Erfahrungen beim Einsatz von Klassenbibliotheken für C++

Dipl.-Wirtsch.-Inf. Stefan Kuhlins
Universität Mannheim
Lehrstuhl Wirtschaftsinformatik III
Schloß
68131 Mannheim
http://www.wifo3.uni-mannheim.de/~kuhlins/

Zusammenfassung

Im Rahmen eines C++-Projekts, bei dem ein objektorientiertes CASE-Tool für unterschiedliche Systeme zu entwickeln war, wurden die drei Klassenbibliotheken BIDS (Borland International Data Structures), Tools.h++ von Rogue Wave und die STL (Standard Template Library) eingesetzt. Anhand eines Implementationsmusters werden der Einsatz der drei Bibliotheken und die dabei auftauchenden Probleme demonstriert. Abschließend werden einige allgemein gültige Ratschläge zur Auswahl einer C++-Klassenbibliothek gegeben.

1 Einleitung

Am Lehrstuhl für Wirtschaftsinformatik III der Universität Mannheim wird das sogenannte *MAOOAM*Tool* entwickelt, das die „Mannheimer objektorientierte Analysemethode“ unterstützt [5]. Die Implementation erfolgt mit der Programmiersprache C++.

In den von Benutzern des Tools angelegten Modellen sind u.a. Klassen mit Attributen und Methoden zu verwalten. Faßt man Attribute und Methoden als spezielle Elemente einer Klasse auf, so ergibt sich in der Notation von Coad/Yourdon [2] die links in Abbildung 1 dargestellte Struktur. Als Implementationsmuster („Pattern“) ist die gleiche Struktur rechts abstrakt gezeigt. Eine Klasse A setzt sich demnach aus beliebig vielen B-Objekten zusammen. Die Klasse B ist eine abstrakte Basisklasse, von der die speziellen Klassen A1 und A2 abgeleitet sind.

Zur Implementation einer solchen Struktur in C++ kann z.B. eine doppelt verkettete Liste eingesetzt werden, die Zeiger auf die Basisklasse B verwaltet. (Die Auswahl einer geeigneten Containerklasse hängt im konkreten Fall vom geplanten Einsatz ab. Für die hier verfolgten Zwecke bietet sich eine doppelt verkettete Liste aufgrund ihres einfachen Aufbaus an.)

Als Basisklasse sollte B über einen virtuellen Destruktor verfügen [4]. Die rein virtuelle Elementfunktion vf() repräsentiert eine typische Operation, die das Objekt modifiziert. Die Operatoren == und < werden zur Verwaltung in einer Containerklasse benötigt. Damit die Objekte ausgegeben werden können, wird der Ausgabeoperator definiert. Weitere Funktionen werden hier nicht benötigt.

```
class B {
public:
   virtual ~B();
   virtual void vf() = 0;
   friend bool operator==(const B&, const B&);
   friend bool operator<(const B&, const B&);
   friend ostream& operator<<(ostream&, const B&);
   /* ... */ };
```

In: Dieter Pötschke und Mathias Weber (Hrsg.): Anwendungen für Kommunikations-Highways, Konferenzband zur INFO’96, 8–9. November 1996 in Potsdam, R. v. Decker’s Verlag, 1997, S. 351–356

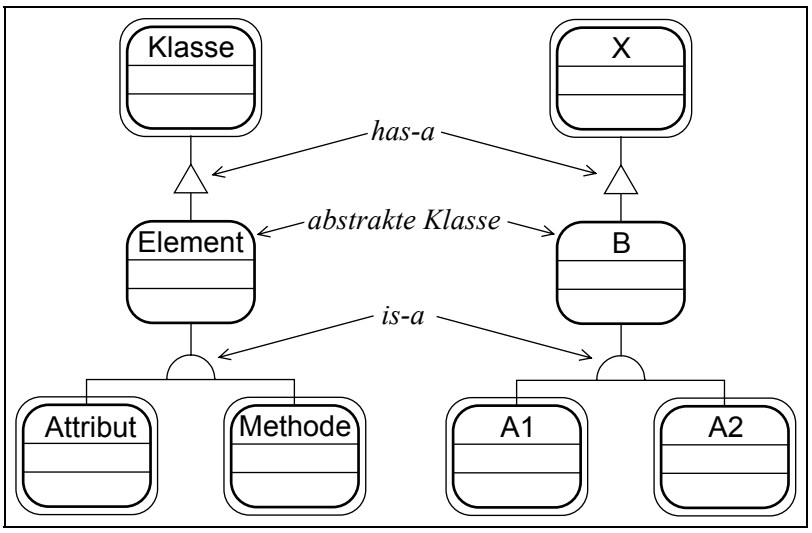

Abbildung 1: Das zu implementierende Modell

```
class A1 : public B {       // Klasse A2 ähnlich
   int i;
public:
   A1(int);
   void vf();
   /* ... */ };
```

Im weiteren Verlauf wird das Beispiel mit jeder der Klassenbibliotheken implementiert. Dabei sollen für die Klasse X die folgenden Elementfunktionen definiert werden.

```
class X {
   Liste b;
public:
   void einfuegen();
   void ausgeben() const;
   void suchen() const;
   ~X(); };
```

Die Elementfunktionen enthalten Operationen, die auch in realen Programmen benötigt werden.

2 BIDS

Die „Borland International Data Structures“ werden zusammen mit dem C++-Compiler von Borland ausgeliefert [1]. Dies war auch der Grund für den anfänglichen Einsatz in unserem Projekt. Wenn schon eine Klassenbibliothek dabei ist, warum dann eine andere zusätzlich erwerben?

Die Containerklasse TIDoubleListImp ist eine doppelt verkettete Liste, die, wie am I für „indirekt“ zu erkennen ist, Zeiger verwaltet. Ein eventueller Wechsel zu einer anderen Containerklasse wird erleichtert, wenn mittels typedef zwei Namen für die Liste und den zugehörigen Iterator vereinbart werden; außerdem spart man so bei länglichen Klassennamen Tipparbeit.

```
#include<classlib/dlistimp.h>
class X {
   typedef TIDoubleListImp<B> Liste;
   typedef TIDoubleListIteratorImp<B> Iterator;
   /* ... */ };
```

Zum Einfügen von neuen Listenelementen kann die Elementfunktion Add() eingesetzt werden.

```
void X::einfuegen() { b.Add(new A1(1)); }
```

Zur Ausgabe der Listenelemente gibt es zwei mögliche Verfahren: per Iterator oder Callback-Funktion. Mit einem Iterator gestaltet sich die Ausgabe folgendermaßen:

```
void X::ausgeben() const { for (Iterator it(b); it; it++) cout << *it.Current() << endl; }
```

Dabei setzt der Konstruktor den Iterator auf das erste Listenelement. Die Prüfung, ob der Iterator auf ein gültiges Element zeigt, erfolgt mit Hilfe der Konversionsfunktion operator int(). Zum nächsten Element gelangt man mit dem Inkrementoperator ++. Die Elementfunktion Current() liefert das Listenelement, auf das der Iterator gerade zeigt.

Hier zeigen sich Mängel in der Implementation der Iteratorklasse. Da der Borland C++-Compiler den logischen Datentyp bool unterstützt, sollte nach bool und nicht int konvertiert werden. Außerdem sind die Konversionsfunktion und Current() nicht const, was der Borland Compiler allerdings lediglich mit einer Warnung rügt, obwohl es gemäß C++-Standard ein Fehler ist.

```
void test(const Iterator& i) { if (i) cout << *i.Current(); }
```

Wenn man eine Callback-Funktion wie z.B. print() definiert, die ein Listenelement ausgibt, können mit der Elementfunktion ForEach() alle Listenelemente ausgegeben werden.

```
void print(B& b, void*) { cout << b << endl; }
void X::ausgeben() const { b.ForEach(print, 0); }
```

Der Aufruf von ForEach() wird bemängelt, weil die Funktion nicht const ist. Außerdem kann das erste Argument der Callback-Funktion print() nicht als const B& deklariert werden. Solche Probleme ließen sich durch Überladen von ForEach() mit einer konstanten Version vermeiden.

Objekte können mittels Find() gesucht werden, wobei der Vergleichsoperator == der Klasse B benutzt wird. Auch diese Elementfunktion der Listenklasse ist nicht const.

```
void X::suchen() const {
   B* z = b.Find(&A1(1));
   if (z != 0) cout << "gefunden: " << *z << endl; }
```

In einer konstanten Elementfunktion der Klasse X können B-Objekte lediglich lesend bearbeitet werden, da const auch als „read-only“ interpretierbar ist. Um so überraschender ist, daß bei Borlands Implementation der Aufruf einer nicht konstanten Elementfunktion der Klasse B möglich ist.

```
void X::lesendBearbeiten() const { for (Iterator it(b); it; it++) it.Current()->vf(); }
```

Auch hier ist mangelnde const-Korrektheit die Ursache. Der Konstruktor der Iteratorklasse erwartet als Argument eine konstante Liste, die Elementfunktion Current() liefert aber einen Zeiger auf ein nicht konstantes Objekt. Um diese Schwachstelle zu umgehen, ist eine zweite Iteratorklasse notwendig, so daß ein Iterator für konstante und einer für nicht konstante Listen zur Verfügung steht.

Der Destruktor der Klasse X soll die eingefügten Objekte wieder löschen. Dazu ruft er die Elementfunktion Flush() auf, über deren Argument gesteuert wird, ob nur die Listenelemente oder auch die über die Zeiger erreichbaren Objekte gelöscht werden.

```
X::~X() { b.Flush(1); }        // löscht Listenelemente und B-Objekte
```

Die BIDS wurden aufgrund der angesprochenen Mängel und der eingeschränkten Portierbarkeit auf andere Betriebssysteme und Compiler in unserem Projekt von der Bibliothek Tools.h++ abgelöst.

3 Tools.h++

Die Klassenbibliothek Tools.h++ der Firma *Rogue Wave* ist nahezu für alle relevanten C++-Compiler und Betriebssysteme verfügbar [3].

```
#include <rw/tpdlist.h>
class X {
   typedef RWTPtrDlist<B> Liste;
   typedef RWTPtrDlistIterator<B> Iterator;
   /* ... */ };
```

Das Einfügen neuer Listenelemente kann mit `b.append(new A1(1));` realisiert werden.

Für die Ausgabe aller Listenelemente steht wieder ein Iterator zur Verfügung. Dieser erwartet jedoch eine nicht konstante Liste, was abermals zu Problemen führt. Auch Tools.h++ fehlt somit eine zweite Iteratorklasse. Eine Eigenheit der Iteratorklasse ist, daß der Iterator direkt nach der Initialisierung durch den Konstruktor noch nicht benutzt werden darf. Statt dessen muß er erst durch einen Aufruf von `operator()` auf das erste Element bewegt werden. Der Rückgabewert signalisiert, ob der Iterator gültig ist. Mit `key()` wird auf das aktuelle Element zugegriffen.

```
for (Iterator it(const_cast<Liste&>(b)); it(); ) cout << *it.key() << endl;
```

Die gleiche Aufgabe kann mittels Callback-Funktion gelöst werden. Wobei auch hier wie bei Borland das Fehlen einer konstanten Version der Elementfunktion `apply()` zu bemängeln ist.

```
void print(B* b, void*) { cout << *b << endl; }

   const_cast<Liste&>(b).apply(print, 0);
```

Die Funktion zum Suchen von Objekten in der Liste heißt `find()`. Sie ist `const`, liefert aber einen Zeiger auf das gefundene Objekt, über den es modifiziert werden kann.

```
B* z = b.find(&A1(1));
if (z != rwnil) cout << "gefunden: " << *z << endl;
z->vf();    // Oops!
```

Mit `b.clearAndDestroy();` werden alle Listenelemente und die zugehörigen `B`-Objekte zerstört.

Wir haben in unserem Projekt Tools.h++ u.a. deshalb durch die STL ersetzt, weil letztere in frei verfügbaren Versionen zugänglich ist und so keine Kosten anfallen.

4 STL

Die „Standard Template Library“ ist Teil des aktuellen C++-Standards und gehört damit ebenso zur Sprache wie z.B. die Stream-Bibliothek [6]. Es erscheint deshalb aufgrund der zu erwartenden Verbreitung vorteilhaft, die STL für alle Aufgaben zu nutzen, die mit ihrer Hilfe lösbar sind.

Im Gegensatz zu den beiden anderen Bibliotheken bietet die STL keine speziellen Container für Zeiger an. Instanziert man einen Container einfach mit `T*` statt `T`, werden zwar prinzipiell Zeiger verwaltet, jedoch arbeiten einige Operationen nicht wie gewünscht. Beispielsweise werden beim

Suchen von Objekten Zeigeradressen verglichen, so daß zwar identische Objekte gefunden werden, nicht aber welche mit gleichen Werten. Solche Probleme sind dadurch lösbar, daß ein „Smart-Pointer“ benutzt wird, der sich bis auf die Vergleichsoperatoren wie ein „normaler“ Zeiger verhält.

```
template<class T> class SmartPointer {
   T* p;
public:
   explicit SmartPointer(T* t = 0) : p(t) { }
   T& operator*() { return *p; }
   const T& operator*() const { return *p; }
   T* operator->() { return p; }
   const T* operator->() const { return p; }
   T* get() { return p; }
   const T* get() const { return p; }
   bool operator<(const SmartPointer& r) const { return *p < *r.p; }
   bool operator==(const SmartPointer& r) const { return *p == *r.p; } };
```

Die Listenklasse der STL heißt list und ist im namespace std definiert.

```
#include<list>
using namespace std;
class X {
   typedef list<SmartPointer<B> > Liste;
   /* ... */ };
```

Im Gegensatz zu den beiden anderen Bibliotheken stellt die STL zwei Iteratorklassen für die Liste zur Verfügung, auf die mittels Liste::iterator und Liste::const_iterator zugegriffen wird, so daß zusätzliche typedef-Deklarationen unnötig sind. Durch die beiden Iteratorklassen sind die oben beschriebenen Probleme bezüglich der const-Korrektheit gelöst.

Neue Elemente können mit b.push_front(SmartPointer<B>(new A1(1))); eingefügt werden. Die Ausgabe aller Listenelemente läßt sich mit Hilfe eines konstanten Iterators bewerkstelligen.

```
for (Liste::const_iterator it(b.begin()); it != b.end(); ++it) cout << **it << endl;
```

Eine Alternative dazu stellt die Funktion for_each() dar. Sie ist in der Header-Datei algorithm definiert und erwartet als drittes Argument eine Funktion oder ein Funktionsobjekt. Letzteres ist eine Klasse, für die der Funktionsaufrufoperator definiert ist. Gegenüber einem Funktionszeiger hat dies den Vorteil, daß der Compiler den Aufruf inline realisieren kann.

```
template<class T> struct Print {
   void operator()(const SmartPointer<T>& s) const { cout << *s << endl; } };

   for_each(b.begin(), b.end(), Print<B>());
```

Zum Suchen eines Listenelements kann die Funktion find() verwendet werden. Für ein konstantes Listenobjekt liefert sie einen konstanten Iterator zurück.

```
Liste::const_iterator z = find(b.begin(), b.end(), SmartPointer<B>(&A1(1)));
if (z != b.end()) cout << "gefunden: " << **z << endl;
```

Das Löschen der von den Listenelementen referenzierten B-Objekte erledigt die folgende Schleife:

```
for (Liste::iterator it(b.begin()); it != b.end(); ++it) delete (*it).get();
```

Ein einfaches b.erase(b.begin(), b.end()); löscht dagegen lediglich die Listenelemente und wird vom Destruktor der Liste ausgeführt.

Ein Kennzeichen der STL ist die Trennung von Algorithmen und Datenstrukturen, wodurch einerseits eine maximale Wiederverwendung erreicht wird, andererseits aber auch der „objektorientierte Gedanke“ im Sinne der Einheit von Daten und Operationen verletzt wird. Ein weiteres Kennzeichen ist der massive Gebrauch von Templates, weshalb mancher C++-Compiler damit noch überfordert ist. Es dürfte aber nur eine Frage der Zeit sein, bis alle Compilerhersteller nachgezogen haben.

5 Ratschläge

Vor dem Kauf einer Bibliothek sollten zunächst die Rahmenbedingungen wie Betriebsystem(e), Compiler usw. ausgelotet werden. Dies ist aber gerade bei Projekten mit langer Laufzeit schwierig, weil sich die Anforderungen im Zeitablauf ändern können. Deshalb ist Flexibilität anzustreben.

Um zumindest theoretisch die Möglichkeit zu haben, auftretende Unzulänglichkeiten einer Klassenbibliothek beheben zu können, sollten Klassenbibliotheken mit Sourcecode erworben werden. Allerdings sollten Änderungen am Sourcecode die Ausnahme sein, weil bei Erscheinen der nächsten offiziellen Version solche Korrekturen u.U. wieder vorzunehmen wären. Statt dessen ist es besser, Fehler direkt an den Hersteller zu melden, damit dieser entsprechende Korrekturen durchführt.

Der Umstieg von einer Bibliothek zu einer anderen ist i.d.R. aufwendig, weil einfaches „Suchen und Ersetzen“ aufgrund unterschiedlicher Parameterlisten oft nicht zum Erfolg führt. Selbst wenn das Programm anschließend wieder fehlerfrei übersetzbar ist, können noch Laufzeitfehler versteckt sein.

Unabhängigkeit bezüglich der verwendeten Bibliothek läßt sich durch eine selbst definierte Klassenschicht zwischen den Bibliotheksklassen und dem Anwendungsprogramm erreichen. Dabei werden Funktionsaufrufe mittels inline Funktionen durchgereicht. Als Nachteil fällt die dadurch entstehende proprietäre Schnittstelle ins Gewicht.

Für eine Bibliothek spricht auch eine große Verbreitung und ein damit verbundenes Angebot an Know-how auf dem Arbeitsmarkt und in der Literatur.

Die kompletten Listings können beim Autor angefordert werden.

Literatur

[1] Borland International: Referenzhandbuch, Scotts Valley, CA, 1996

[2] Coad, P. und E. Yourdon: Object-Oriented Analysis (2nd ed.), Prentice Hall, 1991

[3] Rogue Wave: Tools.h++, User’s Guide, Rogue Wave Software, Corvallis, Oregon, 1996

[4] Schader, M. und S. Kuhlins: Programmieren in C++ – Einführung in den Sprachstandard C++, 4. Aufl., Springer-Verlag, 1996

[5] Schader, M. und M. Rundshagen: Objektorientierte Systemanalyse, 2. Aufl., Springer-Verlag, 1996

[6] Working Paper for Draft Proposed International Standard for Information Systems – Programming Language C++, X3J16/95-0087, WG21/N0687, 1996