VL3: Softwareprojekt - Modellierung/Design

Inhalt

1. Struktur eines Softwareprojektes
2. Modellierung/Design allgemein
3. ppmp2ps virtuelle Maschine/ ppmp2ps PIM
4. Übersetzung von ppmp2ps-PIM nach ppmp2ps-PSM

1. Struktur eines Softwareprojektes

In der vorausgehenden Vorlesung wurde ein allgemeines Schema für Softwareprojekte eingeführt, wie es im Rahmen der *Model Driven Architecture (MDA)* der *OMG* angenommen wird (siehe Bild 1). An einem einfachen Beispiel wurde gezeigt, was es bedeuten kann, ein konkretes Anwendungsbeispiel als ein *computerunabhängiges Modell (CIM)* in Form eines *Anwendungsfalldiagramms (Use Case)* zu konstruieren. In dieser Vorlesung geht es nun darum, zu zeigen, wie man auf der Basis eines Anwendungsfalls ein *plattformunabhängiges Modell (PIM)* konstruieren kann.

Ein plattformunabhängiges Modell ist innerhalb der MDA eine *virtuelle Maschin*e --man könnte auch sagen ein *abstraktes System*-- die sich definiert über Teile, Services und Interaktionen untereinander (vgl. MDA Guide 1.0.1. paragraph 3.2).

Die Aufgabe besteht also darin, die Menge der Anforderungen aus dem Anwendungsfall in eine abstrakte Struktur zu überführen, von der man annehmen kann, dass sie sich in ein plattformspezifisches Modell überführen lässt, das dann genau diese Anforderungen erfüllt. Die Konstruktion solch eines plattformunabhängigen Modells hat dabei ein unausweichlich *kreatives* Moment, da sich aus der Anwendungsfallanalyse solch ein plattformunabhängiges Modell nicht direkt ableiten lässt. Selbst wenn man davon ausgeht, dass es in Softwarefirmen grössere Sammlungen von schon bekannten plattformunabhängigen Modellen gibt, ist die Auswahl des geeigneten Modells kein rein automatischer Vorgang (und wird es aus prinzipiellen Gründen auch niemals sein werden).

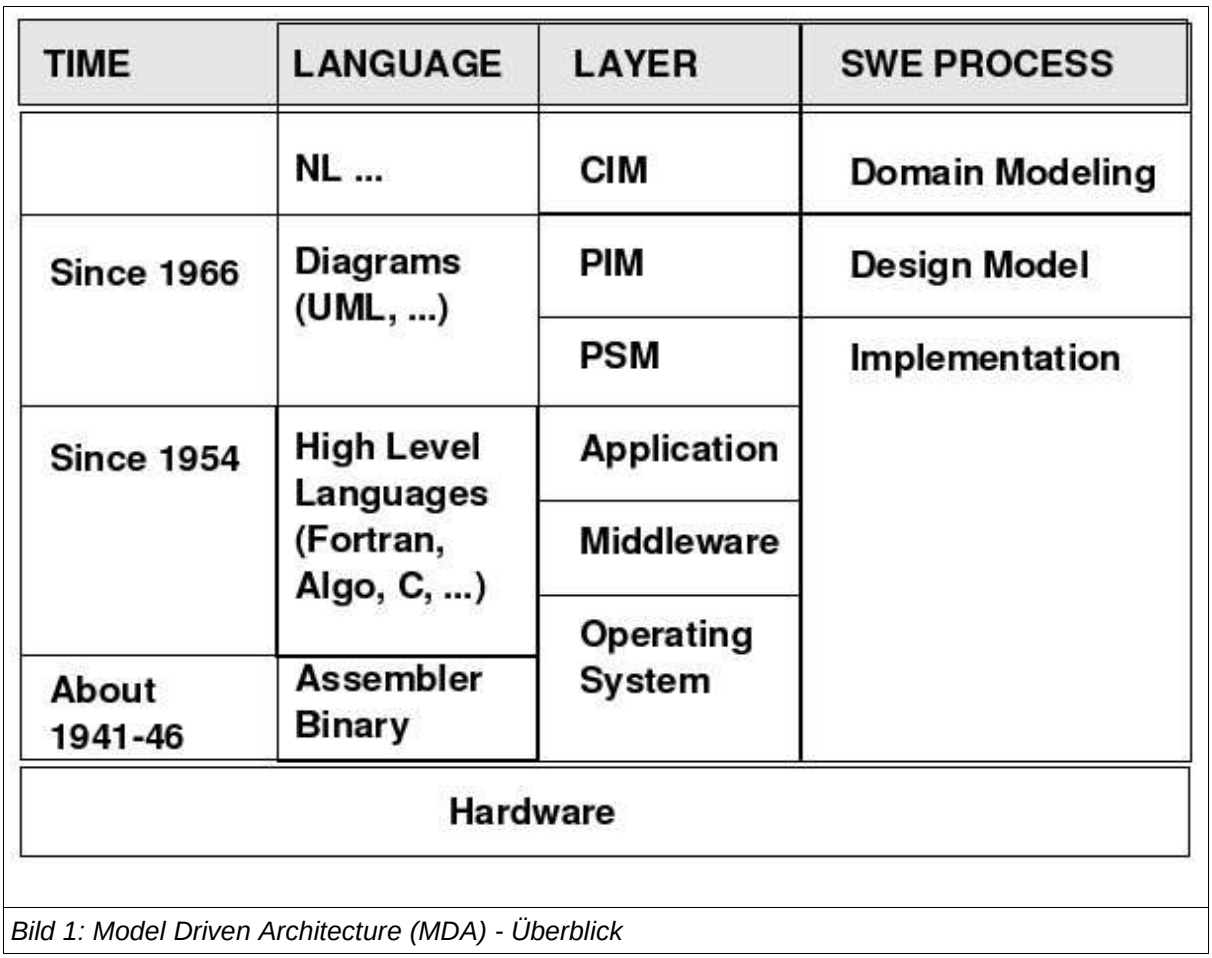

Bild 1: Model Driven Architecture (MDA) - Überblick

2. Modellierung/Design allgemein

Im Rahmen der MDA spricht man allgemein von Modellierung, wenn ein konkretes Problem einer Lösung zugeführt werden soll. Ausserhalb der MDA ist auch der Begriff des *Designs* anstatt des plattformunabhängigen Modells (evtl. unter Einschluss des plattformspezifischen Modells) sehr verbreitet. Dies sei hier nur erwähnt.

Die Frage ist nun, was man sich unter einer *virtuellen Plattform* bzw. einem *abstrakten System* vorstellen soll?

Zu einem abstrakten System gehören *Teile (parts), Dienste (services)* sowie *Beziehungen (relations*) zwischen diesen Teilen.

Teile (parts) werden als *Klassen* dargestellt, deren Instanzen *Objekte* sind. Klassen können *Attribute* und *Methoden* haben.

In einer *statischen* Sicht haben Klassen eine bestimmte *Struktur*, die sich mithilfe von *Klassendiagrammen* und *Komponentendiagrammen* beschreiben lässt; zusätzlich kann

man bestimmte *statische Beziehungen* zwischen Klassen angeben (Verallgemeinerungen, Assoziationen, Aggregationen, Kompositionen).

In einer *dynamischen* Sicht kan man das Aufeinanderfolgen von Interaktionen und Zuständen beschreiben (Sequenzdiagramme, Zustandsdiagramme, Aktivitätsdiagramme sowie Kollaborationsdiagramme (= in UML 2.0 Kommunikationsdiagramme)).

Ein abstraktes System (= virtuelle Maschine = plattformunabhängiges Modell) im Sinne der MDA ist also eine Menge von Klassen, die in ihren statischen und dynamischen Eigenschaften soweit spezifiziert werden, dass die intendierten Verhaltensleistungen damit eingelöst werden können. Aus der Sicht des *Verhaltens* ist ein PIM also ein *kausales Modell* zur Bereitstellung von gewünschten Verhaltenseigenschaften unter vorgegebenen Umweltbedingungen.

3. ppmp2ps virtuelle Maschine/ppmp2ps PIM

Hier nun weitere Überlegungen für ein einfaches Beispiel für eine virtuelle Maschine, die die zuvor geschilderte Anwendungssituation realisieren soll. Diese Überlegungen gehen *evolutionär* vor, d.h. es werden zu Beginn sowenig Annahmen wie möglich gemacht, diese werden umgesetzt, und dann wird überprüft, inwieweit diese Umsetzung schon den Anforderungen aus dem Anwendungsfall entspricht. Falls nicht, wird das Modell verfeinert; falls alle Anforderungen erfüllt sind, wird das Modell als abgeschlossen betrachtet.

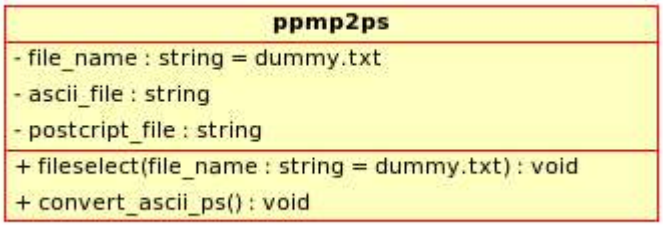

Bild 2: Eine erste Systemkomponente, die Klasse pmp2ps

Bild 2 zeigt einen Vorschlag für eine erste Komponenten des abstrakten Systems ppmp2ps in Form eines *Klassendiagramms*. Eine UML-Klasse kann aus drei Teilen bestehen: (i) dem *Namen* (immer notwendig), (ii) den *Attributen* (nicht notwendig) sowie (iii) den *Operationen* (nicht notwendig) (*Methoden* sind die Instanzen von Operationen und spezifizieren konkrete Algorithmen und Prozeduren).

Der Namen ist Teil eines *Namensraumes (namespace)* und muss innerhalb dieses Namensraumes eindeutig sein. Die *Attribute* beschreiben Eigenschaften und die

Operationen repräsentieren Dienste.

Im vorliegenden Fall werden zwei Operationen angenommen: *file_select* sowie *convert_ascii_ps.* Beide Operationen sind *nach aussen sichtbar*, d.h. sie sind *public*. Diese Form der Sichtbarkeit wird wird durch ein "+"-Zeichen repräsentiert.

Die Operation *file_select* hat die Signatur *(file_name: string = dummy.txt):void.* Dies besagt, dass es einen *Parameter* gibt vom Typ *string* mit einem Defaultwert "dummy.txt". Der Rückgabewert ist vom Typ *void*. Diese Operation soll den Dienst anbieten, dass man der Klasse ppmp2ps einen Dateinamen übergeben kann, mittels dessen dann nach einer entsprechenden ASCII-Datei gesucht werden kann.

Die Operation *convert_ascii_ps* hat die signatur *():void.* Dies besagt, dass es keinen *Parameter* gibt. Der Rückgabewert ist vom Typ *void*. Diese Operation soll den Dienst anbieten, dass man die Klasse ppmp2ps auffordern kann, eine zuvor ausgewählte ASCII-Datei in eine Postscript-Datei zu übersetzen.

Alle angeführten Attribute der Klasse ppmp2ps sollen *nicht sichtbar* sein, d.h. sie sind -- in diesem Fall-- *privat (private)*. Sie können nur von den Operationen der Klasse selbst benutzt werden.

Man kann jetzt diskutieren, ob die Angabe dieser privaten Attribute an dieser Stelle schon gerechtfertigt ist, da sie ja schon eine Detaillierung implizieren, die an dieser Stelle noch nicht unbedingt notwendig ist. Sie setzen ja voraus, dass man schon konkrete Überlegungen anstellt, wie denn möglicherweise die Operationen ihren behaupteten Dienst realisieren. Dies stellt den Übergang von der *Operation* zur *Methode* dar. Falls man sich auf den Standpunkt stellt, dass Methoden noch nicht zur plattformunabhängigen Modellierung gehören, dann müsste man diese Attribute an dieser Stelle weglassen. Steht man aber auf dem Standpunkt, dass die Angabe von Methoden noch zum PIM gehören und nicht zum PSM, dann kann man auch Annahmen über private Attribute machen. Für letztere Variante spricht, dass mit der Annahme von Methoden und zugehörigen privaten Attributen noch keine konkreten Plattformen impliziert werden. Diese Position wird hier im weiteren Verlauf eingenommen.

Die angegebenen privaten Attribute *file_name, ascii_file, postscript_file* sind allesamt vom Typ string und unterstellen, dass der Dateiname des zu übersetzenden ASCII-Textes gespeichert wird. Mit diesem Wert wird dann ein ASCII-text geladen, der in dem Puffer *ascii_file* zwischengespeichert wird. Aus diesem Puffer heraus wird er dann in einen Postscript-Text übersetzt, der in dem Puffer *postscript_file* zwischengespeichert wird. Von dort wird der Text mit einer entsprechenden Endung ".ps" dann als neue Postscript-Datei abgespeichert.

4. Übersetzung von ppmp2ps-PIM nach ppmp2ps-PSM

Da wir bei der Konstruktion des Systems ppmp2ps *evolutionär* vorgehen wollen, versuchen wir den ersten Vorschlag für ein ppmp2ps-PIM sogleich in ein *plattformspezifisches Modell*, also in ein *ppmp2ps-PSM*, zu übersetzen. Für solch eine Übersetzung gibt es unterschiedliche Wege.

Als erstes muss man klären, wie denn die Zielplattform überhaupt aussehen soll. Im vorliegenden Fall genügt die Angabe, dass das Modell in der *Sprache C++* implementiert werden soll. Dies bedeutet, dass wir das ppmp2ps-PIM, das in der Sprache UML 1.5 repräsentiert worden ist, nun in ein ppmp2ps-PSM übersetzen müssen, das in der Sprache C++ realisiert wird.

Eine mögliche Realisierung ist hier die, dass man eine Übersetzungszuordnung herstellt zwischen den Elementen des ppmp2ps-PIM einerseits und den Konzepten der C++-Sprache. Der MDA-Standard spricht in diesem Fall von *Marken (marks)*, d.h. die Konzepte in der Sprache C++ sind Marken und diese werden dann den Elementen des ppmp2ps-PIM *angeheftet*.

Im vorliegenden Beispiel besitzt das ppmp2ps-PIM nur einen Typ von Elementen, nämlich Klassen, und auch davon nur eine einzige. Dem Element UML-Klasse entspricht auf Seiten der Sprache C++ die C++-Klasse.

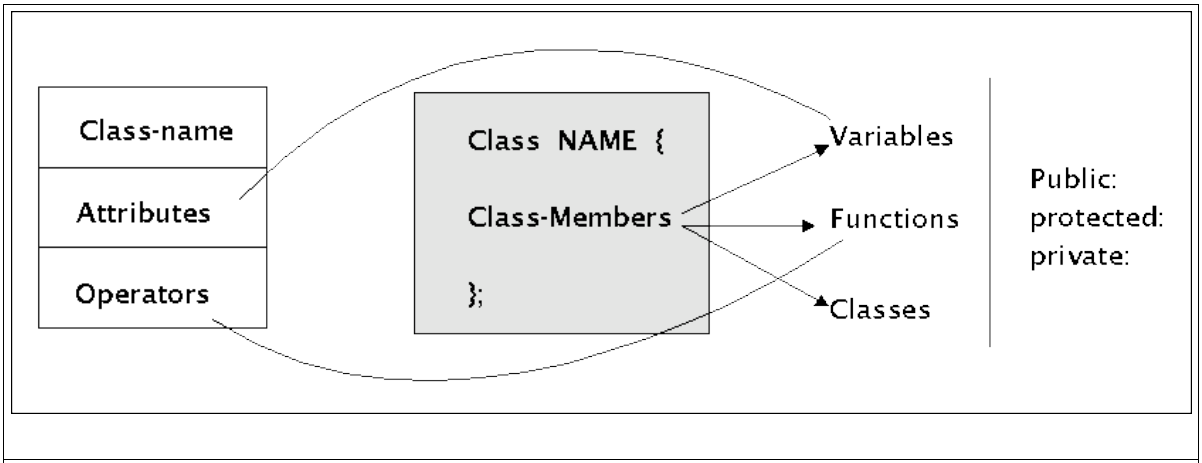

Bild 3: Abbildung UML-Klasse zu C++-Klasse

Aus dem Schaubild kann man entnehmen, dass dem Rechteck, das in UML eine Klasse repräsentiert, in C++ das struct-Gebilde 'class NAME { ... }; entspricht.

Die *UML-Attribute* werden innerhalb einer C++-Klasse als Elemente dargestellt, die wie *Variablen* deklariert werden.

Die *UML-Operationen* werden in einer C++-Klasse ebenfalls als Elemente aufgeführt, die wie *Funktionen* (und zwar normalerweise nur die Prototypen) deklariert werden.

Die *UML-Sichtbarkeitsindikatoren* '-', '#' sowie '+' werden in C++ entsprechend durch die Schlüsselwörter 'private:', 'protected:' sowie 'public:' dargestellt.

Während die Anordnung der Attribute und Operationen in einer UML-Klasse streng geregelt ist, kann man in einer C++-Klasse die Elemente (inklusive der Sichtbarkeitsindikatoren) beliebig anordnen und mischen.

Aufgrund dieser Zuordnung von UML-Klassen zu C++-Klassen kann man das ppmp2ps-PIM direkt in ein ppmp2ps-PSM übersetzen-

Als erstes die Übersetzung der UML-Klasse in eine C++-Headerdatei.

```
/**************************************************************************
 name: ppmp2ps.h

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

This file was generated on Sa Apr 16 2005 at 23:41:31
/home/gerd/public_html/fh/II-PPmP/VL/VL3/ppmp2ps.h
**************************************************************************/
#ifndef PPMP2PS_H
#define PPMP2PS_H
#include <string>

/**
 * 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 ausgewählten ASCII-Text in eine Postscript Datei
   */
  void  convert_ascii_ps ();


/**
 * Private stuff
 */
private:
  /**
   * Fields
   */
  /**
   * Enthält 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
```

Dann die Angabe einer C++-Implementierungsdatei, die zunächst nur einen Rumpf bildet der gerade die Signaturen der UML-Operatoren enthält.

```
/**************************************************************************
name: ppmp2ps.cpp
author: gerd d-h
first time: april-16, 2005
last change: april-16, 2005
intention: Read an ascii-textfile.txt and convert it into postscript textfile.ps

This file was generated on Sa Apr 16 2005 at 23:41:31
The original location of this file is
/home/gerd/public_html/fh/II-PPmP/VL/VL3/ppmp2ps.cpp
**************************************************************************/
#include "ppmp2ps.h"
#include <string>

/**
 * Methods
 */
/**
 * Wählt einen  ASCII-Text aus und lädt ihn
 */
void ppmp2ps::fileselect (string file_name) {

}
/**
 * Konvertiert einen zuvor ausgewählten ASCII-Text in eine Postscript Datei
 */
void ppmp2ps::convert_ascii_ps () {

/**
* Ende der Implementierungsdatei
*/
```

Als nächstes wird die Implementierung durch Erläuterung der C++-Plattform weiter ausgearbeitet.