Das Modulkonzept in C++

Holger Szillat

2001-05-16

1 Einführung

In der Softwaretechnik kommt dem Modulkonzept eine große Bedeutung zu. Nur mit diesem Konzept ist es möglich, sehr früh in der Softwareentwicklung zu definieren, was ein Modul können muss ohne auch nur eine Zeile implementiert zu haben. Auch das Austauschen der Implementierung zu einem späteren Zeitpunkt, beispielsweise aus Performancegründen o.ä., wird durch das Modulkonzept ermöglicht.

In diesem Essay möchte ich das Modulkonzept unter C++ betrachten und es mit dem Modulkonzept von Modula-2 vergleichen.

2 Austauschbarkeit?

Um eine möglichst gute Austauschbarkeit von Modulen zu erreichen, ist eine Trennung von Interface und Implementierung wünschenswert. In C++ ist es normalerweise üblich, das Interface in einer Header-Datei zu anzulegen und die Implementierung in einer separaten (Quell-)Datei zu schreiben.

Dadurch ist es natürlich möglich die Implementierung eines Moduls zu ändern ohne dabei das Restsystem zu beeinträchtigen.

Probleme entstehen in der Praxis allerdings meistens dadurch, dass es bei der Implementierung von Klassen (OOP) üblich ist, manche Funktionen “inline” zu schreiben. Beispiel:

```
class Foo {
private:
  int bar;

public:
  Foo( int initBar = 0 ) { bar = initBar; };
  // Oder: Foo( int initBar = 0 ) : bar( initBar ) {};
}; // Foo
```

Teilweise wird auch, wie oben gezeigt, eine noch kürzere Schreibweise verwendet. Die gilt sogar als guter Programmierstil in C++, deren Berechtigung aber dahingestellt sei.

Mitunter ist es auch notwendig, da die Verwendung von “template”s nur einen solchen Programmierstil, d.h. Interface und Implementierung in einer Datei, zulässt. Beispiel:

```
template<class T>
void swap( T& a, T& b ) {
  T tmp = a;
  a = b;
  b = tmp;
} // swap
```

Eine solche “generische” swap-Funktion ist nur dadurch möglich, dass Interface und Implementierung zusammenfallen. Nur so kann der Compiler überhaupt bei der Verwendung Code erzeugen. Es ist zwar möglich, das Interface getrennt zu spezifizieren, aber der Compiler braucht zur Compilierungs-Zeit die Definition der Funktion, zur Link-Zeit ist es zu spät.

Damit wird es natürlich auch erschwert, zu einem späteren Zeitpunkt ein Modul, das Template-Funktionen verwendet, auszutauschen ohne das Restsystem zu beeinträchtigen.

2.1 Versionkontrolle von Modulen?

In Modula-2 gibt es normalerweise eine Überprüfung, ob ein Definitionsmodul auch zu einem Implementationsmodul passt, bzw. ob die Verwendung eines übersetzten Moduls zu seiner Spezifikation passt. Damit soll es erschwert werden, dass ein Client ein Modul linken will, sich aber auf eine (möglicherweise veraltete) Spezifikation abstützt.

Unter C++ gibt es diesen Mechanismus nicht. Der Compiler generiert zwar Typinformationen in das entsprechende Linksymbol (dies ist auch notwendig, um den Mechanismus der “überladenen Funktion” zu bekommen), aber dieser Mechanismus funktioniert nicht bei benutzerdefinierten Typen.

Beispiel: Die Datei “bar.h” definiert einen benutzerdefinierten Typ namens “Bar_t”, sowie zwei Funktionen, die das entsprechende Feld aus dem Typ extrahieren.

```
#ifndef __BAR_H__
#define __BAR_H__

typedef struct {
  int i; // (1)
  char c; // (2)
} Bar_t;

extern int get_i( Bar_t bar );
extern char get_c( Bar_t bar );

#endif
```

Die Modulimplementierung “bar.cc” sieht folgendermaßen aus:

```
#include "bar.h"

int get_i( Bar_t bar ) {
  return( bar.i );
} // get_i

char get_c( Bar_t bar ) {
```

```
  return( bar.c );
} // get_c
```

Ferner gibt es einen Clienten des Moduls, namens “mainbar.cc”:

```
#include <iostream>
#include "bar.h"

using std::cout; using std::endl;

int main() {
  Bar_t somebar;

  somebar.i = 42;
  somebar.c = 'a';

  cout << "somebar.i = " << get_i( somebar ) << endl;
  cout << "somebar.c = " << get_c( somebar ) << endl;

  return( 0 );

} // main
```

Angenommen, der Programmierer des Servermoduls entscheidet sich nun die Felder in “Bar_t” in ihrer Reihenfolge zu vertauschen, also die Zeilen, die oben mit “(1)” und “(2)” markiert sind. Er kompiliert jetzt versehentlich nur die Datei “mainbar.cc” (der Client) neu, nicht aber die Datei “bar.cc” (der Server) und linkt nun das alte Objektmodul des Servers mit dem neuen Objektmodul des Clients, so funktioniert das Programm (scheinbar) trotzdem, die Ausgabe ist aber nicht “42 a”, wie vor der Veränderung, sondern nun “97 *”. Weder der Compiler noch der Linker haben den Unfall bemerkt.

Es ist allerdings anzumerken, dass ein (gutes) Make-Tool und ein korrektes Makefile die Veränderung bemerkt hätten und einen korrekten Compile- und Link-Aufruf getätigt hätten. Allerdings sind dies Tools, die weder in den Umfang der Sprache fallen, noch bei einer Compiler-Suite[1] mitgeliefert werden müssen!

Eine Versionskontrolle wird bei C++ also nur durch (externe) Tools oder Programmierhilfen ermöglicht.[2]

3 Geheimnisprinzip?

In C ist das Geheimnisprinzip nur schwach ausgelegt. Der Modulanbieter kann zwar den Standpunkt vertreten, das, was nicht dokumentiert ist, auch nicht existiert, der Client ist aber dennoch nicht davor geschützt, “versehentlich” eine Variable oder Funktion des Servers zu verwenden. Erschwert wird dieses Versehen dann noch dadurch, dass erst der Linker “meckert”, nicht aber der Compiler.

In C++ existiert dieses Problem so nicht mehr. Folgendes Beispiel soll dies verdeutlichen:

[1] “wie es auf Neuschwäbisch heißt”

[2] Fairerweise sei angemerkt, dass Modula-2 dies normalerweise auch so macht.

```
#ifndef __FOO_H__
#define __FOO_H__

namespace Foo {
  extern float pi;
  extern int fak( int n );
};
#endif
```

Das obige Programmstück repräsentiert die Exportschnittstelle “foo.hpp” des Moduls, das folgende den Modulrumpf “foo.cpp”:

```
#include "foo.hpp"

namespace Foo {
  float pi = 3.14159;
  int fak( int n ) {
    if ( n == 0 ) return 1;
    else return( n * fak( n - 1 ) );
  } // fak

  static double e = 2.72;
  int invisible() { return -1; }

}; // Foo
```

Durch Verwendung des Namensraums “Foo” werden Namenskonflikte vermieden, ähnlich wie in Modula-2.

Das Client-Modul sei:

```
#include <iostream>

#include "foo.hpp"

using std::cout; using std::endl;
using namespace Foo;

void main() {
  // Oder: std::cout << "pi = " << pi << std::endl;

  cout << "pi = " << pi << endl;
  cout << "fak(5) = " << fak(5) << endl;

  cout << "Dies tut nicht: invisible() = " << invisible() << endl;

  return;
} // main
```

Das Client-Modul benutzt zum den qualifizierten Import mit “`using std::cout;`” und “`using std::endl;`” um die Namen “cout” und “endl” aus dem Namensraum

“std” zum importieren. Ohne diese Direktiven ließen sich diese Namen auch verwenden, was aber einen “impliziten Import” darstellen würde, der so in Modula-2 nicht existiert.

Andererseits gibt es auch den unqualifizierten Import mit “`using namespace Foo;`”, was aber alle “exportierten” Namen aus “Foo” sichtbar macht und so *nicht* dem unqualifizierten Import in Modula-2 entspricht.

Das Server-Modul lässt sich problemlos übersetzen, das Client-Modul jedoch nicht; der C++-Compiler “meckert” mit:

```
mainfoo.cpp: In function ‘int main(...)’:
mainfoo.cpp:11: implicit declaration of function ‘int invisible(...)’
mainfoo.cpp:13: ‘return’ with no value, in function returning non-void
```

Es wird im übrigen auch keine Objektdatei erzeugt, die man trotzdem verwenden könnte! In C funktioniert dieses Experiment, es lässt sich dort die Funktion `invisible()` verwenden, obwohl sie nicht exportiert ist.

Es lässt sich also folgern:

- Ein Server-Modul kann seine “Leistungen”[3] über eine Exportschnittstelle zur Verfügung stellen und ein Client-Modul kann auch nur diese benutzen.
- Ein Client-Modul kann(!) über die `using`-Direktiven eine Importschnittstelle bilden. Die Betonung liegt auf “kann”, da der Client dies durch implizite Verwendung des Namensraums wie z.B. in “`int i; std::cin >> i;`” unterwandern kann. Die Importschnittstelle unterliegt also eher der Disziplin des Programmierers.
- Durch `namespace`s werden Namensräume gebildet, die Namenskonflikte wie z.B. in C verhindern.
- Das Geheimnisprinzip ist in C++ scheinbar also erfüllt.

3.1 Geheimnisprinzip doch verletzt?

In Modula-2 gibt es den Mechanismus der “opaquen Typen”. Ein entsprechender Mechanismus existiert auch in C++ oder kann entsprechend nachgebildet werden. Gewöhnlich werden aber Methoden in Klassendefinitionen “inline” programmiert, liegen also in der Exportschnittstelle des Moduls und sind somit für den Client sichtbar. Ferner wird so auch die interne Arbeitsweise oder zumindest Teile davon sichtbar.

Zudem gibt es in C++ die sog. “Templates”, die ein sehr mächtiges Werkzeug darstellen, aber von ihrer Arbeitsweise her das Geheimnisprinzip komplett verletzen.

Das Geheimnisprinzip ist in C++ also vorhanden, aber nicht so stark ausgeprägt wie in Modula-2. Es liegt zu großen Teilen in der Verantwortung des Programmierers das Geheimnisprinzip zu unterstützen.

[3] Variablen, Typdeklarationen und Funktionen

4 Ist Modula-2 schlechter als C++?

Dies ist natürlich eine sehr ketzerische Frage, gilt Modula-2 doch neben Ada als *die* Softwaretechniksprache schlechthin[4].

Aber kann ein Argument wie "Es wird Software in C++ geschrieben und nicht in Modula-2" gelten? Modula-2 hinkt den neuesten "Programmiersprachengimmicks" (wie z.B. generisches Programmieren, etc.) jedenfalls deutlich hinterher.

Ein Argument für Modula-2 ist sein Modulkonzept. Ein schmerzlich vermisstes Feature jedoch ist das Fehlen von "Templates" wie, sie in C++ oder Ada existieren. Ein so generisches Modul wie eine Stackimplementierung für jeden Datentyp quasi neu zu schreiben, ist fehleranfällig und zeitaufwendig. Wesentlich eleganter ist dies in C++ gelöst, auch wenn Templates missbraucht werden können, um schwer durchschaubaren Code zu schreiben.

5 Schlussfolgerung

Vom Standpunkt des Modulkonzepts ist Modula-2 sicherer zu verwenden als C++. Das versehentliche Verwenden von Moduldeklarationen und "veralteten" Implementierungen wird besser aufgefangen als bei C++, wo erst zur Link-Zeit Alarm geschlagen wird.

Dennoch ist das Fehlen von Template-Parametern in Modula-2 ein großer Nachteil, da damit das Schreiben von Bibliotheken zur Verwendung auch von benutzerdefinierten Typen praktisch unmöglich ist.

Letztendlich hat C++ auch über die vergangenen Jahre sehr auf Modula-2 aufgeholt, es vielleicht sogar schon überholt. C++ kann also durchaus als Softwaretechniksprache gelten.

[4] Warum sind freie Modula-2-Compiler trotzdem praktisch nicht zu bekommen? Der "Gardens-Point Modula-2"-Compiler für Linux ist in der Sep96-Ausgabe für jede neuere Linux-Distribution unbrauchbar, da sie eine veraltete libc-Bibliothek braucht.