Softwaretechnik
2002
Prof. Dr. H. Klaeren und M E Leypold

Eberhard Karls
Universität
Tübingen


Übungsblatt 8

Strukturierte Analyse, MVC-Pattern

Ausgabe: 25.6.2002
Abgabe: 4.7.2002, 12:00

Inhaltsverzeichnis

Bitte das Vorwort sorgfältig lesen!

1 Vorwort

Dieses Übungsblatt ist das letzte Übungsblatt zur Softwaretechnik in diesem Semester, das testiert und für die Scheinkriterien mit herangezogen werden wird. Dieses Mal stellen wir 3 verschiedene Aufgaben zur Auswahl, die jeweils mit 100 Punkten gewertet werden: Es sollte nur eine Aufgabe bearbeitet werden.

In der ersten Aufgabe würdet Ihr Euch mit strukturierter Analyse (Datenflussdiagramme, Data Dictionaries und Mini-Spezifikationen) beschäftigen – das ist im Wesentlichen der aktuelle Stoff aus der Vorlesung. Alternativ dazu könnt Ihr ein einfaches objektorientiertes System (einen einfachen Textbetrachter mit einer minimalen Bearbeitungsfunktionen) unter Anwendung des Model-View-Controller-Pattern entweder implementieren oder entwerfen und dokumentieren.

Wir hoffen, dass es bei dieser Auswahl für jede(n) – ob PraktikerIn oder mehr planerisch veranlagt – möglich ist, sich eine Aufgabenstellung herauszusuchen, die ihr/ihm entspricht. Tendenziell wird wohl die erste Aufgabe mit der wenigsten Arbeit verbunden sein, die Implementation des Dateibetrachters mehr an die PraktikerInnen appellieren und der reine Entwurf des Dateibetrachters für die Teilnehmer interessant sein, die sich mit C++ unsicher fühlen.

2 Strukturierte Analyse des Disassemblers

2.1 Übersicht

Im Übungsblatt 5 habt Ihr Euch bereits mit einem einfachen Disassembler beschäftigt. In dieser Aufgabe soll es darum gehen, das Geschehen im Disassembler mit den Mitteln der strukturierten Analyse zu beschreiben: Datenflussdiagramme (DFD), Data Dictionaries und Mini-Spezifikationen.

Ich werde zur Orientierung den Verarbeitungsprozess im Disassembler in natürlicher Sprache beschreiben. Ihr könnt allerdings beim Zeichnen der DFDs auch andere Verarbeitungsschritte annehmen, wenn sich diese sinnvoll rechtfertigen lassen.

2.2 Natürlichsprachige Beschreibung

Beim Aufruf des Disassembler `disAs65` muss der Benutzer einen Dateinamen auf der Kommandozeile übergeben, der als der Name eines 6502-Programm-Image (wie in Blatt 5 beschrieben) interpretiert wird. Diese Datei wird eingelesen, disassembliert (nur das Code-Segment) und das resultierende Listing seitenweise auf den Drucker ausgegeben.

Diesen Prozess beschreibe ich nun näher im Detail. In der Programmdatei befindet sich eine Folge von Bytes. Diese kann als Header, ein Speicherabbild des Programmcodes und ein Speicherabbild der statischen Programmdaten interpretiert werden. Für den nun folgenden Übersetzungsprozess sind sowohl die Ladeadresse des Code, als auch sein Speicherabbild wichtig, die Programmdaten jedoch uninteressant.

Das Programm liest aus einer Datei mit fest kodiertem Namen (`/usr/share/DisAs65.table`) die Zuordnung der Opcodes zu den symbolischen Namen (Mnemoniks) und Adressierungsarten ein. Eine weitere externe Datei enthält zusätzliche Informationen über die Adressierungsarten (Länge der Operanden, Druckformat der Operanden). Die Information

in beiden Dateien ermöglicht es dem Disassembler, das Code-Segment in eine Folge von Instruktionen zu unterteilen. Jede Instruktion besteht aus einem Opcode, besitzt entweder keinen Operanden oder einen Operanden von ein oder zwei Byte Länge. Jede Operation kann einer bestimmten Ladeadresse zugeordnet werde – das ist wichtig, da ja später die disassemblierte Form der Instruktion auch die Ladeadresse enthalten soll (siehe Blatt 5).

Die Folge von Instruktionen kann nun unter Zuhilfenahme der bereits erwähnten Definitionstabellen in eine Folge von Zeilen übersetzt werden. Diese werden dann in Seiten zu je 60 Zeilen unterteilt und auf einen Drucker ausgegeben.

Der in der Aufgabenstellung von Blatt 5 mit auszugebende Header wurde hier der Einfachheit unterschlagen und muss in dieser Aufgabe auch nicht in den Datenflussdiagrammen berücksichtigt werden.

2.3 Aufgabenstellung

Erstelle – wie in Vorlesung und Skript erläutert – eine Hierarchie von Datenflussdiagrammen für den Disassembler, inklusive der dazugehörigen Data-Dictionaries und Mini-Spezifikationen. Es sollte sich mehr als eine Ebene der Detaillierung ergeben und der Detaillierungsgrad sollte genau genug sein, dass der Datenfluss in der Übersetzung einer einzelnen Instruktion sichtbar wird.

Abzugeben sind: DFDs, DDs, Minispecs in geordneter und sinnvoll nummerierter Form.

3 Ein Dateibetrachter

Der Dateibetrachter muss als eine Verlegenheitslösung gelten. Didaktisches Ziel war ein vollständiges *Programm zu implementieren, in dem ein einigermaßen vollständiger Pattern die Architektur stellt. Inzwischen ist mir ein bessere Beispiel eingefallen: Ein Tool zum interaktiven Betrachten von Verzeichnissen und löschen von Dateien. Das ist nicht ganz ungefährlich :-), und ein springender Punkt ist, dass ein Teil des Zustandes, der dargestellt wird außerhalb des Programms liegt. Sehr schön.*

3.1 Übersicht

In dieser Aufgabe soll ein einfacher Dateibetrachter unter Verwendung des Modell-View-Controller Pattern für ein zeichenorientiertes Terminal entweder entworfen oder implementiert werden. Für welche dieser beiden Aufgaben Ihr Euch entscheidet, bleibt Euch überlassen. Bewertungsmaßstab ist in beiden Fällen die Erkennbarkeit der angewandten Pattern und die deutliche Teilung des Programms in lose gekoppelte Komponenten mit klaren Zuständigkeiten.

3.2 Der MVC-Pattern

Mit diesem Aufgabenblatt geben wir wieder einige Kopien aus, diesmal einen Abschnitt aus dem Buch *Programmieren in Smalltalk mit Visualworks* (Bücker u. a., 1995). Dieser Abschnitt beschreibt – weitgehend sprachunabhängig – den Modell-View-Controller-Pattern[1].

[1] Dieser wird dort als MVC-*Paradigma* bezeichnet, da der Text aus einer Zeit stammt, in der Begriff Pattern noch nicht derart in Mode gekommen war. Modernere Literatur sieht im MVC-Pattern zudem eine Kombination

Kurz gesagt sind an einem MVC-Pattern 3 Arten von Klassen beteiligt: Das Datenmodell (Model) kapselt den Zustand bzw. den Datenbestand der Applikation (man kann dies durchaus als ADT verstehen). Der View ist für die Darstellung dieses Zustandes auf einem Ausgabegerät (in unserem Fall dem Terminal) verantwortlich. Der View erhält vom Modell Nachricht, wenn sich das Modell ändert, damit er nötigenfalls die Darstellung auf dem Ausgabegerät auffrischen kann. Der Controller andererseits liest das Eingabegerät (in unserem Fall die Tastatur) und übersetzt sie auf Operationen auf dem Datenmodell.

Für eine detaillierte Darstellung möchte ich auf die ausgegebenen Kopien verweisen – wirklich wichtig erscheinen mir vor allem die Seiten 242-246.

3.3 Die Anforderungen

Implementiert (oder entworfen) werden soll ein einfacher Dateibetrachter für ein zeichenbasiertes Terminal. Dieser soll das ganze Terminal für die Anzeige des Textes verwenden. Mittels bestimmter Tasten soll der Benutzer den angezeigten Ausschnitt des Textes verschieben (scrollen) können – und zwar zeilenweise oder bildschirmweise. Mit einer weiteren Taste soll der Benutzer die erste Zeile auf dem Bildschirm aus dem Text löschen können[2].

Sowohl für Entwurf, als auch Implementation sei die Wahl der Programmiersprache freigestellt. Dabei sollten jedoch folgende Regeln eingehalten werden.

- Die verwendete Sprache muss entweder objektorientiert sein (etwa Java, C++, Modula-3 oder OCAML) oder sie muss modular sein und dann müssen bestimmte OO-Fähigkeiten mit „zu Fuß“, d. h. mit Funktionszeigern, nachgebildet werden (Modula-2). Perl, Fortran und TCL seien explizit ausgeschlossen.
- Es darf kein vorhandener GUI-Toolkit o. ä. verwendet werden! Die Begründung ist die, dass diese Toolkits bereits ihre eigenen Implementationen des MVC-Pattern oder ähnlicher anderer Pattern enthalten. STL (C++) und Klassenbibliotheken, die ADTs implementieren (z. B. Datencontainer in Java) oder Ein- und Ausgabe abstrahieren, dürfen aber nach Belieben verwendet werden.
- Vermeidet komplizierte (unbegründete) Klassenhierarchien und Klassen mit wenig Funktionalität, also z. B. Klassen, die nur eine Methode enthalten, oder nur zur kurzfristigen Aufbewahrung von Daten dienen.

3.4 Tipps

Ein- und Ausgabegeräte: Diese können jeweils durch Objekte dargestellt werden, die den Zustand des Geräts kapseln[3] . Wer in C++ implementiert, kann die Module *tty*, *display*, *keyboard* und eventuell *events* aus dem EFASS-Editor verwenden und hinter einem Objekt verstecken.

Tastenbelegung: Wird eine Taste auf dem Keyboard gedrückt, setzt der Treiber des Betriebssystems dieses Ereignis in eine Sequenz von ein oder mehreren Zeichen um. Während die meisten Tasten in nur einem Zeichen resultieren, erzeugen gerade die für die Steuerung der Applikation interessanten Tasten (Up, Down) mehrere Zeichen. Diese müssten von

aus verschiedenen anderen einfacheren Pattern, wie *Observer*, *Strategy*, *Composite* und *Factory*. Diese Dekomposition des MVC-Pattern lohnt im vorliegenden einfachen Fall aber nicht.

[2] Diese Anforderung ist der ist der Versuch, so etwas wie eine einfache Bearbeitungsfunktion in die Aufgabe einzubringen, ohne sie über Gebühr aufzublähen.

[3] Dies ist eine Anwendung des Proxy-Pattern.

Standard-Eingabe (bzw. vom Terminaldevice) sequentiell gelesen und zu Eingabeereignissen geparst werden.

Hier bieten sich zwei Lösungsmöglichkeiten an: Zum einen kann das Modul *events* aus dem EFASS-Parser verwendet werden, zum anderen kann aber auch (für Zwecke dieser Aufgabe) auf eine Verwendung dieser Tasten verzichtet werden: Gesteuert wird die Applikation dann über die Eingabe von Buchstaben, etwa „u“ als Abkürzung für „up“.

Position der Ansicht: Bei Verwendung des MVC-Pattern kapselt das Modell den Zustand der Applikation, während der View nur dazu dient, diesen Zustand bzw. einen Ausschnitt desselben zu visualisieren. Bei genauerer Betrachtung verhält es sich allerdings komplizierter. Der Zustand des Dateibetrachters zum Beispiel setzt sich zusammen aus dem Text, den der Benutzer lesen möchte, und einer Position, die angibt, welcher Ausschnitt gerade auf dem Terminal dargestellt wird.

Streng genommen sollte das Modell nur aus dem Text bestehen. Die Position des Textfensters dagegen ist Bestandteil der View. Damit wird auch die View zustandsbehaftet und der Controller muss von Fall zu Fall auch Nachrichten an die View schicken, wenn dieser Zustand zu verändern ist.

Eine Alternative dazu besteht in der Einführung eines intermediären Editor-Objekts, wie dies in den ausgehändigten Kopien angedeutet ist. Für den vorliegenden Fall sind beide Ansätze äquivalent, der erstere aber vielleicht einfacher umzusetzen.

3.5 Aufgabenstellung 1 – Entwurf

Entwerfe und beschreibe den Dateibetrachter unter Einsatz der bekannten Mittel: Klassen-, Zustands- und Sequenzdiagramme sowie die syntaktische Beschreibung der Klassen-Schnittstellen.

Der Entwurf sollte detailliert genug sein, dass sich Fragen der folgenden Art im Prinzip daraus beantworten lassen:

- Was geschieht, wenn bestimmten Tasten gedrückt werden? Welche Klassen erhalten dann welche Nachrichten?
- Wann darf eine gegebene Operation aufgerufen werden? Was geschieht, wenn eine Operation aufgerufen wird? Welche Nachrichten werden wann verschickt.

Abzugeben sind: Die oben beschriebenen Diagramme, Skelettklassen und wo nötig verbale Beschreibungen.

3.6 Aufgabenstellung 2 – Implementation

Implementiere den Dateibetrachter unter Einsatz des MVC-Pattern. Die Implementation sollte funktionsfähig und sinnvoll gegliedert sein.

Abzugeben sind: Ausdrucke des Quelltext mit (lesbaren und verständlichen) Kommentaren.

Viel Erfolg.

Literatur

[Bücker u. a. 1995] BÜCKER, Matthias C. ; GEIDEL, Joachim ; LACHMANN, Matthias F.: *Programmieren in Smalltalk mit Visual Works: Smalltalk - nicht nur für Anfänger*. Springer, 1995. – ISBN 3-540-58813-2