VL5+6: Softwareprojekt - Implementierung Teil 1+2

Inhalt

1. Einleitung

In den vorausgehenden Vorlesungen wurde ein allgemeines Schema für Softwareprojekte eingeführt, wie es im Rahmen der *Model Driven Architecture (MDA)* der *OMG* angenommen wird. An einem einfachen Beispiel wurde erklärt, was es bedeuten kann, ein konkretes Anwendungsbeispiel als ein *computerunabhängiges ppmp2ps-Modell (ppmp2ps-CIM)* in Form eines *ppmp2ps-Anwendungsfalldiagramms (ppmp2ps-Use Case)* zu konstruieren. Dann wurde erklärt, wie man auf der Basis eines Anwendungsfalls ein *ppmp2ps-plattformunabhängiges Modell (ppmp2ps-PIM)* konstruieren und dieses dann in ein *ppmp2ps-plattformspezidisches Modell (ppmp2ps-PSM)* übersetzen kann. Dabei zeigte sich, dass im ersten Anlauf zwar die *statischen Eigenschaften* im Modell ppmp2ps-PIM erfasst und diese auch automatisch mit umbrello nach ppmp2ps-PSM übersetzt worden sind, es fehlten jedoch noch alle Angaben darüber, wie sich dieses Modell *dynamisch* verhält. Dies zeigte sich dann u.a. darin, dass im ppmp2ps-PSM alle Funktionen nur als Funktionsrümpfe, d.h. nur mit ihren *Signaturen* vorkommen. Es wurde dann erklärt, wie man die *dynamische* Struktur des computerunabhängiges ppmp2ps-Modells (ppmp2ps-CIM) in das ppmp2ps-PIM Modell als ppmp2ps-PIM2-Modell *integrieren* kann. In dieser Vorlesung beginnen wir nun damit, eine erste *Implementierung* des ppmp2ps-PIM2-Modells in ein plattformspezifisches ppmp2ps-PSM-Modell vorzunehmen.

2. Übergang von pmp2ps-PIM2 zu ppmp2ps-PSM

In Vorlesung 3 war schon erklärt worden, was man sich grundsätzlich unter einem Übergang von einem PIM zu einem PSM vorzustellen hat, dies insbesondere am Beispiel von C++ als eines plattformspezifischen Modells. Die Kernaussage in VL3 war gewesen, dass sich die UML-Klassen direkt mit C++-Klassen labeln lassen (siehe nochmals das Schaubild Bild 1). Deser Sachverhalt soll nun eingehender untersucht werden.

Neben der allgemeinen Tatsache, dass bei der Übersetzung von ppmp2ps-PIM2 nach ppmp2ps-PSM die UML-Klassen mit C++-Klassen gelabelt werden, liefert Bild 2 einen Hinweis auf eine mögliche Struktur des plattformspezifischen Modells. Es handelt sich um

eine spezifische Anordnung von C++-Dateien.

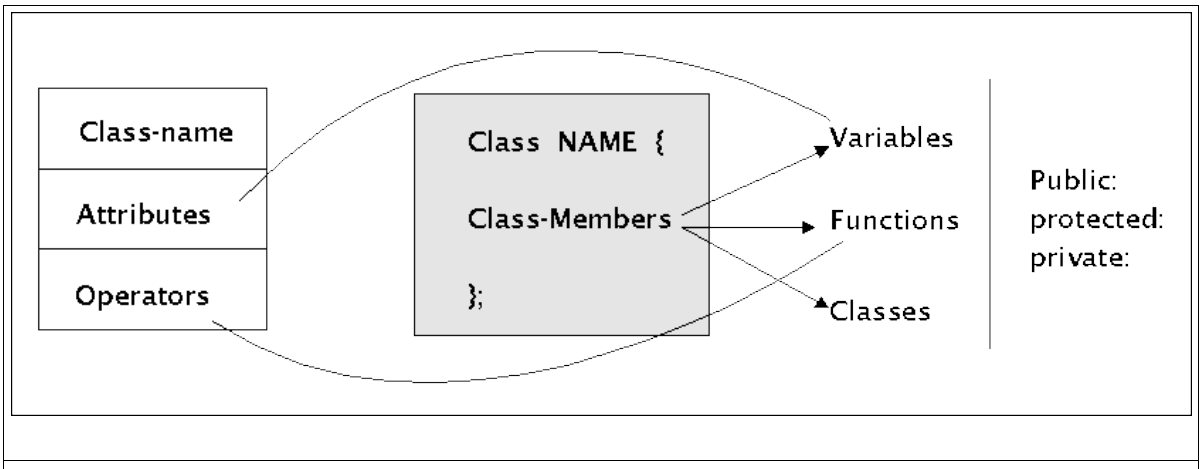

Bild 1: Abbildung UML-Klasse zu C++-Klasse

In den *Headerdateien (___.h)* wird jeweils eine Klasse definiert. In der zugehörigen *CPP-Methoden-Datei* werden die Methoden der Klasse definiert. Und in der zugehörigen *CPP-Usage-Datei* werden die so vereinbarten Klassen und Methoden aufgerufen und benutzt.

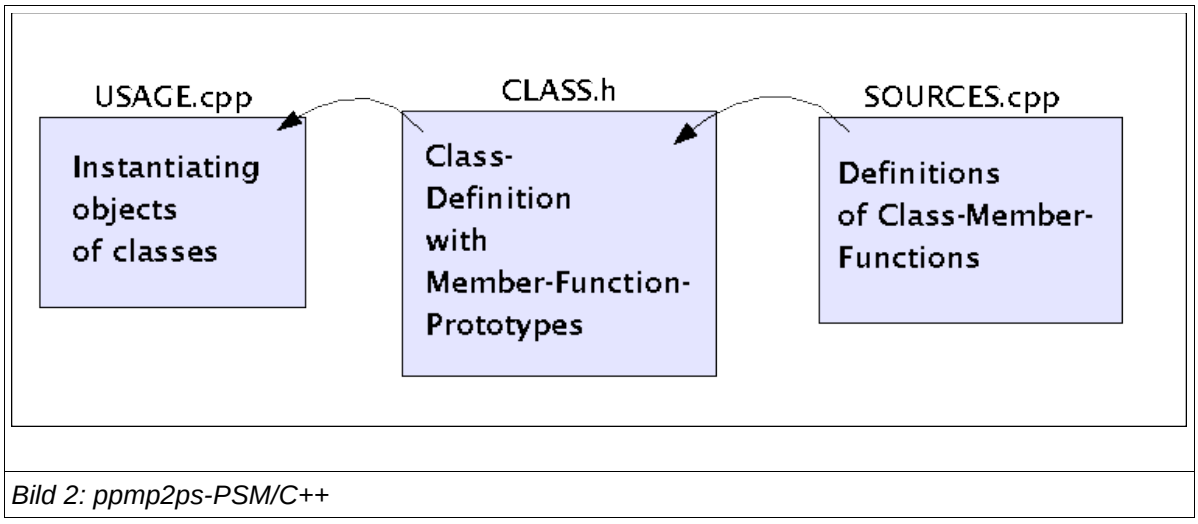

Bild 2: ppmp2ps-PSM/C++

Betrachten wir dies an einem Beispiel. In der vorausgehenden Vorlesung wurde mittels UML ein ppmp2ps-PIM2-Modell konstruiert. Als Kern besitzt dieses plattformunabhängige Modell die UML-Klasse ppmp2ps (vgl. Bild 3). Ein erster Übersetzungsschritt besteht darin, diese UML-Klasse in eine C++-Klasse zu überführen.

2.1 UML-Klasse --> C++Klasse ´

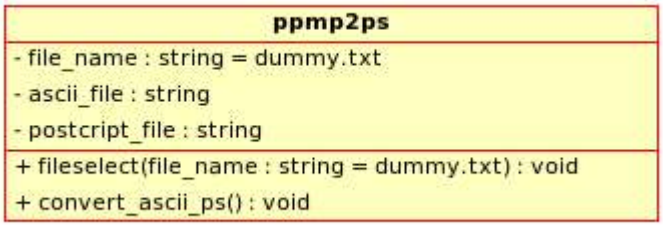

Bild 3: Die Klasse pmp2ps

ASCII-Text der C++-Headerdatei zur Klasse ppmp2ps:

```
/*************************************************************************

name: ppmp2ps.hpp

author: gerd d-h
first time: april-16, 2005
last change: may-1, 2005
intention: Read an ascii-textfile.txt and convert it into postscript
textfile.ps

Location: /home/gerd/public_html/fh/II-PPmP/VL/VL5/ppmp2ps.hpp

COMPILATION: Has to be included in another file

*************************************************************************/

#ifndef PPMP2PS_H
#define PPMP2PS_H

#include <string>

using namespace std;

/********************
 * Class ppmp2ps
 *
 ********************/

class ppmp2ps {

/*************************
 * Public stuff
 *************************/

public:

  /*********************
   * Operations
   *********************/
```

```
  /*************************
   *
   * @param file_name Bekommt vom User einen Dateinamen fuer eine ASCII-Datei
mit Endung .txt und
                    liest dann diese Datei in einen internen Puffer
   ********************************************************/

  void  fileselect (string file_name="dummy.txt");


  /*************************************************************
   * Konvertiert einen zuvor ausgewÃ¤hlten ASCII-Text in eine Postscript Datei
   ************************************************************************/

  void  convert_ascii_ps ();

  /*******************
 * Private stuff
 ***************************/

private:

  /*****************************
   * Fields
   ***********************************/

  /*************************************************************
   * EnthÃ¤lt den Dateinamen der aktuell zu konvertierenden ASCII-Datei
   ************************************************************/

   string file_name;

   /*********************
   * Ein Textpuffer mit der eingelesenen ASCII-Datei
   ************************************************/

   string ascii_file;

   /************************************************
   * Textpuffer fuer die neu konvertierte Postcript Datei
   ************************************************/

   string postcript_file;


};
#endif //PPMP2PS_H
```

Für die allgemeine Kommentierung con C++-Klassen siehe bitte Vorlesung Nr.3.

Man sieht direkt, wie die Attribute der UML-Klasse ppmp2ps als private Variablen in die C++-Klasse ppmp2ps übernommen wurden; ebenso die öffentlichen Operationen. Eine Besonderheit ist die Anweisung:

using namespace std;

Damit wird gesagt, dass für alle Klassen aus dem Bereich des Namensraumes "std" das Präfix "std." automatish ergänzt werden soll. Man muss also bei dem Aufruf einer Methode aus einer Klasse der STD-Klassen dieses Kürzel nicht mehr explizit hinschreiben.

2.2 Implementierung der Methoden in dr C++-Klasse

Da die Operationen in der C++-Klasse nur als *Signaturen* vorliegen (Operationsname, Typangaben, Parameter, kein definierender Rumpf), müssen jetzt in einer zugehörigen *Implementierungsdatei* diese Daten nachgeliefert werden. Diese Implementierungsdatei bekommt den Namen ppmp2ps.cpp. Eine erste, noch unvollständige Version nur mit den Signaturen würd wie folgt aussehen (siehe nachfolgenden ASCII-Text).

```
/*************************************************************************

name: ppmp2ps.cpp

author: gerd d-h
first time: april-16, 2005
last change: may-1, 2005
intention: Read an ascii-textfile.txt and convert it into postscript
textfile.ps


LOCATION: /home/gerd/public_html/fh/II-PPmP/VL/VL5/ppmp2ps.cpp

COMPILATION: g++ -o ppmp2ps_usage ppmp2ps_usage.cpp ppmp2ps.cpp
USAGE: ppmp2ps_usage

*************************************************************************/

#include "ppmp2ps.h"
#include <string>

/*******************
 * Methods
 *******************/

/*****************************
 *  Holt sich einen Dateinamen und liest die zugehoerige Datei in einen Puffer
 ************************************************************************/

void ppmp2ps::fileselect (string file_name) {

}
/*************************************************************
 * Konvertiert einen zuvor ausgewÃ¤hlten ASCII-Text in eine Postscript Datei
 *************************************************************/

void ppmp2ps::convert_ascii_ps () {

}
```

Eine solche Datei würde natürlich noch nichts tun. Rein zu Testzwecken (evolutionäres Vorgehen, Rapid Prototyping) können wir diese Rumpfimplementierung mit zwei

Testausgaben füllen, um einen ersten Implementierungstest vornehmen zu können.

```
/*************************************************************************

name: ppmp2ps.cpp

author: gerd d-h
first time: april-16, 2005
last change: may-1, 2005
intention: Read an ascii-textfile.txt and convert it into postscript
textfile.ps


LOCATION: /home/gerd/public_html/fh/II-PPmP/VL/VL5/ppmp2ps.cpp

COMPILATION: g++ -o ppmp2ps_usage ppmp2ps_usage.cpp ppmp2ps.cpp
USAGE: ppmp2ps_usage

*************************************************************************/

#include "ppmp2ps.hpp"
#include <string>
#include <iostream>

/*******************
 * Methods
 *******************/

/*****************************
 *  Holt sich einen Dateinamen und liest die zugehoerige Datei in einen Puffer
 ************************************************************************/

void ppmp2ps::fileselect (string file_name) {

  cout << "Dies ist die Metode fileselect!" << endl;

}
/*****************************************************************
 * Konvertiert einen zuvor ausgewÃ¤hlten ASCII-Text in eine Postscript Datei
 *****************************************************************/

void ppmp2ps::convert_ascii_ps () {

  cout << "Dies ist die Metode convert_ascii_ps!" << endl;

}
```

2.3 C++-Usage Datei

Um nun die Klassendeklarationsdatei (.hpp) sowie die Implementierungsdatei (.cpp) tatsächlich nutzen zu können, benötigen wir noch eine *Usage-Datei*, in der Objekte der definierten Klasse erzeugt werden und in der von den öffentlich zugänglichen Methoden dieser instantiierten Objekte Gebrauch gemacht wird. Ein einfaches Beispiel solch einer Datei ist die folgende:

```
/*************************
 *
 * name: ppmp2ps_usage.cpp
 *
 * author: gerd doeben-henisch
 * first: may-2, 2005
 * last: ---
 *
 * intention: Usgae file for ppmp2ps
 *
 * LOCATION: /home/gerd/public_html/fh/II-PPmP/VL/VL5/ppmp2ps_usage.cpp
 * COMPILATION: g++ -o ppmp2ps_usage ppmp2ps_usage.cpp ppmp2ps.cpp
 * USAGE: ppmp2ps_usage
 *
 ************************************/

#include "ppmp2ps.hpp"
#include <string>
#include <iostream>

using namespace std;

int main(){

  /* Initialisierung eines Objektes der Klasse ppmp2ps */

  ppmp2ps a1;

  /* Test, ob sich die Methoden der Klasse aufrufen lassen */

  a1.fileselect ("xyz.txt");

  a1.convert_ascii_ps();


}
```

Ausgabe der Konsole:

gerd@kant:~/public_html/fh/II-PPmP/VL/VL5> g++ -o ppmp2ps_usage ppmp2ps_usage.cpp ppmp2ps.cpp

gerd@kant:~/public_html/fh/II-PPmP/VL/VL5> ./ppmp2ps_usage

Dies ist die Metode fileselect!

Dies ist die Metode convert_ascii_ps!

In dieser Usage-Datei passiert noch nicht viel. Es wird auch der Namensraum *std* aktiviert. Dann wird ein Objekt *a1* der Klasse ppmp2ps gebildet. Schliesslich werden die öffentlichen Methoden des Objektes *a1* aufgerufen. Wie zu erwarten bekommt man von diesen Methoden nur eine einfache Meldung zurück.

2.4 Fortsezung der Implementierung der Methoden:

Zusätzliche Aktivitätsiagramme nutzen

Damit die Methoden das tun, was man von ihnen eigentlich erwartet, muss man die entsprechenden Informationen in die Implementierung der Methoden einbauen, d.h. man muss die *Definition* der Methoden entsprechend erweitern. Eine Möglichkeit, dies zu tun, besteht darin, die Modellierung im Kontext des plattformunabhängigen Modells (PIM) soweit voranzutreiben, dass man hier genügend Informationen für eine Implementierung erhält. Dies haben wir in Vorlesung Nr.4 getan. Im Rahmen des Modells *ppmp2ps_PIM2* wurden mit Hilfe von Aktivitätsdiagrammen das Verhalten der Methoden *fileselect()* sowie *convert_ascii_ps()* weiter analysiert.

2.4.1 Die Methode fileselect()

Für die Methode *fileselect()* wurde folgendes Aktivitätsdiagramm erstellt (siehe Bild 4). In diesem Diagramm wird zu Beginn angenommen, dass eine Datei geoeffnet und gelesen werden kann. Um dies zu implementieren, muss man wissen, wie dies in C++ möglich ist. Im folgenden wird zuerst eine konkrete Lösung vorgestellt und dann im Anschluss wird die zugehörige "Theorie" nachgeliefert.

Eine einfache Lösung zum Öffnen einer Datei bietet folgender Kodeausschnitt:

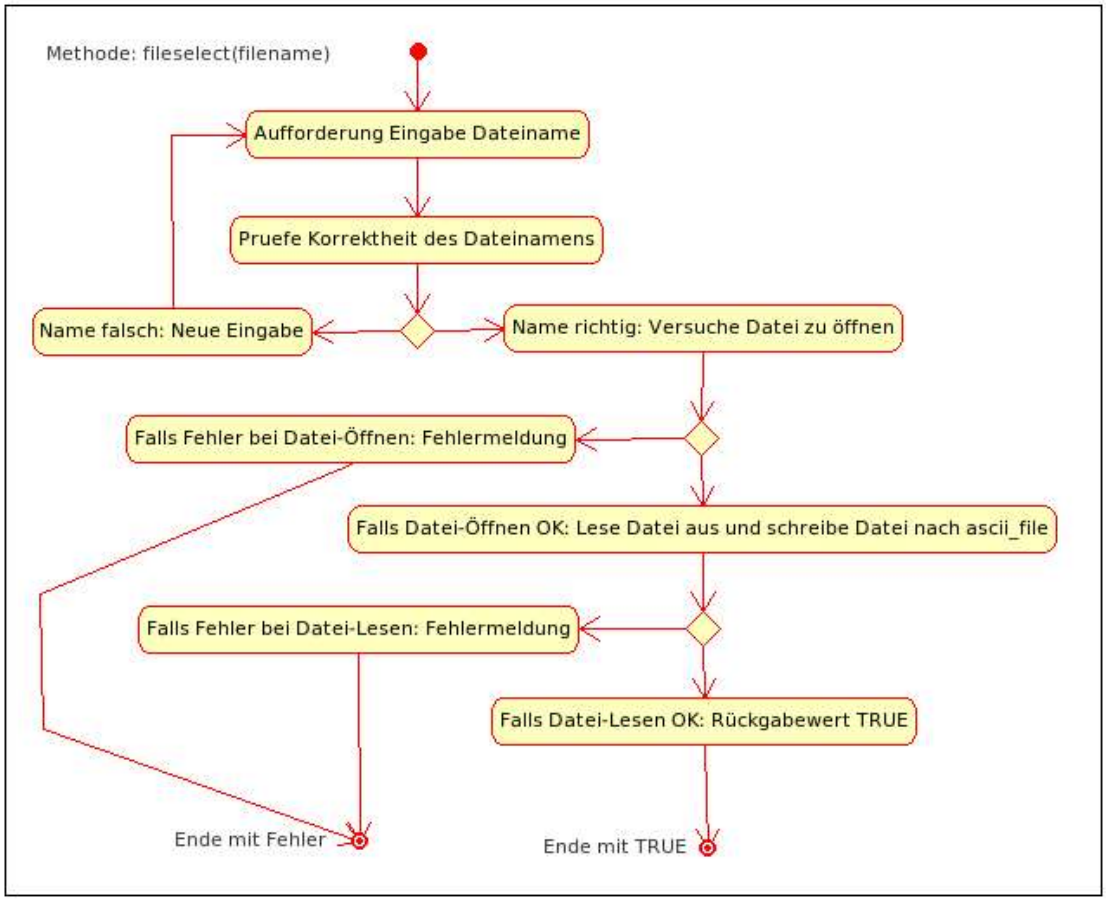

Bild 4: ppmp2ps-Aktivitätsdiagramm für die Methode fileselect(filename)

Kodeausschnitt zu fileselect() in ppmp2ps.cpp

```
#include "ppmp2ps.hpp"

#include <string>       // for strings
#include <iostream>     // for I/O
#include <fstream>     // for I/O
#include <cstdlib>      // for exit()



/*******************
 * Methods
 *******************/

/*******************************
 *  Holt sich einen Dateinamen und liest die zugehoerige Datei in einen Puffer
 ********************************************************************************/

void ppmp2ps::fileselect (string file_name) {

  cout << "Dies ist die Metode fileselect!" << endl;

    // Oeffne input file

    ifstream file("dummy.txt");
```

```
    // Datei geoeffnet?

    if (! file) {

        // NEIN, beende Programm
        cerr << "Datei laesst sich nicht oeffnen \"" << file_name << "\""
             << endl;
        exit(EXIT_FAILURE);
    }

    // Kopiere Dateiinhalt auf den Bildschirm

    char c;
    while (file.get(c)) {
        cout.put(c);
    }

}
```

Als erstes sieht man, dass eine zusätzliche Klasse *fstream* dazugeladen werden muss. Diese Klasse wird zum Lesen und Schreiben von Dateien benötigt. Die Klasse *cstdlib* wird benötigt, um im Fehlerfall das Programm mit Fehlermeldung verlassen zu können.

Im Beispiel wird die Datei mit folgender Anweisung geöffnet:

ifstream **file**("dummy.txt");

Hier ist der Dateiname noch fest vorgegeben. Der test, ob das Öffnen der Datei funktioniert hat, findet sich in den folgenden Zeilen:

```
    if (! file) {

        // NEIN, beende Programm
        cerr << "Datei laesst sich nicht oeffnen \"" << file_name << "\""
             << endl;
        exit(EXIT_FAILURE);
    }
```

Falls sich die Datei normal öffnen lässt, wird der Inhalt gelesen und zunächst auf den Bildschirm ausgegeben:

```
// Kopiere Dateiinhalt auf den Bildschirm

    char c;
    while (file.get(c)) {
        cout.put(c);
    }
```

Damit wir eine Datei zum testen haben erzeugen wir diese einfach mit dem Befehl *echo*:

```
gerd@kant:~/public_html/fh/II-PPmP/VL/VL5> echo "Dies ist ein Dummy-Inhalt" >>
dummy.txt
```

Kompiliert man dann die neue Version von ppmp2ps.cpp und ruft sie auf der Konsole

auf, dann erhält man folgende Ausgabe:

```
gerd@kant:~/public_html/fh/II-PPmP/VL/VL5> ./ppmp2ps_usage
Dies ist die Metode fileselect!
Dies ist ein Dummy-Inhalt
Dies ist die Metode convert_ascii_ps!
```

Will man einen beliebigen Dateinamen angeben, dann erreicht man dies durch folgende Zeilen:

```
void ppmp2ps::fileselect (string file_name) {


  cout << "Dies ist die Metode fileselect!" << endl;

  cout << "Geben sie bitte einen Dateinamen ein mit Endung .txt: ";

  cin >> file_name;

    // Oeffne input file

  ifstream file(file_name.c_str());
```

Auf der Konsole erhält man nach dem Kompilieren dann die Ausgabe:

```
gerd@kant:~/public_html/fh/II-PPmP/VL/VL5> ./ppmp2ps_usage
Dies ist die Metode fileselect!
Geben sie bitte einen Dateinamen ein mit Endung .txt: dummy.txt
Dies ist ein Dummy-Inhalt
Dies ist die Metode convert_ascii_ps!
```

Bei einer fehlerhaften Eingabe erscheint:

```
gerd@kant:~/public_html/fh/II-PPmP/VL/VL5> ./ppmp2ps_usage
Dies ist die Metode fileselect!
Geben sie bitte einen Dateinamen ein mit Endung .txt: dummy
Datei laesst sich nicht oeffnen "dummy"
```

Lassen wir an dieser Stelle die Frage des korrekten Namens vorläufig beiseite und betrachten wir, wie wir die Datei nicht nur auf den Bildschirm, sondern auch in den Puffer mit Namen *ascii_file* schreiben können. Dies geht z.B. mit der Funktion *getline*:

```
// Kopiere Dateiinhalt in Puffer und dann  auf den Bildschirm

    char c;
    while (file) {
      getline(file,ascii_file);
      cout << ascii_file << endl;
    }
```

Die Ausgabe auf der Konsole sieht dann wie folgt aus:

```
gerd@kant:~/public_html/fh/II-PPmP/VL/VL5> ./ppmp2ps_usage
Dies ist die Metode fileselect!
Geben sie bitte einen Dateinamen ein mit Endung .txt: dummy.txt
Dies ist ein Dummy-Inhalt

Dies ist die Metode convert_ascii_ps!
```

Für einen ersten Überblick zu Ein-/Ausgabe unter C++ sei verwiesen auf die Webseite http://www.fbmnd.fh-frankfurt.de/~doeben/I-PROGR2/I-PROGR2-TH/VL8/i-progr2-th-vl8.html. Eine genauere Besprechung wird noch stattfinden.

2.4.2 Die verbesserte Methode fileselect() sowie die Methode convert_ascii_ps()

In Bild 5 ist noch einmal das Aktivitätdiagramm zu sehen, das als Entwurf für die Methode convert_ascii_ps() erstellt worden ist.

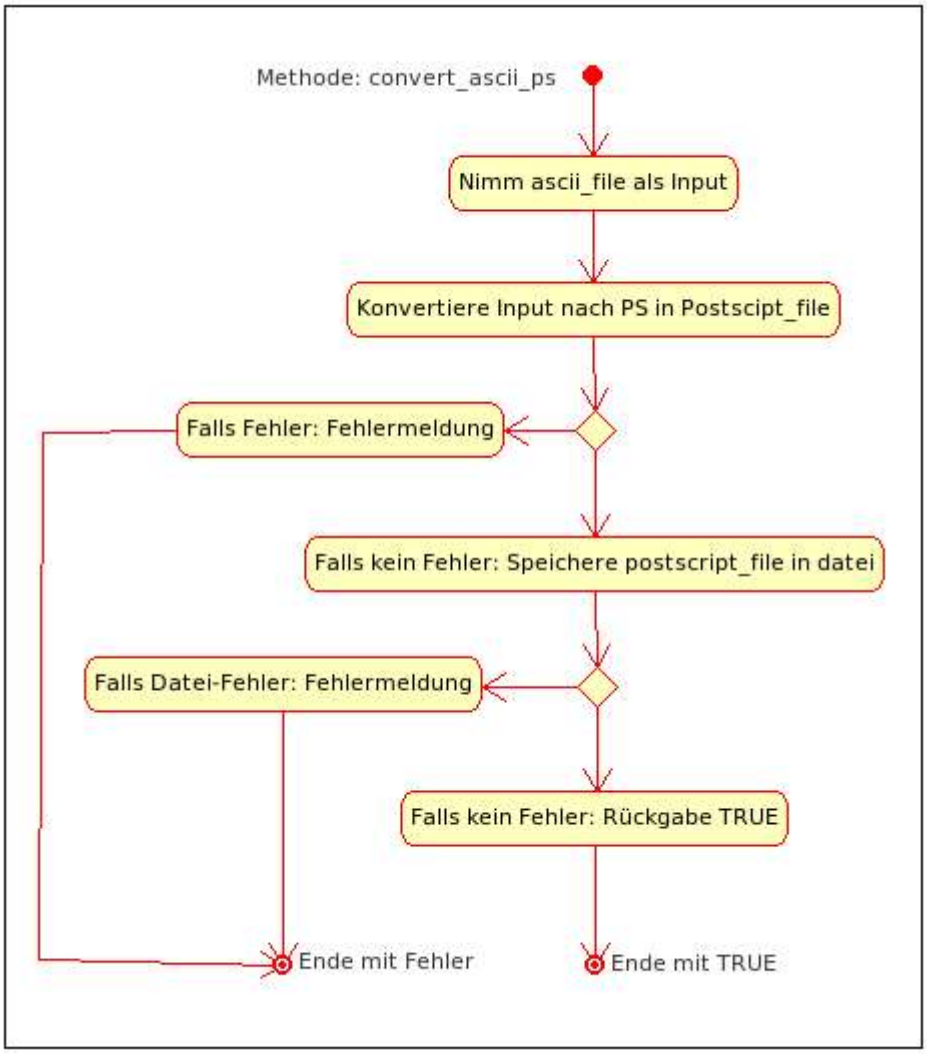

Bild 5: Aktivitätsdiagramm zur Methode convert_ascii_ps()

Die Grundidee dieses Diagramms ist, dass der ASCII-Text aus dem Puffer *ascii_file* gelesen und dann in ein Postscript-Programm umgewandelt wird. Über die Details dieser Umwandlung wird in dem Aktivitätsdiagramm noch nichts gesagt. Wir geben daher hier zunächst nur eine grobe Implementierung, bei der die Details der Übersetzung noch ausgepart sind. Ausserdem wird die Implementierung der Methode *fileselect()* aus der letzten vorlesng nochmals abgeändert; für Kommentare siehe den nachfolgenden Text.

```
/*************************************************************************

name: ppmp2ps.cpp

author: gerd d-h
first time: april-16, 2005
last change: may-8, 2005
intention: Read an ascii-textfile.txt and convert it into postscript
textfile.ps

LOCATION: /home/gerd/public_html/fh/II-PPmP/VL/VL5/ppmp2ps.cpp

COMPILATION: g++ -o ppmp2ps_usage ppmp2ps_usage.cpp ppmp2ps.cpp
USAGE: ppmp2ps_usage

*************************************************************************/

#include "ppmp2ps.hpp"

#include <string>       // for strings
#include <iostream>     // for I/O
#include <fstream>     // for I/O
#include <cstdlib>      // for exit()


/*******************
 * Methods
 *******************/

/*****************************
 *  Holt sich einen Dateinamen und liest die zugehoerige Datei in einen Puffer
 ************************************************************************/

void ppmp2ps::fileselect (string filename) {


  cout << "Dies ist die Metode fileselect!" << endl;

  cout << "Geben sie bitte einen Dateinamen ohne Endung ein ";

  cin >> filename;

  file_name = filename;

  filename = filename + ".txt";

    // Oeffne input file

  ifstream file(filename.c_str());

    // Datei geoeffnet?

    if (! file) {

        // NEIN, beende Programm
```

```
        cerr << "Datei laesst sich nicht oeffnen: \"" << filename << "\""
             << endl;
        exit(EXIT_FAILURE);
    }

    // Kopiere Dateiinhalt in Puffer und dann  auf den Bildschirm

    char c;
    while (file.get(c)) {
      ascii_file.push_back(c);
    }

    cout << "TEST: Inhalt von ascii_file: " <<  endl <<
"-------------------------" << endl <<  ascii_file << endl;
}


/*****************************************************************
 * Konvertiert einen zuvor ausgewÃ¤hlten ASCII-Text in eine Postscript Datei
 *****************************************************************/

void ppmp2ps::convert_ascii_ps () {

  cout << "Dies ist die Metode convert_ascii_ps!" << endl;

  cout << "Die Methode liest zunaechst den ASCII-Text von ascii_file und gibt
ihn dann auf dem Bildschirm aus" << endl;


  cout << "TEST: content of ascii_file : " <<  endl <<
"-------------------------" << endl << ascii_file << endl;

  for( int i = 0; i < ascii_file.length( ); i++ ){
        cout << ascii_file[ i ];
  }
    cout << endl;

    cout << "Es wird jetzt eine Datei zum Schreiben geoeffnet." << endl;

    file_name = file_name + ".ps";

    ofstream file(file_name.c_str());

    // Test auf Fehler beim Oeffnen der Datei

    if (! file) {
        // Fehler; Stoppe Programm
        cerr << "folgende Datei kann nicht geoeffnet werden: \"" << file_name
<< "\""
             << endl;
        exit(EXIT_FAILURE);
    }


    cout << "Es wird jetzt die Datei geschrieben." << endl;

    file << ascii_file;

}
```

Die Änderungen in der Methode fileselect() beziehen sich einmal auf die folgende Stelle: der Dateiname wird nun ohne Endung eingegeben und die Variable für diesen Namen ist verschieden von dem Klassenattribut *file_name*. Dadurch kann man die Eingabe unverändert intern speichern und den eingegebenen Namen mit einem Suffix ".txt"

versehen. Die Operationen der Stringzuweisung mit "=" sowie "+" gehören zur Klasse String (siehe dazu: http://www.fbmnd.fh-frankfurt.de/~doeben/I-PROGR2/I-PROGR2-TH/VL1/i-progr2-th-vl1.html#String).

```
cout << "Geben sie bitte einen Dateinamen ohne Endung ein ";

cin >> filename;

file_name = filename;

filename = filename + ".txt";
```

Die andere Änderung bezieht sich auf die Art und Weise, wie die aus der Datei gelesenen Zeichen in den String eingefügt werden. Hier wird die Methode .push_back() benutzt. Damit ist es möglich, am Ende eines String jeweils ein neues Zeichen anzuhängen.

```
char c;
while (file.get(c)) {
  ascii_file.push_back(c);
}
```

In der neuen Methode convert_ascii_ps() werden die Elemente des Puffers ascii_file mittels der Methode [idx] ausgelesen, d.h. man addresiert die elemente des Pufers einzeln. Vor Beginn dieser Abfrage besorgt man sich die Länge des Puffers.

```
for( int i = 0; i < ascii_file.length( ); i++ ){
      cout << ascii_file[ i ];
}
```

Die ausgabedatei soll eine Postscrpt-Datei sein. Diese benötigt als Suffix ".ps". Dazu holt man sich den ursprünglichen Dateinamen und fügt mit der Moethode "+" das suffix an. Dann öffnet man eine File zum Schreiben mittels der Klasse ofstream.

```
cout << "Es wird jetzt eine Datei zum Schreiben geoeffnet." << endl;

file_name = file_name + ".ps";

ofstream file(file_name.c_str());
```

Falls beim Öffnen der Datei zum Schreiben kein Fehler auftritt, kann man den Inhalt des Puffers --oder einer anderen Datei-- in die Datei schreiben.

```
cout << "Es wird jetzt die Datei geschrieben." << endl;

file << ascii_file;
```

In der endgültigen Version wird man möglicherweise den Postscripttext erst im Puffer *postscript_file* zwischenspeichern und erst zum Schluss in eine Datei schreiben.

Eine Anwendung des programms sieht dann wie folgt aus:

```
gerd@kant:~/public_html/fh/II-PPmP/VL/VL5> cat dummy.txt
Dies ist ein Dummy-Inhalt.
Jetzt haben wir Zeile 2.
Jetzt Zeile 3.


gerd@kant:~/public_html/fh/II-PPmP/VL/VL5> g++ -o ppmp2ps_usage
ppmp2ps_usage.cpp ppmp2ps.cpp
gerd@kant:~/public_html/fh/II-PPmP/VL/VL5> ./ppmp2ps_usage
Dies ist die Metode fileselect!
Geben sie bitte einen Dateinamen ohne Endung ein dummy
TEST: Inhalt von ascii_file:
------------------------
Dies ist ein Dummy-Inhalt.
Jetzt haben wir Zeile 2.
Jetzt Zeile 3.


Dies ist die Metode convert_ascii_ps!
Die Methode liest zunaechst den ASCII-Text von ascii_file und gibt ihn dann auf dem
Bildschirm aus
TEST: content of ascii_file :
------------------------
Dies ist ein Dummy-Inhalt.
Jetzt haben wir Zeile 2.
Jetzt Zeile 3.


Dies ist ein Dummy-Inhalt.
Jetzt haben wir Zeile 2.
Jetzt Zeile 3.


Es wird jetzt eine Datei zum Schreiben geoeffnet.
Es wird jetzt die Datei geschrieben.
```

3. C++-Sprachelemente

In den vorausgehenden Vorlesungen haben wir gesehen, wie man eine UML-Klasse in eine C++-Klasse übersetzen kann. Ferner wurde erklärt, wie man eine C++-Klasse benutzen kann: man benutzt den Klassennamen als neue Typbezeichnung und führt dann ein Objekt dieser Klasse dadurch ein, dass man eine neue Variable zu dem Klassentyp einführt. Im Programm ppmp2ps_usage.cpp war dies mit der Deklaration

ppmp2ps a1;

erreicht worden. Hier war *a1* ein neues Objekt vom Typ ppmp2ps. In der sogenannten *Standard Library (STL)* stellt die Sprache C++ dem Benutzer eine Reihe von vordefinierten Klassen zur Verfügung, die man innerhalb von C++ wie einen normalen Datentyp benutzen kann. Dazu gehört z.B. die Klasse *string* sowie die Klassen *iostream, ifstream* sowie *ofstream*.

3.1 Die Klasse *string*

Wie alle Klassen hat die Klasse *string* eine Schnittstelle, über die die öffentlich zugänglichen Funktionen zugänglich sind. Einige der wichtigsten dieser Funktionen finden sich in der nachfolgenden Tabelle.

Von diesen verfügbaren string-Methoden wurden bislang die folgenden benutzt:

```
string file_name;
string ascii_file;
string postcript_file;
```

In allen Fällen handelt es sich darum, ein neues Objekt vom Typ string einzuführen. Wie man aus der Liste der verfügbaren string-Methoden entnehmen kann, könnte man bei der Einführung neuer string-Objekte sehr wohl auch Startwerte mit übergeben.

Ferner wurde von der Operation "=" Gebrauch gemacht, indem man einem String den Wert eines anderen Strings direkt zuweisen kann. Zusätzlich die Konkatenationsoperation "+" mit einer Stringkonstanten ".txt". Schliesslich die Generierung eines Zeigers auf einen C-String mittels ".c_str()".

```
file_name = filename;
filename = filename + ".txt";
ifstream file(filename.c_str());
```

Schliesslich wird auch noch die Methode .length() benutzt, um die Länge eines Strings feststellen zu können in Verbindung mit dem Zugriff über den Index [idx].

```
for( int i = 0; i <  ascii_file.length( ); i++ )
cout << ascii_file[ i ];
```

string c	Creates empty string c with no elements
string c1(c2)	Creates a copy of string c2 into c1
string c(n,a)	Creates a string c with n copies of character a
string c(beg,end)	Creates a string c initialized with the characters of the range [beg,end)
c.~string()	Destroys all elements and frees the memory
c.size()	Returns the actual number of elements
c.empty()	Returns whether the string is empty c
c.max_size()	Returns the maximum number of elements possible
c.length()	Returns the number of elements
c.reserve()	Enlarges capacity if not enough
c1 == c2	Whether equal
c1 != c2	Not equall
c1 < c2	Lesser than
c1 > c2	Greater than
c1 >= c2	greater than or equal
c1 <= c2	lesser than or equal
c1 = c2	Assigns all elements of c2 to c1
c.assign(n,elem)	Assigns n copies of elem to c
c.assign(beg,end)	Assigns the elements of the range beg - end
c1.swap(c2)	Swaps the data of c1 and c2
swap(c1,c2)	Swaps the data of c1 and c2
c.at(idx)	Returns the element with index idx. In case of 'out of range' throws an error (the only range checking function !)
c[idx]	Returns element with index idx (no range checking!)
c.getline()	Read the value of a stream
c.push_back(elem)	Appends a copy of element elem at the end
c1 + c2	Concatenates two strings
c.resize(num)	Changes the number of elements to num; new elements are created by the default constructor
c.clear()	Removes all elements
c.c_str()	Return the string as C-String
c.data()	Return the string as an array of characters without terminating 0.

3.2 Ein-/Ausgabe

Sämtliche input-Output-Prozesse werden in C++ als Streams behandelt, d.h. als 'Strom von Daten', wobei ein Stream als ein Objekt konstruiert ist, das bestimmte Eigenschaften hat und mittels bestimmter Streamfunktionen manipuliert werden kann.

Die grossen Unterscheidungen sind Input und Output. Diese werden zusätzlich unterschieden nach 'normalem' I/O, Datei-I/O und Stream-I/O.Zusätzlich gibt es für diese verschiedenen Streams auch noch Buffer, die mit den Streams verknüpft werden können. Eine allgemeine Übersicht gibt das nächste Schaubild:

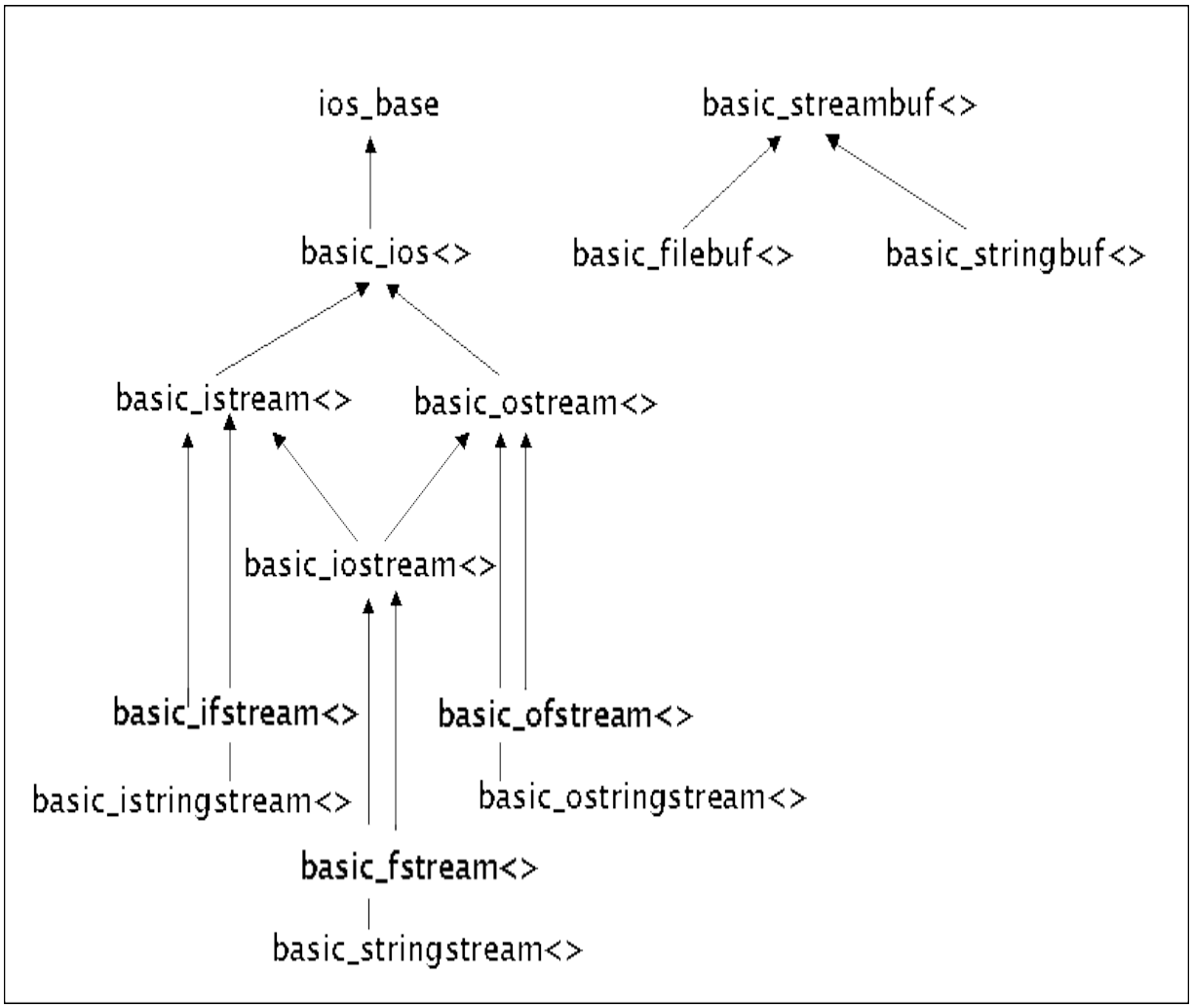

Bild: I/O-Basisklassen

Man kann in dem Schaubild u.a. erkennen, dass eine allgemeine Input-Output-Klasse ios_base mit einem Template basic_ios<> aufgespalten wird in zwei Templates für input-stream und output-stream. Der input-stream ist zum Lesen von Daten und der output-

stream zum Schreiben von Daten. Ein kombinierter input-output-stream ist auch verfügbar. Diese Basisklassen sind dann nochmals zusätzlich spezifiziert für Dateioperationen und Stringoperationen.

Entsprechend der Struktur von Klassen kann man bei einer Klasse neben den Attributen auch Elementfunktionen unterscheiden. Die öffentlich zugänglichen Elementfunktionen der wichtigen I/O-Klassen zeigt das nächste Schaubild:

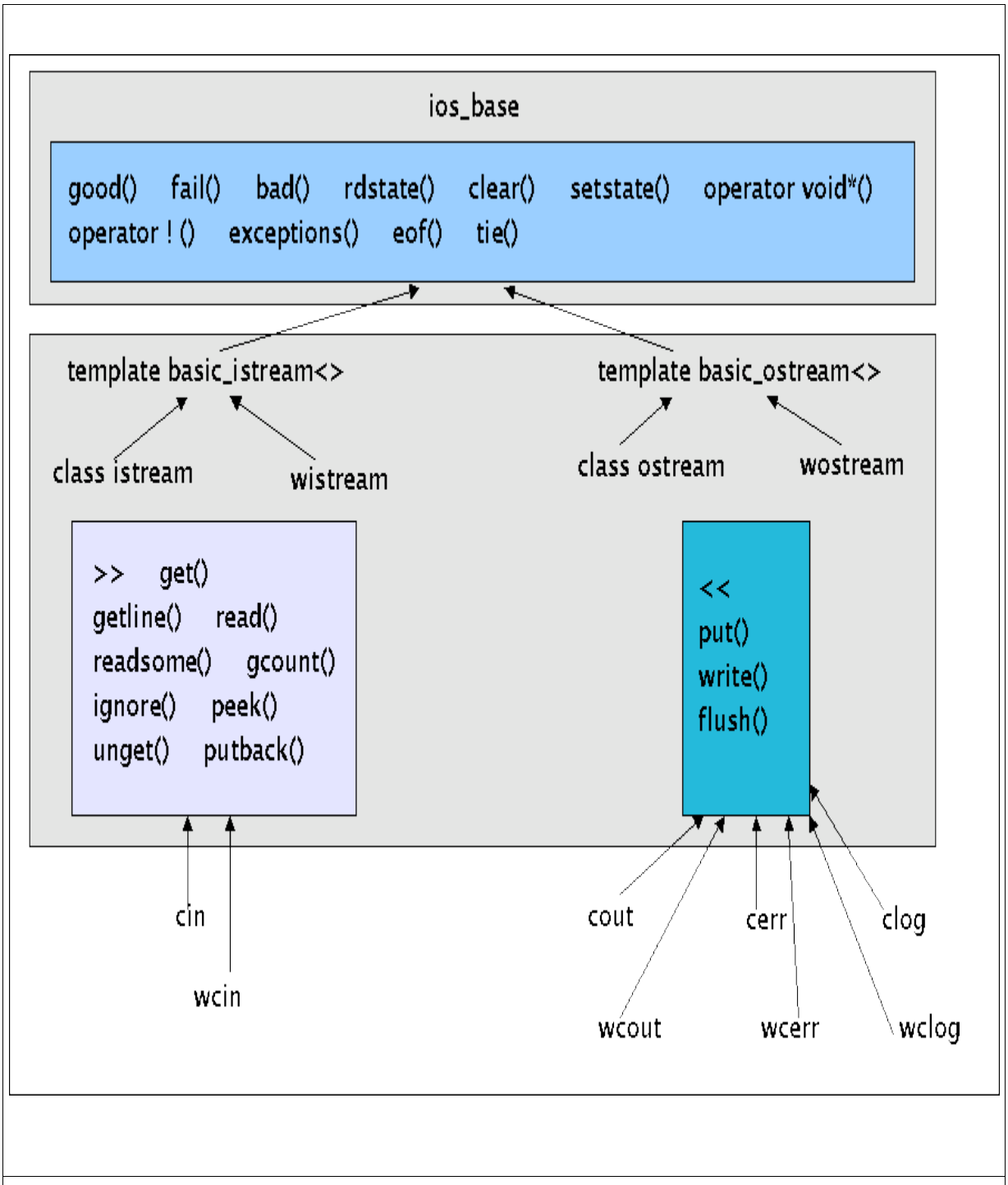

Bild: Elementmethoden der IO-Klassen

Schon die Basisklasse *ios_base* stellt einige Funktionen zur Verfügung:

- *good()* := Gibt 'true' zurück wenn das goodbit gesetzt ist.
- *fail*() := Gibt 'true' zurück wenn das failbit or badbit gesetzt ist.
- *bad*() := Gibt 'true' zurück wenn das badbit gesetzt ist.
- *eof*() := Gibt 'true' zurück wenn das eofbit gesetzt ist.
- *rdstate* := Gibt die aktuell gesetzten Flags zurück
- *clear*() := Klärt (und setzt) alle (state) Flags
- *setstate*(state) := Setzt zusätzliche state-Flags
- *operator* void* () := Gibt zurück, ob der Stream nicht in einen Fehler gelaufen ist
- *operator* ! () := Gibt zurück, ob der Stream in einen Fehler gelaufen ist
- *exceptions*(flags) := Setzt flags, die Ausnahmen triggern
- *exceptions*() := Gibt die flags zurück, die Ausnahmen triggern
- *tie*() := Bindet zwei Streams zusammen

Zusätzlich haben die speziellen Klassen für Input bzw. Output spezielle Elementfunktionen:

Elementfunktionen für **Input**:

- >> := Liest aus dem Stream
- *get*() := liest das nächste Zeichen
- *getline*() := liest ganze Zeile
- *read*(count) := liest count-viele Zeichen
- *readsome*(count) := liest bis zu count-viele Zeichen
- *gcount*() := Gibt die Anzahl der gelesenen Zeichen durch die letzte read-Operation zurück

- *ignore*() := ignoriert Zeichen
- *peek*() := Liest das nächste Zeichen ohne es aus dem Stream zu entfernen
- *unget*() := Setzt das letzte gelesene Zeichen wieder zurück in den Stream
- *putback*() := wie unget()

Elementfunktionen für **Output**:

- << := Schreibt in den Stream
- *put*(c) := Schreibt c in den Stream
- *write*(count, str) := Schreibt count-viele Zeichen von str in den Stream
- *flush*() := flushed die Puffer des Streams

Auf der Basis der zuvor definierten Klassen und Templates kann man Objekte generieren. Einige Objekte werden standardmässig generiert. Für den Inputstrema sind dies *cin* und *wcin*; für den output-stream sind dies *cout, cerr* und *clog* sowie die *w-Versionen*.

Für die *Dateioperationen* gibt es einige zusätzliche Operationen:

Elementfunktionen für **Input und Output**:

- open() := Öfffnet eine Datei für einen Stream
- close() := Schliesst eine Datei für einen Stream
- is_open() := Gibt zurück, ob eine Datei noch offen ist.

Elementfunktionen für **Input**:

- tellg() := Gibt die Leseposition zurück
- seekg() := Setzt die Leseposition

Elementfunktionen für **Output**:

- tellp() := Gibt die Schreibposition zurück
- seekp() := Setzt die Schreibposition

Elementfunktionen für **Buffer-Klasse**:

- rdbuf() := Gibt einen Zeiger zum Stream-Buffer zurück

- rdbuf(*streambuf**) := Richtet einen Stream-Burfer ein, auf den das Argument zeigt, und gibt einen Zeiger auf den vorher benutzten Stream-Buffer zurück

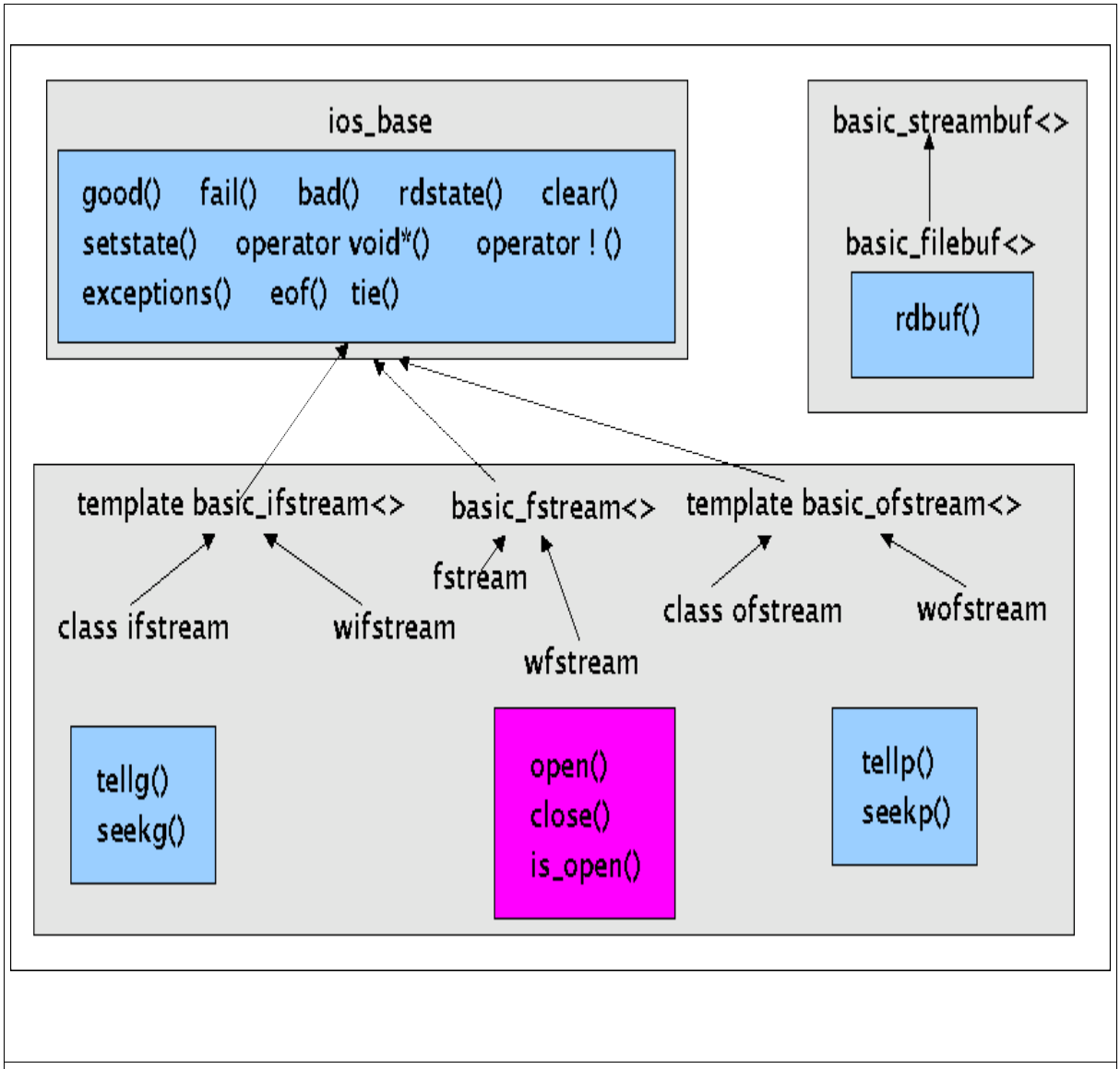

Bild: Spezielle Methoden für Dateioperationen

Manipulatoren sind spezielle Objekte, um Streams zu manipulieren, so das bekannte 'endl', das schon mehrfach benutzt wurde:

```
std::cout << std::endl;
```

'endl' fügt bekanntlich ein '\n' ein. Zusätzlich wird der Output-Puffer geleert. Hier die Liste der Manipulatoren (noch nicht vollständig):

MANIPULATOR	KLASSE	BEDEUTUNG
endl	ostream	Gibt '\n' aus und flushed den output-Puffer
ends	ostream	Gibt '\0' aus
flush	ostream	flushed den output-Puffer
ws	istream	Liest whitespace und wirft es weg
setiosflags (*flags*)	ios_base	Setzt *flags* als format-Flags (ruft setf(*flags*) für den Stream); benötigt Inklude-Datei <iomanip> (s.u. bei ---> Formatierung)
setiosflags (*mask*)	ios_base	Klärt alle flags die durch *mask* identifiziert werden (Ruft setf(0,*mask*) für den Stream); benötigt Inklude-Datei <iomanip> (s.u. bei ---> Formatierung)
boolalpha	ios_base	erzwingt alphanumerische Representation(s.u. bei ---> Formatierung)
noboolalpha	ios_base	erzwingt numerische Representation(s.u. bei ---> Formatierung)
setw(*val*)	ios_base	setzt die Feldbreite für input und output auf *val* (s.u. bei ---> Formatierung)
setfill(*c*)	ios_base	definiert *c* als Füllzeichen (s.u. bei ---> Formatierung)
left	ios_base	adjustiert den Wert links (s.u. bei ---> Formatierung)
right	ios_base	adjustiert den Wert rechts (s.u. bei ---> Formatierung)
internal	ios_base	adjustiert das Zeichen links und den Wert rechts (s.u. bei ---> Formatierung)
oct	ios_base	schreibt und liest Oktalzahlen (s.u. bei ---> Formatierung)
hex	ios_base	schreibt und liest Hexzahlen (s.u. bei ---> Formatierung)
dec	ios_base	schreibt und liest Dezimalzahlen (s.u. bei ---> Formatierung)
...	ios_base	...

Im Rahmen der Klasse ios_base sind mehrere Zustandsbits definiert, anhand denen man den Zustand eines Streams ablesen kann. Sie stehen allen Objekten vom Typ basic_istream und basic_outstream zur Verfügung. Stream-Puffer haben allerdings keine Zustandsbits.

KONSTANTE	BEDEUTUNG
goodbit	Alles ist OK; kein anderes Bit ist gesetzt
eofbit	Ende der Datei wurde erreicht
failbit	Fehler; eine I/O-Funktion war nicht erfolgreich
badbit;	Schlimmer Fehler; ein undefinierter Zustand wurde erreicht

Diese Zustandsbits können mittels Funktionen der Klasse ios_base manipuliert werden.

- **good()** := Gibt 'true' zurück wenn das goodbit gesetzt ist.
- **fail()** := Gibt 'true' zurück wenn das failbit or badbit gesetzt ist.
- **bad()** := Gibt 'true' zurück wenn das badbit gesetzt ist.
- **eof()** := Gibt 'true' zurück wenn das eofbit gesetzt ist.
- **rdstate** := Gibt die aktuell gesetzten Flags zurück
- **clear()** := Klärt alle Flags
- **clear(*state*)** := Klärt alle Flags und setzt state-Flags
- **setstate(*state*)** := Setzt zusätzliche state-Flags
- **operator void* ()** := Gibt zurück, ob der Stream nicht in einen Fehler gelaufen ist
- **operator ! ()** := Gibt zurück, ob der Stream in einen Fehler gelaufen ist
- exceptions(flags) := Setzt flags, die Ausnahmen triggern
- exceptions() := Gibt die flags zurück, die Ausnahmen triggern

Aufgrund der Mängel der Ausnahmebehandlung in C++ wird im allgemeinen empfohlen, diese möglichst nicht zu benutzen

C++ bietet auch zahlreiche Möglichkeiten die Formate der ein- und Ausgaben zu beeinflussen. Um dies tun zu können, gibt es mehrere Möglichkeiten:

- **Format-Flags** := jeder Stream besitzt eine Reihe von Flags, die besagen, welche Formateigenschaften aktiviert sind. diese kann man auslesen, kopieren und setzen.

- **Format-Manipulatoren** := Format-Manipulatoren können im Rahmen der standard-Ein-Ausgabe Funktionen '<<' sowie '>>' benutzt werden, um direkt Formateigenschaften ein- oder auszuschalten (die Liste der format-Manipulatoren ist Teil der Liste der Manipulatoren; s.o.)

- **Format-Element-Funktionen** := diese können benutzt werden, um unabhängig von einer aktuellen Ein- oder Ausgabeoperation formateigenschaften abzufragen oder zu setzen.

Einige der Format-Element-Funktionen sind hier aufgelistet.

FUNKTION	BEDEUTUNG
setf(*flags*)	Setzt *flags* als zusätzliche format-flags und gibt die bisherigen Zustand aller Flags zurück.
setf(*flags, mask*)	Setzt *flags* als zusätzliche format-flags für die durch *mask* identifizierte Gruppe und gibt die bisherigen Flags zurück.
unsetf(*flags, mask*)	Klärt *flags*.
flags()	Gibt alle aktuell gesetzten Format-flags zurück.
flags(*flags)*	Setzt *flags* als die neuen Format-flags und gibt die bisherigen Flags zurück.
copyfmt (*stream)*	Kopiert *alle* Format-Definitionen von *stream*.
copyfmt (*stream)*	Kopiert *alle* Format-Definitionen von *stream*.

Von diesen Klassen und Elementfunktionen wurden die folgenden bislang benutzt:

```
cout << "Dies ist die Metode fileselect!" << endl;

cout << "TEST: " <<  endl << "------" << endl << ascii_file << endl;
```

```
cin >> filename;

cerr << "Datei laesst ...nicht oeffnen: \"" << filename << "\"" << endl;
```

Bei *cout, cin* und *cerr* handelt es sich um Objeke von Klassen, die standardmaessig zur Verfügung gestellt werden.

Objekte	***Bedeutung***
std::istream cin;	Standardeingabekanal; entspricht meistens der Tastatur (in C ist dies *stdin*)
std::ostream cout;	Standardausgabekanal; entspricht meistens dem Bildschirm (in C ist dies *stdout*)
std::ostream cerr;	Standardfehlergabekanal; entspricht meistens dem Bildschirm (in C ist dies *stderr*)

Öffnen von Dateien zum Lesen (*ifstream*) bzw. zum Schreiben (*ofstream*). Zugleich sieht man, wie man in eine geöffnete Datei schreiben kann.

```
ifstream file(filename.c_str());

ofstream file(file_name.c_str());

file << ascii_file;
```

Der Operator **!** gibt zurück, ob es eine Fehlermeldung gegeben hat.

```
 if (! file) {
```

Mit der Elementfunktion .get(c) wird auf eine geöffnete Datei *file* zugegriffen

```
   while (file.get(c))
```