen behebt.

Als weitere Voraussetzung war festgelegt, in einem grafischen Interface allfällige Ressourcen- und Terminkonflikte aufzulösen, dh. beispielsweise Arbeitsschritte so zu verschieben, dass die Auslastung der einzelnen Ressourcen (Personen, Personengruppen, Maschinen) in möglichst optimaler Weise gegeben ist. Danach sollten die geänderten Daten wieder in *MS Project* übertragen werden können.

Für die Realisierung der grafischen Darstellung von Terminen bzw. Ressourcen (in einer dem Gantt-Diagramm ähnlichen Form) war in einer ersten Variante der Einsatz von am Markt erhältlichen Tools vorgesehen.

Es stellte sich jedoch rasch heraus, dass diese Tools nur bedingt den Anforderungen entsprochen haben. Deshalb wurde entschieden, eine eigene Grafikbibliothek zu implementieren, mit der es möglich ist, den Kundenwünschen entsprechende Darstellungen zu generieren.

In einem ersten Schritt wurde für die Lösung ein Client geschrieben, der noch viel Programmlogik enthielt und weiters ein Modul, das die Daten mit *MS Project* abgeglichen hat. Es stellte sich jedoch bald heraus, dass dieser Aufbau Probleme mit sich bringt, wenn mehrere Personen geichzeitig mit dem Tool arbeiten möchten. Dazu hätten Mechanismen implementiert werden müssen, die dafür sorgen, dass nur ein Client gleichzeitig einen Teil der Daten ändert. Abgesehen davon müsste jeder Client für eine Auswertung den gesamten Datenbestand laden, da eine Ressource in jedem Projekt involviert sein kann. Folglich wurde entschieden, eine Client/Server Architektur zu wählen, da nur mit einer solchen die Möglichkeit besteht, verschiedene Darstellungen im vorhinein zu errechnen. Diese Maßnahme steigert die Perfomance der Lösung und damit auch die Akzeptanz der Benutzer.

Im Zuge der Anpassung der Lösung für die ersten Kunden wurde der Wunsch geäußert gleich in *aceProject* zu planen und damit auf *MS Project*

zu verzichten.

Mit der Anbindung an ein weiteres System wie *Fabasoft Components* ist es möglich, ein umfassendes Projektplanungstool mit Dokumentenverwaltung, Stundenaufzeichnung, Terminverwaltung und vielem mehr zu implementieren.

Ein weiteres sehr wichtiges Ziel war es, die Anwender nicht mit einer Funktionsvielfalt zu überfordern, was aber nur teilweise gelungen ist. Dies ist nicht zuletzt darauf zurückzuführen, dass diverse Anwenderwünsche bezüglich einer übersichtlichen Bedienung bzw. Funktionserweiterung mittlerweile berücksichtigt wurden. Projektmanagement ist ein äußerst komplexes Gebiet und die Anforderungen der Anwender sind so weit reichend, dass die Verwendung von solchen Tools immer auch einen gewissen Schulungsaufwand mit sich bringt.

Kapitel 2

Grundlagen

In den folgenden Kapitel werden jene Themen behandelt, ohne deren Kenntnis die Implementierung der Lösung in dieser Form nicht möglich wäre. Dabei geht es nicht nur um ein Verständnis der von *Windows 2000* zur Verfügung gestellten Infrastruktur (DCOM) oder der Beherrschung einer Programmiersprache (C++) sondern auch um das Wissen über Methoden (*Observer*) und Gefahren (Zeiger) bei der technischen Umsetzung einer Softwarelösung. Daraus ergibt sich eine Problematik, die sich durch die zunehmende Komplexität bzw. den Umfang der vorliegenden IT-Lösung (*aceProject*) zwangsläufig verschärft.

2.1 Objektorientierung

Die für die Implementierung dieser Softwarelösung besonders relevante Themen werden in den nachfolgenden Kapiteln beschrieben. Dabei wird nur auf C++ spezifische Eigenheiten und Probleme eingegangen. Weiterführende Informationen zu C++ können Bespielsweise aus Bjarne Stroustrup, The C++ Programming Language[1] entnommen werden.

2.1.1 Klassen

Die Idee von Klassen ist es, Daten und Funktionen in geschlossene Einheiten zu gliedern (Kapselung). Dabei trennt man zwischen internen und öffentlichen Daten/Funktionen . Die Klasse *CTask* zeigt diese Trennung beispielhaft.

```
class CTask
{
public:
      void Move(CTime newDate);
private:
      void CalcDuration(...);
private:
      CTime m_Begin;
```

```
    CTime m_Finish;
    CTimeSpan m_Duration;
}
```

Die Klasse enthält eine allgemein zugängliche Funktion *Move*, eine interne Hilfsfunktion *CalcDuration* und ihre Variablen.

public bedeutet, dass die folgenden Funktionen/Variablen öffentlich zugänglich sind. *private* hingegen schränkt den Zugriff auf die Klasse selbst ein.

Es ist sehr wichtig, dass nur die Klasse ihre Daten ändern darf, ansonsten würden unnötige Probleme auftreten.

Als Beispiel sei das Speichern der Daten der Klasse *CTask* in die Datenbank angeführt.

Um das Speichern effizienter zu gestalten, werden nur jene Datensätze gespeichert, die sich geändert haben. Um das zu erreichen, wird eine Statusvariable einführt.

```
class CTask
{
public:
    void Move(CTime newDate);
private:
    void CalcDuration(...);
private:
    CTime m_Begin;
    CTime m_Finish;
    CTimeSpan m_Duration;
    bool m_Dirty;
}
```

Wenn die Daten frei zugänglich wären, müsste jeder, der die Daten ändert, auch *m_Dirty* ändern. Das ist auf Dauer nicht durchzuhalten. Außerdem ist es die Aufgabe des Programmierers der Klasse, dass der Status *m_Dirty* laufend überprüft wird.

Ein weiterer Vorteil der Trennung zwischen internen und externen Daten/Funktionen ist es, dass Änderungen an der Funktionalität transparent sind, d.h. es interessiert niemanden, wie eine Berechnung durchgeführt wird.

2.1.2 Vererbung

"Vererbung" bedeutet, dass eine Klasse die Eigenschaften (Methoden/Variablen) einer anderen Klasse übernimmt und diese um eigene Funktionalität erweitert. Abbildung 2.1 zeigt ein solches Beispiel.

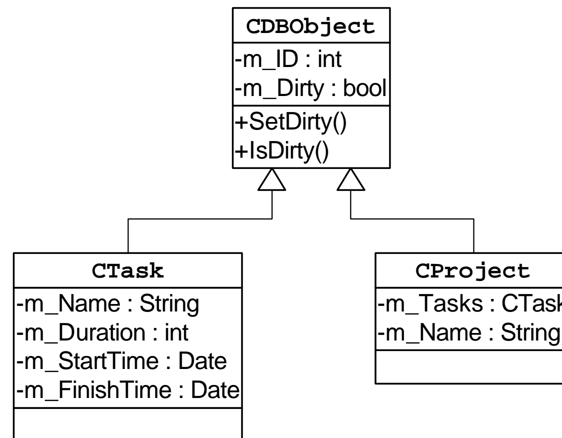

Abbildung 2.1: Ableitungen

Die Klasse *CDBObject* hat die Eigenschaft, den Datenbankstatus eines Objektes zu verwalten. Dazu werden die Variablen *m_ID* (Datenbankschlüssel) und *m_Dirty* (Datensatz ist zu speichern) benutzt. Die Funktionen *SetDirty()* und *IsDirty()* steuern den Status.

Die Klassen *CTask* und *CProject* sind Objekte, die in der Datenbank in unterschiedlichen Tabellen abgespeichert werden. Auch sonst haben sie nicht viel gemeinsam. *CTask* repräsentiert einen Arbeitsschritt mit einem Aufwand (*m_Duration*) und einer Zeitspanne (*m_StartTime*, *m_FinishTime*). *CProject* ist ein Objekt, das Arbeitsschritte zusammenfasst. Gemeinsam ist ihnen jedoch, dass beide in eine Datenbank gespeichert werden und daher eine eindeutigen Schlüssel haben. Ohne der Möglichkeit von Ableitungen müsste die Datenbankverwaltung für beide Klassen implementieren werden. Mit Hilfe von Ableitungen wird dieses Problem gelöst. Für den Benutzer dieser Klassen sieht es so aus, als ob beide Klassen diese Funktionalität "einfach" haben. Änderungen an dieser Funktionalität wirken sich sofort auf beide Klassen aus.

Syntax

Um *CTask* von *CDBObject* abzuleiten wird folgendes geschrieben:

```
class CTask : public CDBObject
{
...
}
```

public bedeutet hier, dass die öffentlichen Funktionen von *CDBObject* wieder öffentlich zugänglich sind. *private* würde den Zugriff verwehren.

Es gibt noch weitere Zugriffsbeschränkungen, wie die Klasse *CDBObject* zeigt:

```
class CDBObject
{
protected:
    void SetDirty();
    bool IsDirty();
private:
    int m_ID;
    bool m_Dirty;
}
```

protected bedeutet hier, dass der Zugriff auf die Klasse und deren abgeleiteten Klassen beschränkt ist. D.h. nur *CTask*, *CProject* und *CDBObject* haben Zugriff auf die Funktionen *SetDirty* und *IsDirty*. Dies ist auch sinnvoll, da die Klassen selbst entscheiden können, wann die Daten gespeichert werden sollen. Auch der eindeutige Datenbankschlüssel ist nur für die betreffenden Klassen von Interesse.

Mehrfachvererbung

Die meisten Programmiersprachen erlauben nur die Ableitung von einer Klasse (wobei man in der Regel von mehreren Interfaces (2.1.3) ableiten kann). In C++ ist die Mehrfachableitung möglich, was allerdings nicht unproblematisch ist.

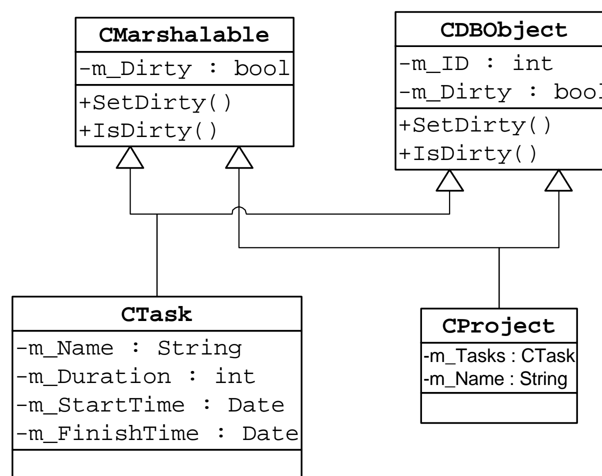

Abbildung 2.2: Mehrfachableitungen

Wie die Abbildung 2.2 zeigt, leiten die Klassen *CTask* und *CProject* nun zusätzlich von *CMarshalable* ab. Problematisch dabei ist allerdings die Tat-

sache, dass beide Basisklassen eine Funktion mit dem selben Namen (*SetDirty()*) haben. Beide Funktionen haben im Wesentlichen dieselbe Aufgabe, nur deren Ergebnis hat unterschiedliche Auswirkungen[1]. Dies ist einer der Hauptgründe dafür, warum die meisten Programmiersprachen nur Einfachableitungen erlauben.

Dieses Dilemma lässt sich allerdings leicht lösen. Beim Aufruf der Funktion wird über die Basisklasse die richtige Funktion gewählt.

```
...
CDBObject::SetDirty();
CMarshalable::SetDirty();
...
```

Zum Thema "Mehrfachableitung" gibt es unterschiedliche Meinungen. Einige besagen, dass die "Mehrfachableitung" einfach nur ein Designfehler ist, und mit Einfachableitung und Verwendung von Interfaces sehr gut das Auslangen zu finden ist. Dieses Projekt zeigt jedoch deutlich, dass es Sinn macht, Mehrfachableitung zu benutzen. Wenn eine Klasse mehr als nur eine Funktionalitätsgruppe benötigt, welche nichts miteinander zu tun haben, macht es Sinn, die Funktionalitätsgruppen auf mehrere Klassen aufzuteilen. Als Beispiel sei die Verwendung von *CDBObject* und *CMarshalable* genannt.

Die Ignoranz von Mehrfachableitungen führt oft zu "wilden" Konstrukten. So werden zum Beispiel Interfaces (siehe 2.1.3) eingeführt, die in jeder Klasse ihre Funktionalität implementieren müssen. Oder es werden Klassen eingeführt, die mehr als eine Aufgabe haben. Dies ist dann so ähnlich wie prozedurales Programmieren.

Virtuelle Ableitung

Bei Mehrfachableitungen gibt es ein weiteres Problem, wie in Abbildung 2.3 zu sehen ist. Die Frage lautet, ob nun die Variable *CSmartPtr::m_Counter* einmal oder zweimal in der Klasse *CTask* existiert. Laut Definition wird die Variable zweimal im Speicher angelegt, da unter Umständen die beiden Klassen *CDBObject* und *CMarshalable* eigene Instanzen benötigen. In diesem Fall wird so etwas nicht eingesetzt, da es sich hier um einen "Smart Pointer" (siehe 2.4) handelt.

Gelöst wird das Problem mit einer virtuellen Ableitung (siehe auch Bjarne Stroustrup, The C++ Programming Language[1]).

```
class CMarshalable : public virtual CSmartPtr
{
...
}
```

[1] *CDBObject* speichert in die Datenbank, *CMarshalable* informiert die Clients, dass eine Änderung stattgefunden hat

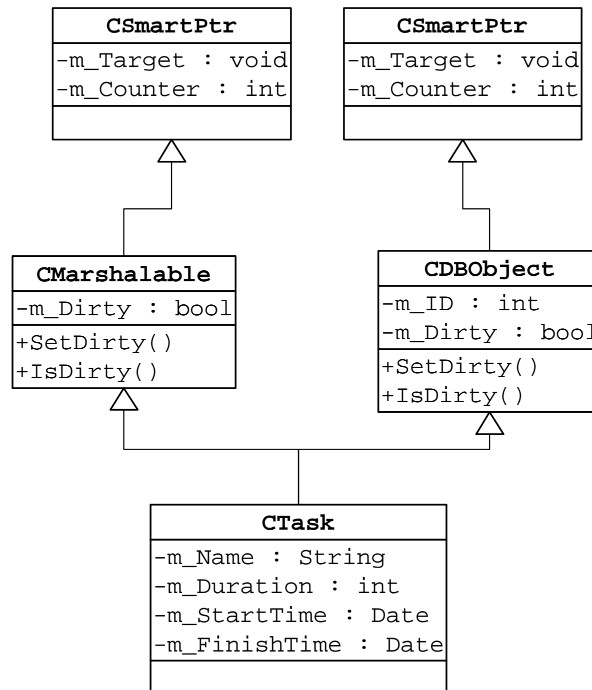

Abbildung 2.3: Mehrfachableitungen - getrennte Basisklassen

```
class CDBObject : public virtual CSmartPtr
{
...
}
```

Mit dieser Schreibweise weist man den Compiler an, eine eventuell schon vorhandene Basisklasse zu benutzen. Das Ergebnis wird in Abbildung gezeigt 2.4.

2.1.3 Interfaces

Ein Interface ist ein Vertrag über das Aussehen und Eigenschaften einer Klasse. Ein Interface selbst implementiert keine Funktionalität, dies ist Aufgabe der Klassen. Die Implementierung wird von Klasse zu Klasse unterschiedlich sein, solange ihr Verhalten nach außen den Vertrag erfüllt.

Abbildung 2.5 zeigt die Verwendung von Interfaces. Die Klasse *CViewProxy* teilt mit, dass sie die Funktionen *notifyUpdate*, *notifyAddTask*und *notifyDeleteItem* implementiert. Wie sie dabei auf einen Aufruf reagiert, ist in diesem Beispiel nicht im Vertrag definiert. Eine andere Klasse kann anders darauf reagieren. Der Vorteil für die aufrufende Funktion ist, dass sie keine Kenntnis über die Implementierung oder die konkrete Klasse, die sie aufruft,

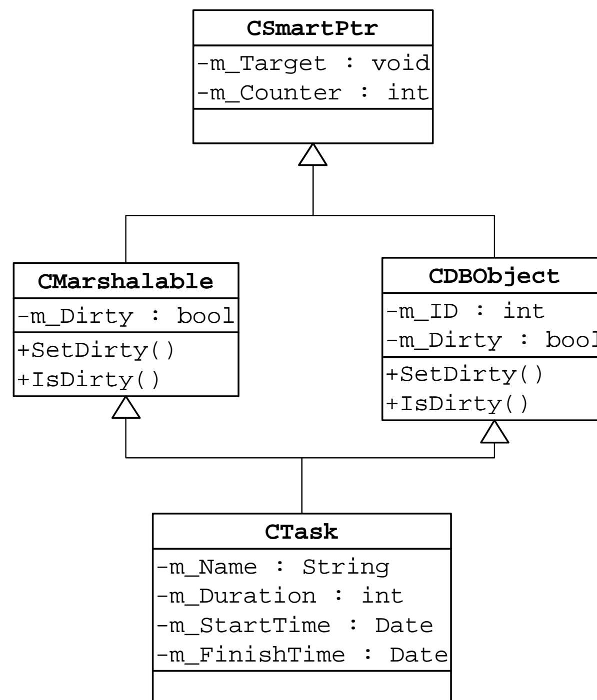

Abbildung 2.4: Mehrfachableitungen - gemeinsame Basisklasse

haben muss (die Funktion sieht ja nur das Interface).

In C++ gibt es, anders als in anderen Sprachen, kein konkretes Schlüsselwort für ein Interface. Statt dessen wird das Interface über virtuelle Funktionen (2.1.4) realisiert.

2.1.4 Virtuelle Funktionen

Abbildung 2.6 zeigt ein Beispiel für ein Interface. Die Funktion *CopyItem()* ist in der Basisklasse definiert, soll aber in den abgeleiteten Klassen *CTask* und *CProject* ausgeführt werden. Angenommen es gibt eine Collection (siehe auch 2.1.6), die *CVersionItem* enthält. *CVersionItem* ist allerdings ein Interface, also befinden sich in der Collection *CTask* und *CProjekt* Instanzen. Damit bei einem Aufruf von *CVersionItem::CopyItem()* die Funktion der abgeleiteten Klasse aufgerufen wird, muss das Programm zur Laufzeit wissen, um welchen Typ von Klasse es sich handelt. Da C++ keine Typeninformationen zur Laufzeit speichert, gibt es eine andere Möglichkeit, die "richtige" Funktion aufzurufen.

C++ baut sich zu diesem Zweck eine sog. *V-Table* (Virtual Table) auf. Zu jeder Instanz einer Klasse wird eine Tabelle erzeugt, in der sich Funktionszeiger befinden, um zur Laufzeit die richtige Funktion aufzurufen (siehe auch 2.3.2). Anders als in Java wird nicht jede Funktion in die *V-Table* aufgenommen. Dies hat hauptsächlich Perfomance-Gründe. Um eine Funktion "virtu-

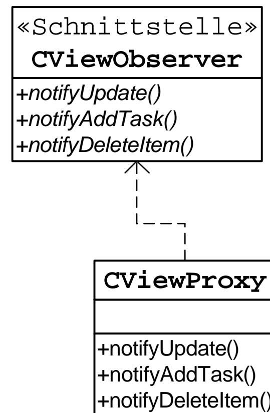

Abbildung 2.5: Interfaces

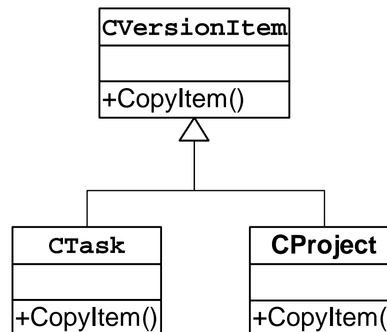

Abbildung 2.6: Virtuelle Funktionen

ell" zu machen, also sie in die *V-Table* aufzunehmen, wird das Schlüsselwort *virtual* verwendet.

```
class CVersionItem
{
    virtual void CopyItem();
}
class CTask : public CVersionItem
{
    virtual void CopyItem();
}
```

2.1.5 Templates

Templates sind - entsprechend dem Namen - Vorlagen. Es können Klassen und Funktionen als Vorlage definieren werden. Es stellt sich die Frage, wozu Templates definiert werden, wo doch einfach von einer Klasse abgeleitet werden kann. Erstens können bei Templates auch Typen als Parameter definieren werden, was bei Collections wichtig ist. Zum anderen wird vermieden, dass zwei Klassen, die nichts miteinander zu tun haben, auf eine gemeinsame Basis gestellt werden. Der Grund ist, dass bei der Instanzierung eines Templates eine neue Klasse entsteht.

Abbildung 2.7 zeigt die Struktur *CVersionItem*, *CTask* und *CProjekt*. *CTask* und *CProject* haben bis auf einige Grundfunktionen nichts miteinander gemeinsam. Wenn sie von einem Template ableiten, und diese unterschiedlich parametrisiert werden, wird eine gemeinsame Basis verhindert. Weiters sind im *CVersionItem* Variablen definiert, deren Typ sich unterscheidet.

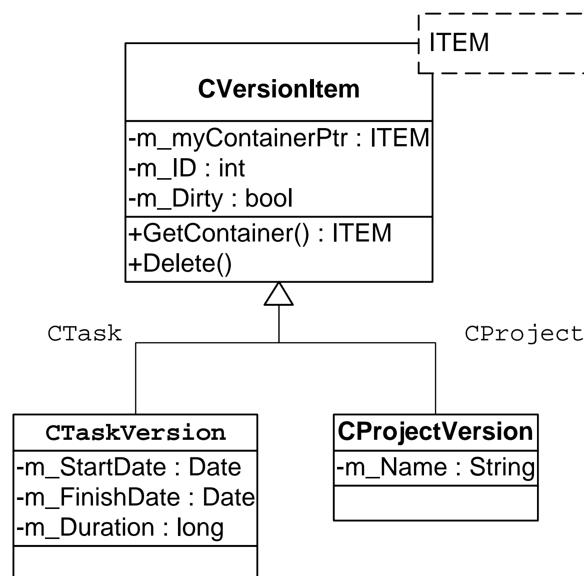

Abbildung 2.7: Templates

```
template <class ITEM> class CVersionItem
{
        typedef CUserSmartPtr< CVersionContainer<ITEM> >
        CONTAINERPTR;
        typedef CUserSmartPtr< ITEM > ITEMPTR;
public:
        CONTAINERPTR GetContainer() { return m_myContainerPtr; };
        void Delete() { m_myContainerPtr->DeleteVersion(this); };
```

```
        virtual void CopyItem(const CFilter& filter,
        ITEMPTR& newItem) const = 0;
private:
        CONTAINERPTR m_myContainerPtr;
        int m_ID;
        bool m_Dirty;
...
}
class CTaskVersion : CVersionItem<CTaskVersion>
{
private:
      CTime m_StartDate;
      CTime m_FinishDate;
      long m_Duration;
...
}
class CProjectVersion : CVersionItem<CProjectVersion>
{
private:
      CString m_Name;
...
}
```

Das Schlüsselwort *template* definiert, dass es sich bei der Klasse um eine Vorlage handelt. *class ITEM* definiert den Parameter. Die beiden *typedef* definieren je einen Zeiger auf die Parameterklasse (*ITEM*) selbst und einen Zeiger auf einen Container (siehe refAusfuehrung). Damit hat die Klasse

CVersionItem<*CTaskVersion*>

die Variable

CUserSmartPtr<C*VersionContainer*<*CTaskVersion*> > *m_myContainerPtr*

und die Funktion

CopyItem(const CFilter& filter, CUserSmartPtr<*CTaskVersion*>*& newItem)*

definiert. Es wurde erreicht, dass die Klassen *CTaskVersion* und *CProjectVersion* nichts miteinander zu tun haben und dennoch über gemeinsame Funktionalität verfügen. Weiters hat jede Template-Instanz eine Variable mit unterschiedlichem Typ (*m_myContainerPtr*) und eine Funktion mit unterschiedlichem Parameter (*CopyItem*).

2.1.6 Collections

Eine Collection ist eine Ansammlung von mehreren Objekten[2]. Ob eine Collection jetzt als verkettete Liste oder als Array implementiert ist, ist von untergeordneter Bedeutung[3].

Bei der Implementierung einer Collection, zum Beispiel einer Hashtable, werden Templates verwendet. Bei der Verwendung von Klassen könnten nur *void** gespeichert werden, was zur Folge hätte, dass bei jedem Zugriff der Typ umgewandelt (*casten*) werden müsste, was wiederum das Type-Checking von C++ aushebelt. Oder es werden pro Variablentyp eigene Collections implementiert.

Der nachfolgende Code zeigt ein einfaches Beispiel einer Collection. Es handelt sich hierbei um ein adaptives Array, welches seine Kapazität selbständig anpasst.

```
template <class T> class Array
{
public:
        Array(int size=0) : m_Data(size?new T[size]:0),
        m_Size(size) { };
        virtual Array() { delete[] m_Data; };

        T& operator[] (int p)
        {
                if(m_Size<=p)
                {
                      T* newData = new T[p];
                      if(m_Data)
                      {
                             memcpy(newData, m_Data,
                             m_Size*sizeof(T));
                      }
                      delete[] m_Data;
                      m_Data = newData;
                }
                return m_Data[p];
        }

        int GetSize() { return m_Size; }

private:
        T* m_Data;
        int m_Size;
```

[2]Wobei auch *build in types* wie *int* gemeint sind

[3]Bei dieser Entscheidung spielen Performance-Überlegungen eine Rolle

```
};
```

template <class T> sagt aus, dass es sich bei der Klasse um ein Template handelt, wobei *T* der Template-Parameter ist. Die Ausdrücke *new T[size]*, *T& operator[]* und *T* m_Data* zeigen, dass der Template-Parameter als Typ genutzt wird. Damit lassen sich alle Typen in die Collection aufnehmen, die einen Defaultconstructer haben und sich byte-weise kopieren lassen[4].

Das Template wird durch

```
Array<int> intArray;
Array<CWnd> wndArray;
```

instanziert. In *intArray* lassen sich nun Integer abspeichern, in *wndArray* Fenster. Ein Versuch, Daten von einem Array in das andere zu kopieren, wird bereits vom Compiler unterbunden.

Vordefinierte Collections

Collections werden in den meisten Klassenbibliotheken mitgeliefert. Die wichtigsten MFC-Collections sind:

Template	Beschreibung	Operatoren

`CArray<VAL>`	Adaptives Array ohne Bereichsprüfung	`Add()`
		`GetSize()`
		`[]`

`CList<VAL>`	Doppelt verkettete Liste	`GetHeadPosition()` `GetNext()`

`CMap<KEY,VAL>`	Hashtable	`SetAt()`
		`[]`
		`GetStartPosition()`
		`GetNextAssoc()`

Die *STL* (Standart Template Library [1]) geht bei den Collections einen eigenen Weg. Anstatt für jede Collection eigene Zugriffsfunktionen zu definieren, definiert die *STL* einen Satz von Zugriffsfunktionen, was den Vorteil hat, dass im Nachhinein der Collection-Typ austauscht werden kann. Weiters ist es möglich, standardisierte Algorithmen zu implementieren.

Solch ein Algorithmus, zum Beispiel eine Sortierfunktion, muss in der Regel die Collection auslesen und Elemente austauschen können. Dies geschieht bei den STL-Collections mit Hilfe von *Iteratoren.*

Ein *Iterator* ist ein Objekt, das einen Datensatz innerhalb der Collection repräsentiert. Weiters muss es möglich sein, mit dem `++` Operator zum nächsten Element zu gelangen. Eine STL-Collection stellt zusätzlich einen Beginn- und ein Ende-Iterator zu Verfügung.

[4] das liegt in der Art und Weise, wie die Daten gespeichert werden

```
vector<int> v;

for(vector<int>::iterator i=v.begin();i!=v.end();++i)
{
	int useInt = *i;
}
```

Mit Hilfe von Templates (siehe 2.1.5) lässt sich sehr einfach ein generischer Algorithmus programmieren. Die wichtigsten STL-Collections sind:

Template	**Beschreibung**
`vector<VAL>`	Adaptives Array ohne Bereichsprüfung
`list<VAL>`	verkettete Liste
`map<KEY,VAL>`	wird als Baum implementiert

2.2 COM/DCOM

2.2.1 Die Idee von COM

COM bedeutet "Compound Object Model", wobei Compound soviel wie "zusammengesetzt, verbunden" bedeutet. COM stellt eine Möglichkeit dar, fertig kompilierte Module wiederzuverwenden. Es stellt sich die Frage, warum eine neue Technologie verwendet werden soll, wo doch Module als Libraries zur Verfügung gestellt werden können.

Libraries haben den Nachteil, dass sie nur in einer Programmiersprache benutzt werden können. Eine Basic Library kann nicht in C++ benutzt werden. In C++ kommt noch ein weiterer Nachteil hinzu. Es müssen für C++, unter Windows, 8 Versionen der Library zur Verfügung stellen, nämlich

- MFC Release
- MFC Debug
- Unicode MFC Release
- Unicode MFC Debug
- Release
- Debug
- Unicode Release
- Unicode Debug.

Kaum ein Programmierer überprüft alle Versionen auf ihre Funktionsfähigkeit, was immer wieder zu Problemen führt.

Eine andere Methode, Module wiederzuverwenden, ist eine DLL zur Verfügung zu stellen. Der Nachteil ist aber, dass es in Visual Basic nicht immer möglich ist, eine Funktion der DLL aufzurufen, insbesondere dann, wenn eigene Datentypen eingeführt werden. Außerdem muss der Programmierer für jede Sprache eine Header-Datei, in Basic ein Modul, zur Verfügung stellen. Fehlt diese Datei, kann die DLL nicht genutzt werden. Ein weiteres Problem ist die Aufrufkonvention. Jede Sprache hat ihre eigene, weshalb beim Aufruf einer Funktion immer aufpasst werden muss, dass die richtige Konvention eingestellt ist. Außerdem ist diese Methode nicht objektorientiert.

COM hingegen umgeht alle diese Nachteile. COM ist objektorientiert, alle Informationen bezüglich Funktionen sind im Modul enthalten, was den Sprachen erlaubt, den Aufbau des Moduls maschinell auszulesen. Alle benutzerdefinierten Datentypen sind in dieser Beschreibung hinterlegt, alle Standardtypen haben einen definierten Aufbau. Auch die Aufrufkonvention ist festgelegt.

COM erlaubt allerdings weit mehr, als nur Funktionalität wieder verwenden. Nicht nur Module haben jetzt die Möglichkeit, ihre Funktionalität zur Verfügung zu stellen, sondern auch Programme. Das beste Beispiel hiefür ist *MSWord*^{TM}. Die Funktionen sind nicht dafür gedacht, "wiederverwendet" zu werden, sondern stellen eine Schnittstelle zu *Word* dar. Damit kann zum Beispiel eine eigene Serienbrieffunktion implementieren werden. Oder es kann eine Bericht erzeugt werden. Weiters ist es mit COM möglich, Programme zu erweitern, indem eine Schnittstelle definiert wird, über die andere Applikationen ihre Funktionen zur Verfügung stellen. Als Beispiel sei hier das Versenden von E-Mails zu nennen.

DCOM bedeutet „Distributed Compound Object Model". Es ist eine Erweiterung von COM um die Eigenschaft, auf Objekte auf einem anderen Computer zugreifen zu können. Damit lässt sich eine Fernwartung einer Applikation sehr bequem implementieren. Oder es wird diese Möglichkeit genutzt, um verteilte Applikationen zu implementieren.

2.2.2 Aufbau/Funktionsweise

Die wichtigsten Komponenten von COM sind

.exe, .dll, .ocx

Wenn eine *.exe* vorliegt, handelt es sich um einen EXE-Server. Dieser wird immer "Out-Of-Process" betrieben und benötigt eine Proxy DLL. Ein *.dll* Modul wird (fast) immer "inproc" betrieben, außer im COM+ Environment[2]. Bei einem *.ocx* Modul handelt es sich meist um ein ActiceX Control [3].

Typelibrary

Die Typelibrary ist die Beschreibung des COM Objektes. Sie befindet sich im Modul selbst oder in einer eigenen Datei (.tlb, .olb).

IDL

IDL - Interface Definition Language - ist die Beschreibungssprache der COM Objekte. Mit ihr werden alle Objekte, Interfaces und Methoden beschrieben.

MIDL

Compiler für die IDL.

Class

Die Class ist die Definition eines COM-Objektes. Ein Modul kann mehrere Klassen enthalten.

Interface

Das Interface ist die Schnittstelle der Klasse. Eine Klasse kann mehrere Schnittstellen haben. Von Schnittstellen kann ableitet werden.

Eine Schnittstelle kann auch in mehreren Modulen vorkommen. Das wird zum Beispiel beim *Index Server* von der Firma Microsoft benutzt, um eigene Dateifilter zu implementieren.

GUID/UUID

Der Global Unique Identifier oder Universal Unique Identifier dient zur Identifikation der Objekte und Schnittstellen. Es handelt sich hierbei um eine 128 Bit große Zahl, die weltweit eindeutig ist. Ihr Wertebereich reicht aus, um während der nächsten 10.782.897.524.560.000.000 Jahre eine Billion GUID pro Sekunde zu erzeugen [2].

CLSID/IID/LIBID

Module, Klassen und Schnittstellen brauchen jeweils eine eindeutige GUID. In allgemeinen Sprachgebrauch wird die GUID für Klassen Class Identifier, den für Module Library Identifier und den für Schnittstellen Interface Identifier genannt.

Class Factory

Die Class Factory ist ein standardisiertes COM Objekt, das in jeder COM-Library vorhanden sein muss. Es dient zur Instanzierung der in der Library definierten Objekte.

IUnknown

IUnknown ist ein standardisiertes Interface, von dem alle anderen Interfaces ableiten müssen. Über dieses Interface werden alle anderen Interfaces der Klasse angefordert und der Lebenszyklus des Objektes geregelt (siehe auch 2.2.4).

IDispatch

Ist ein standardisiertes Interface über das alle Script-Sprachen die Funktionen des Objektes aufrufen. Die Aufrufe erfolgen nicht über eine "V-Table" sondern über eine Funktionsnummer.

Proxy DLL/Marshaling

Eine Proxy DLL wird von lokalen und entfernten Servern benötigt. Die Proxy DLL übersetzt die Aufrufe und deren Parameter in ein Protokoll, um damit das entfernte Objekt aufzurufen. Diesen Vorgang nennt man Marshaling. Proxy DLL sind immer an Schnittstellen gebunden.

Beispiel eines Objektes

```
[
        object,
        uuid(5D938D00-1C1E-11d4-8697-0050DA769BCA),
        helpstring('IApplication Interface')
]
interface IApplication : IUnknown
{
        HRESULT LogIn([in] BSTR clientname, [in] BSTR pwd);
        HRESULT LogOff();

        HRESULT OpenDoc([out] ULONG* nDocID);
        HRESULT CloseDoc([in] ULONG nDocID);
...
};
...
[
        uuid(F0C198A9-E52B-472b-B7FE-99B71EAE88A7),
        version(1.0),
        helpstring('aceProject v1.0.1b')
]
library ResManagementLib
{
        [
                uuid(BBCEC717-99B2-4f36-8BBF-B9C9875E26BF),
```

```
        helpstring('aceProject Class')
    ]
    coclass aceProject
    {
        interface IApplication;
        interface IDocument;
        interface IObjAccess;
        interface IViewFunctions;
        interface IPlugin;
        interface IXMLReports;
        interface IVersionControl;
    };
};
```

2.2.3 Instanzierung

Am Beispiel des COM-Objektes in Abbildung 2.8 wird die Instanzierung gezeigt.

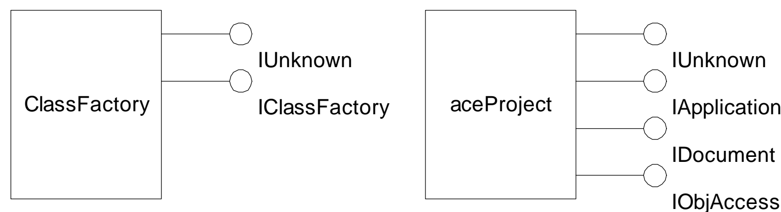

Abbildung 2.8: COM-Object

Bei diesem COM-Objekt handelt es sich um eine Service, das heißt das Programm wird beim Starten des Computer ausgeführt. Das Programm führt zunächst den Befehl *CoInitialize()* aus, um die COM-Library zu initialisieren. Danach wird der Befehl *CoRegisterClassObject()* ausgeführt, um das Objekt beim Service Control Manager anzumelden.

Ein Client, welcher auf das Modul zugreifen möchte, ruft nun

```
CoCreateInstance( CLSID_aceProject, NULL,
                  CLSTX_LOCAL_SERVER, IID_Unknown, (void**)&pUnk);
```

auf. Mit *CLSID_aceProject* teilt der Client dem Service Control Manager mit, dass er ein Objekt vom Typ *aceProject* instanzieren möchte. In der "Registry" ist hinterlegt, welches Modul geladen werden muss. Da es sich um eine *.exe* handelt, wird nun das Programm gestartet, sofern das noch nicht

durchgeführt wurde. In diesem Fall wird das Programm bereits ausgeführt, da es sich um ein Service handelt.

Im Hintergrund geschieht folgendes: Die Class Factory wurde beim Aufruf von *CoRegisterClassObject()* beim Service Control Manager angemeldet. Dieser holt sich nun von der Class Factory die *IClassFactory* Schnittstelle, um darauf *CreateInstance()* aufzurufen. *CreateInstance()* erzeugt ein neues *aceProject* Objekt und gibt die Schnittstelle *IUnknown* an den "Service Control Manager" zurück.

Da es sich bei dem COM-Server um einen "Local Server" handelt wird eine Proxy DLL benötigt. Der "Service Control Manager" instanziert zweimal die DLL und lädt eine Instanz in den Adressraum des Clients und eine in den Addressraum des Server (falls noch nicht vorhanden). Danach wird dem Client nicht die Schnittstelle *IUnknown* des Serverobjektes zurückgegeben sondern die der Proxy DLL. Dieser Vorgang wird in Abbildung 2.9 gezeigt.

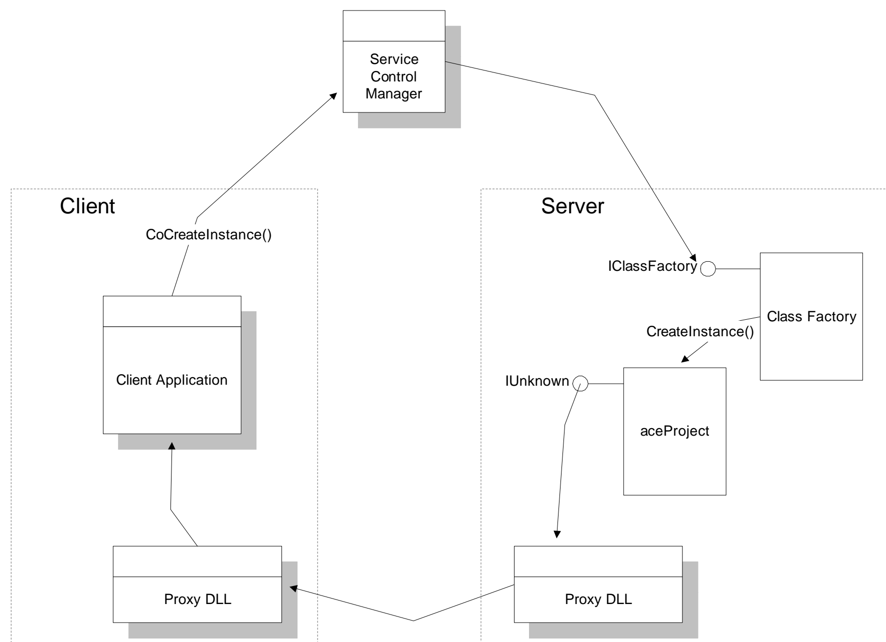

Abbildung 2.9: Instanzierung

Nachdem der Client den Schnittstellenzeiger *IUnknown* erhalten hat, ruft er die Methode *QueryInterface* auf, um die gewünschte Schnittstelle, zum Beispiel *IApplication*, zu bekommen. Auch hier wird gegebenenfalls vom Sevice Control Manager eine Proxy DLL geladen.

2.2.4 Lebenszyklus

Bei der Instanzierung eines COM-Objektes wird bei jedem Aufruf der Methode *CoCreateInstance()* ein neues Objekt erzeugt[5]. Dieses Objekt muss "irgendwann" wieder freigegeben werden. Die Entscheidung über den geeigneten Zeitpunkt trifft das COM-Objekt selbst. Es bedient sich dabei der Methoden von *IUnknown*.

```
interface IUnknown
{
        HRESULT QueryInterface(
                [in] REFIID iid,
                [out, iid_is(iid)] void ppObj );
        ULONG AddRef();
        ULONG Release();
}
```

Jedes Mal, wenn ein Schnittstellenzeiger angefordert (mit *QueryInterface* oder *CoCreateInstance()*) oder ein Zeiger dupliziert wird, wird die Methode *AddRef()* aufgerufen. Im letzteren Fall ist der Programmierer für die Implementierung zuständig. Damit wird das COM-Objekt darüber informiert, dass ein weiterer Zeiger auf das Objekt existiert. Jedes Mal, wenn der Programmierer entscheidet, dass ein Zeiger nicht mehr benötigt wird, **muss** *Release()* aufgerufen werden.

Das COM-Objekt hat intern einen Referenzzähler, der bei *AddRef()* inkrementiert wird und bei *Release()* dekrementiert wird. Erreicht der Referenzzähler 0, vernichtet sich das Objekt selbst.

Daneben gibt es noch einen Objektzähler, der den COM-Server beendet, sobald keine Objekte mehr instanziert sind.

2.2.5 Arten von COM Objekten

Inproc/Out of Process

Hierbei wird zwischen COM-Servern unterscheiden, die entweder in den Adressraum des Clients geladen werden (Inproc), oder ob es sich um eigenständige Applikationen handelt (Services, *MSWord*).

Ein *Out of Process* Server benötigt zur Kommunikation mit einem Client eine Proxy DLL, da die Übergabe von Parametern über einen Kommunikationskanal des Betriebssystems geschehen muss.

Threading Model/Apartment

Hier wird unterschieden, ob ein COM-Server immer nur einen Methodenaufruf gleichzeitig abarbeiten kann (Single Apartment/STA) oder mehrere

[5] In Ausnahmefällen wird nur eine Instanz erzeugt

gleichzeitig MTA. Dabei wird weiter unterschieden, ob auf ein und derselben Schnittstellenzeiger mehrere Aufrufe gleichzeitig ausführt werden können oder nicht.

Threading Model	**Beschreibung**
Single	Es kann immer nur eine Objektinstanz, eine Methode gleichzeitig ausführen
Apartment	Es können mehrere Objektinstanzen, eine ihrer Methode gleichzeitig ausführen
Free	Es können mehrere Objektinstanzen, mehrere ihrer Methode gleichzeitig ausführen
Both	Wie *Free*, dient jedoch nur zur Angabe dass STA Clients ebenfalls zugreifen können

2.3 Zeiger

2.3.1 Grundlagen

Zeiger gehören zu den schwierigsten Kapiteln beim Erlernen von C++. Aufgrund ihrer Grundstruktur (Abbildung 2.10) verleiten sie leicht zur Annahme, dass die Verwendung kein größeres Problem darstellen kann. Probleme treten aber bei Detailfragen auf.

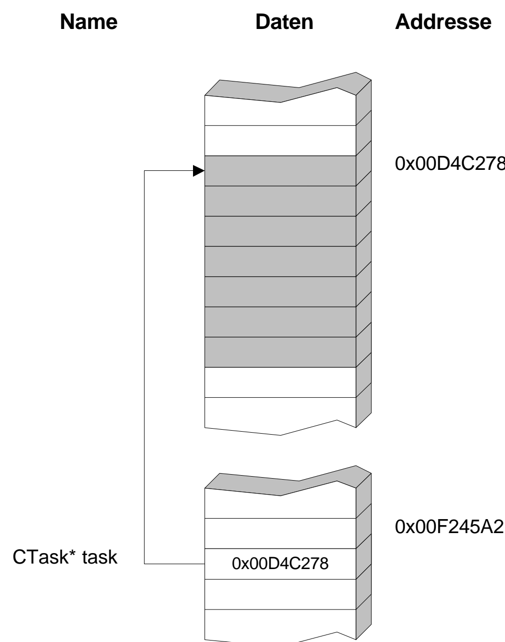

Abbildung 2.10: Zeiger

Ein Zeiger, auch Pointer genannt, ist eine Variable, deren Größe sich an

der Adressbreite orientiert[6] und dessen Inhalt die Adresse des Objektes vom Typ des Zeigers ist. Er kann drei Zustände annehmen:

- **gültig**, der Zeiger enthält eine gültige Adresse, dessen Inhalt ein gültiges Objekt ist
- **NULL**, der Zeiger enthält den Wert 0, er zeigt auf kein Objekt
- **ungültig**, der Zeiger enthält eine Adresse, dessen Inhalt irgendetwas sein kann. Dieser Zustand kann nicht abgefragt werden.

Im Beispiel in Abbildung 2.10 heißt der Zeiger *task*, ist vom Typ *CTask* und zeigt in diesem Fall auf ein *CTask* Objekt, das sich auf der Adresse 0x00D4C278 befindet.

Syntax

```
...
CTask instTask();               // CTask Instanz erzeugen
CTaks* pTask = &instTask;       // Adresse von instTask in pTask laden
CTaks* pTask2 = new CTask();    // Neues Objekt CTask erzeugen
                                // und Adr. in pTask2 speichern
CTask* pTask3 = pTask2;         // Adr. von pTask2 nach pTask3 kopieren
                                // beide zeigen nun auf das selbe Obj.
CTask instTask2 = *pTask;       // Inhalt von pTask nach instTask2
                                // kopieren. Dabei entsteht ein
                                // neues Objekt
instTask.Move();                // Funktion von instTask aufrufen
(*pTask).Move();
pTask->Move();
pTask2->Move();
...
```

new/delete

Mit `CTaks* pTask2 = new CTask();` wird ein neues Objekt vom Typ *CTask* erzeugt. Gelöscht wird es wieder mit `delete pTask;`. Ein Array von *CTask* Objekten kann mit *CTask* pTask = new CTask[128];* instanziert werden. Es ist dabei unerheblich, auf wieviele Objekte ein Pointer zeigt. Bei dem *delete* Operator ist dies anders. Arrays müssen mit `delete[] pTask;` gelöscht werden, da sonst nur der Destructor des ersten Objektes ausgelöst wird.

Im Hintergrund geschieht folgendes: *new* weist das Betriebssystem an, n Byte Speicher zu reservieren und die Addresse zurückzugeben. Die Größe des Speicherblocks wird als Parameter mitgegeben, die zur Compile-Time

[6] Bei den aktuellen Prozessoren sind es 32 Bit = 4 Byte

bestimmt wird. Das Betriebssystem sucht sich mit Hilfe eines komplexen Algorithmus einen freien Speicherblock und markiert diesen als "belegt". Beim Befehl *delete* wird dem Betriebssystem mitgeteilt, dass der Speicherblock nicht mehr benötigt wird. Die Größe muss nicht als Parameter übergeben werden, da das Betriebssystem "weiß", wie groß der Block ist.

Arrays/Felder

Die Bedeutung von `char** p` kann wie folgt interpretiert werden. Es kann ein Zeiger auf einen Zeiger vom Type *char* sein, oder es handelt sich um ein eindimensionales Feld von Zeichenketten. Abbildung 2.11 zeigt beide Möglichkeiten.

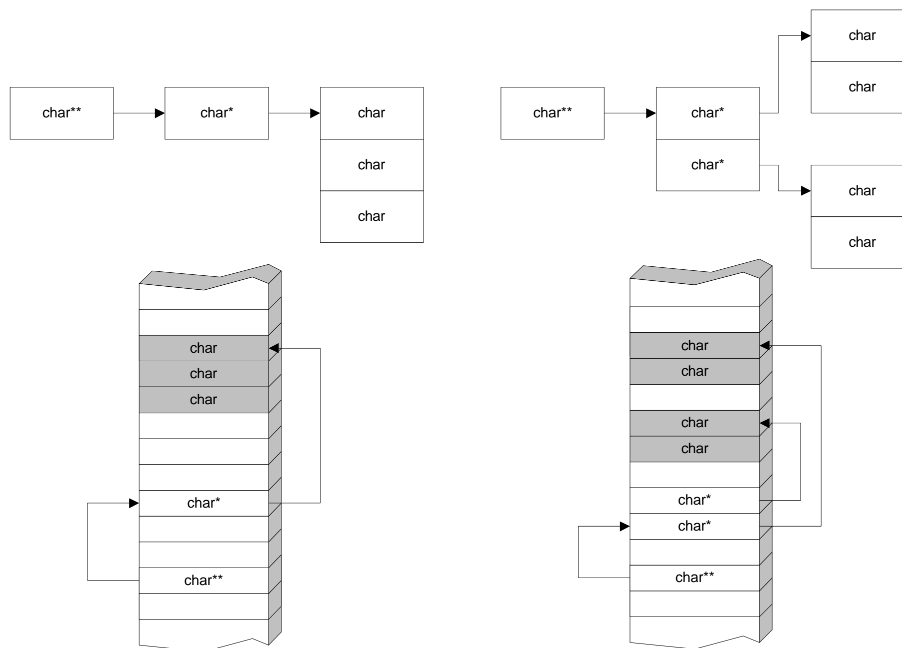

Abbildung 2.11: Zeiger-Arrays

In C++ werden Arrays als Zeiger implementiert. Wenn man ein Array definiert, so ist der Name des Array implizit ein Zeiger. Umgekehrt sind alle Zeiger auch gleichzeitig Arrays.

```
...
char str[128];          // Array of char, Größe 128 char
str[0] = 'a';           // Erste Stelle mit 'a' füllen
str = 'b';              // Erste Stelle mit 'b' füllen
char* i = str;          // Zeiger i initialisieren
*i = 'c';               // Erste Stelle von str mit 'c' füllen
i[1] = 'd';             // Zweiter Stelle von str mit 'd' füllen
...
```

2.3.2 Verwendung/Probleme

Strings

C++ kennt den Datentyp *String* nicht. Man kann aber einen *String* als ein Feld von einzelnen Zeichen auffassen, dessen Ende durch den Wert 0 gekennzeichnet wird[7] (Abbildung 2.12).

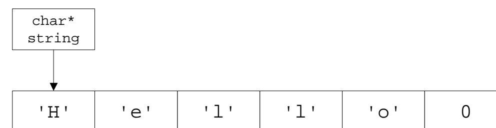

Abbildung 2.12: Strings

```
...
int printf(const char* format, ...);        // Definition von printf
char* gets(char* buffer);                   // Definition von gets
...
char* str = "Enter something: ";
printf(str);
gets(str);
...
```

Der Code zeigt ein Beispiel für einen schweren Fehler. Die Definition von *printf()* und *gets()* verlangen als Parameter *char**. Also werden *char** angelegt, oft auch ohne Initialisierung, und den Funktionen übergeben. Bei der Funktion *printf()* funktioniert das Programm noch, da sie nur lesend auf das Feld zugreift (Zeiger = Feld). Bei der Funktion *gets* kommt es zu einem Fehler. Hier wird das Feld befüllt, ohne Rücksicht auf dessen Kapazität. In diesem konkreten Fall würde nach der Eingabe des 18. Zeichens der Speicher nach dem Feld überschrieben werden (näheres später). Würde der Code in einer Funktion stehen, und das Programm bis zum erneuten Aufruf nicht abstürzen, dann wäre sogar der Text "Enter something: "geändert worden, da er von *gets* überschrieben wurde. Der Grund ist, dass der String "Enter something: "auch nur ein Zeiger auf ein Array vom Typ *char* ist, der beim Programmstart initialisiert wurde. Dieser Zeiger wurde nach *str* kopiert und in weiterer Folge *gets* übergeben, der danach zuverlässig den Inhalt des Zeigers überschreibt.

Korrekt müsste der Code daher folgendermaßen aussehen:

```
...
char str[1024];
```

[7]In anderen Sprachen wird die Größe des Strings am Anfang des Feldes gespeichert

```
printf("Enter something: ");
gets(str);
...
```

Der Zeiger auf ”Enter something: ”wird jetzt nur noch einer Funktion übergeben, die lesend auf das Feld zugreift[8]. Für *gets* wurde ein Speicherblock von 1024 char am Stack reserviert, wohin *gets* gefahrlos schreiben kann. Aber auch hier tritt ein Problem auf:

Wenn mehr eingegeben wird, als Platz vorhanden ist, wird der nachfolgende Speicher überschrieben.

Oft besteht der Wunsch Strings auch zu verändern, um zum Beispiel in einer Stringvorlage einen Namen eingeben zu können. Dazu werden eine Reihe von Funktionen zur Verfügung gestellt:

Funtion	**Beschreibung**
`strcpy`	Kopiert Strings
`strcat`	Fügt an das Ende eines Strings einen zweiten an
`sprintf`	Wie printf (Formatierte Ausgabe) nur wird in einen String geschrieben
`strlen`	Bestimmt die Länge eines Strings

```
...
char name[1024];                          // Variablen definieren
char str[1024];
printf("Enter your name: ");              // Name einlesen
gets(str);
sprintf(str,                              // String formatieren
   "Hello %s, how are you?", name);
printf(name);                             // String ausgeben
...
```

Diese Funktionen haben jedoch alle einen Nachteil. Sie prüfen nicht die Kapazität des Feldes, in das sie hineinschreiben. Daher ist es besser, die Stringklassen der *STL* oder *MFC* zu benutzen.

Heap

Zeiger werden hauptsächlich dafür verwendet, den zur Laufzeit angeforderten Speicher zu verwalten. Dazu speichert man die Adresse, die *new* zurückgibt. Hierfür gibt es eine große Anzahl an Möglichkeiten. Adressen können in Arrays, Hashtables, Bäumen oder aber in lokalen Variablen gespeichern werden. Der Hauptanwendungsfall ist sicherlich die Objektzeiger global in Listen zu verwalten. Hierbei können allerdings einige Probleme auftreten.

[8] Das drückt das *const* in der Definition aus

```
...
static CMap<long, CTask*> m_TaskList;
...
CTask* task = new CTask();
m_TaskList[task->GetID()] = task;
...
// Viele Operationen später
CTask* task = m_TaskList[taskID];          // Gib es das Objekt noch?
...
```

Da C++ keinen *Garbage Collector* hat, ist beim Programmieren darauf zu achten, dass nicht irgendwann ein Zeiger auf eine Instanz erzeugt wurde, der die Instanz löscht, die Liste aber nicht bereinigt. Umgekehrt muss auch darauf geachtet werden, dass die Liste selbst nicht gelöscht wird, ohne die noch in ihr verbliebenen Instanzen zu löschen. Ersteres führt dazu, dass ein oder mehrere Objekte, die in den freien Speicher gelegt wurden, zerstört werden, bzw. das Originalobjekt nicht mehr vorhanden ist. Zweiteres führt zu *Memory Leaks*, was dazu führt, das die Speichermenge, die verbraucht wird, immer größer wird. Das ist auf den ersten Blick nicht problematisch, wenn aber das Programm idealerweise mehrere Jahre auf einem Server ohne Unterbrechung genutzt werden muss, kann dies sehr unangenehme Folgen haben.

Stack

Wie bereits bei den Strings gezeigt wurde, ist das Hinausschreiben über die Grenze eines Feldes ein Problem. Auf dem *Heap* halten sich die dabei entstandenen Schäden noch in Grenzen, da ja *nur* benachbarte Objekte beschädigt werden. Auf dem *Stack* hat ein Überschreiben einen Feldes jedoch schwerwiegendere Folgen, die vom Absturz des Programms bis hin zu einem erfolgreichen Eindringen in das System durch Hacker reichen. Abbildung 2.13 zeigt den Aufbau des *Stacks* für die nachfolgende Methode.

```
void CSomeClass::ReadValue(int param)
{
    int str[4];
    gets(str);
    m_Value = param;
};
```

Was hier geschieht lässt sich folgendermaßen beschreiben: Zunächst wird das Feld *str* aufgefüllt, danach wird der Zeiger auf die Objektinstanz gelöscht, was zur Folge hat, dass diese Methode auf keine Variablen der Klasse zugreifen kann. Wenn die Methode es dennoch versucht, wird sie möglicherweise eine *Access Violation* verursachen. Danach wird der gespeicherte "Extended Base Pointer" gelöscht. Das hat zum gegenwärtigen Zeitpunkt noch keine

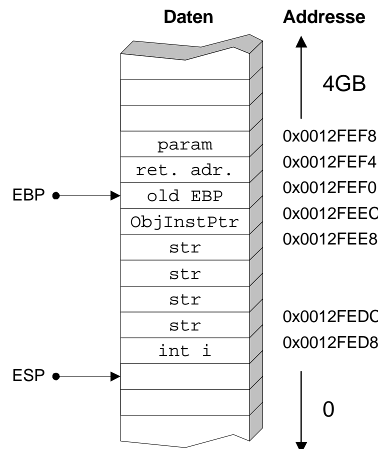

Abbildung 2.13: Stack einer Methode

Konsequenzen, sobald aber die Methode zurückkehrt, kann die aufrufende Funktion nicht mehr auf ihre lokalen Variablen zugreifen. Im Anschluss daran wird die Rücksprungadresse der Methode überschrieben.

Das hat vorerst keine unmittelbaren Konsequenzen. Erst wenn die Methode zur aufrufenden Funktion zurückkehren möchte, folgt ein sofortiger Programmabsturz. Als letztes werden die Methodenparameter überschrieben. Auch das hat erst dann Konsequenzen, wenn ein Zeiger überschrieben wurde. Beim nächsten Zugriff auf diese Zeiger wird vermutlich eine *Access Violation* gemeldet. Wenn jetzt noch weitergeschrieben wird, werden alle Daten der aufrufenden Funktion genauso wie zuvor zerstört.

Ein Hacker kann sich das zum Nutzen machen, indem er absichtlich über eine Feldgrenze schreibt, aber dafür sorgt, dass die Rücksprungadresse mit einem Wert überschieben wird, die ein Programm des Hackers aufruft (Abbildung 2.14). Da sich solche Programme gerne in einem höheren "Securitylevel" befinden, hat der Hacker nun das System unter seiner Kontrolle.

Der Trick funktioniert deshalb, weil sich die Positionen der Werte am Stack bei erneutem Programmstart/Methodenstart selten ändern. Das ist auf den *Protected Mode* der CPU zurückzuführen, wo jedes Programm seinen eigenen 4GB Adressraum bekommt. Damit werden der Programmcode und alle Daten immer auf dieselbe Adresse (logische) geladen[9].

Zeiger und Mehrfachableitungen

Bei der Verwendung von Mehrfachableitungen kann es zu Problemen mit Zeigern kommen. Der nachfolgende Quellcode mit Abbildung 2.15 zeigt ein

[9]Eine Ausnahme stellen die DLLs unter Windows dar. Sie können an einer anderen Adresse geladen werden, da sich alle Anwendungen den Code der DLL teilen

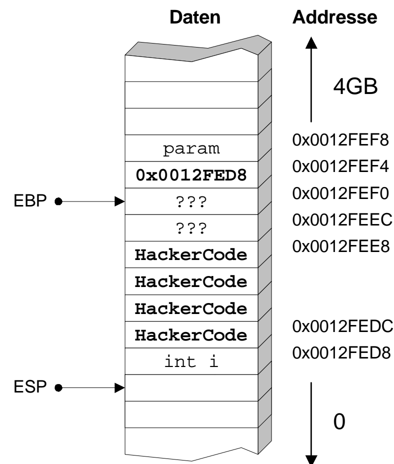

Abbildung 2.14: Stack Exploite

typisches Beispiel.

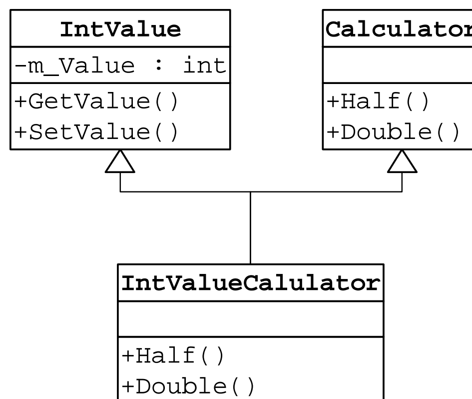

Abbildung 2.15: Zeiger und Mehrfachableitungen

```
#include <stdio.h>
#include <conio.h>

int main(int argc, char* argv[])
{
   IntValueCalulator* c = new IntValueCalulator();

   IntValue* pIntValue = static_cast<IntValue*>(c);
   Calculator* pCalculator = static_cast<Calculator*>(c);
```

```
    // BadPointer !
    Calculator* rpCalculator = reinterpret_cast<Calculator*>(c);

    printf("Original Pointer 0x%p\n", c);
    printf("Pointer to IntValue 0x%p\n", pIntValue);
    printf("Pointer to Calculator 0x%p\n", pCalculator);
    printf("!Pointer to Calculator after reinterpret_cast! 0x%p\n",
       rpCalculator);

    delete c;
    return 0;
}
```

Die Ausgabe des Programms sieht folgendermaßen aus:

```
Original Pointer                                   0x002F2F08
Pointer to IntValue                                0x002F2F08
Pointer to Calculator                              0x002F2F10
!Pointer to Calculator after reinterpret_cast!     0x002F2F08
```

Wenn ein Objekt mit virtuellen Funktionen (alle Objekte haben virtuelle Destruktoren) erzeugt wird, sei es jetzt am *Stack* oder am *Heap*, so wird für jede Klasse der Instanz eine *VTable* angelegt (Abbildung 2.16), über die die virtuellen Funktionen aufgerufen werden. Wie Abbildung 2.16 auch zeigt,

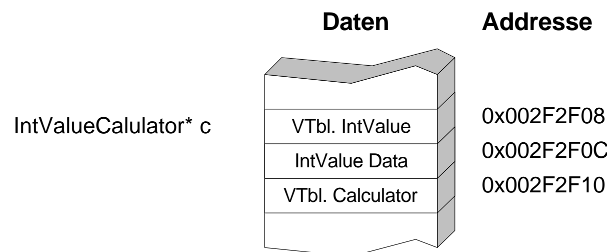

Abbildung 2.16: VTable

werden die Daten des Objektes nach den *VTable* Zeigern abgelegt. Wenn jetzt ein Zeiger auf eine Basisklasse angefordert wird, **muss** der Zeiger verschoben werden, da die Information über den eigentlichen Typ des Objektes verloren geht.

Der Unterschied zwischen `static_cast` und `reinterpret_cast` ist, dass `static_cast` die Verschiebung des Zeigers vornimmt, `reinterpret_cast` jedoch nicht, da `reinterpret_cast` nur eine andere Interpretierung des vorliegenden Bitmusters erlaubt.

Probleme können entstehen, wenn eine Liste von *Calculator** anlegt wird und zu einem späteren Zeitpunkt überprüft werden soll, ob eine abgeleitete Instanz eines *Calculator* Objektes sich in der Liste befindet. Dazu muss die zu vergleichende Instanz vorher unbedingt in ein *Calculator* Objekt gecastet werden, um die korrekte Adresse zu erhalten.

2.4 Smart Pointer

2.4.1 Definition

Im Kapitel 2.3.2 wurde ausführlich auf die Probleme bei der Verwendung von Zeigern eingegangen. Da C++ ein vollständig objektorientiertes Umfeld bietet, liegt es nahe, die Standardaufgaben der Zeigerverwaltung in Klassen zu kapseln. Als erstes sollte überlegt werden, einen *Garbage Collector* zu implementieren.

Dies ist allerdings eine sehr anspruchsvolle Aufgabe, die auch noch die Performance des Programmes beeinträchtigt.

Noch sinnvoller ist es, sich auf ein teilweise automatisches Löschen von Objekten zu konzentrieren und Zeigerfunktionalität in Klassen zu kapseln. Diese Aufgabe lässt sich in ca. 150-200 Zeilen realisieren.

Betrachtet man die Operationen, die auf einen Zeiger ausgeführt werden können, mit dem zusätzlichen Wunsch, dass die Objekte automatisch gelöscht werden sollen, ergibt sich folgende Aufgabenstellung:

- Speichern einer Adresse
- Rückgabe des Objektes (dereferenzieren)
- Zeiger vergleichen
- Zeiger kopieren
- Zeiger konvertieren
- Ungültige Adressen verhindern
- Objekt löschen, wenn kein Zeiger auf die Instanz mehr verfügbar ist

Das Speichern einer Adresse stellt kein Problem dar, es wird einfach eine entsprechende Variable definiert. Hierbei gibt es zwei Möglichkeiten. Entweder wird ein *void** benutzt, um die Adresse zu speichern. Das hat den Nachteil, dass jedesmal, wenn dereferenziert wird, der Zeiger in den gewünschten Typ umgewandelt werden muss (casten). Außerdem kann es passieren, dass dabei der falsche Typ verwendet wird und sich der Fehler erst zur Laufzeit bemerkbar macht. Oder es werden Templates benutzt (siehe auch Kapitel 2.1.5). Bei dieser Methode muss weder der Typ umgewandelt werden, noch besteht die Gefahr, in den falschen Datentyp zu konvertieren.

Das Dereferenzieren lässt sich einfach durch Überschreiben der Operatoren `operator->()` und `operator*()` implementieren. Auch der Zeigervergleich und das Kopieren ist durch überschreiben der entsprechenden Operatoren `operator==()`, `operator!=()`, `operator=()` und des *Copy Constructor* einfach zu lösen.

Das Konvertieren von Zeigern in einen anderen Typ ist nicht mehr so einfach zu realisieren. Dazu muss eine Konvertierungsfunktion, die als Template ausgeführt ist, implementiert werden. Ihre Aufgabe ist es, einen neuen *SmartPointer* vom gewünschten Typ zu erzeugen und ihn mit den korrekten Werten zu versorgen.

Ungültige Adressen werden schon vom Konzept her verhindert, da eine ungültige Adresse in der Regel erst entsteht, wenn ein Objekt gelöscht, oder ein Zeiger nicht initialisiert wird. Damit kennt der *Smart Pointer* nur zwei Zustände: gültige Adresse oder NULL.

Die letzte Aufgabe, das automatische Löschen von Objekten, ist ebenfalls einfach zu lösen. Dazu werden die Zeitpunkte genutzt, an denen C++ Objekte löscht.

- Wenn das Objekt mit *new* am Heap angelegt wurde, wird es beim Aufruf von *delete* gelöscht. Genau das gilt es zu automatisieren.
- Wenn das Objekt im globalen Scope angelegt wurde, wird es am Ende des Programms gelöscht.
- Wenn das Objekt im einem lokalen Scope angelegt wurde - das kann eine Funktion oder ein Block sein - wird es am Ende des Scopes gelöscht.
- Wenn der Destructor eines Objektes ausgeführt wird, werden die Destruktoren aller Membervariablen automatisch ausgeführt.

```
#include <stdio.h>
#include <conio.h>

CTask globalTask;                          // global scope

int main(int argc, char* argv[])
{
   CTask localTask;                        //local scope
   {
      CTask blockTask;                     //local scope
   }                                       // blockTask wird gelöscht
   return 0;                               // localTask wird gelöscht
}
                                           // globalTask wird gelöscht
```

Wenn jeder Zeiger, der mit *new* erzeugt wurde, immer sofort in einen *Smart Pointer* gespeichert wird, der innerhalb eines Scopes erzeugt wurde und nur noch mit *Smart Pointern* gearbeitet wird, kann mit Hilfe einer Referenzzählung die Anzahl der *Smart Pointer* auf die Objektinstanz bestimmt werden. Wenn jetzt der letzte *Smart Pointer* auf eine Instanz gelöscht wird, weiß der betreffende *Smart Pointer* aufgrund des Referenzzählers, dass er das Objekt zu löschen hat.

2.4.2 Beispiel eines Smart Pointers

```
template<class TARGET> class CSmartPtr
{
public:
    CSmartPtr(const CSmartPtr<TARGET>& BlockPtr);
    CSmartPtr(TARGET* Block);
    virtual ~CSmartPtr();

    CSmartPtr<TARGET>& operator=(const CSmartPtr<TARGET>& Block);
    TARGET* operator->() const
    {
        if(m_Target == NULL) throw NULL Pointer!";
        return m_Target; };

    int operator== (const TARGET* compare) const
    { return m_Target == compare; }
    int operator!= (const TARGET* compare) const
    { return m_Target != compare; }

    TARGET& operator* () const { return *m_Target; }

private:
    TARGET* m_Target;
    int* m_Counter;
};

// Destructor
template<class T> CSmartPtr<T>::~CSmartPtr()
{
    if(--(*m_Counter) == 0)
    {
        delete m_Target;
        delete m_Counter;
    }
}
```

```
// Copy constructor
template<class T>
CSmartPtr<T>::CSmartPtr(const CSmartPtr<T> &TargetPtr)
    : m_Target(TargetPtr.m_Target), m_Counter(TargetPtr.m_Counter)
{
    (*m_Counter)++;
}
```

Der Konstruktor `CSmartPtr(TARGET* Block);` nimmt als Parameter die Adresse der Objektinstanz, die ab sofort nur noch über Smart Pointer angesprochen werden soll, welche in *m_Target* gespeichert wird. Zusätzlich wird ein Referenzzähler am Heap erzeugt.

Der *Copy Constructor* `CSmartPtr(const CSmartPtr<T> &TargetPtr)` nimmt als Parameter den zu kopierenden *Smart Pointer*. Als erstes wird der Objektzeiger und der Referenzzeiger kopiert. Danach wird der Referenzzähler inkrementiert.

Der Destruktor dekrementiert zuerst den Referenzzähler. Wenn dieser den Wert 0 erreicht, wird das Objekt gelöscht.

Der `operator=(const CSmartPtr<TARGET>& Block)` nimmt als Parameter den zu kopierenden *Smart Pointer*. Zuerst wird der eigene Referenzzähler dekrementiert und gegebenenfalls wird das Objekt gelöscht. Danach werden Objektzeiger und Referenzzähler kopiert und der Referenzzähler wird inkrementiert.

Die Operatoren `operator->()` und `operator*()` geben die Referenz auf das Objekt zurück, die Operatoren `oprator==()` und `oprator!=()` vergleichen die gespeicherte Adresse.

2.4.3 Verwendung

Die großen Stärken von *Smart Pointern* lassen sich am besten anhand einer einfach verketteten Liste zeigen (Abbildung 2.17).

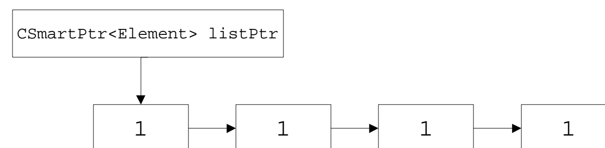

Abbildung 2.17: Einfach verkettete Liste

Bei der Freigabe von *listPtr* wird das erste Element der Liste gelöscht, da außer *listPtr* kein anderes Objekt auf das Element zeigt. In weiterer Folge wird auch das nächste Element gelöscht, da vom ersten Element der Destruktor aufgeführt wird, der automatisch den Destruktor des *Smart Pointers* auf

das nächste Element ausführt. Die gesamte verkettete Liste wird so mit einem Befehl gelöscht.

Bei einer doppelt verketteten Liste ist der Löschvorgang problematischer, wie in Abbildung 2.18 gezeigt wird.

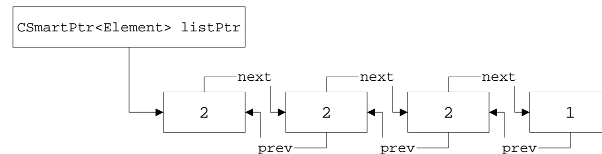

Abbildung 2.18: Doppelt verkettete Liste

Das Problem dabei ist, dass dann wenn der *listPtr* freigegeben wird, das erste Element nicht gelöscht wird, da sein nachfolgendes Element noch einen *Smart Pointer* auf das Element hat. Der Referenzzähler ist in diesem Fall also '2'. Um solch eine Liste zu löschen, muss eine Funktion implementiert werden, die in allen Elementen zumindest den *next* Pointer freigibt.

Damit stellt sich die Frage, warum *Smart Pointer* benutzt werden sollen, wenn der Programmierer das Löschen dieser selbst implementieren muss. Diese Frage wird mit der Darstellung aus Abbildung 2.19 beantwortet.

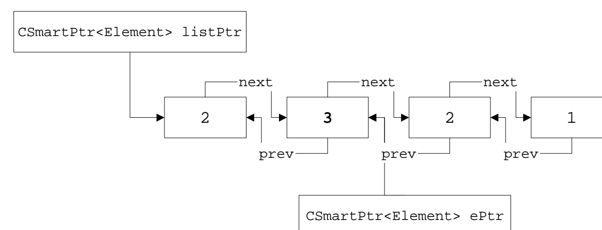

Abbildung 2.19: Doppelt verkettete Liste mit weiterem Zeiger

Die Liste wird bis auf das zweite Element gelöscht, da noch ein *Smart Pointer ePtr* das Element referenziert. Daher kann es zu diesem Zeitpunkt noch nicht gelöscht werden. Der große Vorteil von *Smart Pointern* ist, wie bereits erwähnt, dass sie nur noch zwei Zustände annehmen können. Das Element wird erst gelöscht, wenn *ePtr* das Element freigibt.

2.5 Smart Objects

Wie bereits im Kapitel *Smart Pointer* 2.4 gezeigt wurde, ist es sehr vorteilhaft und praktisch, Standardaufgaben in eine Template-Klasse zu kapseln. Diese Aufgaben beschränken sich aber nicht nur auf Pointer, sondern auch auf die

Verwendung von einem *Mutex* oder einer Datei. In diesem Kapitel wird die Implementierung eines *SmartMutex* gezeigt.

Ein Mutex hat folgende Eigenschaften

- Er kann nur von einem Thread gleichzeitig gesperrt werden.
- Er kann vom selben Thread mehrmals gesperrt werden.
- Jede Sperre muss explizit aufgehoben werden.
- Wurde einmal vergessen, eine Sperre aufzuheben, ist der Mutex unbrauchbar.
- Er ist eine Systemressource, die wieder freigegeben werden muss.

Auch hier wird die Tatsache genutzt, dass am Stack erzeugte Objekte am Scope-Ende wieder gelöscht werden.

```
class CHandle
{
public:
    explicit CHandle(HANDLE hdl) : m_Hdl(hdl) { };
    virtual ~CHandle() { ::CloseHandle(m_Hdl); };
    HANDLE GetHandle() { return m_Hdl; };
private:
    HANDLE m_Hdl;
};

typedef CSmartPtr<CHandle> CHandlePtr;

class CMutexHandle
{
public:
    CMutexHandle() :
    m_MutexHandle(CHandlePtr(new CHandle(
    ::CreateMutex(NULL, FALSE, NULL)))) { };
    virtual ~CMutexHandle() { } ;

    HANDLE GetHandle() const
    { return m_MutexHandle->GetHandle(); };

    CMutexHandle(const CMutexHandle& other)
    : m_MutexHandle(other.m_MutexHandle) { };
    CMutexHandle& operator=(const CMutexHandle& other)
    { m_MutexHandle = other.m_MutexHandle; return *this; };
private:
```

```
    CHandlePtr m_MutexHandle;
};

class CMutexLock
{
public:
    CMutexLock(CMutexHandle& hMutex,
    DWORD dwMilliseconds = INFINITE)
    :m_Mutex(hMutex)
    {
        ::WaitForSingleObject(m_Mutex.GetHandle(),
        dwMilliseconds);
    };
    virtual ~CMutexLock() { ::ReleaseMutex(m_Mutex.GetHandle()); } ;

private:
    CMutexHandle& m_Mutex;
};
```

Der *Smart Mutex* besteht aus vier Klassen:

- Einer generischen Handle-Klasse, die das Handle automatisch schließt.
- Einem Handle *Smart Pointer*, um das Handle weitergeben zu können.
- Einer Mutex Handle Klasse, die den Mutex erzeugt und verwaltet.
- Einer Klasse, die die Sperre und Freigabe des Mutex implementiert.

Die Verwendung ist denkbar einfach, es wird einfach ein *CMutexHandle* Objekt instanziert. Wenn gesperrt werden soll, wird ein *CMutexLock* Objekt instanziert, welches den Mutex sperrt und gegebenenfalls auf die Aufhebung einer anderen Sperre wartet. Am Ende des Scopes, worin die *CMutexLock* Instanz erzeugt wurde, wird der Mutex automatisch freigegeben.

```
CMutexHandle gMutex;                    // Mutex erzeugen,
                                        // automatisch löschen
...
void myFunction()
{
   CMutexLock lock(gMutex);             // Mutex sperren
   ...
}                                       // Mutex freigeben
```

2.6 Observer

Eine weiteres Feature, das in C++ nicht implementiert ist, ist die aktive Benachrichtigung über Ereignisse. Wenn ein Objekt eine Zustandsänderung meldet, sollen jene Objekte, die sich dafür "interessieren" aktiv benachrichtigt werden. Unter "aktiver Benachrichtigung" ist zu verstehen, dass das Objekt nicht permanent nachfragen muss, ob eine Änderung stattgefunden hat, sondern dass es automatisch aufgerufen wird. Auch dieses Problem lässt sich mit einer Templateklasse und einem Makro[10] lösen. Abbildung 2.20 zeigt die Struktur des *Observers*.

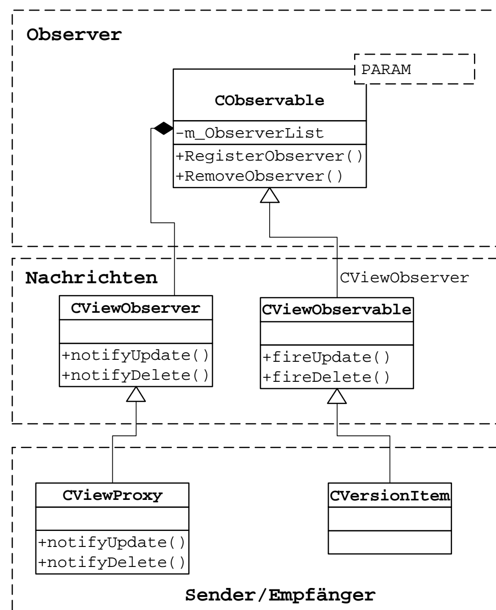

Abbildung 2.20: Observer

Das Objekt, welches Nachrichten versenden möchte, ist in diesem Beispiel *CVersionItem*. Dazu wird eine Klasse definiert, welche die "Feuerfunktionen" implementiert. Die Definition dieser Funktionen ist als Makro implementiert, da jede Funktion die *ObserverList* durchgehen muss, um auf dem *Observer* die entsprechende Notify-Funktion aufzurufen. Die Klasse ist von der Klasse *CObservable* abgeleitet, welche die grundlegende Funktionalität zur Verfügung stellt. Die Klasse *CVersionItem* wiederum ist von *CViewObservable* abgeleitet.

Die Objekte, die Nachrichten empfangen wollen, müssen von einem Interface abgeleitet sein, das die Notify-Funktionen definiert. Das Template *CObservable* nimmt als Template-Parameter dieses Interface. Um die Nachrichten zu empfangen, muss sich das Empfängerobjekt beim Senderobjekt re-

[10] Makros sind zwar nicht besonders gerne gesehen, aber hierbei kann man eine Ausnahme machen.

gistrieren. Diese Funktion ist im Template *CObservable* implementiert. Das Senderobjekt ruft zur Benachrichtigung eine Feuerfunktion auf, die alle registrierten *Observer* aufruft.

Kapitel 3

Ausführung

In den nachfolgenden Kapiteln wird nun, basierend auf dem vorigen Kapitel mit einigen wesentlichen Grundlagen, die IT-Lösung *aceProject* detailliert beschrieben. Ausgehend von einer allgemeinen Darstellung des Funktionsumfanges werden in weiterer Folge die programmtechnischen Detailfragen, soweit diese für das Gesamtverständnis relevant sind, diskutiert.

3.1 Zielsetzung

aceProject wurde für die Planung und Abwicklung von umfangreichen Projekten entwickelt, um den Anwendern eine "ganzheitliche" Sicht auf die gesamten Projektdaten zu ermöglichen. Die wesentlichen Zeilsetungen sind:

- **Einfachste Bedienung** bzw einfache dezentrale Wartung der Projektdaten
- **Schneller zuverlässiger Zugang** - an einer einzigen Stelle sind sämtliche Projektdaten zugänglich
- **Individualisierte Sicht** auf Daten und Informationen
- **Multiprojekt- und Ressourcenplanung**
- **Laufende Wartung** der Daten aus Drittsystemen sowie direkt durch die einzelnen Projektmitarbeiter gewährleistet eine Sicht auf die jeweils aktuellsten Projektdaten (dies ermöglicht wiederum ein frühzeitliches Reagieren auf allfällige Fehlentwicklungen/Termin- oder Kostenüberschreitungen)
- Laufende, automatisierte **Kosten- und Terminüberwachung** (inkl. Benachrichtigung bei Grenzwertüberschreitungen)
- **Projektdaten und -information** sollen zueinander in Beziehung stehen

- **Ganzheitliche Projektdatenverwaltung**, unterschiedliche Sichtweisen auf einen mehr oder weniger umfangreichen Daten- und Informationsbestand
- **Grafische Kapazitätsplanung und -überwachung**

3.2 Integration in eine Gesamtlösung

aceProject lässt sich auch in eine umfassendere Projektmanagementlösung integrieren bei der zusätzliche Themen behandelt werden.

Projektpläne / Projektativitäten

Den Projekten werden *Projektpläne* zugeordnet, diesen wiederum Teilprojekte bzw Arbeitsschritte. Teilprojekten bzw Arbeitsschritten werden Ressourcen/Ressourcengruppen und damit Kosten und Termine zugeordnet. Auf fast allen Ebenen werden laufend Soll-/Ist Kosten verglichen bzw Termine angezeigt.

Diese sind der eigentliche Kern einer solchen Lösung. Projektaktivitäten können auf Basis von "use cases" geplant und durchgeführt werden. Jeder Projektmitarbeiter bekommt Zugriff zu den für ihn relevanten Tätigkeiten, Termine und Kosten werden laufend überwacht, bei Überschreitung der SOLL-Kosten werden die Mitarbeiter oder der Projektleiter automatisiert informiert.

Tickets/Wünsche

Es ist möglich, das gesamte Ticketing bzw das Change-Management in Projekten über ein spezielles Helpdesk-Modul abzuwickeln.

Kommunikation

Alle projektrelevanten Emails, Briefe, Faxe, Papierdokumente, etc können dem Projekt zugeordnet werden.

Projektberichtswesen

Projektberichte werden in der Regel sehr unterschiedlich gestaltet. Durch den Einsatz von Standardtechnologien (SQL, Crystal Reports, PDF) können sehr vielfältige Auswertungen generiert werden. Alternativ zu diesen Auswertungen werden OLAP-basierende Lösungen implementiert.

Zeiterfassung

Die IST-Kosten-Erfassung ist von größter Bedeutung und kann über jedes beliebige Werkzeug erfolgen, diese Daten müssen laufend in das System (automatisiert) eingepflegt werden um so Plan-Ist-Vergleiche durchführen zu können.

Projektteam

Jedem Projekt können Projektteam-Mitglieder zugeordnet werden (Security).

XML-Schnittstellen zu ERP-Systemen

Die Lösung ist völlig offen gestaltet, dh über XML und Biztalk-Server können die Daten mit beliebigen Systemen ausgetauscht werden. Es können auch Views auf die Tabellen in fast beliebiger Weise zur Verfügung gestellt werden.

MS-Project-Import/-Export

Solche Dateien können jederzeit in das System geladen bzw aus diesem sofort extrahiert werden. *aceProject* versteht sich als Ergänzung zu *MSProject*.

3.3 Vorteile der Lösung

- Einheitlicher, schneller online Zugang zu allen aktuellen Daten und Informationen zu einem Projekt ("Single point of access")
- Termin- und Kostenüberwachung (automatisiert)
- Gewährleistung einer Übersichtlichkeit über Projekte und Ressourcen, auch bei hoher Komplexität bzw Vielzahl von Projekten
- Schnelle und weltweite Zugriffsmöglichkeit - auch über PDA's und Handy (IE)
- Laufende Aktualisierung der Daten
- Einfachste Handhabung, Handling weitestgehend selbsterklärend
- Individualisierte Sicht auf Daten - Anwender wird gezielt informiert (Verhindern einer Datenflut)
- Geringer Schulungsaufwand
- Erhöhte Akzeptanz bei den Anwendern durch einfache Bedienung
- Zukunftssicher durch Einsatz von Standardtechnologien

- Vielzahl von Integrationsmöglichkeiten in Drittsysteme (SAP, Components, Web)
- Hohes Maß an Skalierbarkeit

3.4 Funktionsbeschreibung

In den nachfolgenden Kapiteln wird auf die wesentlichen Funktionalitäten der Lösung eingegangen. Dabei können aufgrund der großen Anzahl an Funktionen und Möglichkeiten dieses Produktes nur die wichtigsten beschrieben werden.

Die Grundmotivation dieses Produkt zu entwickeln resultiert aus der Tatsache, das *MSProject* zwar sehr gut mit einem Projekt und den darin vorkommenden Ressourcen umgehen kann, nicht jedoch mit mehreren Projekten gleichzeitig. In einem größerem Unternehmen stellt sich jedoch das Problem, dass es kaum ein Werkzeug gibt, mit dem man eine Ressourcenplanung über alle im Unternehmen befindlichen Projekte durchführen kann. Diese Problemstellung soll das Produkt *aceProject* entschärfen.

3.4.1 Projektansicht

Die Projektansicht ist jene Ansicht, die aus *MSProject* bekannt ist (Abbildung 3.1) und nur insoweit eine Neuerung darstellt, als diese beliebig viele Projekte umfasst.

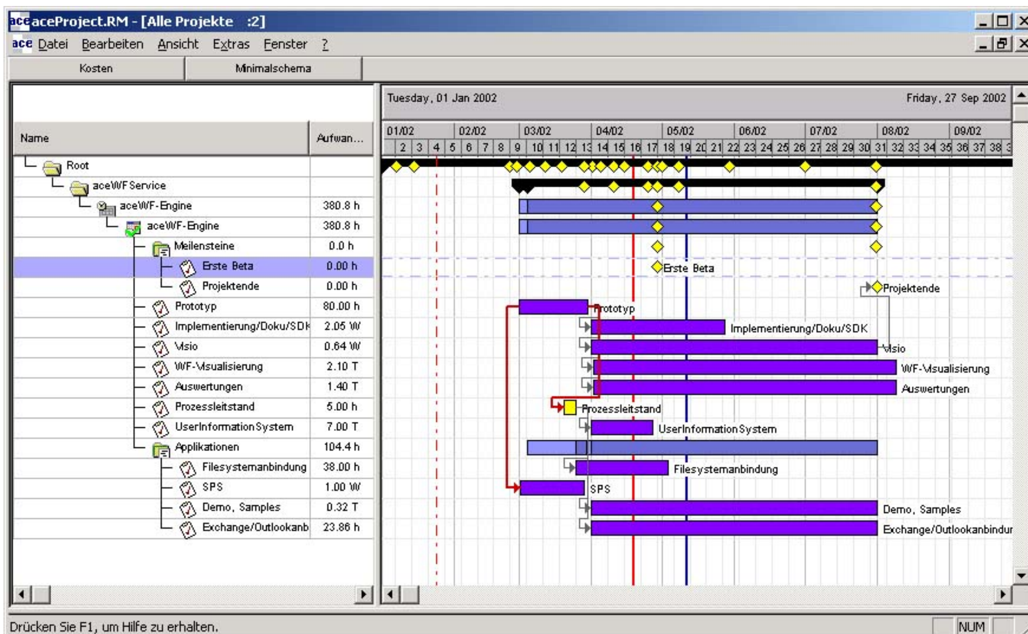

Abbildung 3.1: Projektansicht

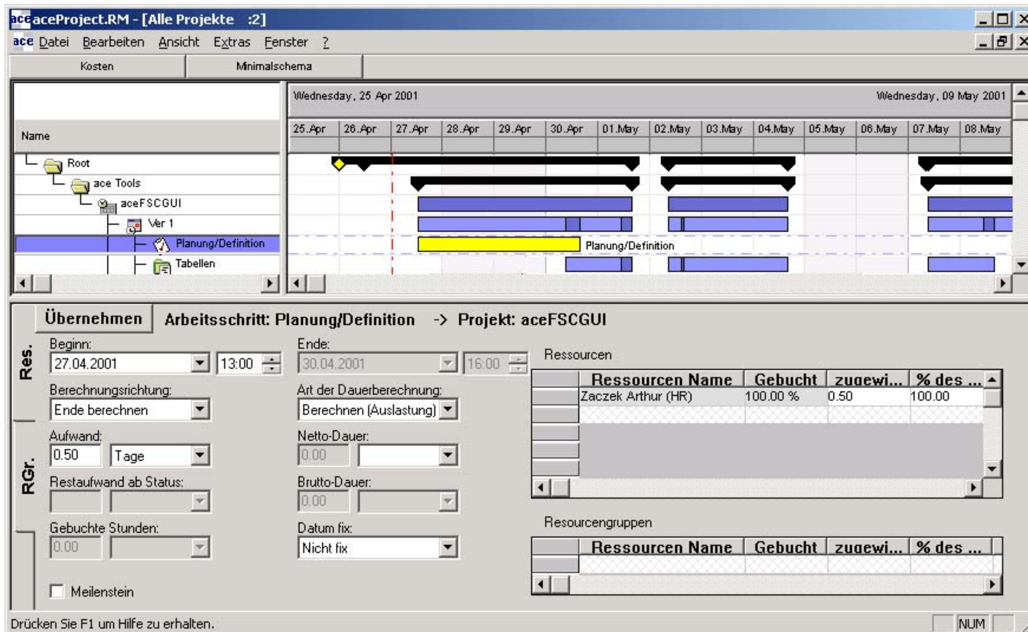

Abbildung 3.2: Eingabe der Daten

Im linken Bildschirmbereich befindet sich der Projektbaum. *Projekte*, symbolisiert mit einem grauen Icon, können in Ordnern abgelegt werden (Element *Root* mit gelbem Icon). "Unterhalb eines *Projektes*" können mehrere *Projektpläne* abgelegt werden. Jeder *Projektplan* stellt eine Version des Projektes zu einem bestimmten Zeitpunkt dar. "Unter den Projektplänen" befinden sich *Sammelvorgänge* und *Arbeitsschritte*. Weiter besteht die Möglichkeit, sich individuell Spalten mit zusätzlichen Informationen aus Drittsystemen einblenden zu lassen. In Abbildung 3.1 und 3.3 wird der Aufwand eingeblendet.

Auf der rechten Seite werden die *Arbeitsschritte*, entsprechend dem bekannten Gantt-Diagramm, grafisch dargestellt. Dabei kann die Farbe des *Arbeitsschrittes* beliebig gewählt werden. *Projekte*, *Sammelvorgänge* und *Ordner* stellen die darunterliegenden *Arbeitsschritte* komprimiert dar. Somit lässt sich sofort erkennen, über welchen Zeitraum sich ein *Projekt* erstreckt.

Um die Eingabe der Daten (Beginnzeiten, Dauer, Aufwand) zu erleichtern, können diese Daten in einem Bereich unter der Ansicht eingegeben werden (Abbildung 3.2).

3.4.2 Ressourcenansicht

Die Ressourcenansicht, die in Abbildung 3.3 gezeigt wird, ist in vielen Fällen die wichtigste Ansicht, die dieses Werkzeug bietet. Es zeigt, "welche *Arbeitsschritte*" in "welchen *Projekten*" eine Ressource (Person, Maschine) zugeordnet hat. Im Gegensatz zu *MSProject* werden dabei alle Projekte berück-

sichtigt. Damit ist eine einfache und schnelle Ressourcenplanung möglich, da Terminkonflikte bzw. Ressourcenkonflikte vorzeitig erkannt werden.

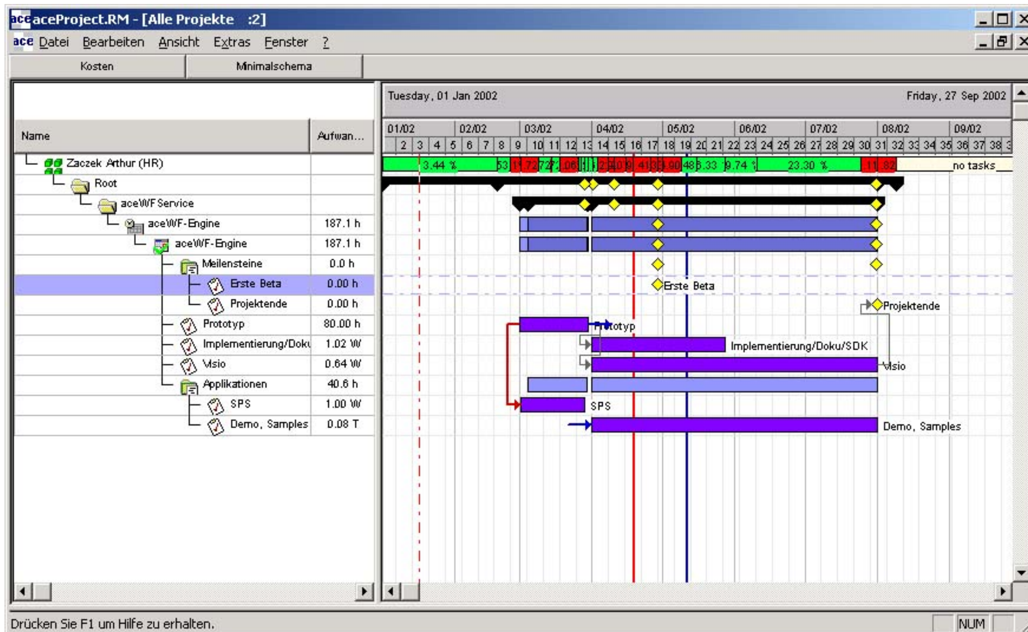

Abbildung 3.3: Ressourcenansicht

Die linke Ansicht stellt wie bei der Projektansicht den hierarchischen Projektbaum dar, mit dem Unterschied, dass hierbei die Wurzel die Ressource ist und nur jene Projekte gezeigt werden, in denen die Ressource eingeteilt ist. Auf der rechten Seite werden die *Arbeitsschritte* grafisch dargestellt. Zusätzlich wird bei der Ressource ihre Auslastung über die Zeit angezeigt. Dabei ist es nicht notwendig, das alle Projekte ausgewählt werden, es werden alle Arbeitsschritte aller Projekte berücksichtigt.

3.4.3 Versionen

Um aus abgeschlossenen Projekten lernen zu können, müssen Vergleiche zu älteren Planungen gezogen werden. Dies unterstützt *aceProject* mittels einer sehr leistungsfähigen Versionierungsfunktion. Dazu werden zu bestimmten Zeitpunkten Duplikate der Projekte erstellt, die dann in der Datenbank abgespeichert werden. Von einem Projekt lässt sich immer nur eine Version bearbeiten, alle anderen werden nicht angezeigt. Um zwei oder drei Versionen vergleichen zu können, werden diese grafisch "übereinander gelegt" (Abbildung 3.4). Diese Einstellung bzw. Ansicht kann über den Filter gesteuert werden.

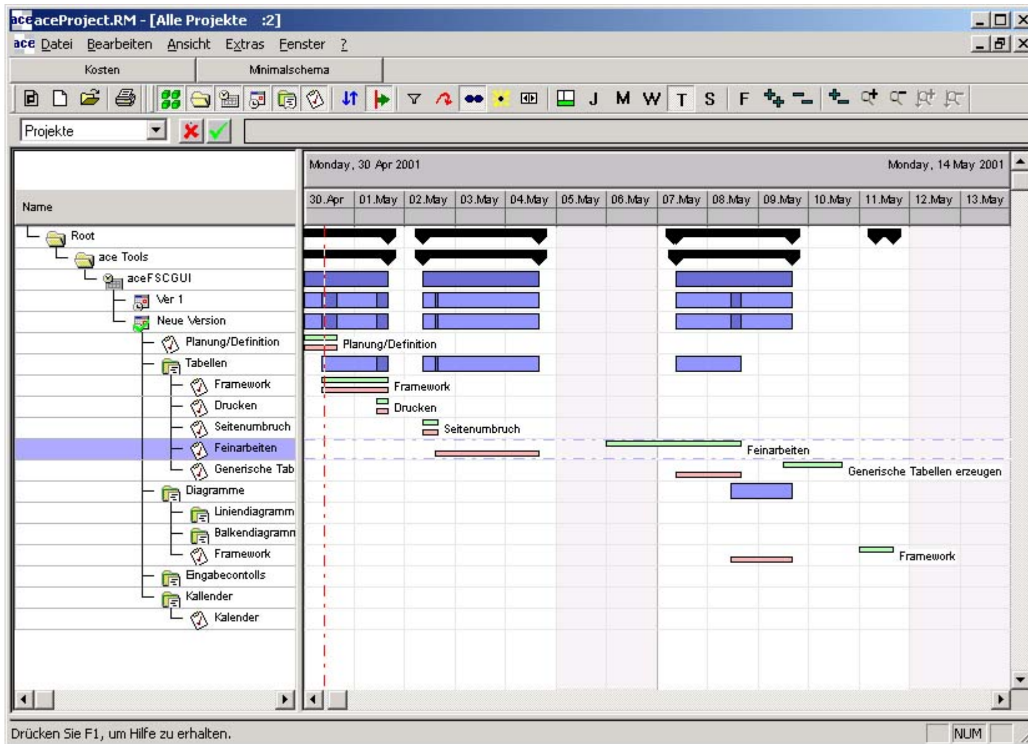

Abbildung 3.4: Versionsvergleich

3.4.4 Filter

Die Menge der Daten bzw. die Größe und Anzahl der Projekte steigt mit der Größe des Unternehmens. Damit der Benutzer nicht mit einer Datenflut konfrontiert wird, kann die Ansicht über *Filter* eingeschränkt werden (Abbildung 3.5). Dabei besteht die Möglichkeit, einzelne *Projekte* oder *Ressourcen* wegzufiltern. Diese Einstellungen können auch kombiniert angewendet werden. Weiter kann im Filter definiert werden, welche Version der Projekte angezeigt werden soll. Außerdem ist es möglich nach Texten zu filtern.

3.4.5 Kapazitätsdiagramm

Das Kapazitätsdiagramm (Abbildung 3.6) zeigt die Auslastung einer Ressource über ein Zeitintervall. Dabei kann wahlweise die Auslastung in Prozent oder die Mannstunden angezeigt werden. Dieses Diagramm ist vom Filter unabhängig. Als Zeitintervall können Tage, Wochen, Monate oder Jahre gewählt werden. Die Werte sind als durchschnittliche Auslastung innerhalb eines Intervalls zu sehen. Wenn eine Ressource, wie in Abbildung 3.6 zu sehen ist, in einer Woche überlastet ist, in den übrigen jedoch nicht, so ist die Auslastung im gesamten Monat ausgeglichen. Laut Abbildung 3.6 würde die Ressource dennoch überlastet sein.

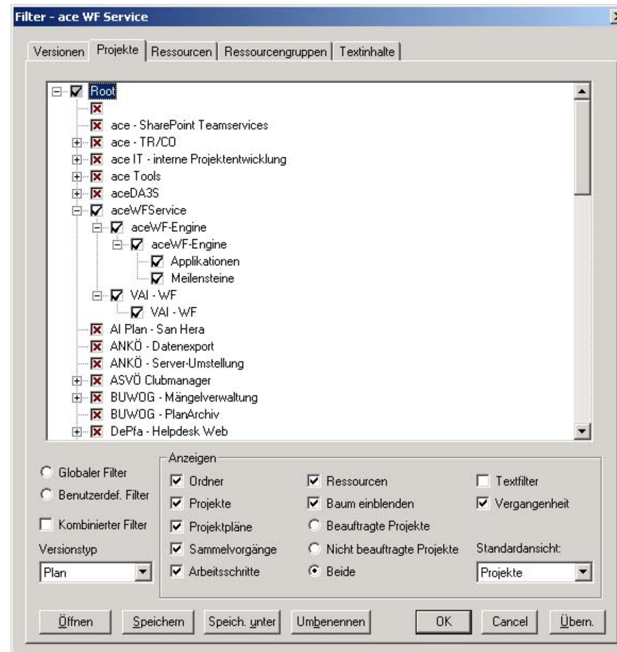

Abbildung 3.5: Filter

Es kann auch für *Projekte* ein Kapazitätsdiagramm erstellt werden. Die Werte geben dann über die benötigten Mannstunden des Projektes Auskunft.

3.5 Das Basiskonzept

Abbildung 3.7 zeigt das Basiskonzept dieser Applikation. Es handelt sich hierbei um eine klassische *3-Tier-Architektur* mit einer Datenbank, einer Business-Logic und einer Präsentationsschicht. Als Datenbank kann wahlweise ein *SQL-Server* oder eine *Oracle-Datenbank* verwendet werden. Außerdem besteht die Möglichkeit, *Fabasoft Components* als Datenbank zu benutzen. Mit dieser objektorientierten Erweiterung einer SQL-Datenbank ist es möglich, den Funktionsumfang einer Projektmanagement-Lösung noch einmal deutlich zu erhöhen.

Die Business-Logic besteht aus mehreren Teilen, die im Kapitel 3.6 ausführlich beschrieben werden. Die Philosophie bei der Implementierung dieses Servers war, dass alle zur Anzeige relevanten Daten im Speicher gehalten werden, im Gegensatz zu den üblichen *3-Tier-Applikationen*. Der Grund liegt in der Forderung nach schnell verfügbaren, bereits aufbereiteten Daten.

Der Server kommuniziert über vier Schnittstellen mit seiner Umwelt, das sind die Datenbank, der Client, die Erweiterungen (z.B. eine Web-Applikation) und die *Fabasoft Components* Anbindung. Die *Client Anbindung* ist die Schnittstelle für den *Win32-Client*. Über diese Schnittstelle werden hauptsächlich für die Anzeige aufbereitete Daten übertragen, wie zum Bei-

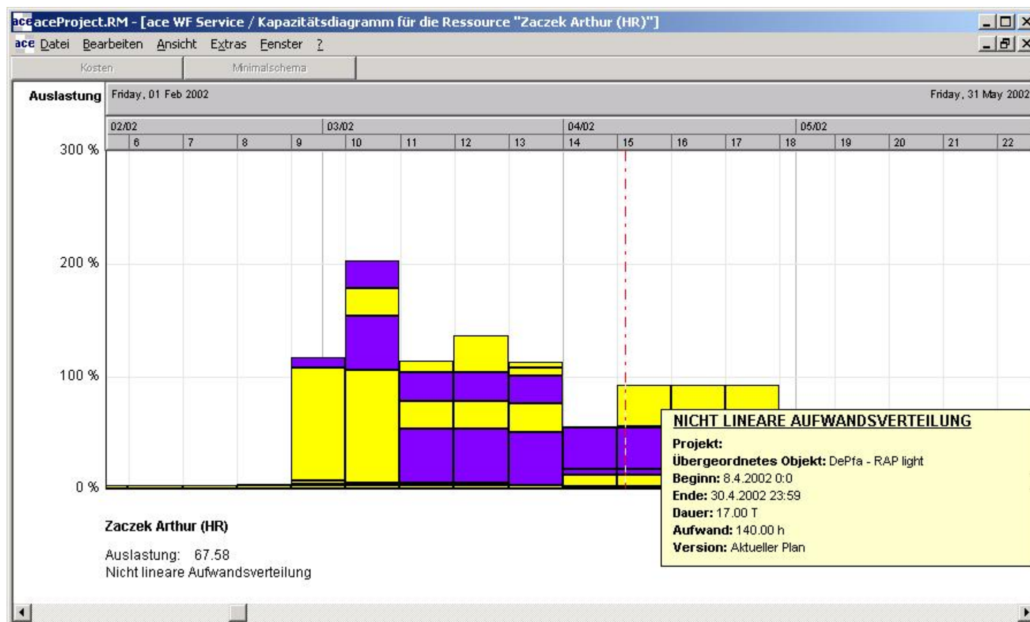

Abbildung 3.6: Kapazitätsdiagramm

spiel die Position eines Objektes innerhalb einer Liste oder einer Zusammenfassung von Werten anderer Objekte. Über die *Plugin-Schnittstelle* können andere Applikationen, wie der *Internet Information Server*, mit dem Server Daten austauschen. Für eine Aufbereitung der Daten hat die Fremdapplikation (z.B. eine Web-Applikation) zu sorgen. Mit den Datenbanken wird über jeweils eigene Schnittstellen kommuniziert, wobei *SQL-Server* und *Oracle* dieselbe Schnittstelle verwenden.

Alle Schnittstellen, bis auf die Datenbankanbindung, sind als COM-Objekte implementiert, diese sind im Kapitel 2.2 behandelt worden.

3.6 Server

3.6.1 Datenmodell

Abbildung 3.8 zeigt Teile der Ableitungshierarchie der einzelnen Klassen. Ein vollständiges Diagramm des Datenmodells ist aufgrund des Umfanges der Lösung nicht mehr darstellbar.

CVersionContainer

Die Aufgabe von *CVersionContainer* ist es, die Versionen und Transaktionen der einzelnen Objekte zu verwalten (siehe 3.6.3 und 3.6.4). Beide Mechanismen werden in den Kapiteln 3.6.3 und 3.6.4 beschrieben. Es werden keine Objekte vom Typ dieser Klasse instanziert, sondern es werden nur Typen

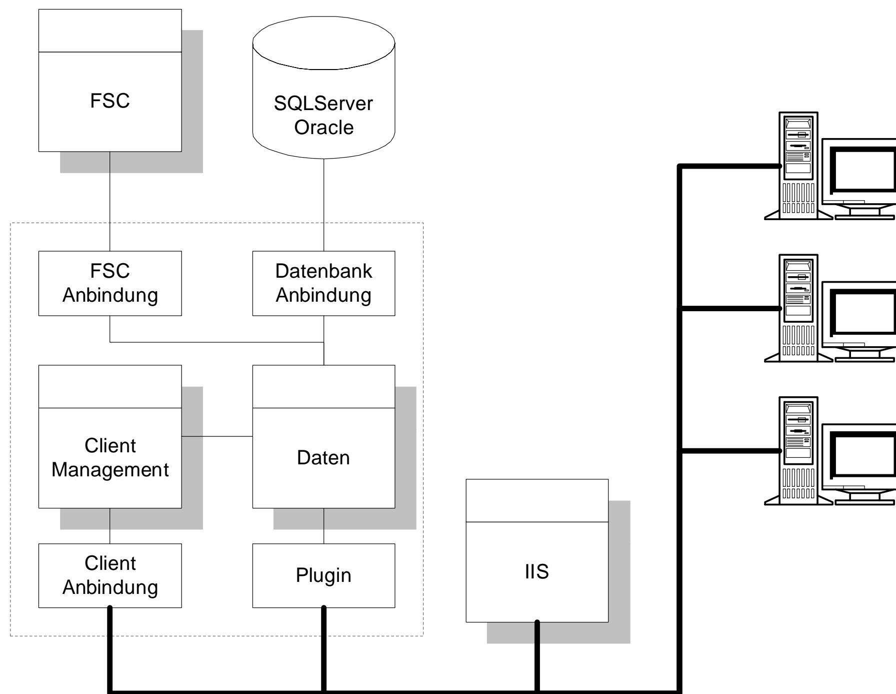

Abbildung 3.7: Grundkonzept

definiert, wie zum Beispiel *CProject* oder *CTask* (siehe Kapitel 2.1.5).

CVersionItem

CVersionItem ist die Basis aller Datenobjekte. Ihre Aufgaben sind Basisfunktionalität für die Datenbankanbindung und ein Interface für Versionierung und Transaktionen zur Verfügung zu stellen.

Observer

Die Observerklassen *CViewObserver*, *CViewObservable* und *CObservable* dienen zur Signalisierung von Änderungen im Datenbereich an die Clientschnittstelle. Die Funktion der "Observer" wurde im Kapitel 2.6 beschrieben.

3.6.2 Datenobjekte

Abbildung 3.9 zeigt die Beziehungen der Datenobjekte untereinander. Der Ausgangspunkt ist die Klasse *CDoc*. Sie hat die absolute Hoheit über alle Datenobjekte. Ihre Aufgabe ist es, die Daten von der Datenbankanbindung (*CDocDatabase* und *CDocFSC*) zu lesen und schreiben, sowie das Erzeugen und Löschen von Objekten während der Laufzeit.

Die Struktur der Daten beginnt mit dem Zeiger auf das erste Projekt. Jeder Zeiger auf ein Datenobjekt ist ein Zeiger auf die *CVersionContainer*

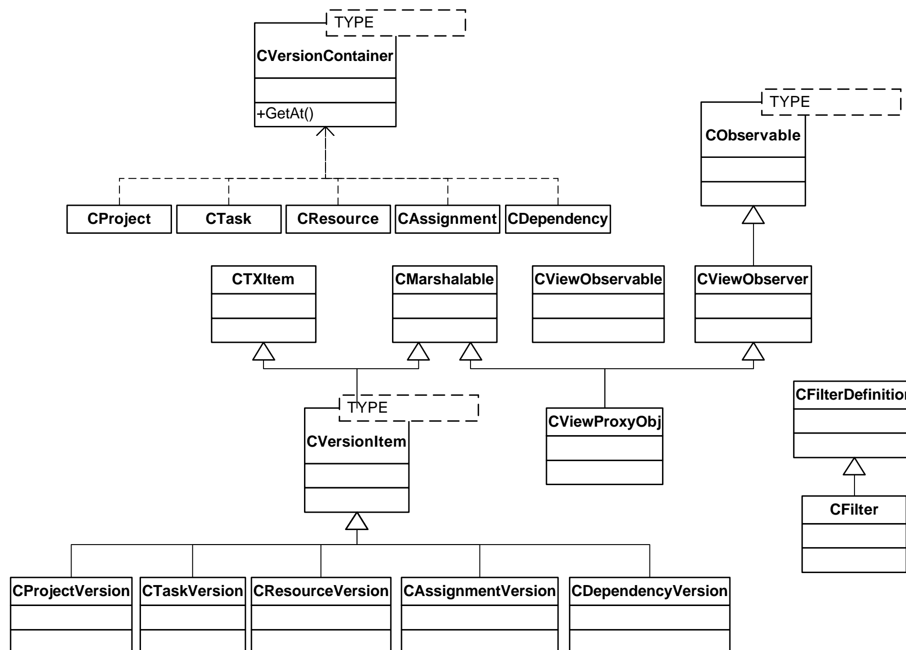

Abbildung 3.8: Ableitungsdiagramm

Template-Instanz. Nur mit Hilfe eines *CFilter* Objektes kann auf das eigentliche Datenobjekt zugegriffen werden.

Alle Datenobjekte sind untereinander mit *SmartPointern* verbunden, welche in den Kapiteln 2.3 und 2.4 beschrieben wurden.

CProject/CProjectVersion

Projekte sind baumartig strukturiert. Ein Projekt kann mehrere Subprojekte oder Arbeitsschritte haben. Diese werden in entsprechenden Collections gespeichert (siehe Kapitel 2.1.6). Jedes Projekt hat einen Zeiger auf sein übergeordnetes Projekt. Die Wurzel dieses Projektbaumes hat eine Sonderfunktion. Sie wird automatisch erzeugt, hat kein übergeordnetes Projekt und wird als einziges Projekt in der Klasse *CDoc* gespeichert.

Es gibt mehrere Typen von Projekten, die sich nur bei der Anzeige durch das Icon unterscheiden und in ihrem Verhalten in der Listenansicht. Folgende Projekttypen sind im Augenblick definiert:

- Ordner
- Projekt
- Projektplan
- Aktueller Projektplan

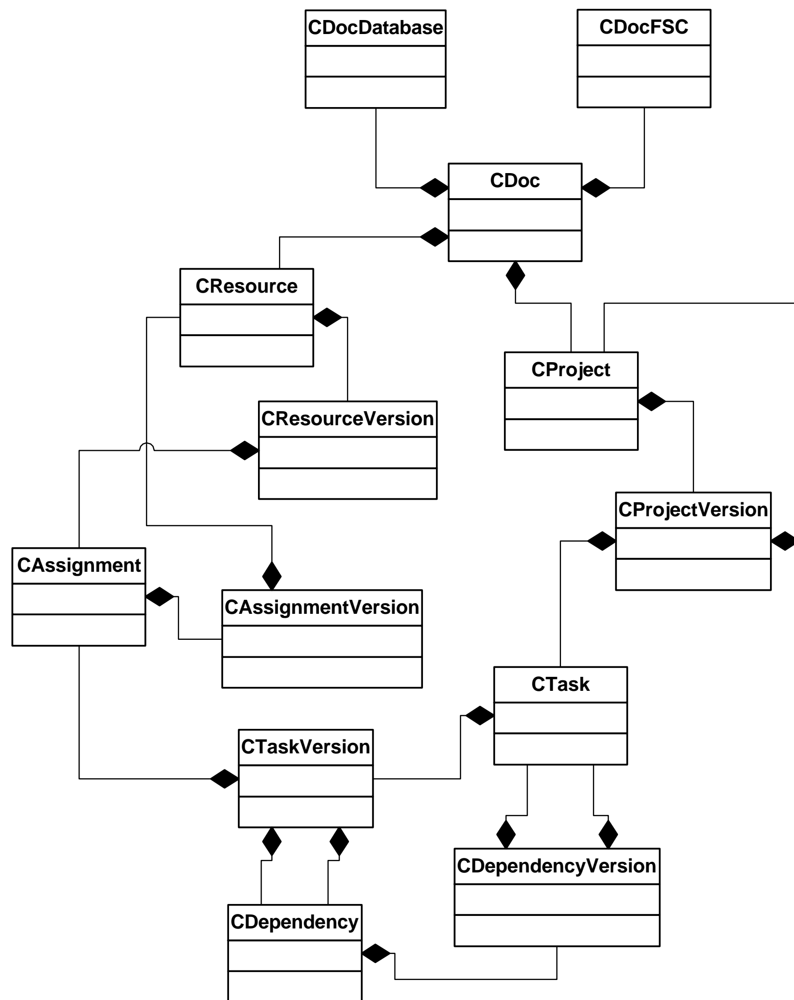

Abbildung 3.9: Strukturdiagramm Daten

- Projektplanvorlage
- Sammelvorgang/Subprojekt

Für den schnelleren Aufbau der Ressourcenansicht sind im Projekt alle Ressourcen der Arbeitsschritte innerhalb des Projektes indiziert.

CTask/CTaskVersion

Ein Arbeitsschritt ist immer einem Projekt zugeordnet. Er besteht aus folgenden Informationen:

- Datum von/bis
- Aufwand

- Nettodauer
- Bruttodauer
- Meilenstein
- Art der Berechnung

Arbeitsschritte können über Abhängigkeiten miteinander verbunden werden. Außerdem können einem Arbeitsschritt ein oder mehrere Ressourcen bzw. Ressourcengruppen zugeordnet werden.

CDependency/CDependencyVersion

Dieses Datenobjekt stellt eine Abhängigkeit zwischen zwei Arbeitsschritten dar. Es gibt vier Arten von Abhängigkeiten:

- Anfang-Ende
- Ende-Ende
- Anfang-Anfang
- Ende-Anfang

Außerdem kann eine Verzögerung zwischen den Arbeitsschritten definiert werden.

CResource/CResourceVersion

Eine Ressource stellt die ausführenden Personen/Maschinen der Arbeitsschritte dar. Sie wird über ein Assignment einem Arbeitsschritt zugeordnet. Jede Ressource verfügt über einen Kalender, in dem die Verfügbarkeit eingetragen wird. Es wird zwischen Personen/Maschinen oder einer Gruppe/Abteilung unterscheiden.

Die maximale Verfügbarkeit einer Ressource wird über den Kalender und einem Multiplikator definiert. Damit kann bei Abteilungen einen Kalender für die Normarbeitszeit hinterlegt werden und über den Multiplikator die Anzahl der Personen bestimmt werden.

CAssignment/CAssignmentVersion

Das Assigmnent ist das Bindeglied zwischen Arbeitsschritten und Ressourcen. Mit dem Aufwand, mit dem die Ressource am Arbeitsschritt beteiligt ist und unter Berücksichtigung des Kalenders, wird die Auslastung der Ressource berechnet. Diese Daten werden am Client in Form von Diagrammen dargestellt.

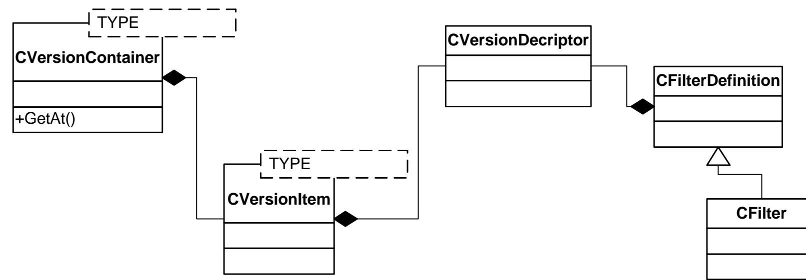

Abbildung 3.10: Versionen

3.6.3 Versionen

Abbildung 3.10 zeigt die an der Versionierung beteiligten Klassen. Wie bereits erwähnt, wird jedes Datenobjekt durch sein *CVersionContainer* Objekt angesprochen. Um jetzt ein Versionsobjekt zu erhalten, muss auf dem Container die Methode *GetAt()* aufgerufen werden. Diese Methode nimmt als Parameter einen Filter, der festlegt, welche Version zurückgegeben werden soll.

Das Template *CVersionContainer* hat einen Zeiger auf die aktuelle Version (MasterVersion) und eine Liste aller Nebenversionen. Das Template *CVersionItem*, das die Version darstellt, hat einen Zeiger auf einen *CVersionDescriptor*, der die Version identifiziert. Die Klasse *CFilter* hat eine Liste mit allen *CVersionDectiptor* Objekten, die der Benutzer angezeigt haben möchte. Beim Aufruf der Methode *CVersionContainer::GetAt()* wird nach einem *CVersionDescriptor* gesucht, der in beiden Listen von *CVersionContainer* und *CFilter* vorkommt. Wenn kein *CVserionDescriptor* gefunden wird, wird die aktuelle Version zurückgegeben. Damit ist sichergestellt, dass die Methode *GetAt()* immer ein gültiges Objekt zurückgibt. Sollten mehrere Versionen gefunden werden, gilt der Grundsatz "*first come, first serve*".

3.6.4 Transaktionen

Abbildung 3.11 zeigt die an einer Transaktion beteiligten Klassen. Für den Transaktionsmechanismus wurde im *CVersionContainer* die Liste der Versionen abgeändert auf eine Liste von Transaktionsobjekten *TX_ITEM*. Ein Transaktionsobjekt hat einen Zeiger auf das Versionsobjekt und einen auf das sich in der Transaktion befindliche Versionsobjekt. Abbildung 3.12 zeigt die Vorgehensweise für die Aufnahme der Objekte in die Transaktion.

Objekt in die Transaktion aufnehmen

Um ein Objekt bearbeiten zu können, muss zuerst eine Transaktion eröffnet werden. Dies geschieht mit dem Aufruf *BeginTransaction()* am Filter. Bei

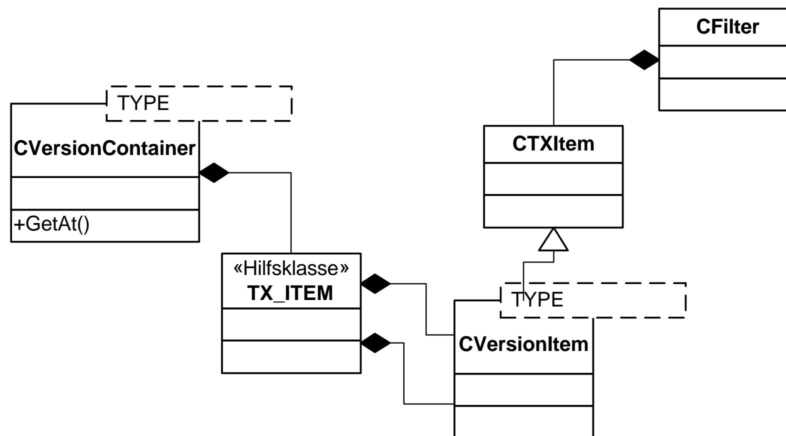

Abbildung 3.11: Transaktionsobjekte

diesem Aufruf wird zunächst nur der Filter in den Transaktionsmodus versetzt und eine Transaktionsnummer (*TransactionID*) generiert. Beim Aufruf *CVersionContainer::GetAt()* wird zunächst geprüft, ob der Filter eine Transaktion hat. Wenn dies nicht der Fall ist, wird einfach das Versionsobjekt (Kapitel 3.6.3) zurückgegeben. Sollte versucht werden, an diesem Objekt etwas zu ändern, wirft *CVserionItem* beim Aufruf von *SetDirty()* eine Exception.

Wenn es sich bei dem übergebenen Filter um einen Transaktionsfilter handelt, wird versucht, das Objekt im Schreibmodus zu bekommen. Da das Objekt unter Umständen von einer anderen Transaktion gesperrt sein kann, wird der Zugriff mehrmals versucht. Danach wird festgestellt, ob sich das Objekt bereits in einer Transaktion befindet. Wenn nicht, und es sich aus Effizienzgründen um einen Lesezugriff handelt, wird das Versionsobjekt zurückgegeben. Andernfalls wird das Objekt in die Transaktion aufgenommen. Dazu wird ein neues Versionsobjekt erstellt, dass sofort darüber in Kenntnis gesetzt wird, dass es als Transaktionsobjekt erstellt wurde. Danach werden die Daten des Originalobjektes in das neue Objekt kopiert. Als letztes wird in der Struktur *TX_ITEM* das neu erstellte Objekt zusammen mit der Transaktionsnummer gespeichert.

Sollte sich das Objekt bereits in einer Transaktion befinden, wird geprüft, ob es sich um dieselbe Transaktion handelt. Wenn es sich nicht um dieselbe Transaktion handelt, wird auf die Freigabe des Objektes gewartet.

Transaktion abschließen

Um die Änderungen zu speichern, wird am Transaktionsfilter die Methode *CFilter::CommitTransaction()* aufgerufen. Dieser Vorgang wird in Abbildung 3.13 gezeigt.

Das *Commit* besteht aus drei Phasen. Zunächst werden alle Objekte in die Datenbank geschrieben. Das Ergebnis des Datenbank-Commits wird

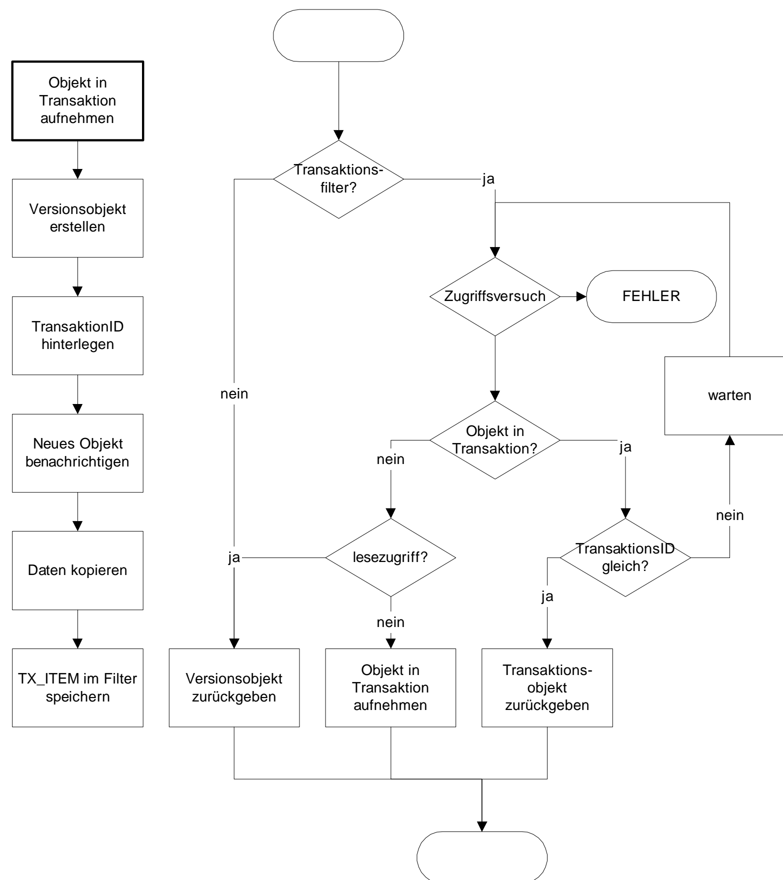

Abbildung 3.12: Objekt in Transaktion aufnehmen

zunächst zwischengespeichert. Abschließend kommt die kritischste Phase. Wenn jetzt ein unvorhergesehener Fehler auftritt, muss der Server neu gestartet werden, da sonst einige Objekte in der Transaktion hängen bleiben könnten und die Daten inkonsistent wären.

Alle Objekte werden im Speicher in die Versionsobjekte zurückgeschrieben, wenn das Datenbank-Commit erfolgreich war. War das Commit nicht erfolgreich, werden alle Transaktionsobjekte darüber benachrichtigt und anschließend gelöscht. Wenn ein Objekt als gelöscht markiert ist, wird das betreffende Versionsobjekt darüber informiert und anschließend aus dem Speicher entfernt. Ansonsten wird das Transaktionsobjekt über das erfolgreiche Commit informiert und die Daten des Objektes werden in das Versionsobjekt kopiert. Abschließend wird die Transaktionsnummer in der Struktur *TX_ITEM* zurückgesetzt und das Transaktionsobjekt gelöscht.

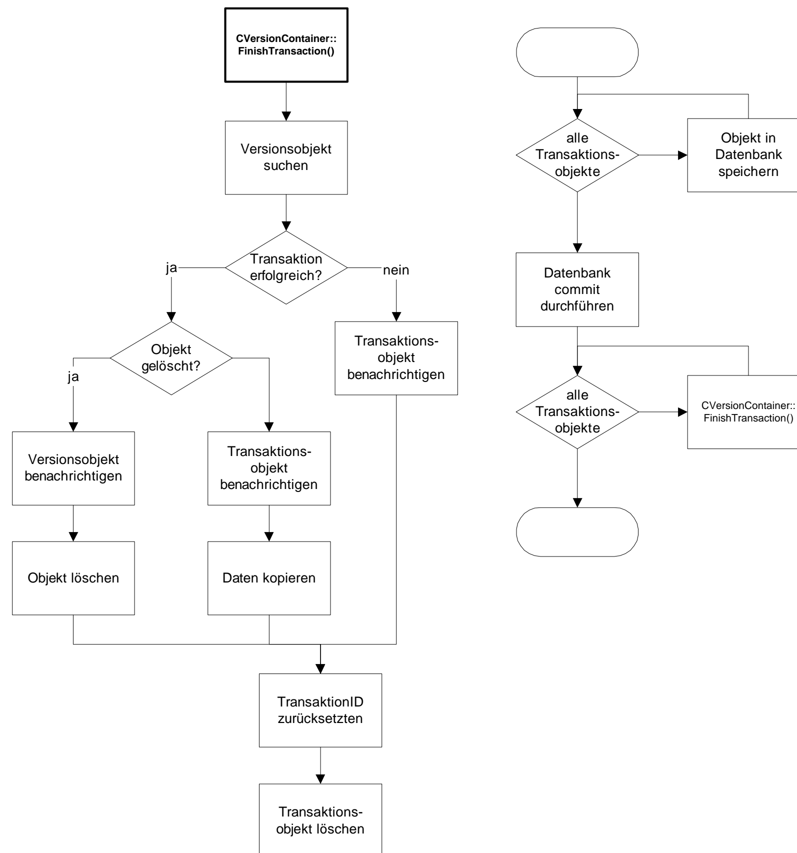

Abbildung 3.13: Transaktion abschließen

Transaktion abbrechen

Wird die Transaktion abgebrochen, so werden alle Objekte wie beim Commit aufgerufen, allerdings mit der Information, dass das Commit fehlgeschlagen ist.

3.6.5 Datenbank

SQL Server

Die Anbindung an einen SQL Server besteht aus einer Reihe von Klassen, die in der Lage sind, ein Objekt in die Datenbank zu schreiben und aus der Datenbank zu lesen. Weiters haben diese Klassen Funktionalitäten eingebaut, um selbständig die Datenbankstruktur aufzubauen. Abbildung A.1, A.2 und A.3 im Anhang zeigen das Datenbankmodell.

Als Grundlage dieser Klassen wird von der *MFC-Library* die Klasse *CRecordset* benutzt. Davon ist die Klasse *CCheckRecordset* abgeleitet, die den Aufbau der Datenbank übernimmt. Von dieser Klasse sind alle anderen Datenbankklassen abgeleitet. Die Anbindung an die Datenbank erfolgt über

ODBC mit der *MFC-Klasse CDatabase*.

Für das Speichern und Laden der Objekte ist ausschließlich die Klasse *CDoc* verantwortlich, da bei einer Anbindung an *Fabasoft Components* je nach Objekt in *Fabasoft Components* oder in die Datenbank geschrieben bzw. gelesen wird. Pro Objekt stehen entsprechende Funktionen zur Verfügung.

Fabasoft Components (FSC) Server

Die Anbindung von *Fabasoft Components* an den Server gestaltet sich etwas schwieriger, da *Fabasoft Components* mehr als nur eine objektorientierte Datenbank ist. Es besteht die Möglichkeit, Daten über mehrere Wege zu verändern, zum Beispiel über den Client oder das Web. Das Problem dabei ist, die Datenbestände am Server und in *Fabasoft Components* synchron zu halten. Eine Richtung ist trivial, wenn nämlich am Server Daten verändert werden. Dazu werden die Daten wie bei der (reinen) SQL-Server-/Oracle-Variante sofort zurückgeschrieben.

Wenn Daten in *Fabasoft Components* selbst geändert werden, muss der Server darüber informiert werden. Dazu wird bei der *Fabasoft Components* Methode *ObjectCommitted()* eine Funktion definiert, die den Server über die Veränderung eines Datensatzes benachrichtigt. Abbildung 3.14 zeigt diesen Vorgang.

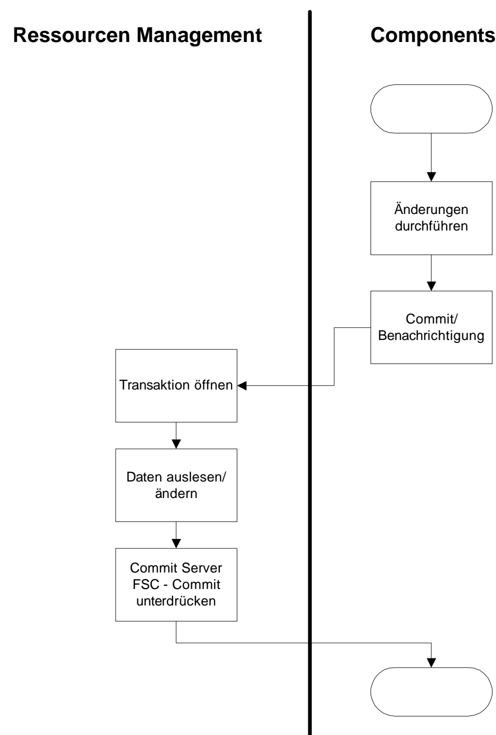

Abbildung 3.14: Commit im Components

Das Commit der Daten am Server stellt eine technische Herausforderung dar. Hier besteht das Problem, dass Objekte in *Fabasoft Components* geändert werden, die danach ein Commit in *Fabasoft Components* verursachen, es aber keine Möglichkeit gibt, die Benachrichtigung zu unterdrücken. Es gibt zwar die Möglichkeit eine *Transaktionsvariable* zu definieren, allerdings findet das *Commit* in einer neuen, eigenen Transaktion statt, so dass man keine Möglichkeit hat, die Benachrichtigung zu unterdrücken. Die einzige Möglichkeit eine "Unendlichschleife" zu verhindern ist, die Benachrichtigung zuzulassen und anschließend im Server das Änderungsdatum des Objektes zu prüfen. Abbildung 3.15 zeigt diesen Vorgang.

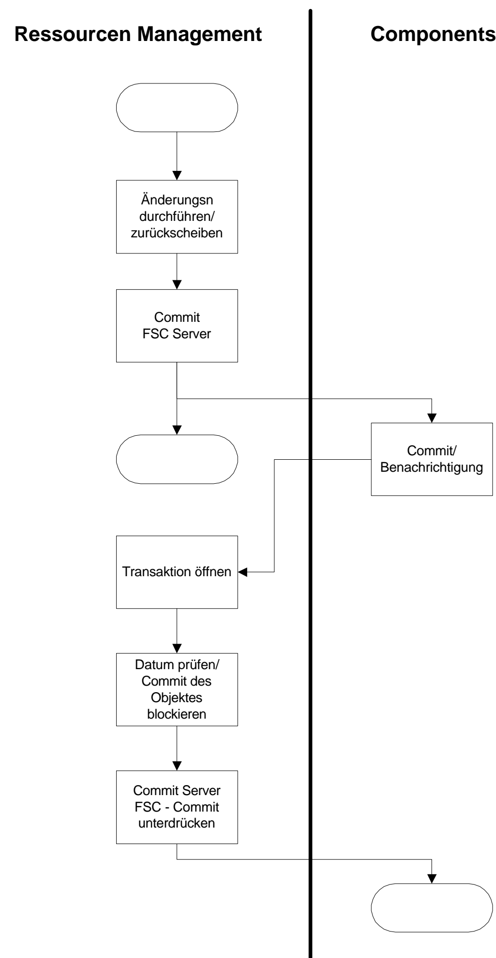

Abbildung 3.15: Commit im Server

Zunächst wird serverseitig eine Transaktion geöffnet, um die Daten in den Speicher, bzw. ins *Fabasoft Components* zu schreiben. Danach wird die Transaktion sowohl im Speicher als auch in *Fabasoft Components* committed. Hier trennt sich der Ablauf zwischen Server und *Fabasoft Components*. *Fabasoft Components* startet sofort nach dem Commit eine weitere Transaktion, um

die Methode *ObjectCommitted* auszuführen. Diese Methode benachrichtigt den Server über ein geändertes Objekt, das allerdings der Server geändert hat. Beim Aufruf der Servers prüft dieser das Änderungsdatum, um das Objekt nicht noch einmal zu ändern.

3.6.6 Security

Berechtigungen auf Daten werden in zwei Gruppen eingeteilt. Die erste Gruppe betrifft allgemeine Einstellungen, wie Kalender oder Balkenfarben. Diese Einstellungen können nur von einem Administrator durchgeführt werden.

Die zweite Gruppe von Berechtigungen bezieht sich auf den Zugriff auf Projekte. Abbildung 3.16 zeigt das Datenmodell.

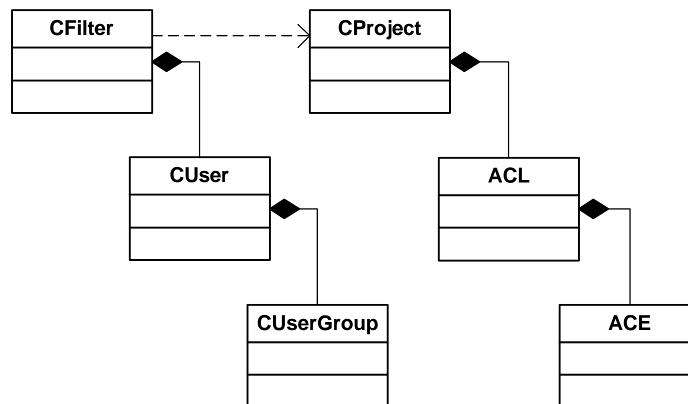

Abbildung 3.16: Security

Bei jedem Projekt ist eine *Access Control List (ACL)* hinterlegt. Diese legt fest, welche Benutzer/Gruppen welche Rechte auf dem Projekt haben. Diese Information wird in einem *Access Control Entry* gespeichert. Dieses Modell ist stark an das *Windows* Security Modell angelehnt.

Bei jedem Filter ist der ausführende Benutzer gespeichert, der sich zwingend authentifizieren muss. Die Authentifizierung erfolgt mit der *Windows-API*. Jeder Benutzer kann Mitglied von einer oder mehreren Gruppen sein. Beim Zugriff auf ein Projekt wird geprüft, ob sich der Benutzer oder eine seiner Gruppen in der ACL befindet. Wenn nicht, wird der Zugriff mittels Exception verhindert.

3.6.7 Berechnungen

Kalender

Der Kalender ist die Basis aller Zeitberechnungen. Jede Ressource muss zwingend einem Kalender zugeordnet sein. Für die Berechnung der Nettodauer wird ein Standardkalender verwendet.

Der Kalender unterscheidet zwischen Arbeitszeiten und Ruhezeiten. Ein Eintrag ist wie folgt aufgebaut:

Datum	Wochentag	Beginnzeit	Endzeit	Arbeit/Ruhe

- Ist das Feld *Datum* gültig, gilt dieser Eintrag nur für diesen Tag. Das Feld *Wochentag* wird ignoriert (Tagesregel).
- Ist das Feld *Datum* ungültig und das Feld *Wochentag* gültig, gilt dieser Eintrag an jedem *Wochentag* (Wochentagsregel).
- Sind die Felder *Datum* und *Wochentag* ungültig, gilt dieser Eintrag an jedem Tag (allgemeine Regel).

Ein Kalender kann von einem Basiskalender abgeleitet werden. Die Vererbung kann beliebig oft durchgeführt werden. Ein abgeleiteter Kalender enthält nur die Änderungen zum Basiskalender.

Kalendereinträge suchen

Zuerst wird nach einem Eintrag für das aktuelle Datum gesucht (Tagesregel), danach für den aktuellen Wochentag (Wochentagsregel). Als letztes wird eine allgemeine Regel gesucht. Wird beim aktuellen Kalender kein passender Eintrag gefunden, wird im Basiskalender nach einem zutreffenden Eintrag gesucht. Die Suche im Basiskalender erfolgt für jeden Suchschritt. Ein Beispiel: Wird kein Eintrag für das aktuelle Datum gefunden (Tagesregel), wird im Basiskalender gesucht.

Berechnung der Dauer

Vom Startdatum ausgehend wird der Kalender solange durchlaufen, bis der mitgezählte Aufwand größer oder gleich dem gegebenen Aufwand ist.

Berechnung der Auslastung

Die Auslastung ergibt sich aus der Formel

$$\text{Auslastung}[\%] = \frac{\text{Aufwand}}{\text{Aufwand lt. Kalender}} \times 100$$

Beispiel: Ein Arbeitsschritt dauert ein Monat, der Kalender definiert diesen Monat mit maximal 173 Arbeitsstunden, der Aufwand beträgt 100 Stunden.

$$\frac{100}{173} \times 100 = 58\%$$

Berechnung der Dauer für mehrere Ressourcen

Wenn einem Arbeitsschritt mehrere Ressourcen zugeordnet sind, so kann die Dauer nur näherungsweise berechnet werden, da die Ressourcen unterschiedliche Kalender haben können.

Als erstes wird allen Arbeitsschritten der gleiche Aufwand zugewiesen.

$$\text{RessourcenAufwand}[i] = \frac{\text{Aufwand}}{\text{Anzahl der Ressourcen}}$$

Danach werden die Endtermine für jede Ressource berechnet.

$$Endtermine[i] = CalcFinishDate(start,\ dur[i])$$

Aus den Endterminen wird der Durchschnitt gebildet und damit ein "virtueller Endtermin" festgelegt (Abbildung 3.17).

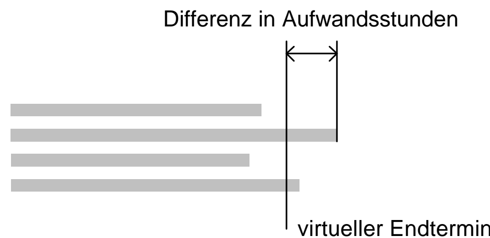

Abbildung 3.17: Berechnung der Dauer für mehrere Ressourcen

Allen Arbeitsschritten, die über den virtuellen Endtermin hinausgehen, werden die überschüssigen Aufwände abgezogen, allen Arbeitsschritten, die darunter liegen, werden diese Aufwände hinzugerechnet. Danach werden wieder die Endtermine berechnet.

Die Schleife bricht ab

- nach den maximal möglichen Durchgängen
- wenn die größte Zeitabweichung unter einer Stunde liegt.

Zusammenfassung der Ressourcen

Bei jeder Ressource wird ein Auslastungsdiagramm über die Zeit gespeichert, die am Client in Form von gelben, grünen bzw. roten Balken dargestellt wird. Für diese Zusammenfassung werden alle Arbeitsschritte, an der die Ressource beteiligt ist, herangezogen. In einer Liste werden die Anfangszeiten und die positive Auslastung des *Assignment* sowie die Endzeiten und die negative Auslastung eingetragen. Ist an einem Zeitpunk bereits ein Eintrag vorhanden, so wird der Auslastungswert korrigiert. Mit diesen Werten kann später am Client die tatsächliche Auslastung berechnet werden, indem über eine Schleife ein Zähler den aktuellen Auslastungswert mitzählt. Abbildung 3.18 veranschaulicht diesen Vorgang.

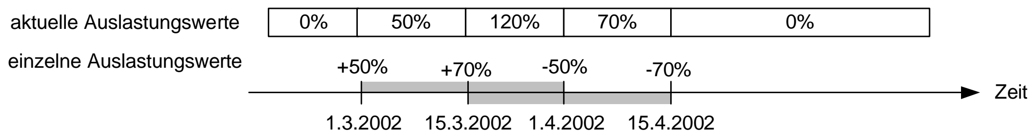

Abbildung 3.18: Zusammenfassung der Ressourcen

Zusammenfassung der Projekte

Bei jedem Projekt sind alle Arbeitsschritte noch einmal zusammengefasst gespeichert. Die Berechnung dieser Zusammenfassung ist ident mit der Ressourcenzusammenfassung, nur wird anstatt der Auslastung der Wert +1 bzw. -1 verwendet. Ein weiterer Unterschied besteht darin, dass alle Werte über 2 aus der Zusammenfassung gelöscht werden.

Zusammenfassung der Projekte in der Ressourcenansicht

In dieser Ansicht werden nur jene Arbeitsschritte zur Berechnung herangezogen, bei der die betreffende Ressource zugeteilt ist. Dazu wird bei jedem Projekt ein Ressourcenindex aufgebaut. In einem solchen Eintrag wird die Ressource und ihre Häufigkeit gespeichert. Der Eintrag wird vom betreffenden Arbeitsschritt verwaltet. Ansonsten ist die Berechnung die gleiche wie bei der Projektzusammenfassung.

3.6.8 Schnittstellen

Der Server hat nach außen zwei Schnittstellen. Beide Schnittstellen sind als COM-Schnittstellen ausgeführt (siehe Kapitel 2.2). Die erste Schnittstelle ist für die Anbindung des Win32 Clients zuständig. Ihre Aufgabe ist es, dem Client die Daten so aufzubereiten, dass er sie schnell und einfach darstellen kann. Die zweite Schnittstelle dient zur Anbindung von Fremdapplikationen, wie zum Beispiel einem Web-Portal.

Client

Jedes Datenobjekt am Server hat eine eindeutige Nummer, eine *ID*. Dazu hat jedes Datenobjekt, welches zum Client übertragen wird, eine zweite eindeutige Nummer, die *Referenz*. Jedes Datenobjekt, das am Client in einem Baum dargestellt wird, hat am Server pro Ansicht am Client ein *Proxyobjekt*, das auch über eine *Referenz* verfügt. Abbildung 3.19 zeigt den Zusammenhang und die beteiligten Klassen.

Für die Übertragung der Objekte vom und zum Client ist die Klasse *CMarshalable*, von der alle Datenobjekte und die *Proxyobjekte* abgeleitet sind, zuständig. Diese Klasse zwingt die Objekte, Methoden zu implementieren, die dafür sorgen, dass die Daten des Objektes in einen *Stream* verpackt

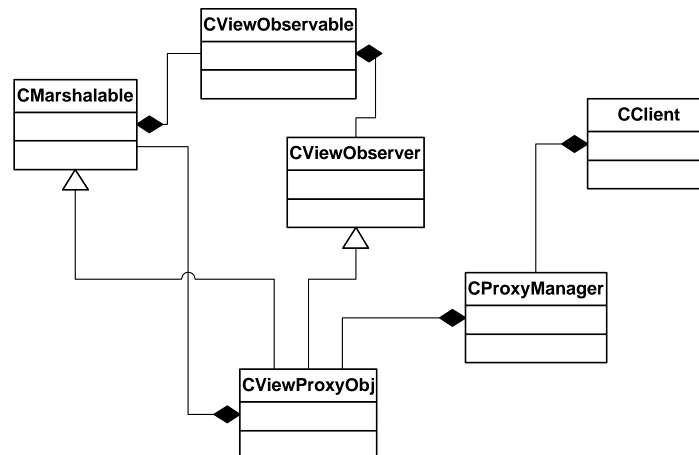

Abbildung 3.19: Strukturdiagramm Clientanbindung

und von dort wieder auspackt werden. Dieser Vorgang wird als *Marshaling* bezeichnet. Der *Stream* wird dann zum Client übertragen, der daraus wieder ein Objekt erzeugt, das er darstellen oder verarbeiten kann. Werden Daten am Client geändert, so wird dieser Vorgang einfach umgekehrt.

In der Projekt- und Ressourcenansicht ist es notwendig, die darzustellenden Objekte in einem Baum abzubilden. Da die einzelnen Elemente der Bäume, wovon jeder Client mindestens einen sein eigen nennt, dasselbe Datenobjekt darstellen, muss dieses Element ein anderes Objekt sein als das Datenobjekt. Dazu wurden die *Proxyobjekte* eingeführt. Diese Proxyelemente enthalten neben dem Zeiger auf das Datenobjekt auf ansichtsspezifische Informationen, wie die Position innerhalb des Baumes oder der Status *offen/geschlossen*. Diese Informationen werden deshalb serverseitig gespeichert, um zu verhindern, dass der Client den vollständigen Baum übertragen und speichern muss. Bei größeren Bäumen könnte das zu lange dauern. Die Philosophie bei dieser Anbindung ist es, nur jene Daten zu übertragen, die gerade zur Anzeige notwendig sind. Abbildung 3.20 zeigt den Aufbau des Baumes und seine Hilfslisten.

Der Baum wird aufgebaut, indem aufgrund der Filterdefinition alle betreffenden Datenobjekte aufgefordert werden, ein *Proxyobjekt* zu erzeugen. Dieses *Proxyobjekt* wird an geeigneter Stelle in den Baum gehängt. Dabei kann es vorkommen, dass ein Datenobjekt zwei *Proxyobjekt* erzeugen muss.

Da der Baum in der Darstellung eigentlich eine Liste ist, muss diese vorher noch erzeugt werden. Diese Liste wird *Indexlist* genannt. Zum Aufbau dieser Liste wird der Baum, unter Berücksichtigung des offen/geschlossen Status, durchlaufen und die Referenz des *Proxyobjektes* wird in die Liste eingetragen. Öffnet oder schließt der Benutzer einen Zweig im Baum, muss die Liste neu aufgebaut werden.

In Abbildung 3.20 wird beispielsweise gezeigt, dass der Zweig *SV 1* ge-

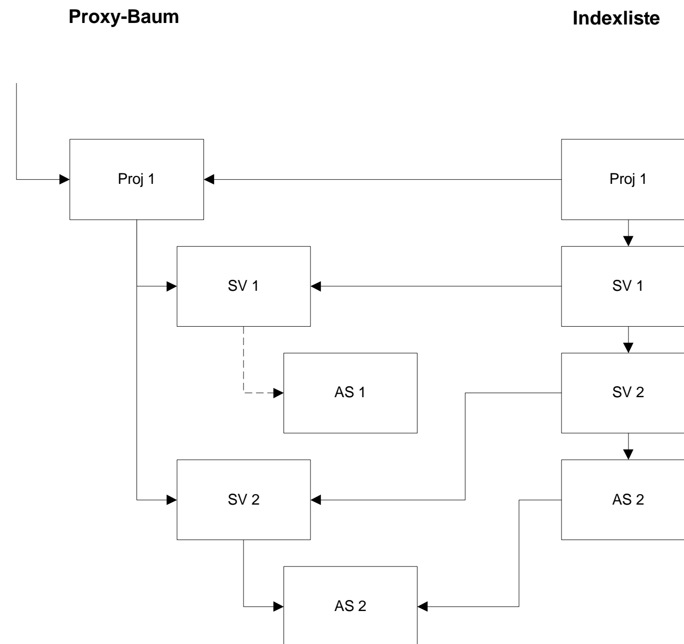

Abbildung 3.20: Clientanbindung - Baum

schlossen ist. Dies ist daran zu erkennen, dass der Zweig unter *SV 1*, nämlich *AS 1*, nicht in der *Indexliste* vorkommt.

Jeder Client, der sich am Server anmeldet, bekommt ein *CClient* Objekt zugewiesen. Dieses Objekt ist für die Verwaltung der Ansichten und für die Übertragung der Objekte zuständig. Jede Ansicht wird vom *CProxyManager* verwaltet. Dieser ist für den Aufbau des Baumes und der *Indexliste* zuständig.

Die *Proxyobjekte* werden über die *Observer* über Änderungen der Daten informiert. Diese wurden im Kapitel 2.6 beschrieben.

Plugin

Über diese COM-Schnittstelle haben andere Applikationen, wie zum Beispiel ein Web Portal, die Möglichkeit mit dem Server zu kommunizieren. Der Name kommt von der ursprünglichen Idee, den Server um kundenspezifische Funktionalität zu erweitern. Bis jetzt wird allerdings nur der Datenimport aus *MSProject* über diese Schnittstelle abgewickelt.

Das Interface *IDocumentPlugin* ist der Ausgangspunkt aller Anfragen. Über dieses Interface können nicht nur Daten angefordert sondern auch Daten hinzugefügt werden. Jedes Objekt, wie zum Beispiel ein Arbeitsschritt, hat ebenfalls ein COM-Interface, das über *IDocumentPlugin* angefordert werden kann.

3.6.9 Anbindung MSProject

Wie bereits erwähnt, werden Daten aus *MSProject* über die Pluginschnittstelle des Server importiert. Im Augenblick gibt es zwei Möglichkeiten, Daten aus *MSProject* zu übernehmen.

Die erste Möglichkeit ist über den Client einzelne Dateien zu importieren. Dazu wird sowohl am Server als auch am Client eine Import-DLL geladen. Serverseitig dockt die Import-DLL an das Interface *IDocumentPlugin*. Clientseitig lädt die DLL Daten aus *MSProject* und überträgt diese direkt an die serverseitig geladene DLL. Diese fügt dann die Daten in den Server ein.

Die zweite Möglichkeit, Daten aus *MSProject* zu übernehmen, ist jene mit einem Batchjob. Das ist ein Programm, das ein angegebenes Verzeichnis nach Project-Dateien durchsucht und diese über *IDocumentPlugin* in den Server lädt.

3.7 Client

Die Implementierung des Client orientiert sich im wesentlichen an der Client-Schnittstelle des *aceProject* Servers. Die wesentlichen Funktionen zur grafischen Darstellung werden in der entsprechenden Fachliteratur behandelt[4].

Der Client meldet sich über DCOM (siehe Kapitel 2.2) an einem Server an und lässt den Benutzer über die NT-Security authentifizieren. Der Server erzeugt daraufhin eine *CClient* Instanz und lädt benutzerspezifische Informationen, die an den Client übertragen werden.

Der Benutzer wählt einen Filter, der serverseitig den Baum und die Indexliste aufbaut.

Jene Teile des Baumes, die gerade angezeigt werden, werden vom Client "geholt" und angezeigt. Das Holen einzelner Datenobjekte geschieht über das *Marshaling*. Jedes serverseitige Objekt, ohne Unterschied ob Daten- oder Proxyobjekt, hat clientseitig ein "Gegenüber". Diese Objekte sind ausschließlich für die Speicherung und Anzeige der Daten verantwortlich.

Bei Änderung eines solchen Objektes durch den Benutzer wird dieses sofort an der Server zurückübertragen, wo es innerhalb einer Transaktion vom betreffenden Datenobjekt ausgepackt wird.

Kapitel 4

Wirtschaftliche Aspekte

4.1 Einsatz beim Kunden

aceProjekt wurde bisher dreimal - in jeweils unterschiedlicher Form sowie jeweils kombiniert mit anderen Dritt-Lösungen bei Kunden in Österreich und Deutschland eingeführt: Das gesamte Kernmodul bei der Firma *Magna Steyr Fahrzeugtechnik* in Graz, eine Kombination mit SAP und anderen Drittsystemen (SAP) bei der VA-TECH-Tochter *SAT*, sowie bei der Immobilientochter einer Deutschen Bank (deren Namen aus Datenschutzgründen hier nicht genannt werden kann), hier wiederum mit völlig anderen Teil- und Drittsystemen (Helpdesk, CRM, Aufwandserfassung, etc) kombiniert.

4.2 Fallbeispiel Magna Steyr Fahrzeugtechnik (SFT) Graz

Bei der Firma *Magna Fahrzeugtechnik* wird das Produkt derzeit primär als Auswertungstool eingesetzt, das jene Teile bzw. Funktionalitäten abdeckt, die mit MS Project nicht realisiert werden können. Dazu kommen eine Reihe von Spezialfunktionen, die für diesen Kunden implementiert werden mußten. Schwerpunkt sind hier die multiprojekt- und multiressourcenfähigen Auswertungsmöglichkeiten. Frühzeitig müssen Zielkonflikte in der Ressourcenplanung erkannt werden. Dies nicht zuletzt deshalb, weil die Entwicklungszeiten von ca 7-8 Jahren (zu Beginn der 90-iger Jahre) auf derzeit ca 2.5 Jahre zurückgegangen sind und Terminverzögerungen jeder Art unbedingt vermieden werden müssen, zumal sich die Unternehmen sofort mit enormen Pönale-Zahlungen konfrontiert sehen, wenn vereinbarte Termine überschritten werden. Hier wird die Ressourcenverwaltung zudem auch für die Kapazitätsplanung von Prüfständen (die nicht in beliebigem Umfang zur Verfügung stehen und die aufgrund der enormen Investitionskosten ständig ausgelastet sein müssen) verwendet. Gerade in solchen Entwicklungsprojekten mit neuesten Technologien, wie sie von Magna SFT Graz durchgeführt werden,

können Terminverzögerungen sehr leicht auftreten, die meist auch unkritisch sind, wenn sie frühzeitig bemerkt werden und damit geeignete Gegenmaßnahmen ergriffen werden können. Das schnelle bzw. kurzfristige Hochfahren in der Verfügbarkeit von Ressourcen ist in vielen Fällen bei diesem Kunden nicht möglich. Konkret werden bei Magna alle *MS Projekt* Dateien in der Nacht in den Ressourcenmanager eingelesen, um daraus eine jeweils aktuelle Kapazitätsauswertung zu bekommen. Es ist geplant, das Projekt, welches erfolgreich abgeschlossen wurde, zu erweitern. Konkret geht es darum, die gesamten Planungstätigkeiten (die neben den Terminen vor allem auch die Kosten umfassen) generell mit einem einzigen Tool durchzuführen. Der Kunde verspricht sich beträchtliche wirtschaftliche Erfolge sowie vor allem auch das Vermeiden von Terminüberschreitungen, die im Falle von Pönalen (wie sie heute üblich sind) sehr schnell im zweistelligen Mio-EURO-Bereich liegen können.

4.3 Fallbeispiel VA-TECH SAT Wien

Bei der Firma *VA-TECH SAT* wird die gesamte weltweite (dh auch bei den Auslandstöchtern) Projektplanung und Abwicklung vollständig auf eine Lösung der ace umgestellt, die auch *aceProject* enthält. In diesem Fall dient *Fabasoft Components* als Datenbank, um weitere Funktionalität und Informationen leichter einbinden zu können. Weiters gibt es mittlerweile Schnittstellen zu einer Reihe von Drittsystemen, die meist mit XML bedient werden (bidirektional). Der wirtschaftliche Nutzen eines solchen Systems für SAT ist enorm, weil dies das gesamte Kerngeschäft abdeckt. Projektlaufzeiten werden weiter verkürzt, Fehlentwicklungen in der Planung bzw. Abwicklung noch früher erkannt. Der wirtschaftliche Nutzen eines solchen Systems liegt auf mittlere Sicht sicherlich im Mio-EURO-Bereich.

4.4 Fallbeispiel Immobilientocher einer deutschen Bank

Bei der *Immobilientochter einer deutschen Bank* wird - wie bei *VA-TECH SAT* - die gesamte weltweite Projektplanung und Abwicklung vollständig auf eine Lösung der ace IT GmbH umgestellt. Auch hier wird als System für die Datenhaltung Fabasoft Components eingesetzt. Auch hier erwartet sich der Kunde eine deutliche Verbesserung in Planung und Abwicklung von Projekten. Auch hier ist es besonders wichtig gewesen, dass die Ressourcen- und Kapazitätsplanung in eine deutlich umfangreichere Lösung als ein wesentliches Teilsystem integriert werden konnte.

4.5 Bedeutung für die ace

Bisher sind in die Entwicklung der Multiprojektplanung mehr als 10.000 Mannstunden investiert worden und es wird derzeit laufend weiter daran gearbeitet, wobei vor allem spezielle Wünsche von Kunden implementiert werden. Nicht zuletzt aufgrund der enormen finanziellen Aufwendungen ist die gesamte Lösung für die ace IT GmbH von außerordentlich großer wirtschaftlicher Bedeutung.

Das gesamte Projekt ist zudem in der Startphase vom FFF (Forschungs-Förderungs-Fond der gewerblichen Wirtschaft) unterstützt worden.

Kapitel 5

Diskussion

5.1 Client/Server Architektur

Die Architektur dieser Lösung ist eine klassische *3-Tier-Architektur*, wobei der *Middle-Tier*, die Business-Logic, eine kleine Ausnahme darstellt. Normalerweise ist die Business-Logic stateless. Bei dieser Lösung scheint es, als ob die Business-Logic einem riesigem Datenbank-Cache gleicht. Dies ist nur teilweise richtig. Die Daten werden zwar sofort in die Datenbank zurückgeschrieben, was bei einer größeren Anzahl von Änderungen längere Zeit dauern kann, jedoch werden die Daten nur beim Start des Server gelesen. Der Grund ist, dass beim Aufbau von Listen, Zusammenfassungen und bei Berechnungen immer eine größere Anzahl von Objekten involviert sind. Wären diese nicht bereits fertig im Speicher aufgebaut, würde jeder Zugriff auf Daten sehr langsam vor sich gehen. Als bestes Beispiel sei hier der Aufbau des Baumes für den Client genannt.

Ein anderer interessanter Aspekt der Architektur ist die Anbindung des Clients. Es handelt sich hierbei zwar um eine COM-Schnittstelle, jedoch werden die Daten als Paket (*Stream*) zum Client übertragen. Diese Vorgehensweise hat zwei große Vorteile:

Erstens muss der Client nicht wegen jedem Attribut eines Objektes einen Call absetzten, was günstig für die Performance ist. Zweitens kann der Client die Datenobjekte in einem Cache ablegen. Er muss den Server nur noch fragen, ob sich das Objekt geändert hat.

5.2 Transaktionen

Transaktionen wurden im Kapitel 3.6.4 ausführlich behandelt. Beim Design wurde entschieden die Transaktionen selbst zu verwalten, da die Möglichkeit *MTS - Microsoft Transaction Server* ausfällt. Die Philosophie beim *MTS* lautet "stateless". Da der Server allerdings alles andere als stateless ist, kommt diese Variante nicht in Betracht.

Selbstverständlich wird gegen die Datenbank, egal ob *SQL-Server* oder

Fabasoft Components, mit einer Transaktion gefahren. Schlägt diese fehl, müssen auch die Objekte im Speicher zurückgesetzt werden. Das könnte man mit einem erneuten Einlesen der Daten aus der Datenbank lösen. Jedoch gibt es Einträge bei den Objekten, die nicht in der Datenbank abgelegt sind. Zum Beispiel hat jedes *CProject* Objekt einen Index auf alle *CRessource* Objekten, die in den Arbeitsschritten unter dem *CProject* Objekt vorkommen. Außerdem wäre es nicht möglich, während einer Änderung der Daten, welche unter Umständen "lange" dauern kann, einen lesenden Zugriff auf die Daten zu gewährleisten.

5.3 Performance

Einige Routinen verlangen ein Höchstmaß an Performance, da viele Objekte im Spiel sind und der Benutzer in der Regel sehr ungeduldig ist. Das beste Beispiel für solch eine Routine ist das Erstellen des Baumes. Hier sind viele Datenobjekte involviert, es werden viele neue Instanzen erzeugt und es handelt sich hierbei um eine Rekursion.

Die Anzahl der Datenobjekte die bei den Berechnungsroutinen involviert sind, lässt sich nicht optimieren. Sehr wohl optimiert werden können die Zugriffe auf jene Objekte, die nicht in den Baum aufgenommen werden, weil zum Beispiel in der Ressourcenansicht die betreffende Ressource gar nicht angezeigt wird. Dieses Problem kann mit Indexlisten gelöst werden.

Indexlisten

Jedes *CProject* Objekt hat einen Index auf alle Ressourcen, die in den Arbeitsschritten unter der *CProject* Instanz vorkommen. Damit kann beim Aufbau des Baumes für die Ressourcenansicht die Routine sofort feststellen, ob dieser Zweig von Bedeutung ist. Die Verwaltung dieses Indexes obliegt dem *CTask* Objekt, da nur er weiß, welche Ressourcen an dem Arbeitsschritt beteiligt sind.

Außer der Indexliste für Ressourcen gibt es noch eine weitere Indexliste, nämlich die für den Baum. Damit hat der Client die Möglichkeit dem Server eine absolute Position zu übergeben, um als Antwort sofort eine Referenz auf das betreffende *Proxyobjekt* zu erhalten.

Rekursionen/Inlining

Es sollten Rekursionen grundsätzlich vermieden werden, da jeder Aufruf einer Funktion Zeit kostet. Generell sollte versucht werden, jeden Funktionsaufruf als *inline* Aufruf auszuführen. Was bei einem Funktionsaufruf geschieht kann im Kapitel 2.3.2 nachvollzogen werden.

Wenn schon eine Rekursion benötigt wird, sollte man darauf achten, die Anzahl der Parameter der Funktion einzuschränken (Kapitel 2.3.2). Weiters sollte darauf geachtet werden, dass keine Instanzen von Objekten übergeben

werden, da diese kopiert werden, was unter Umständen einen *CopyConstructor* auslöst. Wenn dieses Objekt in der Funktion nicht geändert wird, sollte es mit `const&` übergeben werden.

```
void Recursion(int level, const CResourcePtr& resource)
{
...
}
```

Schulung der Benutzer

Dies ist sicher kein technischer Ansatz zur Steigerung der Performance. Es geht hierbei darum, dass die Benutzer angehalten sind, sich die Auswahl der angezeigten Informationen genau zu überlegen. Bei dem Projekt hat sich gezeigt, dass die Aufbereitung von Information wesentlich mehr Zeit in Anspruch nimmt, als die Suche nach der Information. Mehr Daten bedeuten nicht zwangsläufig mehr Information.

Kapitel 6

Ausblick

Eine Softwareentwicklung ist, solange sie benutzt wird, nie "fertig". Software ist erst dann "fertig", wenn die Benutzer diese Software als unbrauchbar abstoßen. Da dieses Produkt noch ein paar Mal verkauft wird, muss es auch laufend weiterentwickelt werden. Abgesehen von einigen Kundenwünschen müssen noch folgende Dinge implementiert werden.

Cluster

Irgendwann werden so viele Daten in den Server geladen, dass der Speicher ausgehen wird. Das Problem liegt nicht in der Anzahl der Projekte, sondern vielmehr in der Anzahl der Versionen der einzelnen Projekte. Bis sich die 64 Bit Maschinen durchgesetzt haben, muss ein Teil der Daten auf einen anderen Server ausgelagert werden. Dazu muss folgendes implementiert werden.

- **Links:** Der Server muss innerhalb seines Projektbaumes Einträge erzeugen können, die auf Projekte zeigen die auf anderen Servern liegen können. Nur so ist es möglich, Teile der Daten auszulagern. Allerdings hat das den Nachteil, dass eine gleichmäßige Verteilung der Daten nicht möglich ist.
- **Redirect:** Der Server muss in der Lage sein, Anfragen eines Clients auf einen anderen Server umzuleiten.
- **Synchronisierung:** Bestimmte globale Daten, wie Ressourcen, Benutzer oder Farbeinstellungen müssen auf alle Server im Cluster repliziert werden.
- **Berechnungen:** Berechnungen die an einem Server begonnen wurden, müssen an einen anderen Server weitergeleitet werden, da die benötigten Daten verteilt sind.

Entladen von Versionen/Archivierung

Im Augenblick werden die Versionen auf Anfrage in den Speicher geladen, jedoch nicht mehr entladen, wenn diese Daten nicht mehr benötigt werden.

Dazu muss ein Referenzzähler, ähnlich wie bei COM-Objekten, implementiert werden.

Weiters muss es eine Möglichkeit zur Archivierung von Versionen geben, die nicht mehr geändert werden.

Plugin-Schnittstelle

Es muss eine Infrastruktur geschaffen werden, um kundenspezifische Wünsche ohne Einfluss auf andere Funktionen implementieren zu können. Dazu gehört eine *EventSink*, die Möglichkeit eigene Datenfelder zu definieren oder Eingabemasken anzupassen.

Literaturverzeichnis

[1] Bjarne Stroustrup, The C++ Programming Language, Special Edition, 2000, AT&T Labs, ISBN 0-201-70073-5

[2] Guy Eddon, Henry Eddon, Inside Distibuted COM, 1998, Microsoft Press, ISBN 3-86063-459-3

[3] Adam Denning, ActiveX Controls Inside Out, Second Edition, 1997, Microsoft Press, ISBN 1-57231-350-1

[4] Richard J. Simon, Win32 Programmierung, Band 1, 1996, SAMS, ISBN 3-8272-4502-8

Abbildungsverzeichnis

2.1 Ableitungen 11
2.2 Mehrfachableitungen 12
2.3 Mehrfachableitungen - getrennte Basisklassen 14
2.4 Mehrfachableitungen - gemeinsame Basisklasse 15
2.5 Interfaces 16
2.6 Virtuelle Funktionen 16
2.7 Templates 17
2.8 COM-Object 25
2.9 Instanzierung 26
2.10 Zeiger 28
2.11 Zeiger-Arrays 30
2.12 Strings 31
2.13 Stack einer Methode 34
2.14 Stack Exploite 35
2.15 Zeiger und Mehrfachableitungen 35
2.16 VTable 36
2.17 Einfach verkettete Liste 40
2.18 Doppelt verkettete Liste 41
2.19 Doppelt verkettete Liste mit weiterem Zeiger 41
2.20 Observer 44

3.1 Projektansicht 49
3.2 Eingabe der Daten 50
3.3 Ressourcenansicht 51
3.4 Versionsvergleich 52
3.5 Filter 53
3.6 Kapazitätsdiagramm 54
3.7 Grundkonzept 55
3.8 Ableitungsdiagramm 56
3.9 Strukturdiagramm Daten 57
3.10 Versionen 59
3.11 Transaktionsobjekte 60
3.12 Objekt in Transaktion aufnehmen 61
3.13 Transaktion abschließen 62
3.14 Commit im Components 63

3.15 Commit im Server 64
3.16 Security 65
3.17 Berechnung der Dauer für mehrere Ressourcen 67
3.18 Zusammenfassung der Ressourcen 68
3.19 Strukturdiagramm Clientanbindung 69
3.20 Clientanbindung - Baum 70

A.1 Datenbankmodell 83
A.2 Datenbankmodell 84
A.3 Datenbankmodell 85

Anhang A

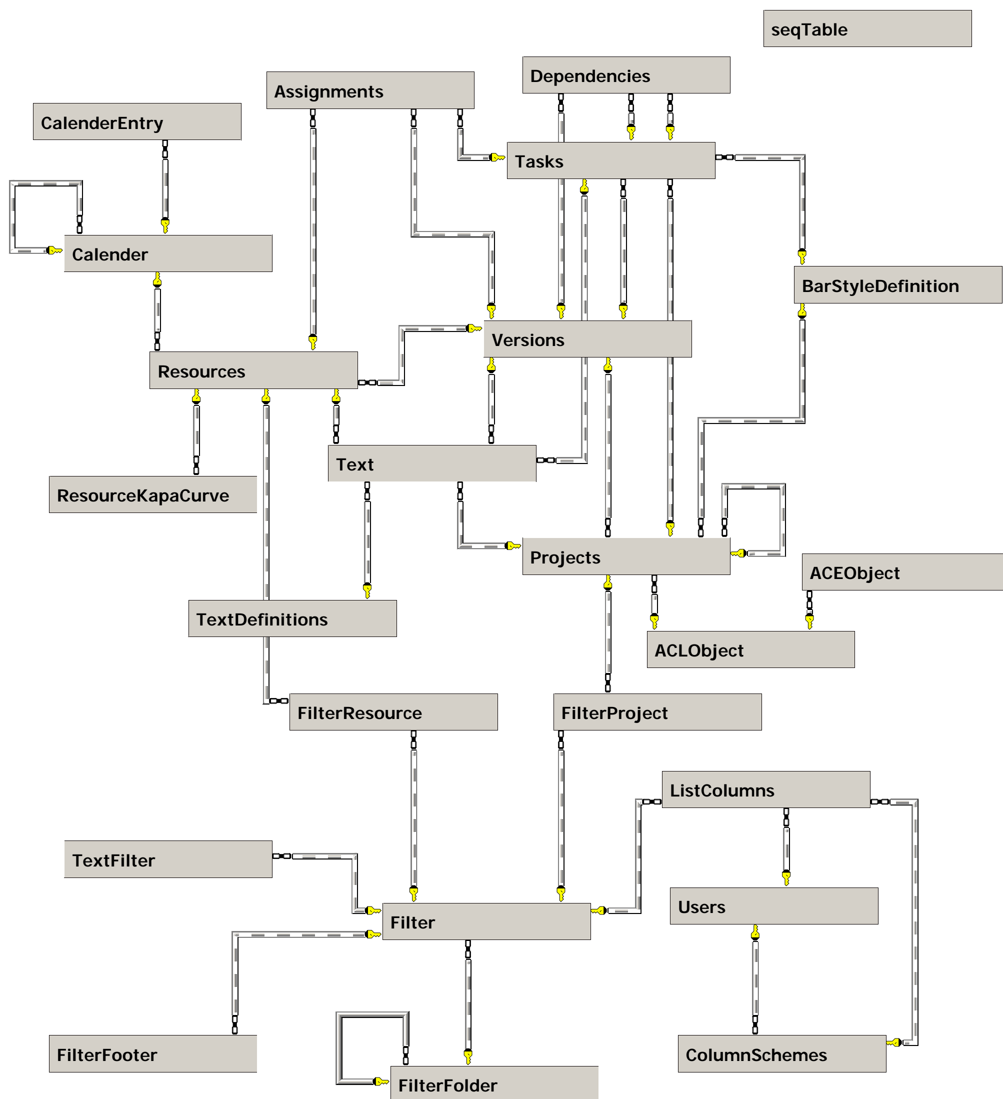

Abbildung A.1: Datenbankmodell

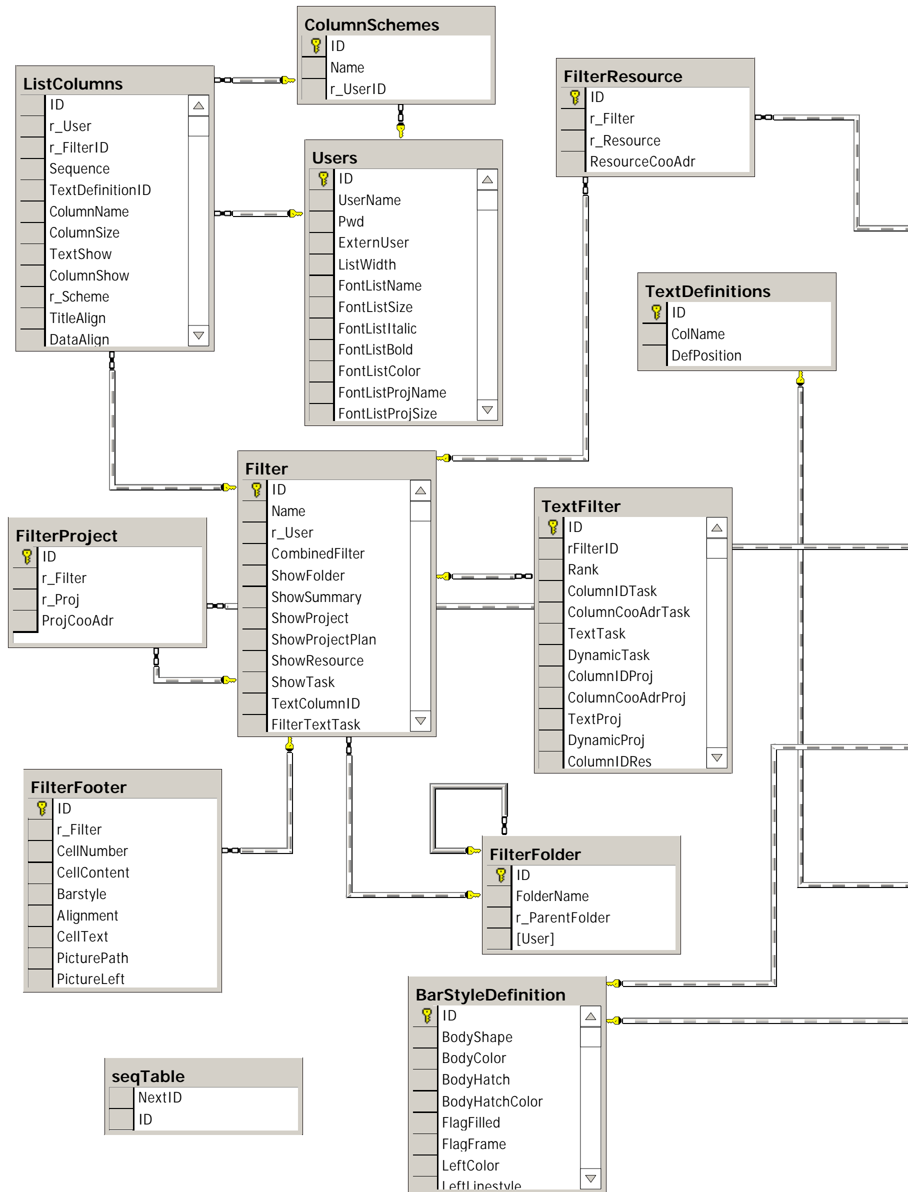

Abbildung A.2: Datenbankmodell

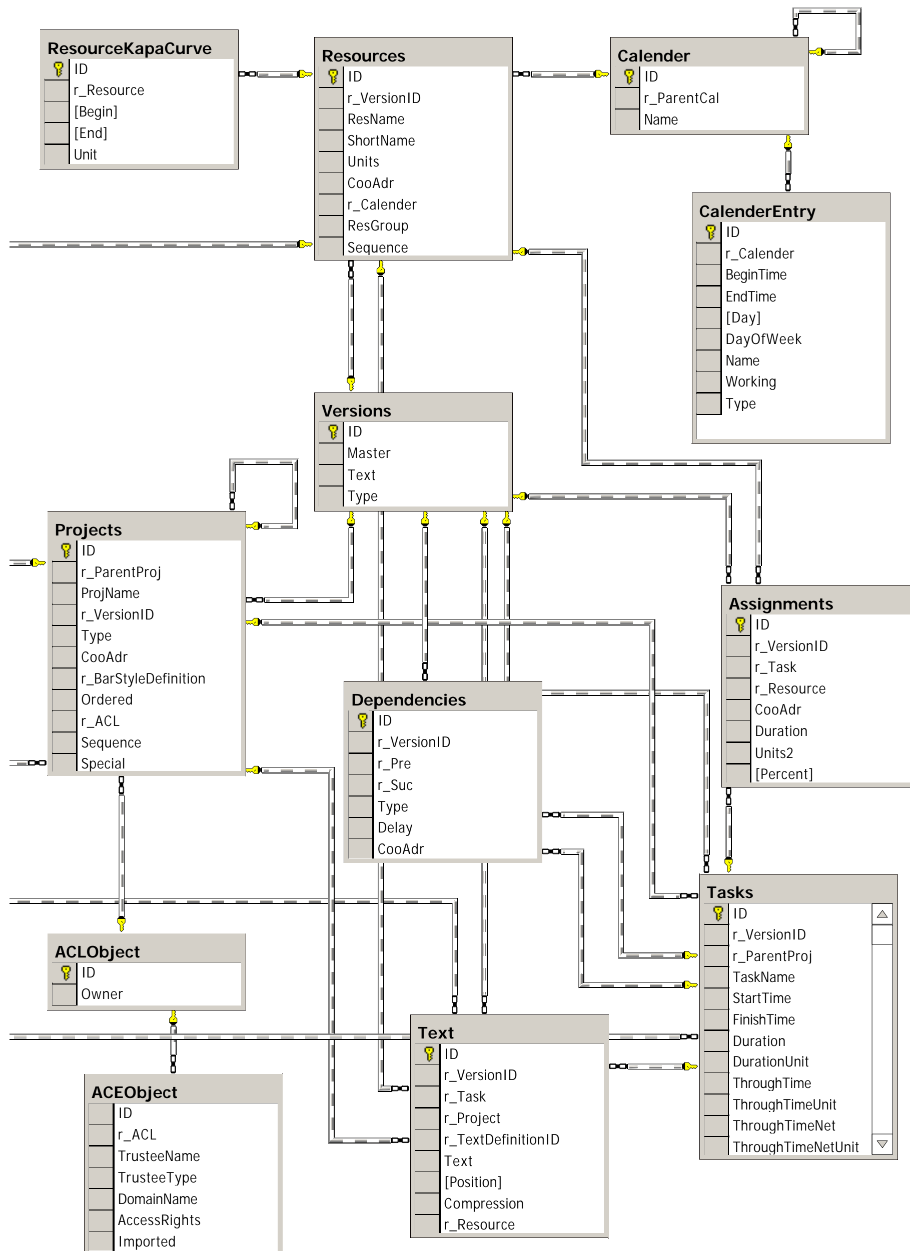

Abbildung A.3: Datenbankmodell