Stephan Brumme, SST, 3.FS, Matrikelnr. 70 25 44
mail@stephan-brumme.com

Zum besseren Verständnis sollte man sich meine Lösung zur Aufgabe O1.2 durchlesen.

Virtuelle Methoden

Die Aufgabenstellung verlangte eine Umsetzung von `CCashOffice` und `CRoom` mit Hilfe von virtuellen Methoden. Eine wesentliche Eigenschaft, die unbedingt vorausgesetzt werden muss, ist die Signaturgleichheit. Eine Stelle im Programm, an der dies notwendig war, ist die Vergleichsoperation. Der Parameter war eine Instanz der gleichen Klasse. Um jetzt aber die Signatur unverändert zu lassen, muss der Parameter in allen abgeleiteten Klassen stets vom gleichen Typ. Dies widerspricht aber dem Konzept, dass abgeleitete Klassen die Basisklassen erweitern und u.a. auch neue Attribute hinzufügen. Bei einem Vergleich müssen auch sie in Betracht gezogen werden.
Ein Ausweg aus dieser Situation sind Zeiger. Es ist eine oft sinnvolle und hilfreiche Eigenschaft von C++, dass man einer Variablen, die einen Zeiger auf eine Klasse darstellt, auch von dieser Klassen abgeleitete Klassen zuweisen kann. Folgender Code stellt eine korrekte Sequenz dar:

```
CCashOffice cash1, cash2;
CRoom *pcash1, *pcash2;

pcash1 = &cash1;
pcash2 = &cash2;
```

Genau diesen Trick kann man jetzt bei der Vergleichsoperation benutzen: man definiert den Parameter als Zeiger auf die Basisklasse:

```
bool CRoom::EqualValue(CRoom *room)
{
	if (room != NULL)
		return (m_nArea == room->m_nArea &&
			m_nNumberOfRoom == room->m_nNumberOfRoom);
	else
		return false;
}
```

Wenn diese Methode in der Headerdatei als `virtual` deklariert ist, so kann sie problemlos überladen werden:

```
bool CCashOffice::EqualValue(CRoom *room)
{
	if (room != NULL)
	{
		CCashOffice *cashoffice = reinterpret_cast<CCashOffice*> (*room);
		return (CRoom::EqualValue(room) &&
			m_nNumberOfCounter == cashoffice->m_nNumberOfCounter);
	}
	else
		return false;
}
```

Doch mit Zeigern handelt man sich ein ziemlich großes Laufzeitproblem ein: man muss zur Laufzeit stets überprüfen, ob sie auf einen gültigen Speicherbereich zeigen. In der Theorie sollte ein Vergleich auf `NULL` ausreichen, in der Praxis hat man allerdings oft mit ziemlich vielen wildgewordenen Zeigern zu tun. Unvorsichtiges Vorgehen führt unter Windows dann schnell zu einer Allgemeinen Schutzverletzung.
Die nächste Schwachstelle besteht darin, dass ich `CCashOffice::EqualValue` mit einem `CRoom*` als Parameter aufrufen darf (laut Parameterliste ist das okay, der Compiler hindert mich nicht daran). Wenn der Code dann aber zur Laufzeit eine Typumwandlung in `CCashOffice*` vornimmt, gibt es wieder einen bösen Absturz. Solche Fehler sind i.d.R. nur schwer zu entdecken.

Nun habe ich viele Fallen und Hindernisse bei der Verwendung von Zeigern aufgezählt. Warum sollte man sie trotzdem benutzen ? Meine bisherige Implementation hat doch `CRoom::EqualValue` und `CCashOffice::EqualValue` derart angeboten, dass größtmögliche Typsicherheit besteht !
Man stelle sich folgenden Fall vor, der das kleine Codestück vom Beginn dieses Textes wiederholt:

```
CCashOffice cash1, cash2;
CRoom *pcash1, *pcash2;

pcash1 = &cash1;
pcash2 = &cash2;

// now do some stuff with cash1/cash2

cout << pcash1->EqualValue(pcash2);
```

Was passiert ? `pcash1` und `pcash2` sind vom Typ `CRoom*`. Der Compiler sieht in der Virtuelle-Methoden-Tabelle von `pcash1` nach und holt sich die Adresse der virtuellen Methode `EqualValue`. Diese ist die Implementation von `CCashOffice`, es wird ein korrekter Vergleich von zwei CashOffices durchgeführt.

Und wie hätte mein ursprünglicher Code reagiert ? Er hätte nur den Datentyp `CRoom*` gesehen und `EqualValue` eines `CRoom` aufgerufen. Dieser führt nur einen Vergleich auf `CRoom` -Basis durch und unterschlägt das neue Attribut `m_nNumberOfCounter`. So könnten zwei CashOffices, die die gleiche Fläche und Raumnummer haben als gleich angesehen werden, obwohl sie sich in der CashOffice-Nummer unterscheiden.

Vergleich beider Vorgehensweisen

In meinen Augen ist es wesentlich sinnvoller, ein Programm so zu designen, dass möglichst selten Zeiger benutzt werden. Zeiger haben eindeutige Vorteile, doch ihre Nachteile sind ebenso unbestritten.
Die ganze Problematik ist schließlich nur entstanden, weil ich es erlauben will, diese Zeilen ordentlich funktionieren sollen:

```
CCashOffice cash1, cash2;
CRoom *pcash1, *pcash2;

pcash1 = &cash1;
pcash2 = &cash2;
```

Doch warum sollte ich ein implizites Typ-Casting anstreben ? Wann sind Zeiger mit unterschiedlichen Typen notwendig ? Mir kommen da nur Datenstrukturen wie Listen usw. in den Sinn. Es ist aber viel sinnvoller, in diesen Fällen ein Template zu generieren, das mir wieder volle Typ-Sicherheit gibt. Java funktioniert schließlich auch ohne explizit benutzbare Zeiger.

Ich würde es stets anstreben, dass die Parameter Referenzen sind. Das erhöht die Übersichtlichkeit im Quellcode und gibt dem Compiler die Möglichkeit, die meisten logischen Fehler zu eliminieren. Denn es gibt nichts schlimmeres für einen Programmierer als nur schwer reproduzierbare Laufzeitfehler aufzufinden.

Anmerkung

Ich habe den Code in diesem Dokument einfach frei Hand niedergeschrieben und nicht im Compiler getestet. Eventuell sind deshalb kleine Syntaxfehler drin, aber es geht ja auch mehr um das inhaltliche Verständnis der Virtualität.