Context-Oriented J Programming

Bastian Kruck

Hasso-Plattner-Institut Potsdam
bastian.kruck@student.hpi.uni-potsdam.de

Abstract. In diesem Seminar werde ich ein context-oriented programming framework in der Array-Programmiersprache J implementieren. Mit dessen Hilfe werde ich das Datenbanksystem jdb refaktorieren um Funktionalitäten ein und auszuschalten zu können. So können andere Pakete das Verhalten der Datenbank auf die jeweilige Situation optimieren. Zum Beispiel ist das importieren einer validen, sortierten Datenbank schneller, wenn die Validierung und Sortierung dafür temporär ausgeschaltet wird.

Keywords: context-oriented programming, software, j programming language, array programming

1 Background

Array Programming ermöglicht das präzise Manipulieren multidimensionaler Daten. Ingenieure schreiben darin Skripte, um wegleitende Erkenntnisse aus z.B. sensorischen oder statistischen Daten zu gewinnen.

In der Array-Programming-Sprache J gibt es Namensräume (locales) [1]. Diese (a) helfen Namenskollisionen zu vermeiden, (b) bieten einen Gültigkeitsbereich für lokale Variablen und (c) haben einen lookup-Mechanismus, der das Nachschlagen von globalen Variablen ermöglicht. Das Verhalten des lookup-Mechanismus wird durch eine Liste von Namensräumen bestimmt (lookup-Pfad). Bei Benutzung einer nicht definierten Variablen wird transparent der Reihe nach versucht, diesen Namen in einem der Namensräume im lookup-Pfad aufzulösen und bei Erfolg der gefundene Wert zurückgegeben. Das funktioniert für Operanden (Nomen), benannte Operatoren (Verben) und benannte Funktionen höherer Ordnung (Adverbien und Konjunktionen). Der lookup-Mechanismus wird in J benutzt um prototypbasierte Objekt-Orientierung zu implementieren. Die Standardbibliothek (stdlib.ijs) implementiert dafür u.a. die Hilfsmethoden `coclass`, `coinsert`, `conew` [2].

`coclass 'Collection'` erzeugt einen neuen Namensraum namens 'Collection' und wählt ihn als aktuell aus.

[1] Quelle: http://www.jsoftware.com/help/dictionary/dx018.htm und http://www.jsoftware.com/help/learning/24.htm, aufgerufen am 24.10.2014

[2] Quelle: http://www.jsoftware.com/help/learning/25.htm, aufgerufen am 24.10.2014

`coinsert 'AbstractClass'` hängt den Namensraum AbstractClass vorne an den lookup-Pfad vom aktuellen Namensraum. Collection ist nun eine Unterklasse von AbstractClass.

`conew Collection` erzeugt einen neuen (namenlosen) Namensraum und nimmt Collection in den lookup-Pfad auf.

Um in J zwischen Kontexten zu differenzieren werden bisher Verzweigungen benutzt. Zum Beispiel wird in trace.ijs abhängig vom Modus-Parameter (`t_mode`) entweder die `execute`-Funktion für das Nachverfolgen (Trace) oder Klammern (Parenthesis) benutzt (siehe Listing 1.1).

Listing 1.1. Verzweigungen tauschen alternative Implementierungen abhängig vom Modus aus trace.ijs Zeile 185

```
if. 'trace' -: t_mode do.
 execute=. executet
 move    =. movet
else.
 execute=. executep
 move    =. movep
end.
```

J hat ein ausgereiftes Paketsystem. Es gibt Mechanismen zum Beschreiben von Abhängigkeiten von anderen Paketen und zum Definieren von Demos und Tests.

`jdb` ist das Standarddatenbanksystem von J. Im Paket `'data/jdb'` wird eine Implementierung bereitgestellt, die schnell und zuverlssig funktioniert. Eine Unterscheidung verschiedener Modi wird nicht vorgenommen (siehe Listing 1.2). Sortierung findet nur zur Anfragezeit statt und die Validierung von Datensatzformat und Fremdschlüsseln ist unausweichlich. Könnte man die automatischen Transaktionen abschalten, so wären manuelle Transaktionsbündelung und schnelles Importieren möglich.

Listing 1.2. Im Paket `data/jdb` sind Transaktionen und die Validierung des inneren Zustands unumgänglich. jdb.ijs Zeile 1180.

```
Delete=: 3 : 0
   delete y
   commit''
   dbwritetrans''
)
```

2 Solution

Fast alle Skripte in der J Standard-Umgebung sind innerhalb eines speziellen Namensraums formuliert. Wenn es nicht explizit angegeben ist, so ist der aktuelle Namensraum `base` und `z` der einzige Namensraum im lookup-Pfad.

J bietet genügend Reflektion, um COP zu implementieren. Es lassen sich Methoden aus einem speziellen Namensraum im Variablenbereich eines anderen ausführen sowie die Methoden eines Namensraums auflisten und verändern.

3 Impact

Layern von `jdb` macht J schneller und besser. J ist eine Sprache zur Datenverarbeitung. Das Datenbanksystem stellt einen wichtigen Bestandteil der Sprachumgebung dar. Durch das Layern von `jdb` wird dieser Bestandteil beherrschbarer und mächtiger. Entwickler kïnnen von mehr Funktionalität und von stellenweise besserer Geschwindigkeit profitieren.

Context-oriented J Programming macht die Arbeit mit Paketen in J konfliktfreier und bedürfnisgerechter. Mit Context-based J Programming lassen sich Bibliotheksfunktionen auf die wechselnden Anforderungen an die Sprachumgebung dynamisch optimieren. Ein Paket, das Konsumenten eine Schnittstelle zum gezielten Ein- und Ausschalten von Funktionalität bietet, erfüllt mehr Bedürfnisse von konsumierenden Paketen. Das dynamische Ein- und Ausschalten verhindert Konflikte bei der Benutzung mehrerer Pakete (der Importierer möchte keine Validierung, das Wartungsfrontend schon).

4 Appendix I: Proof of Concept

Dies sind Screenshots von einer Proof-of-Concept-Implementierung vom Einfrieren einer Collection.

Hilfe zur Unterstrich Notation Sei `obj` ein Objekt der Klasse `Class`. `Method__obj 0` ist equivalent zu `Method_8_ 0` wobei `8` die id von `obj` ist. `Method_8_` ist nicht definiert, aber `Method_Class_` kann aufgelöst werden und so wird die Methode `Method` aus dem Namensraum `Class` auf dem Object `obj` ausgefhrt.

```
NB. define a Collection
coclass 'Collection'
   create  =: 3 : 'items =: 0 $ 0'
   add     =: 3 : '# items =: (< y) , items'
   remove  =: 3 : '# items =: items -. < y'
   inspect =: 3 : 'items'
   destroy =: codestroy
```

```
   NB. testing a collection
   cocurrent 'base'
      C1 =: 0 conew 'Collection'
      add__C1 'foo'
1
      add__C1 37
2
      inspect__C1 0
┌──┬───┐
│37│foo│
└──┴───┘
      remove__C1 'foo'
1
      inspect__C1 0
┌──┐
│37│
└──┘
```

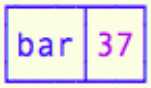
```
   NB. describe the writeonly layer
   'Collection' layer 'writeonly'
      add =: 3 : '# items'
      remove =: 3 : '# items'
      destroy =: 3 : '0'

   disable_layer 'writeonly'

   NB. tesing a layered collection
   cocurrent 'base'
      inspect__C1 0
```


```
      enable_layer 'writeonly'
      add__C1 'bar' NB. no modification
1
      inspect__C1 0
```


```
      disable_layer 'writeonly'
      add__C1 'bar'
2
      inspect__C1 0
```



```
   NB. how does it work?
      NB. there is three locales involved:
      NB. - Collection contains non-layered members
      ownProps 'Collection'
┌──────┬───────┐
│create│inspect│
└──────┴───────┘
      NB. - CollectionWriteonly contains the layered implementations
      ownProps 'CollectionWriteonly'
┌───┬───────┬──────┐
│add│destroy│remove│
└───┴───────┴──────┘
      NB. - CollectionZ contains the non-layered implementations
      ownProps 'CollectionZ'
┌───┬───────┬──────┐
│add│destroy│remove│
└───┴───────┴──────┘


      NB. the add method from CollectionWriteonly
      NB. only return the number of items
      add_CollectionWriteonly_
3 : '# items'

      NB. the add method from CollectionZ
      NB. adds the item and returns the number of items
      add_CollectionZ_
3 : '# items =: (< y) , items'

      NB. enable the 'writeonly' layer, i.e.
      NB. pluck the CollectionWriteonly locale into the lookup path
      enable_layer 'writeonly'

      NB. show add method from Collection
      add_Collection_
3 : '# items'

      NB. disable the 'writeonly' layer, i.e.
      NB. remove the CollectionWriteonly locale from lookup path
      disable_layer 'writeonly'

      NB. show add method from Collection
      add_Collection_
3 : '# items =: (< y) , items'
```