VL5: Softwareprojekt - Implementierung Teil 1

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.

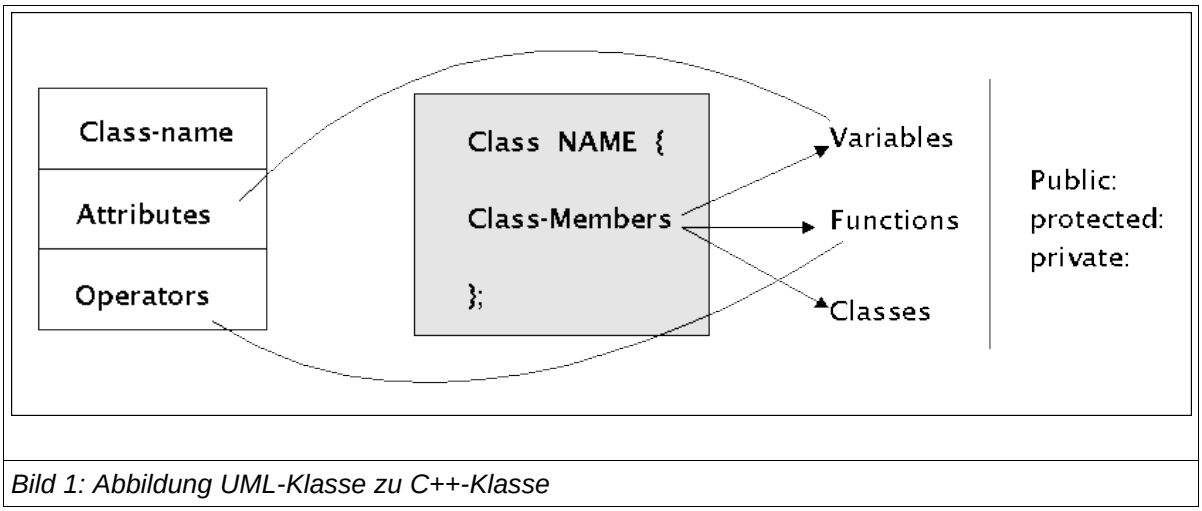

Bild 1: Abbildung UML-Klasse zu C++-Klasse

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.

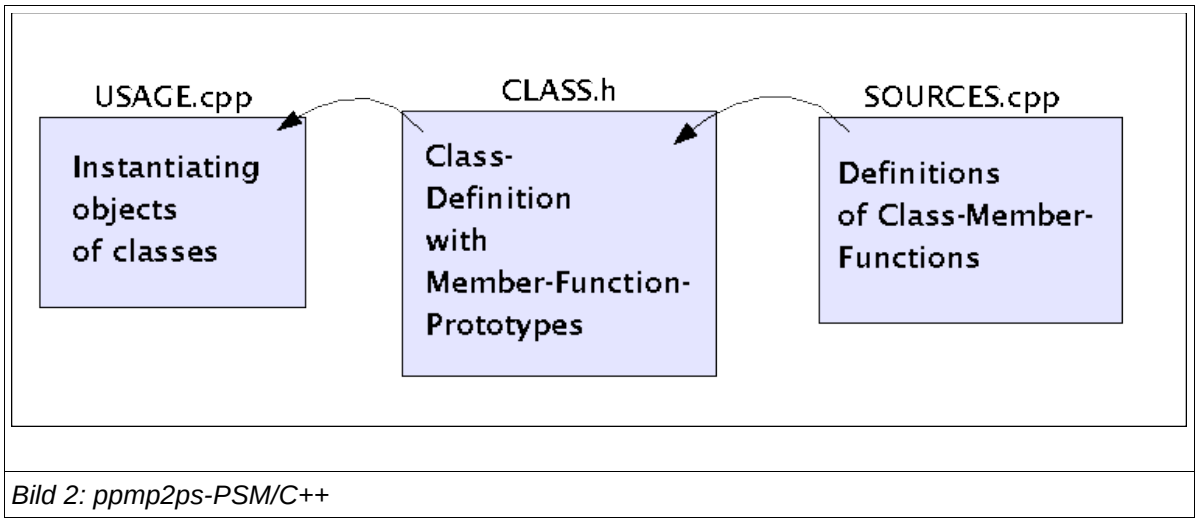

Bild 2: ppmp2ps-PSM/C++

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. Siehe den folgenden Text:

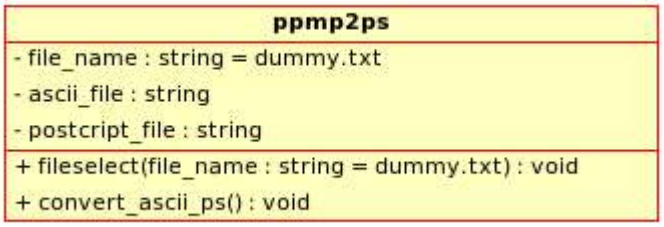

Bild 3: Die Klasse pmp2ps

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.

Da die Operationen in dieser 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;

}
```

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.

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.

Für die Methode *fileselect()* wurde folgendes Aktivitätasdiagramm 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:

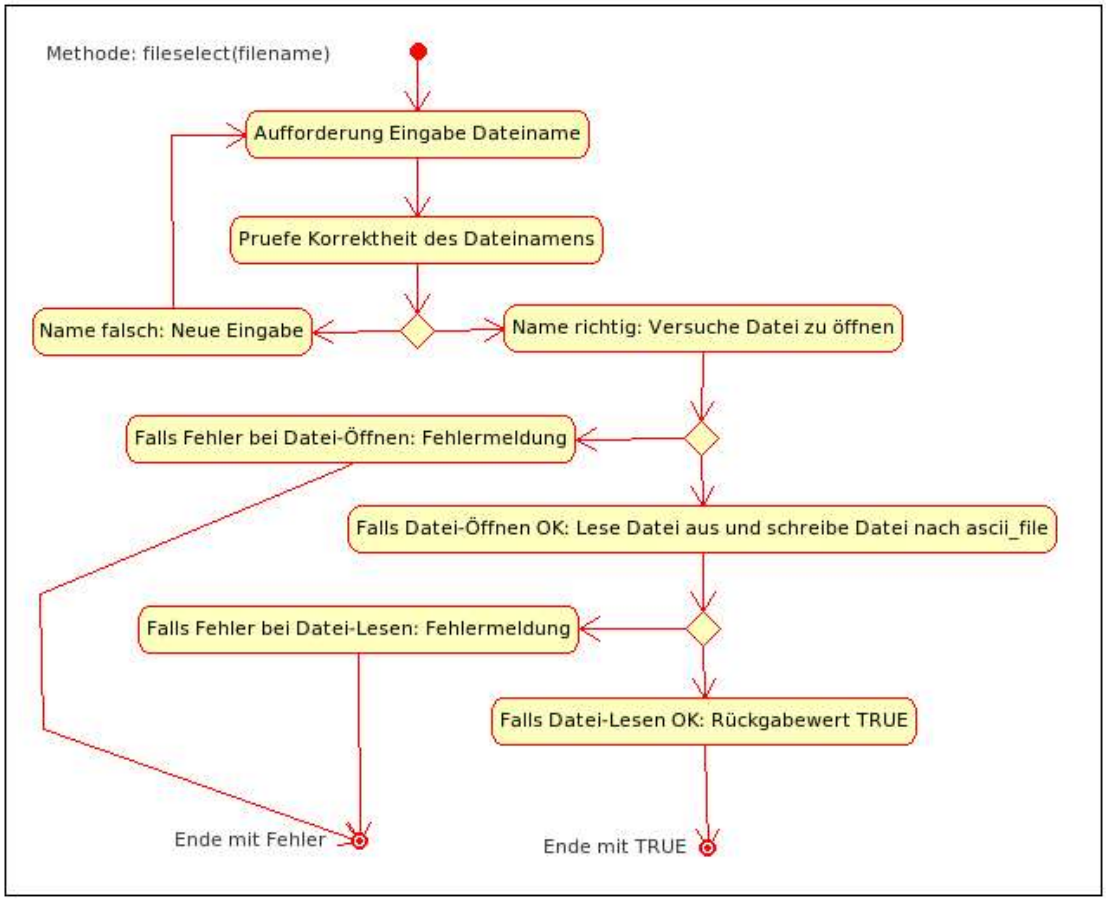

Bild 4: ppmp2ps-Aktivitätsdiagramm für die Methode fileselect(filename)

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.

In der folgenden Vorlesung wird eine erste Implementierung der Funktion *convert_ascii_ps()* besprochen werden. Dann folgt eine ausführliche Kommentierung.