Eberhard Karls
Universität
Tübingen

Übungsblatt 2

Implementation eines FiFo

Ausgabe: 7.5.2002
Abgabe: 14.5.2002, 11:45

Inhaltsverzeichnis

Einleitung

Mit diesem Übungsblatt werden wieder zwei Handouts ausgegeben:

1. *Von der Spezifikation zur Implementation* versucht einige Punkte bezüglich der Korrespondenz zwischen Spezifikation und Implementation zu erhellen.
2. *Module und Namensräume* liefert Hintergrundinformationen zum Export- und Importmechanismus bei Modula-2.

Ihr benötigt das erste Handout, um eine Teil der Aufgabenstellung der ersten Aufgabe zu verstehen. Das zweite Handout ist vor allem für diejenigen nützlich, die noch nie modular programmiert haben und Aufgabe 2 bearbeiten wollen.

1 Implementation der Druckerwarteschlange

Aufgabenstellung: Im letzten Aufgabenblatt solltet Ihr eine (sehr einfache) Drucker-Warteschlange spezifizieren. Es ist nun an der Zeit, dass Ihr diese Warteschlange implementiert und Euch Gedanken macht über die Beziehung zwischen Spezifikation und Implementation.

Wir möchten (in Kürze zusammengefasst), dass Ihr

1. Die Warteschlange und ein dazugehöriges Testbett (siehe Vorlesung) modular in C++ implementiert,
2. Entweder Übereinstimmung zwischen Spezifikation und Implementation argumentiert (im Sinne des ausgegebenen Handout) oder Nicht-Übereinstimmung dokumentiert und Implementation bzw. Spezifikation so korrigiert, dass Übereinstimmung herrscht.

Erläuterungen: Ich werde die einzelnen Ziele kurz näher erläutern: C++ ist, wie hoffentlich inzwischen klargeworden ist, wenn man die Check-Liste aus der Vorlesung durchgeht, nicht wirklich eine Softwaretechniksprache: Einige Mechanismen, die Softwaretechniksprachen zur Verfügung stellen sollten, sind in C++ nicht vorhanden und müssen, wie in der Vorlesung erläutert wurde, durch entsprechende Programmierdisziplin ersetzt werden:

- Systematischer Einsatz von Headerdateien,
- Kapselung der öffentlichen Symbole in Namensräume (C++-Schlüsselwort: *namespace*).
- Kein Objektmodul sollte andere Namen exportieren, als die in der Headerdatei aufgeführten. Zur Kontrolle kann das Unix-Tool '*nm*' verwendet werden.

Weder Compiler noch Sprachdefinition sind in dieser Hinsicht eine große Hilfe, da diese Disziplin nicht von ihnen überwacht oder gar erzwungen wird.

Wir wünschen uns, dass die von Euch angefertigte Implementation der Warteschlange und der Module des Testbetts im oben erwähnten Sinne modular ist.

Weiterhin möchten wir, dass Ihr – um zu demonstrieren, dass Euer Modul funktioniert – ein Testbett um die Warteschlange herum baut: Also zusätzlich eine Dummyimplementation für das Modul *Jobs*, und einen Testtreiber *TestSchedule*, d. h. ein Modul (bzw. Programm),

welches die Warteschlange bedient. Diese beiden Module sollten sich mit der Warteschlange zu einem ausführbaren Programm linken lassen, das einen Bericht etwa dieser Art ausgibt:

```
testing PrintQueue ...
 creating 2 queues ...
 putting entries into queue 1 ...
 putting entries into queue 2 ...
 pulling entries from queue 2 ...
 : 150 - Sonderauftrag 250/8
 :   8 - Testdruck Farbkalibrierung
 pulling entries from queue 1 ...
 : 100 - Artikel-Reprints Prof. Unrath
 :  17 - Diplomarbeit Schmitt
 :   1 - Dissertation Illec
OK. Testing done.
```

Wir haben eine sehr ähnliche, aber nicht vollständig identische[1] Warteschlange für die Vorlesung in Modula-2 implementiert. Diese ist jetzt von der Website zur Vorlesung verfügbar, damit Ihr Euch eventuell daran orientieren könnt.

Das Handout *Von der Spezifikation zur Implementation* erklärt, dass beim Übergang von der Spezifikation zur Implementation gewissen Brüche auftreten können: Die erste – zu idealisierte – Spezifikation kann dann nicht vollständig umgesetzt werden, da die Zielsprache oder -umgebung gewissen Einschränkungen unterworfen ist, die für die Idealisierung in der Spezifikation nicht gelten. Dem kann auf 2 Arten begegnet werden: Entweder werden die Unvollständigkeiten der Implementation als implementationspezifische Einschränkungen dokumentiert, oder die Spezifikation kann ihrerseits so korrigiert (eingeschränkt) werden, dass die Entsprechung zwischen Spezifikation und Implementation vollständig hergestellt werden kann.

Dokumentiert als Erstes solche Einschränkungen in Eurer Implementation, und korrigiert dann Eure Spezifikation und Eure Implementation entsprechend.

Abzugeben sind: Ordentliches Listing (Ausdrucke) aller Programmquellen (Nach meiner Zählung sollten das 5 Dateien sein). Zugrundeliegende Spezifikation der Warteschlange, sofern sie von der bereits Abgegebenen abweicht. Dokumentation der spezifischen Beschränkungen der Implementation (in natürlicher Sprache) und entsprechend modifizierte Teile der Spezifikation, die die Implementationsbeschränkungen wiederspiegeln.

2 Die Gefahren pauschalen Imports

Importmechanismen in Modula-2 und Java: Das Handout *Module und Namensräume* versucht, den Import- und Exportmechanismus von Modula-2 etwas näher zu motivieren.

Modula-2 bietet 2 Mechanismen an, mit welchen Konstrukte aus anderen Modulen zur Übersetzungszeit verfügbar gemacht werden können:

- Eine Deklaration wie '`IMPORT Foo;`' macht Konstrukte unter ihrem vollen Namen verfügbar, also beispielsweise als '`Foo.bar`'.
- Eine Deklaration wie '`IMPORT bar from foo`' erklärt, dass der importierte Name unter einer Kurzform verfügbar sein soll, im angegebenen Beispiel also als '`bar`'.

[1] Es handelt sich dabei um eine prioritätsgeordnete Warteschlange, nicht um einen simplen FiFo.

Hier fragt man sich natürlich, warum es keine Möglichkeit gibt, alle Konstrukte eines Moduls unter Ihrem verkürzten Namen zu importieren. Java bietet im Gegensatz zu Modula-2 genau diese Möglichkeit. Als modulare Einheit muss in Java das *Package* gelten (nicht die Klasse!), dabei sind die importierten Konstrukte aber ausschließlich Klassen.

Mit einer Deklaration der Art '`import thinkcrime.foo.*`' können alle Klassen im Package (das hier '`thinkcrime.foo`' heisst) unter Ihrem Kurznamen verfügbar gemacht werden (hier z. B. '`bar`', wenn das Paket eine solche Klasse enthält – also ohne den Klassenpfad bzw. Paketnamen). Was jedoch so harmlos und praktisch aussieht, hat subtile Fallstricke, über die Ihr im Folgenden nachdenken sollt.

Eine Fallgeschichte: Betrachten wir den folgenden Fall: Firma *B* entwickelt gerade eine Applikation, die aus vielen Klassen *B.wonderapp.a*, *B.wonderapp.b*, ... usw., besteht. Dabei setzen die Entwickler von *B* eine Klassenbibliothek der Firma *A* ein, die (unter anderem) das Package *A.superbib* enthält. Bis Version 0.95 der Klassenbibliothek compiliert 'Wonderapp' ohne Fehler.

Nachdem jedoch Firma *B* ein Upgrade auf Version 0.98 der Klassenbibliothek durchgeführt hat, kommt es beim Übersetzen der Applikation zu Fehlern (den genauen Fehlertext verschweigen wir hier), in – sagen wir – *B.wonderapp.q*. In dieser Klasse finden die Entwickler die Importdeklaration '`import A.superbib.*;`'.

In einem Telefongespräch mit den Entwicklern der Firma *A* schwören diese Stein und Bein, dass Version 0.98 "aufwärts kompatibel" zu Version 0.95 sei: Version 0.98 enthalte alle Klassen von Version 0.95 und diese würden genau die gleichen Funktionen zur Verfügung stellen. So ist es auch, es findet sich kein Hinweis, dass Klassen in 0.98 fehlen oder anders funktionieren.

Aufgabenstellung: Erkläre nun in einigen wohlgesetzten Worten: Was könnte passiert sein? Warum wäre es eine schlechte Idee, wenn *B* jetzt ein Rollkommando von Anwälten gegen *A* in Marsch setzen würde? Wer ist eigentlich verantwortlich für das Problem (denke an 'Design by Contract')? Was lässt sich daraus als Verhaltensmaßregel zum Umgang mit '`import`' (und für wen?) ableiten?

Lasst Euch nicht davon abschrecken, wenn Ihr kein Java könnt, dies ist kein Java-spezifisches Problem und müsste sich sehr gut mit den Informationen aus dem Handout lösen lassen.

Die Aufgabe beruht, nebenbei bemerkt, auf einem beim Übergang von JDK 1.2 auf JDK 1.3 wirklich aufgetretenen Problem.

Übrigens ist die Take-Home-Message nicht die, dass Java Mist ist (ich weiß, es gibt einige, die es ärgert, dass wir Java nicht zu den Softwaretechniksprachen zählen :-). Vielmehr soll diese Aufgabe dazu anregen, über Import- und Exportmechanismen verschiedener Sprachen bei Modulen, der Zweckmässigkeit, Anwendungbereich und Regeln für die Koordination von Entwicklern (Kontrakte) etwas nachzudenken.

Abzugeben ist: Eine (natürlichsprachige) Erläuterung, was wahrscheinlich im obigen Fall schief ging, einige Antworten auf die obigen Fragen.

Viel Spaß!