Portierung der Dozenteneinsatzplanung

Wozu brauchen wir OOP? Diskussion am 18.12.18

Johannes Brauer

August 28, 2019

Warum?

Im Rahmen der Diskussionsrunde *Wozu brauchen wir OOP?*, die auf dem Pamphlet mit dem Titel Brauchen wir objektorientierte Programmierung? basierte, wurde vereinbart, dass der Autor versucht, die Erfahrungen, die in dem genannten Pamphlet lediglich auf der Erstellung einer funktionalen Version für *NoNopoly* (einer stark vereinfachten Monopoly-Simulation) basierte, dadurch zu vertiefen, dass eine kleine – aber nicht-triviale – Anwendung von einer objektorientierten in eine funktionale Version portiert werden sollte.

Als Untersuchungsgegenstand wurde die seit mehr als einem Jahrzehnt an der NAK benutzte Dozenteneinsatzplanung gewählt. Dabei sollte insbesondere auch die Erstellung einer Web-Oberfläche einbezogen werden, auf die bei der NoNopoly-Portierung verzichtet wurde.

Ausgangspunkt

Die Dozenteneinsatzplanung in der heute im Fachbereich benutzen Form ist um 2004 in einer ersten Fassung mit VisualWorks-Smalltalk entstanden. Sie ist nach dem Model-View-Controller-Prinzip entwickelt worden und war zu Beginn mit einer nativen VisualWorks-Oberfläche ausgestattet. Diese wurde später durch eine Weboberfläche unter Verwendung des Frameworks Seaside (Smalltalk) ersetzt. Aufgrund der konsequenten Nutzung von MVC konnte dieser Vorgang ohne Änderung an den Model-Klassen vollzogen werden.

Der letzte größere Entwicklungsschritt war dann die Portierung nach Pharo-Smalltalk (open source).

Eckdaten der Smalltalk-Version

- 8 Model-Klassen mit ca. 220 Methoden; ca. 1500 LoC
- keine Hierarchie unter den Model-Klassen
- 19 Seaside-Klassen (Komponenten) mit 199 Methoden; ca. 2600 LoC

Ziel

Wesentliche Komponenten der Dozenteneinsatzplanung sollten von Smalltalk (Pharo) nach Clojure/ClojureScript (s. u.) portiert werden.

Zur Wahl der Zielsprache

Die Begründung, als funktionale Sprache Clojure(script) zu verwenden, ist hauptsächlich in den Vorkenntnisse des Autors zu sehen. Ein irgendwie geartetes Auswahlverfahren hat nicht stattgefunden, wenn man von einigen Versuchen mit der Sprache *Elm* absieht.

Exkurs: Stichworte zu den Versuchen mit Elm

- Eigenschaften:
  - dient der Entwicklung von Webanwendungen
  - strenge, statische Typisierung
  - übersetzt nach Javascript
  - „Fazit": Wenn man den Type-Checker zufrieden gestellt hat, funktioniert die Anwendung,
  - „Nachteil": „Quick-and-dirty"-Programmierung im Sinne von „mal eben etwas ausprobieren" ist nicht möglich.
  - Elm kennt keine Komponenten (im Sinne von React, s. u.)
  - Diskussion über die Architektur größerer Elm-Anwendungen im vollen Gange
  - Elm noch nicht sehr stabil
  - Ein Minibeispiel für eine Dozenten-Pflege existiert: Quelltext

```
cd /Users/jb/Dropbox/elm/dep
elm reactor
... Go to <http://localhost:8000> to see your project dashboard.
```

Eckdaten der Clojure/ClojureScript-Version

Vorbemerkung zu Clojure vs. ClojureScript

Clojure ist ein Lisp-Dialekt, der auf der JVM läuft. ClojureScript hingegen ist ein Compiler für Clojure, der nach JavaScript übersetzt. Die erste Idee für die Portierung der Dozenteneinsatzplanung bestand in der Entwicklung einer Client-Server-Anwendung. Dabei sollten die Model-Klassen nach Clojure portiert und dann nach einer geeigneten Variante für die Realisierung der Weboberfläche Ausschau gehalten werden.

Der erste Teil hat stattgefunden. Im Zuge der Überlegungen zur Weboberfläche hat sich der Autor dann entschieden, statt einer Client-Server-Anwendung eine Single-Page-Application zu bauen und dafür dann ClojureScript zu verwenden. Die Anwendung läuft also vollständig im Web-Browser ab. Die schon entstandenen Clojure-Dateien für den Model-Part konnten dabei im Wesentlichen unverändert übernommen werden. (Die Dateiendungen müssen von .clj in .cljc geändert werden.)

Web-Technologie

Für die Entwicklung der Single-Page-Application lag es nun nahe, diese nach dem Konzept des *Functional Reactive Programming* zu bauen. Für JavaScript gibt es dafür verschiedene Bibliotheken (z. B. React, Angular, oder Vue.js). Das entscheidende Auswahlkriterium für den Autor, der zuvor noch nie eine SPA entwickelt und von JavaScript keine Ahnung hat, war, eine ClojureScript-Bibliothek zu finden, mit deren Hilfe FRP möglich ist, ohne in JavaScript programmieren zu müssen. So ist die Entscheidung für Reagent (A minimalistic ClojureScript interface to React.js) zustande gekommen.

Vereinfacht gesprochen besteht das Grundprinzip von Reagent darin, in einer (oder mehreren) ClojureScript-Variablen den Weltzustand zu speichern. Sollte sich durch Benutzer-Interkation dieser Zustand ändern, sorgt Reagent automatisch für das Rerendering der zugehörigen React-Komponenten im Browser.

Zahlen

Bei den in der Folge genannten Definitionen handelt es überwiegend um Funktionen ergänzt um ein paar Variablendefinitionen.

- Model: 5 cljc-Dateien mit 71 Definitionen; ca. 550 LoC
- Weboberfläche (cljs-Dateien):

- core.cljs - 4 Definitionen; 45 LoC
- state.cljs - 4 Definitionen; 40 LoC
- 5 Reagent-Komponenten mit 21 Definitionen; ca. 360 LoC
- 12 Hilfsfunktionen; ca. 230 LoC

Vergleich der Eckdaten

	Smalltalk		ClojureScript	
	Anz. Methoden	LoC	Anz. Definitionen	LoC
Model	220	1500	71	550
Oberfläche	199	2600	47	675
Summe LoC		4100		1225

Selbst wenn man berücksichtigt, dass

- der ClojureScript-Variante noch ein paar Kleinigkeiten fehlen (Der aktuelle Prototyp kann unter `https://johbra.github.io/dep/` besichtigt werden), so z. B.
  - eine Auswahlbox für Dozenten in Lehrveranstaltungen,
  - das Anlegen eines neuen Geschäftsjahres,
  - ein paar Sicherheitsabfragen z. B. beim Löschen von Moduln oder Manipeln oder der Neuplanung eines Quartals sowie
  - der Änderung der Semesterzuordnung für Manipel
- ein Redesign der Smalltalk-Variante hier und dort Einsparungen erbringen könnte,

dürfte die Smalltalk-Variante sicherlich um den Faktor 2 bezogen auf die LoC größer bleiben.

Erfahrungen mit Clojure/ClojureScript

Wie bereits in Brauchen wir objektorientierte Programmierung? ausgeführt, lautet die Standard-Antwort von Clojure-Entwicklern bezüglich der Frage „Was machen wir mit den Objekten/Klassen?“: Man nehme Hashmaps. Insbesondere im Abschnitt *NoNopoly mit Hashmaps* ist dargelegt worden, dass eine strikt funktionale Implementierung nicht nur möglich, sondern in einer funktionalen Sprache wohl auch die angemessene Vorgehensweise ist. Die Verwendung einer DSL, die die Benutzung von Klassen und Objekten

im Sinne der gängigen objektorientierten Sichtweise ermöglicht, hat keinen wirklichen Nutzen erkennen lassen. Auch die Verwendung von Clojure-Datentypen und Protokollen scheint nur dann sinnvoll zu sein, wenn für die zu erstellende Anwendung die Java-Interoperabilität von besonderer Bedeutung ist.

Der Beschränkung auf die grundständigen Clojure-Datenstrukturen (Neben den Hashmaps ist noch von Vektoren Gebrauch gemacht) worden, hat auch bei der Implementierung der Dozenteneinsatzplanung keinerlei Schwierigkeiten bereitet. Im Gegenteil bietet die Vorgehensweise einige Vorteile:

- Man spart das Anlegen von Klassen sowie von Getters und Setters.
- Man kann jederzeit aus Einzelheiten eine Struktur (Hashmap) erstellen, um sie z. B. als Argument an eine Funktion zu übergeben. Man erzeugt quasi „Ad-hoc“-Objekte. Dass nicht zuvor eine Klasse erstellt werden muss, ist überaus nützlich.
- Gerade bei der objektorientierten Programmierung von Oberflächen hat man es überwiegend mit Singletons zu tun. Keine Programmiersprache sollte einen Programmierer dazu zwingen, erst eine Klassen anlegen zu müssen, nur um davon anschließend ein einziges Exemplar zu erzeugen.

Eines der zentralen Anliegen der Objektorientierung besteht in der Vereinigung von Struktur und Verhalten. Nach den nunmehrigen Erfahrungen des Autors wird dadurch Wiederverwendung stark eingeschränkt. Es ist selten sinnvoll, dass eine Funktion/Methode nur im Kontext eines Objekts (bzw. einer Klasse) verwendet werden darf. Dies führt eher zu einer „Explosion“ der Anzahl der Methoden. Das Zitat von Alan Perlis: „It is better to have 100 functions operate on one data structure than 10 functions on 10 data structures.“ hat sich bei der Portierung der Dozenteneinsatzplanung bestätigt:

- Es wurden kaum eigene Funktionen entwickelt.
- Es gibt fast immer eine Standard- oder Bibliotheksfunktion, die das gegebene Problem löst.
- Dabei ist die intensive Verwendung der klassischen Funktionen höherer Ordnung (map, filter, reduce) überaus hilfreich.

Aber auch selbst definierte Funktionen höherer Ordnung führen zu kompakten Lösungen. Dies ist in einer Funktion, die alle modalen Fenster für

die unterschiedlichen Pflege-Dialoge (für Dozenten, Manipel, Module und Lehrveranstaltungen) erstellt, „auf die Spitze getrieben worden“, was dazu führte, dass die Funktion zwar kompakt aber auch sehr komplex ist.

Objektgeflechte

Bei der Entwicklung der Smalltalk-Variante schien es dem Autor naheliegend, immer wieder mit Objektgeflechten zu arbeiten. So hat ein Lehrveranstaltungs-Objekt je einen Verweis auf ein Dozenten-, ein Modul- und ein Manipel-Objekt. Durch Verfolgen der Verweise kann zwischen den Objekte programmtechnisch bequem navigiert werden.

Allerdings hat dies zur Folge, dass dem Anwender nicht so ohne weiteres gestattet werden kann, z. B. ein nach aktuellen Studienplänen nicht mehr gültiges Modul zu löschen, solange des noch Lehrveranstaltungs-Objekte gibt, die auf das Modul verweisen. Der Benutzer bekommt daher den Hinweis, dass das Modul nicht gelöscht werden kann, solange es noch Lehrveranstaltungen gibt, die den Modul-Verweis besitzen. Andernfalls drohte ein Crash der Anwendung.

In der funktionalen Variante gibt es keinerlei Objektverweise. In der Lehrveranstaltungs-Hashmap steht der Name des Dozenten, die Modulnummer und die Manipelkennung. Sollte das Modul mit der Nummer gelöscht werden, ist die „Lebensfähigkeit“ der Lehrveranstaltungs-Hashmap und der Anwendung insgesamt nicht bedroht. Gleichwohl könnte es im Sinne der Benutzungsfreundlichkeit sinnvoll sein, einen entsprechenden Warnhinweis zu geben, was zur Zeit – wie oben erwähnt – bisher nicht implementiert ist.

Selbstverständlich könnte man in der objektorientierten Variante auch auf direkte Objektverweise verzichten. Aber wäre das nicht ein weiteres Argument dafür, auf Objekte zu verzichten?

Web-Oberfläche

Bei der Erstellung der Web-Oberfläche unter Verwendung der Bibliothek Reagent kann natürlich nicht im engeren Sinne von einer Portierung der Smalltalk-Seaside-Oberfläche gesprochen werden. Dafür sind die verwendeten Technologien zu unterschiedlich. Die Smalltalk-Variante ist als Client-Server-Anwendung realisiert, bei der auf die Verwendung von JavaScript-Elementen fast vollständig verzichtet wurde, während die ClojureScript-Version vollständig im Browser läuft. Selbst wenn die funktionale Version auch als Client-Server-Anwendung implementiert worden wäre, wären Bibliotheken zum Einsatz gekommen, die keinerlei Verwandtschaft mit dem

Seaside-Framework aufweisen.

Eine Gemeinsamkeit gibt es dennoch. Typisch für Seaside ist, dass der Programmierer keinen HTML-Code schreibt, sondern stattdessen die von Seaside bereitgestellten Nachrichten benutzt, aus denen dann HTML-Code generiert wird. Eine ähnliche Vorgehensweise ist auch in Clojure/ClojureScript üblich. In Reagent aber auch in anderen Web-Bibliotheken wird die Bibliothek Hiccup verwendet, die erlaubt, HTML in Clojure-Syntax aufzuschreiben. HTML-Tags werden als Vektoren und Attribute als Hashmaps aufgeschrieben.

Vorläufiges Fazit (sehr vorläufig)

Der Autor hat sich im Verlauf der Arbeiten an der Portierung gefragt, welches der die Objektorientierung kennzeichnenden Prinzipien (Klassenprinzip, Kapselung, Polymorhie und Vererbung) er vielleicht in der funktionalen Welt vermisst hat.

> Nebenbei bemerkt hat die Objektorientierung der Welt der Programmiersprachen lediglich die Vererbung als Neuerung hinzugefügt. Alles andere gab es auch schon vorher. Vererbung gilt gleichzeitig als hoch problematisch. Dies hat viel damit zu tun, dass in der Objektorientierung nie geklärt wurde, welchem Zweck Klassenhierarchien eigentlich dienen sollen.

Um es vorweg zu nehmen, vermisst wurde nichts. Die Aussage bezieht sich aber nur auf das hier betrachtete Beispiel der Dozenteneinsatzplanung. Dabei handelt es sich ja um eine administrative Anwendung mit einer überschaubaren Anwendungslogik. Es ist durchaus denkbar, dass in einem anderen Kontext – z. B. in einer Simulation, bei der man vielleicht reale Objekte im Rechner rekonstruieren möchte – andere Erkenntnisse gewonnen werden könnten.

Kapselung

Kapselung von Strukturdetails von Hashmaps ist in der funktionalen Programmierung generell und auch in Clojure/ClojureScript kein Thema. Begrenzungen der Sichtbarkeit von Definitionen sind an Modulgrenzen möglich. Davon wurde in der Dozenteneinsatzplanung aber kein Gebrauch gemacht.

Klassen, Vererbung

Wie oben bereits erläutert, sind Klassen als Baupläne für Objekte vollkommen entbehrlich. Zu der Frage, ob Klassen als Elemente von Hierarchien nützlich sind, liefert die Portierung keine Anhaltspunkte, da unter den Model-Klassen von Hierarchien kein Gebrauch gemacht wurde.

Ob das Klassenprinzip überhaupt ein wesentlicher Bestandteil der Objektorientierung ist, kann durchaus infrage gestellt werden. Schließlich gibt es auch klassenlose objektorientierte Programmiersprachen.

Klassenhierarchien, Vererbung und Unterklassen-Polymorphie gehören jedenfalls nicht dazu, wenn man die Definition des Terminus „objektorientiert“ betrachtet, die Alan Kay, dem Erfinder des Begriffs, gegeben hat:

> „OOP to me means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things.“ ~Alan Kay

Mit anderen Worten, gemäß Alan Kay sind die wesentlichen Bestandteile von OOP:

- Message passing
- Encapsulation
- Dynamic binding

Polymorphie

Die gängigen objektorientierten Sprachen kennen dynamische Bindung nur in der primitiven Form der Unterklassen-Polymorphie. Da in den Model-Klassen der Smalltalk-Variante keine Klassenhierarchien existieren, spielt darauf aufbauende Polymorphie keine Rolle. Insofern trat diesbezüglich der Portierung auch kein Problem auf.

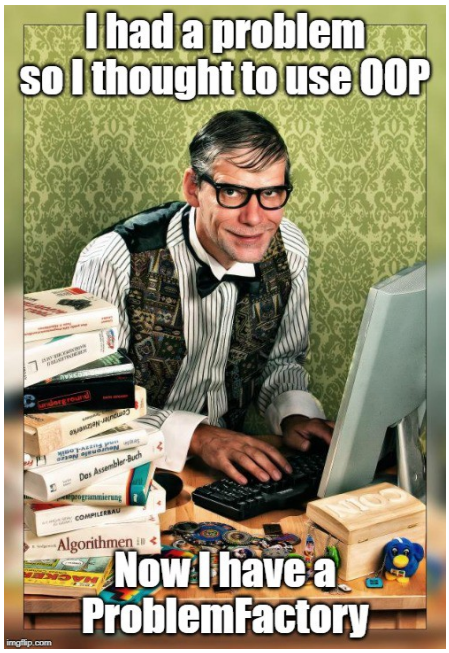
Es soll an dieser Stelle nicht unerwähnt bleiben, dass Clojure/ClojureScript mit dem Konzept der Multimethods über ein mächtiges Werkzeug für späte Bindung verfügt. Während in Smalltalk oder Java bei einem Ausdruck der Form `a.f(b)` die Auswahl der Methode `f` (absurderweise) ausschließlich von der Klassenzugehörigkeit von `a` abhängt (die von `b` bleibt unberücksichtigt), würde in Clojure eine Multimethod `f` mit einer Dispatcher-Funktion ausgestattet, über die der Bindungsmechanismus nach Belieben ausgestaltet werden kann.

Entwicklungswerkzeuge

Hier gibt es eigentlich nichts Bedeutendes zu berichten. Aber wenn auch die Arbeit mit der REPL einen dynamischen und produktiven Entwicklungsprozess erlaubt, so ist dem Autor zum wiederholten Male deutlich geworden, dass die Leistungsfähigkeit einer Smalltalk-Entwicklungsumgebung, wie sie von VisualWorks oder Pharo bereitgestellt wird, unerreicht ist.

Zum Schluss ...

... noch zwei „Zitate":

- 

- Object oriented programs are offered as alternatives to correct ones...

  Edsger W. Dijkstra, pioneer of computer science