Objektorientierte Programmierung für Dummies - Errata

Stand: 28.12.2003

Leider ist es – wie schon vielen Autoren vor mir – auch diesmal wieder nicht gelungen, ein Buch zu 100% frei von Fehlern zu halten. Zu den obligatorischen Druckfehlern schleichen sich doch immer wieder Ungenauigkeiten und kleine Unsauberkeiten ein.

Ich bitte vielmals um Entschuldigung.

Fehler und Irrtümer im Buch

Seite 7

Unter den Comics stehen Seitenzahlen, aber die Comics der Seiten 379 und 425 sind vertauscht.

Seite 66

Dort steht, dass die `DummiesTools` im Anhang abgedruckt sind. Aus Platzgründen wurde der Anhang aber noch aus dem Buch entfernt und existiert nur noch online, folglich befinden sich die `DummiesTools` nun nur noch in den downladbaren Programmdateien, zu finden unter

`http://www.mitp.de/dummies/2984/2984.htm "Beispiele aus dem Buch"`

Seite 88

```
AA objectAA(-1);
```

und

```
AA objectAA = -1
```

machen nicht exakt das Gleiche.

Ersteres ist Direct-Initialization. Letzteres Copy-Initialization. Letzteres ruft den Copy-Konstruktor auf. Dieser Aufruf kann zwar eliminiert werden (dann ist es das gleiche), erfordert aber immer, dass der Copy-Konstruktor von `AA` aufrufbar wäre.

Ein schönes Beispiel zum Verdeutlichen:

```
class Foo
{
   public:
      Foo(int i) {}
   private:
      Foo(const Foo&);
};
int main()
{
    Foo obj(1);     // ok;
    Foo obj2 = 1;   // <-- ERROR.
}
```

Der Comeau C/C++ Compiler sagt dazu:

```
error:  "Foo::Foo(const Foo &)" is inaccessible
        (Even though the copy was eliminated, the
         standard still requires it to be accessible)
```

Danke an HumeSikkins alias Benjamin Kaufmann für den Hinweis.

Seite 104

Dort steht

"Ein virtueller Destruktor kann niemals pure virtual sein, eine Defintion wie `virtual ~Vehicle() = 0` ist nicht erlaubt"

Das ist falsch!

Ein pure virtual Destruktor ist nicht nur erlaubt, sondern häufig auch eine gute Sache. Richtig ist, dass ein Destruktor IMMER eine Definition haben muss, selbst wenn er pure virtual ist.

Das ganze sieht dann so aus:

```
class Foo
{
   public:
```

```
      /* pure */ virtual ~Foo() = 0;
};
// Definition
Foo::~Foo
{}
```

Damit kann man erzwingen, dass die Klasse auf jeden Fall abstrakt ist. Trotzdem wird der Destruktor der Basisklasse definiert und ist aufrufbar im Rahmen der Zerstörung der gesamten Objekthierarchie.

Danke an HumeSikkins alias Benjamin Kaufmann für den Hinweis.

Seite 157

Dort steht, dass der `op=` einer abgeleiteten Klasse den `op=` der Basisklasse aufrufen soll. Verwendet man aber das kurz vorher eingeführte *Create-Temporary-And-Swap-Prinzip*, so ist das unnötig - der Copy-Konstruktor erledigt dies ja bereits.

Die Aussage bleibt dann gültig, wenn man den Zuweisungsoperator `op=` von Hand ausprogrammiert und alle Elemente einzeln kopiert, in diesem Fall muss die Zuweisung der Objekte der Basisklassen explizit aufgerufen werden.

Danke an HumeSikkins alias Benjamin Kaufmann für den Hinweis.

Seite 160

Mitte der Seite steht:

Die Zeile

```
a = (b < c) ? b : c;
```

ist identisch zu

```
if(b < c)
   a = b;
else
   a = b;
```

Das ist natürlich falsch, richtig muss dies lauten

ist identisch zu

```
if(b<c)
    a=b;
```

```
else
    a=c;
```

Seite 160

In der Aufzählung für "Operatoren die nicht überladen werden können" fehlt der Operator `->*`.

Seite 227f

Dort wird aus den Funktionen getX1 und getString ein const auto_ptr zurück gegeben. Der Rückgabewert von getX1 wird später einem auto_ptr zugewiesen. Das ist in Standard-C++ nicht erlaubt, leider erlaubt der Microsoft Visual C++ 6 diese Konstruktion. Der Visual C++ 7 meckert übrigens an dieser Stelle, genau wie der Borland C++ Builder und die ganzen GNU-Compiler.

Ein const auto_ptr ist ein auto_ptr der niemals einen ownership Transfer durchführen kann. Er ist also immer das Ende einer Kette. Korrekterweise müssten beide Funktionen jeweils einen auto_ptr (ohne const) liefern. Leider gibt es bisher nur wenig Compiler die diesen Fehler bemerken.

Seite 310

Der Grundtyp 2 logarithmische Laufzeit ist sehr schlecht beschrieben, logarithmische Laufzeit heißt nicht, dass bei Vervierfachung der Problemgröße sich der Aufwand nur verdoppelt (das wäre vielleicht O(x^(1/2))), sondern dass eine Multiplikation der Problemgröße mit einem Faktor zu einer additiven Vergrößerung des Aufwandes führt, z.B. beim Suchen in einem binären Baum: Verdoppelt sich die Größe, kommt eine Ebene dazu, also nur ein Schritt im Algorithmus. Nochmal verdoppeln -> noch ein Schritt usw.

Seite 351

In der Themenübersicht wird versprochen, dass der Container `map<Key, T>` vorgestellt wird. Aus Platzgründen wurde dieser Teil noch kurzfristig gekürzt.

Hier nun der fehlende Teil als PDF

`http://www.c-plusplus.de/oop_dummies/kap19-reste.pdf`

Ach ja, außer der `map<Key, T>` taucht nun auch noch der `stack<T>` auf, den hat irgendwie keiner vermisst.

Seite 351

Ein sozusagen klassischer Fehler, hier steht "Die Deque ist eine Schlange mit zwei Köpfen, ähnlich der berühmten Medusa". Ein aufmerksamer Leser hat mich darauf hingewiesen, dass nicht bei Medusa, sondern der Hydra die Köpfe nachwachsen... peinlich. Zur Aufklärung hier die volle Beschreibung aus der zugehörigen griechischen Mythologie:

Medusa, (griech. „Herrin”) In der griechischen Mythologie ein Ungeheuer. Sie ist eine Tochter des Phorkys und der Ketos. Medusa war eine der drei Gorgonen. Obwohl anders als Stheno und Euryale sterblich, galt sie für die furchtbarste der drei Schwestern. Jeder, der Medusas Haupt erblickte, wurde auf der Stelle zu Stein. Als die Medusa gerade von Poseidon schwanger war, schlug Perseus ihr das Haupt ab. Aber ihr Kind, Pegasos, kam dennoch zur Welt. Das Haupt bekam die Göttin Athene, die es an ihrem Aigis trug. Steinerne Medusenhäupter finden sich an Gebäuden, um Unheil abzuwehren.

Hydra, auch als Lernaea hydra oder Lernäische Schlange bekannt ist die Hydra (griech. „Wasserschlange”), ein siebenköpfiges Ungeheuer, dessen Giftatem alles vernichtete. Sie trieb im argolischen See bei Lerna ihr Unwesen. Sie ist Tochter von Echidna und Typhon. Ihre Geschwister sind Chimaira, Kerberos und Sphinx. Als Eltern der Hydra werden auch die Styx, eine Tochter des Erebus und der Nyx sowie ein sonst unbekannter Piras genannt. Die zweite Aufgabe des Herakles war es, dieses Ungeheuer zu vernichten, was allerdings erhebliche Schwierigkeiten mit sich brachte. Denn immer, wenn ihr ein Kopf abgeschlagen wurde, wuchsen an der Stelle zwei neue nach. Daher heißt sie auch Excetra (von lat. excrescendo, „auswachsen”). Erst als Herakles Gefährte Iolaos die Stümpfe mit einem brennenden Holzscheit die Wunden ausglühte, fand die Hydra ihr Ende. Erschwerend kam hinzu, daß mit ihr ein gewaltiger Krebs, der Cancer, aus dem See auftauchte, der den Herakles empfindlich mit den Scheren knippte und daher zuerst erschlagen werden mußte. Weil einer der Köpfe der Hydra unsterblich war, vergrub Herakles diesen tief in der Erde und sicherte das Loch zusätzlich mit schwerem Geröll. In die Wunden der Schlange, bzw. ihre Galle, tunkte er seine Pfeile, deren Spitzen dadurch tödlich giftig waren. Seitdem leuchten Hydra und Cancer als Sternbilder vom Himmel.

Seite 411

Aus Kapitel 23 wurde aus Platzgründen ein Abschnitt über das State-Pattern gekürzt:

`http://www.c-plusplus.de/oop_dummies/kap23-reste.pdf`

Seite 412

Im ersten Absatz steht "Sie sollten eigentlich viel häufiger `auto_ptr` verwenden", zwei Absätze tiefer "`auto_ptr` ist der einzige Smart-Pointer, der derzeit in der Standardbibliothek integriert ist. Aus diesem Grund wird er häufig zu oft eingesetzt."

Dies ist unglücklich (da hier widersprüchlich) formuliert, der zweite Absatz würde so deutlicher:

`auto_ptr` sind nicht die einzige Möglichkeit für Smart Pointer, aber die einzige in der Standardbibliothek. In den Boost-Libraries gibt's zum Beispiel noch einige Varianten von Smart Pointern. Das bezieht sich auch darauf, dass es

keine Smart Pointer für Arrays (also z.B. `new int[50]` o.ä.) gibt, oder dass man sich auch noch andere Formen der automatischen Zeigerverwaltung denken kann.

Seite 417

"Maps wurden im Kapitel *Mehr Container und Adapter* ausführlich behandelt"

Wie schon weiter oben geschrieben, werden Maps überhaupt nicht (mehr) behandelt.

Seite 422

Im unteren Quellcode in Listing 23.14 steht gleich zu Beginn die Funktion `register_all()` – diese ist aber überflüssig und wird nicht verwendet, in den ladbaren Quelltexten ist sie gar nicht vorhanden.

Hintergrund ist der, dass die Idee zunächst war die Instanzen der `Producer`-Objekte als lokale statische Objekte in einer Funktion anzulegen, aber danach hatte ich mich dazu entschieden diese doch als ganz gewöhnliche private Instanzen in der Klasse `RegisterViews` zu erzeugen. Im Code wurde es geändert, aber das Listing im Buch wurde nicht mehr angepasst.

Seite 423

Im Absatz *Die so genannte RTTI…* heißt es im zweiten Satz:

Also Ergebnis von `typeid`…

Da ist ein "o" zu viel, richtig ist

Als Ergebnis von `typeid`…

Seite 424

In Absatz 4 heißt es

Getrennt implementieren, an einer zentralen Stelle implementieren, das war's.

Sollte eigentlich lauten

Getrennt definieren, an einer zentralen Stelle implementieren, das war's.

Seite 427

In der Aufzählung "In diesem Kapitel" steht *Lladen* statt *Laden*

Seite 427

Im Kapitel 24 über Persistenz ("Speicherung") von Daten ist der interessante Teil herausgefallen, wie man Zeiger in einer Datei über Objekt-IDs abbilden kann:

`http://www.c-plusplus.de/oop_dummies/kap24-reste.pdf`

Seite 443

Eine Leserfrage:

"Wenn ich das Gras-Schaf-Wolf-Beispiel starte, endet das Programm immer sofort mit einem "abnormal program termination", alles was es davor ausgibt ist Datum 0". Der Compiler: Microsoft Visual C++ 6.

Kein Programmfehler, sondern eine kleine Ungenauigkeit – für die Lauffähigkeit des Programms muss man in den Projektoptionen die Option "RTTI" einschalten. Dies wurde zwar in dem ähnlichen Beispiel auf Seite 421 erklärt, aber man kann es leicht vergessen.

Bonuskapitel (nur im Web) B1

Seite 2

Unten auf der Seite steht

Als Template-Parameter sind auch nicht-generische Parameter zugelassen. Hierbei handelt es sich um Konstanten, die bereits zur Laufzeit bekannt sind."

richtig wäre hier *Compilezeit*, darum geht es hier ja gerade, dass Ausdrücke während der Compilierung bereits vor der Laufzeit ausgewertet werden.

Seite 12

Dort steht als letztes auf der Seite

0 enie netsnosna ,tsi tetieleg `B` nov `A` nnew , `1` enie trefeil `devireD::<B ,A>kcehC`

Im Klartext: `check<A, B>::Derived` liefert eine `1`, wenn `A` von `B` geleitet ist, ansonsten eine `0`.

Das soll natürlich eigentlich abgeleitet lauten, folglich muss das tetieleg durch tetielegba ersetzt werden.