Statistische Felder im Universe Type System

Semesterarbeit

Thomas Hächler

Software Component Technology Group
Departement Informatik
ETH Zürich

Sommersemester 2004
Abstract

Das Ziel dieser Arbeit ist, die Verwendung von statischen Feldern in objekt-orientierten Programmiersprachen, im Besonderen Java, zu untersuchen und ins Universe Type System einzubinden. Das Universe Type System ist ein Ownership-Typsysten, welches Repräsentations-Kapselung und Dependency Control sicherstellt. In Kapitel 3 werden vier Ansätze der Einbindung diskutiert, jeweils mit Beispiel erläutert und Vor- und Nachteile bezüglich Typsicherheit und Möglichkeiten der Verwendung dargelegt.

Ein global-Universum wird in Kapitel 4 genauer behandelt, so dass es als Erweiterung zum Universe Type System verwendet werden kann. Dabei musste der Typkombinator und die Invariante angepasst werden.

Es existiert bereits ein Compiler zum Universe Type System, in Kapitel 5 beschreibe ich kurz, wie die Lösung mit global-Universum in diesem Compiler implementiert wurde.
Inhaltsverzeichnis

1 Statische Felder in objektorientierten Programmiersprachen 5
  1.1 Statische Felder in Java ........................................... 5
  1.2 Statische Felder in Eiffel? ........................................ 5
  1.3 Statische Felder sollten vermieden werden ..................... 6
  1.4 Statische Felder in java.lang und java.util ................... 7
  1.5 Wie werden statische Felder verwendet? ....................... 8
    1.5.1 Konstanten ...................................................... 8
    1.5.2 Metainformation über die Instanzen ......................... 8
    1.5.3 globale Variablen ............................................. 8
    1.5.4 Singleton Pattern ............................................ 9

2 Universe Type System – Einführung 11
  2.1 Kurze Einführung zum Universe Type System .................. 11
    2.1.1 Repräsentations-Kapselung .................................. 11
    2.1.2 Dependency Control .......................................... 11
    2.1.3 Invariante im Universe Type System ....................... 12
  2.2 Beispiel und Notation .............................................. 12
INHALTSVERZEICHNIS

3 Verschiedene Ansätze 15

3.1 Statische Referenzen sind readonly .......................... 15
3.2 Zusätzliches classwide-Universum .......................... 17
3.3 readwrite Schlüsselwort ........................................ 20
3.4 Zusätzliches global-Universum ................................ 23

4 Einbindung ins Universe Type System 26

4.1 Lösung mit neuem global Universum ......................... 26
   4.1.1 Default für statische Felder ist readonly .................. 26
   4.1.2 global: Schlüsselwort und Universum .................... 27
4.2 Beispiel – globaler Cache ..................................... 28
4.3 Beispiel – Foo Bar .............................................. 31
4.4 Zwei Varianten der Veranschaulichung ....................... 33
   4.4.1 global-Universum separat von den Instanz-Universen .... 33
   4.4.2 global-Universum umschließt die Instanz-Universen ........ 33
4.5 Typ-Kombinator .................................................. 35
4.6 Erweiterung der Invariante des Universe Type System .... 36
4.7 Problem mit rekursiven global-Strukturen .................. 36
4.8 Zusammenfassung ............................................... 38

5 global im Universes Kompiler 39

5.1 Implementation .................................................. 39
   5.1.1 Neues Schlüsselwort global ................................. 39
   5.1.2 Grammatik .................................................. 39
5.1.3 CUniverseGlobal.java .................................................. 40
5.1.4 CUniverse.java .......................................................... 40

A Einige Details ............................................................... 43

A.1 grep java.util ........................................................... 43
A.2 Quellcode zu Beispiel in Abschnitt 4.4 .............................. 44
A.3 Quellcode von CUniverseGlobal.java ................................. 45
A.4 Quellcode von Methode combine(..) aus CUniverse.java .... 45
Kapitel 1

Statische Felder in objektorientierten Programmiersprachen

1.1 Statische Felder in Java

Werden in Java Felder und Methoden als static deklariert, gehören sie nicht zu den jeweiligen Instanzen der Klasse, sondern zur Klasse direkt. Sie werden deshalb auch Klassen-Felder und -Methoden genannt.

Auf statische Felder und Methoden kann überall her zugegriffen werden, sofern der Zugriff nicht durch ein zugriﬀbeschränkendes Schlüsselwort verhindert wird.


1.2 Statische Felder in Eiffel?

In Eiffel gibt es kein Schlüsselwort, das static in Java entsprechen würde. Für einige Anwendungen, wird jedoch das once-Konzept verwendet.


Als Standard für die Verwendung von Konstanten wird in Eiffel eine separate Klasse geschrieben,
welche an allen Klassen, die die Konstanten benötigen, vererbt wird. Für globale Objekte werden once-Features verwendet. once-Features können in einer Subklasse neu definiert und müssen nicht mehr als once-Feature implementiert werden. Daraus ergibt sich ein ganz anderes Verhalten, als bei statischen Feldern und Methoden in Java.

Karine Arnout und Eric Bezault beschreiben im Artikel "Singleton in Eiffel" [AB04], dass es in Eiffel nicht möglich ist, das Singleton Pattern [GHJV95] als wiederverwendbare Bibliothek zu implementieren. Es kann nicht verhindert werden, dass mehr als eine Instanz einer Klasse erzeugt wird. Ein Feature get_singleton kann zwar als once implementiert werden und somit immer den gleichen Rückgabewert haben, aber es kann nicht verhindert werden, dass mehr als eine Instanz einer Klasse erzeugt wird.

Mit once-Features bestehen aber auch genau die Probleme, die im Abschnitt 1.3 beschrieben werden.

1.3 Statische Felder sollten vermieden werden

Obwohl ich in dieser Arbeit darüber schreibe, wie statische Felder in ein Ownership-Typsystem integriert werden können, muss ich gerade zu Beginn darlegen, dass die Verwendung von statischen, veränderbaren Feldern nicht empfohlen wird.

Statische Methoden, welche als Variablen nur lokale Variablen oder statische Felder verwenden können, sind nicht "wirkungsvolles objekt-orientiertes Design", wie Tony Sintes in [Sin01] beschreibt. Klassen mit statischen Methoden unterstützen, was den statischen Teil der Klasse betrifft, weder Vererbung und Subtyping, noch Polymorphismus.


Im FAQ der Enterprise Java Beans [Sun01] wird erklärt, dass statische, veränderbare Felder problematisch werden, sobald ein Programm nicht mehr auf einer JVM sondern auf mehreren läuft, weil es dann mehr als eine Instanz des statischen Teil einer Klasse gibt, nämlich eine Instanz pro JVM. Es kann sogar vorkommen, dass auf der gleichen JVM mehrere Instanzen auftreten, wie Ted Neward in seinem Artikel "When is a static not static" [New01] beschreibt. Zum Beispiel, wenn verschiedene Classloaders verwendet werden.

<table>
<thead>
<tr>
<th>Fußnote</th>
<th>Bemerkung</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Ein Feld ist veränderbar, wenn es nicht als final deklariert wurde.</td>
</tr>
<tr>
<td>3</td>
<td>Hilfsmethoden: Funktionen, welche nicht von einem Zustand des Objekts oder der Klasse abhängen; im Universe Type System auch &quot;pure Methoden&quot; genannt.</td>
</tr>
</tbody>
</table>
1.4 Statische Felder in *java.lang* und *java.util*

In den Java-Paketen *java.lang* und *java.util* (diese beide habe ich mir genauer angeschaut) fällt auf, dass nur ganz selten statische non-final Felder verwendet werden.


```java
// class data: instance map
private static HashMap instances = new HashMap(7);
// ...
private static Currency getInstance(String currencyCode,
        int defaultFractionDigits) {
    synchronized (instances) {
        // Try to look up the currency code in the instances table.
        // This does the null pointer check as a side effect.
        // Also, if there already is an entry, the currencyCode must be valid.
        Currency instance = (Currency) instances.get(currencyCode);
        if (instance != null) {
            return instance;
        }
    }
    // ...
    instance = new Currency(currencyCode, defaultFractionDigits);
    instances.put(currencyCode, instance);
    return instance;
}
```

Quellcode 1.1: Ausschnitt aus *java.util.Currency*.


Viele statische Felder in den Java Libraries sind deprecated, was so viel bedeutet, dass sie im Laufe der Zeit eliminiert werden sollen.

Die richtige und sichere Anwendung von statischen Feldern ist also nicht trivial und kann häufig vermieden werden. Trotzdem versuche ich in Abschnitt 1.5 herauszufinden, wie statische Felder in der Praxis verwendet werden.
1.5 Wie werden statische Felder verwendet?

Ich versuchte herauszufinden, wie denn statische Felder in objektorientierten Programmiersprachen verwendet werden. Als Quellen verwendete ich meine eigene Erfahrung, Programmierbücher ([Fla99], [Mey97], [GHJV95]) und eine Applikation, an welcher ich mitprogrammiert habe [EGH03].

1.5.1 Konstanten

Im einfachsten Fall werden statische Felder auch als unveränderbar (final) deklariert. So verändern sich ihre Werte nach der Initialisierung nicht mehr und Referenzen oder gegebenenfalls Kopien des Feldes sind per Definition immer auf den gleichen Wert gesetzt.

Beispiel

```java
public static final int MAX_NR_OF_INSTANCES = 99;
```

1.5.2 Metainformation über die Instanzen

Werden Informationen über die Menge aller Instanzen einer Klasse gebraucht, so können diese Metainformationen zum Beispiel statisch in Klassenvariablen gespeichert werden. Typischerweise wird vom Konstruktor der entsprechenden Klasse das neue Objekt in die Menge eingefügt, respektive die Information aktualisiert (zB ein Zähler oder Generator von eindeutigen IDs.)

Beispiel

Im Quellcode[1.2] wird eine Klasse skizziert, welche ihre Instanzen in einer Collection instances speichert, zum Beispiel um später auf allen eine Operation doSomething() auszuführen.

1.5.3 globale Variablen

Manchmal kommt man nicht darum herum Informationen global abzuspeichern. Dafür werden globale Variablen gebraucht. Natürlich kann der Zugriff auch über getter- und setter-Methoden erfolgen (vgl auch [Ven99]).
KAPITEL 1. STATISCHE FELDER IN OBJEKTORIENTIERTEN PROGRAMMIESPRACHEN

```java
class Element {
    static int instanceCounter = 0;
    static Collection instances = new Collection();

    static Iterator iterator() {
        return instances.iterator();
    }

    /** constructor */
    Element() {
        instanceCounter++;
        instances.add(this);
    }
    void doSomething() {
        // ...
    }
}

void main() {
    Iterator i = Element.iterator();
    while (i.hasNext()) {
        i.next().doSomething();
    }
}
```

Quellcode 1.2: Eine Klasse Element registriert alle Instanzen in einer Collection; main()-Programm verwendet diese.

Beispiel

```java
static Song myFavouriteSong;
static Date lastTimeWrittenToStandardOut;
```

1.5.4 Singleton Pattern

Es hat sich herausgestellt, dass das Singleton Pattern \[GHJV95\] in objekt-orientierter Programmierung eine spezielle Rolle spielt. Vorteile dieser Patterns sind in einem Artikel auf "JavaWorld" \[Sin00\] beschrieben. Das Singleton Pattern wird recht häufig verwendet und die darin vorkommende statische Referenz auf die eine Instanz des Singleton-Objekts kann nur mit erheblichem Programmieraufwand entfernt werden. z.B. müsste eine Referenz auf das Singleton-Objekt immer als Parameter mit gereicht werden.

Eine typische Implementation ist in Quellcode \[1.3\] dargestellt.
KAPITEL 1. STATISCHE FELDER IN OBJEKTORIENTIERTEN PROGRAMMIERSPRACHEN

class Singleton {
    /** hold the one and only instance */
    private static Singleton instance = null;
    /** avoid instantiation from outside */
    private Singleton () { /** default constructor */}
    /** provide the one and only instance */
    public static Singleton getInstance () {
        if (instance == null) instance = new Singleton();
        return instance;
    }
}

Quellcode 1.3: Implementation des Singleton-Patterns.
Kapitel 2

Universe Type System – Einführung

2.1 Kurze Einführung zum Universe Type System


Programme, die im Universe Type System implementiert sind, können modular und zur Kompilezeit auf Typsicherheit getestet werden.

2.1.1 Repräsentations-Kapselung


Diese Kapselung wird Repräsentations-Kapselung genannt.

2.1.2 Dependency Control

Dependency Control wird gebraucht, um die Menge aller Objekte, von denen die Invariante einer Klasse abhängt einzugeben.
Invarianten dürfen nur von rep-Objekten abhängen.

### 2.1.3 Invariante im Universe Type System

Für die oben beschriebenen Eigenschaften gilt die folgende Invariante für jedem Ausführungsschritt eines Programms:

Falls ein Objekt $U$ eine direkte Referenz auf ein Objekt $V$ hat, ist entweder

- $U$ der Besitzer von $V$ oder
- $U$ und $V$ gehören zum gleichen Universum oder
- die Referenz ist readonly.

### 2.2 Beispiel und Notation

Ich erlaube mir, das Beispiel aus dem Paper [MPH01] zu übernehmen und, zur Erklärung meiner Notation, mit meinem Layout darzustellen.

**Beispiel einer LinkedList mit Iterator**

Eine Klasse `Node` (Quellcode 2.1) wird verwendet um eine `LinkedList` (Quellcode 2.2) aufzubauen. Die `LinkedList` bietet außerdem einen Iterator `Iter` (Quellcode 2.3) an, mit dem alle Elemente, die in der Liste gespeichert sind abgerufen werden können.

```java
class Node {
    peer Node prev, next;
    readonly Object elem;
}
```

Quellcode 2.1: Die Klasse `Node` implementiert einen Knoten für die `LinkedList` (Quellcode 2.2).

```java
class LinkedList {
    rep Node first, last;
    void add(readonly Object o) {/*...*/}
    Iter getIter() { return new peer Iter(this); }
}
```

Quellcode 2.2: Klasse `LinkedList`
KAPITEL 2. UNIVERSE TYPE SYSTEM – EINFÜHRUNG

```java
class Iter {
    peer LinkedList list;
    readonly Node position;
    Iter(peer LinkedList l) {
        list = l;
        position = ((readonly List l).first;
    }
    readonly Object next() {
        readonly Object result = position.elem;
        position = position.next;
        return result;
    }
}
```

Quellcode 2.3: Die Klasse Iter implementiert einen Iterator, um alle Elemente einer LinkedList (Quellcode 2.2) zu erreichen.

**Objekt-Struktur**


Referenzen werden als Pfeile dargestellt, wobei readonly-Referenzen gepunktet werden. Weitere Referenztypen (wie z.B. global) werden jeweils in Form einer Legende in der jeweiligen Objekt-Struktur-Abbildung angegeben.

In Abbildung 2.4 wird die Objekt-Struktur des LinkedList-Beispiel dargestellt (Quellcode 2.2 Quellcode 2.1 und Quellcode 2.3)
Abbildung 2.4: Objektstruktur für das LinkedList-Beispiel aus [MPH01].
Kapitel 3

Verschiedene Ansätze

In diesem Kapitel werden verschiedene Möglichkeiten behandelt, wie statische Felder im Universe Type System [MPH01] realisiert werden können.

Im ersten Ansatz (3.1) beschreibe ich eine sehr restriktive Möglichkeit, bei dem alle static Referenzen implizit readonly sind.


In Ansatz 3.4 wird ein neues global-Universum eingeführt, in welchem sich Objekte befinden, auf welche potentiell von überall her Lese- und Schreibreferenzen zeigen.

In Kapitel 4 werden dann die Ansätze 3.1 (readonly) und 3.4 (global) zusammengeführt und als Lösung dargestellt.

3.1 Statische Referenzen sind readonly

Grundidee dieses Ansatzes ist, die statische Referenzen als readonly-Referenzen zu behandeln. Dadurch gibt es keine gemeinsame Objekte, wo verschiedene Instanzen Schreibrecht haben, wie dies in Java mit Hilfe des Schlüsselwortes static erreicht werden kann.

Mit diesem Ansatz wird betont, dass der statische Teil der Klasse nicht zu einer internen Repräsentation einer Instanz oder gar zur internen Repräsentation aller Instanzen gehört, sondern separat zu beachten ist. Es kann aber sein, dass ein Objekt eine peer- oder rep-Referenz auf dasselbe Objekt hat, das in einer Klassenvariable gespeichert ist.
**Regel:** static impliziert readonly.

**Beispiel**

```java
class SingletonThread extends Thread {
    static SingletonThread instance = null;
    //** start myself on instantiation */
    protected SingletonThread() { this.start(); }
    public static Singleton getInstance() {
        if (instance == null) instance = new SingletonThread();
        return instance;
    }
    rep Collection collection;
    void run() { // implements Runnable.
        while (collection.size() < 60) {
            collection.add(new rep Date());
            sleep(1000);
        }
    }
    public Iterator iterator() {
        return collection.iterator();
    }
}
```

Quellcode 3.1: Ein SingletonThread, der zu Laufzeit selber Objekte instanziert.

Eine Klasse SingletonThread (Quellcode 3.1), welche Thread erweitert, instanziert in der Methode run() selber Objekte im eigenen Instanz-Universum. Weil getInstance() static ist, ist die Rückgabevereferenz auf das Singleton-Objekt readonly.


Entsprechend der Regeln der bisherigen Universen, erfolgen alle Operationen von main() aus Quellcode 3.3 auf readonly-Referenzen. Diese sind in Abbildung 3.2 dargestellt.

**Diskussion**

Dieser Ansatz strebt an, kein neues Schlüsselwort einzuführen und ist im bestehenden Universe Type System recht einfach zu integrieren, nimmt dafür einige Einschränkungen in Kauf.

Es wäre zum Beispiel nicht ohne Weiteres möglich ein System.out anzubieten, wie es im Quellcode 3.3 verwendet wird, weil println() nicht pure sein kann.
KAPITEL 3. VERSCHIEDENE ANSÄTZE

Abbildung 3.2: Aus Instanz-Universen kann nur `readonly` auf Objekte im statischen Universum zugegriffen werden.

```java
void main() {
    readonly SingletonThread st = SingletonThread.getInstance();
    sleep(10000);
    readonly Iterator i = st.iterator();
    while (st.hasNext()) {
        System.out.println(st.next());
    }
}
```

Quellcode 3.3: Ein `main()`-Programm, das den SingletonThread (Quellcode 3.1) verwendet.

Aus Kapitel 1.5 werden nur Konstanten (Abschnitt 1.5.1) voll unterstützt. Ausserdem eine eingeschränkte Form des Singleton Pattern (Abschnitt 1.5.4), wenn es ausreicht, auf das Singleton Objekt nur lesend zuzugreifen, was durchaus Sinn machen kann, zum Beispiel bei Property-Files, welche eingelesen werden oder bei Hardware-Devices, welche nur Input zulassen (z.B. Sensor).

3.2 Zusätzliches classwide-Universum

Typische Anwendung von klassenweiten Referenzen sind Metadaten über die Instanzen einer Klasse, wie in Abschnitt 1.5.2 beschrieben.


Beispiel Eine Klasse `Element` (Quellcode 3.5) möchte mit einer Methode `static changeAll()` anbieten, alle Instanzen zu verändern. Sie hält alle Referenzen auf Instanzen in einem statischen `classwide` Feld `instances` und bietet mit einer Methode `static classwide Object[] arrayOfInstances()` allen Instanzen den gewünschten Zugriff. (An dieser Stelle kann kein Iterator verwendet werden, weil auf einem `readonly` Iterator die Methode `next()` nicht aufgerufen werden darf, weil dies den internen Zustand des Iterators verändern würde.)

Instanziert nun `main` (Quellcode 3.6) Elemente, werden diese in der `Collection Element.instances` gespeichert (Abbildung 3.4). Greift `main()` nun mit Hilfe des Iterator `Element.iterator()` auf die `Element`-Instanzen zu, hat `main()` nur `readonly`-Zugriff auf einzelne Objekte. `i.next().changeSomething()` aus Quellcode 3.6 ist verboten. Benutzt `main()` jedoch die statische Methode `Element.changeAll()`, können alle Instanzen verändert werden. Ein beschränkter Zugriff auf alle Instanzen wird möglich.
KAPITEL 3. VERSCHIEDENE ANSÄTZE

```java
class Element {
    static int instanceCounter = 0;
    static Collection instances = new Collection();
    static classwide Object[] arrayOfInstances() { return instances.toArray(); }

    /** constructor */
    Element() {
        instanceCounter++;
        instances.add(this);
    }

    pure String readSomething() { /* ... */ }
    void changeSomething() { /* ... */ }

    static void changeAll() {
        classwide Object[] elements = Element.arrayOfInstances();
        for (int i = 0; i < elements.length; i++) {
            elements[i].changeSomething();
        }
    }
}
```

Quellcode 3.5: Eine Klasse Element registriert alle Instanzen in einer classwide Collection. Analog zu Quellcode 1.2 aus Kapitel 1 aber mit Schlüsselwort classwide.

```java
void main() {
    rep Element e1 = new rep Element();
    rep Element e2 = new rep Element();
    readonly Object[] elements = Element.arrayOfInstances();
    for (int i = 0; i < elements.length; i++) {
        elements[i].readSomething(); // ok.
        elements[i].changeSomething(); // fail!!
    }
    Element.changeAll(); // ok, but dangerous.
}
```

Quellcode 3.6: Ein main()-Programm verwendet Klasse Element aus Quellcode 3.5

Diskussion

Dieser Ansatz bietet eine Möglichkeit, beschränkten Zugriff auf Klassenvariablen zu ermöglichen. Diese Eigenschaft ist jedoch schwierig formal zu typisieren und somit zur Kompilezeit zu testen.

Problematisch ist auch, dass classwide-Referenzen auf Objekte aus einem fremden Universum zeigen können. In Abbildung 3.4 zeigen die Knoten der Collection instances zum Beispiel direkt auf die Elemente i und j, ohne durch den Besitzer, in diesem Fall MyProgram zu gehen. Wenn zum Beispiel die Invariante von MyProgram von diesen Elementen abhängt, kann diese durch den Aufruf einer statischen Methode, wie changeAll() aus dem Quellcode 3.5, verletzt werden.
**Dependency Control**  Bei diesem Ansatz muss die Menge der Objekte, von denen eine Klassen-Invariante abhängen darf um *classwide*-Referenzen eingeschränkt werden. Die Invarianten dürfen nicht von *classwide*-Referenzen abhängen.

Ungeklärt ist, wie sich dieser Ansatz in Bezug auf Subklassen und Vererbung verhalten würde.

**Abschliessend**  kann man sagen, *classwide*-Referenzen sind ein einfach zu verstehendes Prinzip, bergen jedoch zu viele Gefahren und bieten nur für spezielle Probleme, wie zum Beispiel Klassenweite Referenzen (Abschnitt 1.5.2), eine befriedigende Lösung.

### 3.3 readwrite Schlüsselwort

Dieser Ansatz beschreibt eine Möglichkeit, statische Felder les- und schreibbar zu markieren. Dazu wird das Schlüsselwort *readwrite* eingeführt.

Auf sichtbare Felder, welche *readwrite* deklariert sind, kann von überall her zugegriffen werden, als wären es *peer*-Referenzen.

Beispiele für die Anwendung von *readwrite*-Feldern ist das Singleton Pattern (Abschnitt 1.5.4) oder applikationsweite Einstellungen\(^1\).

Hier als Beispiel ein Ausschnitt aus einer Applikation mit GUI und mehreren Fenstern:

**Beispiel**

In einer Klasse `Window` (Quellcode 3.8) ist ein statisches *readwrite*-Feld `activeWindow` deklariert, welches immer auf das aktive Fenster des Programms referenziert. Sonst ist von der Implementati-|on uns nur noch die Methode `close()` und das zugehörige Statusfeld `boolean closed` von Interesse.

Ein `User` (Quellcode 3.9) benutzt ein `Window` und schliesst in einer Methode `work()` das vorher active `Window`. Dazu benutzt er das statische *readwrite*-Feld `activeWindow` aus der Klasse `Window`. Anschliessend registriert er sein Fenster als aktiv und macht irgendetwas.

Ein Beispielprogramm führt `main()` (Quellcode 3.10) aus. Bei der Position *snapshot* haben wir eine Objekt-Struktur, wie in Abbildung 3.7 dargestellt. Daraus ist ersichtlich, dass `u1.myWindow` in seinem statischen *readwrite*-Feld eine Referenz auf `u2.myWindow` hat. Schreibzugriffe über diese Referenz, wie zum Beispiel `activeW.close()`, können erfolgen, ohne dabei über den Besitzer des

\(^1\)z.B. unter Verwendung von java.util.Properties.
Abbildung 3.7: Objekt-Struktur des User-Window-Beispiel mit readwrite-Referenz

Quellcode 3.8: Klasse Window mit statischer Referenz auf das aktive Fenster.

betroffenen Objekts zu gehen. Dies ist eine Verletzung der Invariante des Universe Type Systems (siehe auch Abschnitt 2.1.3). Das heisst in unserem Beispiel, User u1 kann nicht verhindern, dass sein Fenster geschlossen wird; bei der zweiten Ausführung von u1.work() (Quellcode 3.10), müsste u1 sich zuerst über den Zustand seines eigenen Fensters informieren.

Vorteile

• Es wird verlangt, dass statische Felder, auf welche Schreibzugriff ermöglicht werden soll, als das deklariert werden (readwrite).

• Es gibt keine Einschränkungen des Zugriffs durch Klassengrenzen, wie im classwide-Ansatz zuvor (Abschnitt 3.2).

• Dieser Ansatz unterstützt folgende Beispiele aus Kapitel 1.5: Konstanten (Abschnitt 1.5.1),
KAPITEL 3. VERSCHIEDENE ANSÄTZE

```java
class User {
    rep Window myWindow = new Window();
    void work() {
        /* before working */
        readwrite Window activeW = Window.activeWindow;
        if (activeW != null) activeW.close();
        Window.activeWindow = myWindow;
        /* do something. */
    }
}
```

Quellcode 3.9: Anwender der Klasse Window (Quellcode 3.8).

```java
void main() {
    rep User u1 = new rep User();
    rep User u2 = new rep User();
    u1.work();
    u2.work();
    // * snapshot*
    u1.work();
}
```

Quellcode 3.10: main()-Programm für die Klassen Window (Quellcode 3.8) und User (Quellcode 3.9).

Globale Referenzen (Abschnitt 1.5.3), Singleton Pattern (Abschnitt 1.5.4)

Nachteile


- Beim Zugriff auf ein Objekt via ein readwrite-Feld, kann der Besitzer des Objekts übergangen werden. Damit wird die Invariante des Universe Type Systems verletzt (siehe Abschnitt 2.1.3).

- Klassenweite Referenzen (Abschnitt 1.5.2) werden zwar unterstützt, aber die Einschränkung, dass nur aus der entsprechenden Klasse zugegriffen werden kann verfällt.
3.4 Zusätzliches global-Universum

Im letzten Ansatz (3.3) haben wir gesehen, dass zum Verändern von Objekten mit Hilfe von readwrite-Referenzen der Besitzer von Objekten übergangen werden kann. Um dies zu vermeiden wurde folgender Ansatz entwickelt:


Im Unterschied zu den herkömmlichen Universen, können Schreibreferenzen von irgendwoher ins global-Universum zeigen. Da kein Objekt sicherstellen kann, dass es das einzige ist, das auf ein global-Objekt zugreift, darf auch die Invariante nicht von global-Objekt abhängen. Dependency Control, wie in Abschnitt 2.1.2 beschrieben schliesst also global-Objekt irgendwo im Pfad von Invarianten aus.

global-Referenz können also als statische Felder, als Instanz-Felder oder als lokale Variablen verwendet werden. Das ist nötig, damit zu jedem Zeitpunkt, insbesonders zu Kompilezeit, zu jeder Referenz bestimmt werden kann, ob sie global ist, also ins global-Universum zeigt, oder nicht.

Abbildung 3.11: Auf global Objekte kann von überall her lesend und schreibend zugegriffen werden.
Beispiel

Das Singleton-Pattern wird mit global-Universum implementiert, wie in Quellcode 3.12 gezeigt wird. Wichtig dabei ist, dass bei der Instanzierung des Singleton-Objekts, also in der Methode getInstance() das Schlüsselwort global verwendet wird. Nur so wird das Objekt auch im global-Universum angelegt.

Wird dann die Methode main() aus Quellcode 3.13 ausgeführt, ergibt sich eine Objekt-Struktur, wie in Abbildung 3.11 dargestellt ist.

class Singleton {
    /** hold the one and only instance */
    static global Singleton instance = null;
    /** avoid instantiation from outside */
    protected Singleton() {}
    /** provide the one and only instance */
    public static global Singleton getInstance() {
        if (instance == null) instance = new global Singleton();
        return instance;
    }
    private global Object globalO = null;
    global Object getGlobalObject() { return globalO; }
    void setGlobalObject(global Object o) { globalO = o; }
    private readonly Object readonlyO = null;
    readonly Object getReadOnlyObject() { return readonlyO; }
    void setReadOnlyObject(readonly Object o) { readonlyO = o; }
}

Quellcode 3.12: Singleton-Pattern mit global-Universum: das Singleton wird als global-Objekt instanziert.

void main() {
    global Singleton s = Singleton.getInstance();
    s.setReadOnlyObject(new peer Object());
    readonly Object o1 = s.getReadOnlyObject(); // gives readonly reference.
    s.setGlobalObject(new global Object());
    global Object o2 = s.getGlobalObject(); // gives global reference.
    s.setReadOnlyObject(new rep Object()); // ok.
    s.setGlobalObject(new rep Object()); // fail!
    s.setGlobalObject(this); // fail! (this is peer)
}

Quellcode 3.13: main()-Programm, das das global Singleton verwendet (Quellcode 3.12).

Diskussion

Repräsentations-Kapselung und Dependency Control bleiben erhalten für alle nicht-global-Referenzen, da ein Objekt, das nicht als global instanziert wurde, immer noch den Regeln des Universe Type System unterstellt ist und nie global werden kann. Diese Eigenschaft folgt aus der Definition von global (siehe Abschnitt 4.1.2) und daher, dass weder an peer noch an rep-Variablen global-Objekte zugeordnet werden können.

Folgende Beispiele aus Kapitel 1.5 sind unterstützt: Globale Referenzen (Abschnitt 1.5.3) und Singleton Pattern (Abschnitt 1.5.4). Klassenweite global-Objekt können zwar implementiert werden, aber die Einschränkung, dass nur aus der entsprechenden Klasse zugegriffen werden kann, verfällt. Das Konzept von Klassenweite Referenzen (Abschnitt 1.5.2) ist also nicht unterstützt.

Dependency Control für global-Referenzen kann nicht erreicht werden, das heisst, Invarianten dürfen nicht von global-Objekt abhängen, da diese auch von anderen Programmteilen verändert werden können.

Innerhalb von global-Objekten, kann Repräsentations-Kapselung sichergestellt werden, da die globalen Objekte den gleichen Regeln unterstellt sind, wie alle anderen Objekte auch.

Aufpassen muss man, wenn ein inneres Objekt, via einer global-Referenz auf einen sich übergeordneten Besitzer gelangen kann. Diesen Fall nenne ich "rekursive global-Referenzen", er wird in Abschnitt 4.7 genauer erläutert.
Kapitel 4

Einbindung ins Universe Type System

Im letzten Kapitel wurden verschiedene Möglichkeiten dargestellt, wie statische Felder ins Universe Type System miteinbezogen werden könnten. Schaut man die Vor- und Nachteile der verschiedenen Ansätze an, kann man eine Kombination von Ansatz 3.1 und Ansatz 3.4 als Lösung darstellen:

4.1 Lösung mit neuem global Universum

4.1.1 Default für statische Felder ist readonly

Statische Felder werden per default als readonly typisiert. Der readonly-Typ ist Supertyp jedes bisher existierenden Types im Universe Type System und es gibt für ihn keine Einschränkungen bezüglich Überschreitungen von Kontext-Grenzen.

Wird also ein statisches Feld deklariert, wird automatisch der Typ readonly verwendet.

```java
static (readonly) Foo f = new peer Foo();
```

Ein Zugriff, wie zum Beispiel

```java
f.change();
```

ist nicht erlaubt.
4.1.2 global: Schlüsselwort und Universum

Um Lese- und Schreib-Referenzen zu ermöglichen, wird ein neues Universum global eingeführt. In diesem Universum befinden sich alle Instanzen, welche mit dem Schlüsselwort global erzeugt wurden. Zum Beispiel in einer Klasse Foo:

```java
static global Foo f = new global Foo();
```

Abbildung 4.1: Objekt f vom Typ global Foo im global-Universum


In Abbildung 4.1 kann zum Beispiel b: Bar über seine global-Referenz globalFoo voll auf das Objekt f: Foo zugreifen.

**Definition** Ein Objekt V ist genau dann global, wenn es mit dem Schlüsselwort global instanziert wurde. Alle Schreibreferenzen von einem Objekt U auf das global-Objekt V müssen global deklariert werden, oder peer, wenn U global ist.

**Subtyp-Beziehungen der Ownership-Typen**

4.2 Beispiel – globaler Cache


class Client {
    int myId;
    global Cache gCache;
    /* constructor */
    Client(int id, global Cache gc) {
        myId = id; gCache = gc;
        gCache.put("key" + myId,
            new global StringBuffer("my personalized SB: "+ myId);
            ((StringBuffer) gCache.get("log")).append("constructclient "+ myId);
        }
    void run() {
        ((StringBuffer) gCache.get("log")).append("runclient "+ myId);
    }
}

main() {
    global Cache gc = new global Cache();
    gc.put("log", new global StringBuffer("starting out");
    rep Client c1 = new rep Client(1, gc);
    rep Client c2 = new rep Client(2, gc);
    c1.run(); c2.run();
}

Quellcode 4.3: class Client und main()-Programm() zum global Cache in Quellcode 4.4

Die Applikation (main()-Programm in Quellcode 4.3) instanziiert einen Cache (Quellcode 4.4) im global-Universum. Diesen globalen Cache gibt sie als Parameter an die entsprechenden Clients weiter.
import java.util.Map;
class Cache {
    Map cache = new Map();

    /** @returns the Object previously in cache for given key or null. */
    Object put(readonly Comparable key, Object value) {
        return cache.put(key, value);
    }

    Object get(readonly Comparable key) {
        return (Object) cache.get(key);
    }

    Object remove(readonly Comparable key) {
        return (Object) cache.remove(key);
    }
}

Quellcode 4.4: Implementation eines globalen Caches.

Die Klasse Cache ist als normaler Cache implementiert. Also zu Entwicklungs- und Kompilezeit ist noch unbekannt, dass Cache als globaler Cache verwendet wird. Das funktioniert, weil die Methoden, die peer zurückgeben, aufgerufen auf einem global-Objekt gemäß TypCombinator (Abschnitt 4.5) global zurückgeben. (Fall global * peer : global)

Da alle Objekte, die im globalen Cache gespeichert sind, global sind, können alle Clients darauf schreibend zugreifen. Nachdem das main()-Programm aus Quellcode 4.3 ausgeführt wurde, entsteht eine Objekt-Struktur, wie in Abbildung 4.5 dargestellt ist.

Anmerkung: Die Schlüssel ("log", "key", ..) könnten auch als statische Felder in den jeweiligen Klassen gespeichert werden und wären somit automatisch für die anderen readonly abrufbar (dabei ist es egal ob sie final oder veränderbar sind).
Abbildung 4.5: Objektstruktur nachdem das main()-Programm aus Quellcode 4.3 ausgeführt wurde.
4.3 Beispiel – Foo Bar

Als Beispiel eine Klasse Foo (Quellcode 4.6), welche ein statisches Feld global Foo staticFoo und ein Instanzfeld readonly Foo otherFoo hat.

class Foo {
    static global Foo staticFoo = null;
    readonly Foo otherFoo = null;
    private int i = 0;
    void change() { i++; }
}

Quellcode 4.6: Klasse Foo

Auf das statische Feld Foo.staticFoo kann von überall \(^1\) her zugegriffen werden, solange dies nicht durch einen access modifiereingeschränkt wird.

Eine andere Klasse Bar (Quellcode 4.7) verwendet Foo: Sie hat ebenfalls zwei Felder, welche auf ein Objekt Foo verweisen. Das eine Feld (rep Foo repFoo) gehört zur internen Repräsentation und somit zum Universum von der jeweiligen Instanz von Bar, während das Feld global Foo globalFoo eine Referenz auf ein Foo-Objekt im globalen Universum ist.

class Bar {
    rep Foo repFoo = new rep Foo();
    global Foo globalFoo = new global Foo();

    void do() {
        globalFoo.otherFoo = repFoo;
        Foo.staticFoo = this.globalFoo;
    }
}

Quellcode 4.7: Klasse Bar

void main() {
    rep Bar bar1 = new rep Bar();
    rep Bar bar2 = new rep Bar();

    bar1.do();
    bar2.do();
}

Quellcode 4.8: main()-Programm für das Foo-Bar-Beispiel

\(^1\)in diesem Beispiel sind alle Felder public, wenn nicht explizit anders angegeben.
Nachdem aus dem Quellcode `main()` ausgeführt worden ist haben wir im Speicher eine Objektstruktur, wie in Abbildung 4.9 dargestellt wird.

Abbildung 4.9: Objektstruktur nachdem `main()` ausgeführt wurde. (’static global’ ist kein Typ, sondern nur in dieser Darstellung grau eingefärbt um statische Referenzen zu markieren.)

In Abbildung 4.9 sieht man schön, wie Referenzen mit Schreibrecht von Instanzuniversen ins `global`-Universum zeigen, jedoch nicht umgekehrt. So ist zum Beispiel `bar2.otherFoo` vom Typ `readonly Foo`.

Außerdem fällt auf, dass vom Objekt `bar2.globalFoo` im `global`-Universum eine Schreibreferenz auf sich selber zeigt. Dieser Spezialfall, einer `global`-Referenz aus dem `global`-Universum wird in Abschnitt 4.7 nochmals aufgegriffen.
4.4 Zwei Varianten der Veranschaulichung

Es gibt mehrere Möglichkeiten, wie das global-Universum dargestellt werden kann. Zwei Naheliegende werden im folgenden erläutert und auf den Unterschied eingegangen:

(Der Quellcode zu den Darstellungen ist im Anhang, Seite 44, Quellcode A.1, A.2 und A.3 zu finden.)

4.4.1 global-Universum separat von den Instanz-Universen

Abbildung 4.10: Darstellung mit global-Universum neben den Instanz-Universen

Das ist die Darstellung, die ich in dieser Arbeit verwende und auch in Abbildung 4.10 zu sehen ist. Das global-Universum befindet sich neben den Instanz-Universen; die beiden Arten von Universen sind überschneidungsfrei dargestellt.

global-Referenz zeigen immer auf ein beliebiges, äußeres Objekt im global-Universum.

Bei dieser Darstellung ist gut sichtbar, wenn rekursive global-Referenzen existieren. Das ist genau dann, wenn global-Referenz vom global-Universum ausgehen. (Siehe auch Abschnitt 4.7)

4.4.2 global-Universum umschliesst die Instanz-Universen

Bei einer Darstellung, wo das global-Universum alle Instanz-Universen umschliesst (Abbildung 4.11), ergibt sich eine ähnliche Situation, wie bekannt aus anderen Ownership-Typsystems.
Abbildung 4.11: Darstellung, wenn global-Universum die Instanz-Universen umschliesst


rep-Objekte des Hauptprogramms unterscheiden sich in dieser Darstellung nicht von den rep-Objekten eines globalen Objekts.

Das Hauptprogramm wird implizit auch als globales Objekt betrachtet.

Diskussion


Wird das Hauptprogramm, oder der statische Teil einer Klasse, von dem eine Methode ² als Hauptprogramm ausgeführt wird, im global-Universum instanziert, können alle Objekte, die eine Referenz darauf haben dieses auch verändern. Da zu Kompilezeit jedoch nicht bekannt ist, ob diese Methode ² als Hauptprogramm ausgeführt wird und somit Teil des global-Universum wird oder nicht, bringt es keinen Vorteil, wenn es in einer Darstellung so aussieht, als wäre das Hauptprogramm global.

²public static void main(String[] args)
KAPITEL 4. EINBINDUNG INS UNIVERSE TYPE SYSTEM

Aus diesen Gründen verwende ich in dieser Arbeit die Darstellung mit dem global-Universum neben den anderen Universen, wie auch in Abbildung 4.10 zu sehen ist.

4.5 Typ-Kombinator

Der Typ-Kombinator aus dem Universe Type System beschreibt, was für ein Typ eine Referenz v.f hat, gegeben ein Objekt v mit einem Feld oder einer Methode f. Mit der Einführung eines neuen Typ global, muss auch dieser Kombinator erweitert werden. Der erweiterte Kombinator ist in Abbildung 4.12 dargestellt.

<table>
<thead>
<tr>
<th>peer</th>
<th>rep</th>
<th>readonly</th>
<th>global</th>
</tr>
</thead>
<tbody>
<tr>
<td>peer</td>
<td>rep</td>
<td>readonly</td>
<td>global</td>
</tr>
<tr>
<td>rep</td>
<td>undefined</td>
<td>readonly</td>
<td>global</td>
</tr>
<tr>
<td>readonly</td>
<td>readonly</td>
<td>readonly</td>
<td>readonly</td>
</tr>
<tr>
<td>global</td>
<td>undefined</td>
<td>readonly</td>
<td>global</td>
</tr>
</tbody>
</table>

* nur auf this erlaubt
* nicht erlaubt, da dadurch eine Referenz in ein Subuniversum entstehen würde
\[
\text{[MPH01, Abschnitt 4.1]}
\]
* readonly ist transitiv
* in Anlehnung an rep-rep siehe Ausführungen in Abschnitt 4.5

Abbildung 4.12: Typkombinator inklusive dem global-Typ

**Fall global - peer**  Da das Objekt v global ist, zeigen seine peer Referenzen ebenfalls ins global- Universum.
Dies ermöglicht, dass eine Klasse sowohl als rep- wie auch als global-Objekt instanziiert werden kann, denn für ein global-Objekt zeigt eine peer-Referenz wieder auf ein global-Objekt (vgl Beispiel "GlobalCache" Abschnitt 4.2).

**Fall global - rep**  Wäre es zugelassen, eine Referenz auf ein internes Objekt eines global-Objekt zu kriegen, wäre Repräsentations-Kapselung der global-Objekt nicht mehr sichergestellt.

**Fall readonly - global**  Weder die Invariante (Abschnitt 4.6) noch die Definition von global (Abschnitt 4.1.2) verbieten es, dass eine global-Referenz durch eine readonly-Referenz erhalten werden kann. Allerdings würde dadurch die Transitivität von readonly verloren gehen. Deshalb hier der Rückgabe-Typ readonly.
(Dies haben wir willkürlich entschieden. Falls trotzdem eine global-Referenz benötigt wird, kann gekastet werden.)
**Fall** global - global  Aus den gleichen Gründen, wie Fall peer - peer, bleibt hier der Referenz-Typ unverändert global. Der Spezialfall von rekursiven global-Referenzen wird in Abschnitt 4.7 behandelt.

### 4.6 Erweiterung der Invariante des Universe Type System

Die Invariante des Universe Type System, wie sie in Abschnitt 2.1.3 und [MPH01] beschrieben wird, muss angepasst werden, denn global-Referenz sind ein zusätzlicher Fall, wann eine Referenz nicht readonly ist.

Werden im Universe Type System global-Referenz erlaubt, wie sie in diesem Kapitel beschrieben sind, so gilt in jedem Ausführungsschritt eines Programms:

**Invariante**  Falls ein Objekt $U$ eine direkte Referenz auf ein Objekt $V$ hat, ist entweder

- $U$ der Besitzer von $V$ oder
- $U$ und $V$ gehören zum gleichen Universum oder
- $V$ ist global oder
- die Referenz ist readonly.

Hier nochmals die Definition von global:

**Definition**  Ein Objekt $V$ ist genau dann global, wenn es mit dem Schlüsselwort global instanziert wurde. Alle Schreibreferenzen von einem Objekt $U$ auf das global-Objekt $V$ müssen global deklariert werden, oder peer, wenn $U$ global ist.
4.7 Problem mit rekursiven global-Strukturen

Ruft ein rep-Objekt \( r \) eines global-Objekts \( g \) über eine global-Referenz eine Methode auf seinem Besitzer auf, könnte es passieren, dass dieser wiederum \( r \) aufruft. Zum Zeitpunkt, wo \( r \) das Objekt \( g \) aufgerufen hat, war es am arbeiten einer Methode und ist nicht verpflichtet, die Klassen-Invariante aufrecht zu halten. So kann es vorkommen, dass \( g \) das Objekt \( r \) in inkonsistentem Zustand vorfindet, obwohl es eine nicht-pure Methode aufruft und somit das Objekt \( r \) in einem konsistenten Zustand erwarten dürfte.

Beispiel In Abbildung 4.13 wurde eine Methode vom Objekt repFoo durch das global-Objekt staticBar aufgerufen. Während der Ausführung der Methode kann repFoo seine Invariante brechen und über MyProgram.staticBar, welches eine static global Referenz ist, staticBar aufrufen. Falls jetzt staticBar erneut auf repFoo zugreifen will, haben wir den problematischen Fall.
4.8 Zusammenfassung

Die Einführung des Schlüsselwortes `global` ermöglicht, dass Objekte in einem separaten `global`-Universum instanziert werden. Jedes Programmstück, das eine `global`-Referenz auf ein solches Objekt hat, kann darauf lesend und schreibend zugreifen. Solange statische Felder nicht als `global` deklariert werden, werden sie implizit als `readonly` behandelt. Dies schützt vor Aliasing und Leaking und knüpft an die Flexibilität an, die das Universe Type System bereits bietet [MPH01].

Da `global`-Objekte auch von anderen Programmteilen verändert werden können, dürfen Klasse-Invarianten nicht von ihnen abhängig sein. Dies ist eine weitere Restriktion zur Dependency Control aus dem Universe Type System (vgl. Abschnitt 2.1.2).

Repräsentations-Kapselung für die Typen `rep` und `peer` bleibt unverändert bestehen. Auch für die interne Repräsentation von `global`-Objekten gilt Repräsentations-Kapselung.

Transitivität von `readonly` bleibt erhalten. Die Flexibilität, die das Universe Type System mit `readonly`-Referenzen anbietet, bleibt erhalten, da `global` ebenfalls ein Subtyp von `readonly` ist.

Vergleich mit Fällen aus Abschnitt 1.5

- Da der default Typ für statische Felder `readonly` ist, können Konstanten (Abschnitt 1.5.1) einfach implementiert werden.

- Für Globale Referenzen (Abschnitt 1.5.3) und das Singleton Pattern (Abschnitt 1.5.4) bietet das Konzept von `global`-Objekten eine flexible und typsichere Lösung.

- Klassenweite Referenzen (Abschnitt 1.5.2) können implementiert werden, das TypSystem bietet jedoch keinen Schutz gegen Aliasing außerhalb der Klasse. Insofern ist dieser Fall nicht unterstützt.

Der Ownership-Type kann so, wie er in diesem Kapitel beschrieben wird in einem Kompiler implementiert werden (siehe Kapitel 5).
Kapitel 5

global im Universes Kompiler

Die SCT Group der ETH Zürich implementiert das Universe Type System in MultiJava [mul], einem Kompiler für Java mit Erweiterungen zur Java Programmiersprache.

Im Rahmen dieser Semesterarbeit ergänzte ich diese Implementation um das global-Universum.

5.1 Implementation

5.1.1 Neues Schlüsselwort global

Die Liste aller Schlüsselwörter (File MjcID.t) wurde ergänzt durch global (vgl Quellcode 5.1).

/* WMD */
@literal peer
@literal readonly
@literal rep
@literal global
@literal pure

Quellcode 5.1: Die Liste der Schlüsselwörter wurden ergänzt durch global.

5.1.2 Grammatik

Die Grammatik musste an einigen Orten angepasst werden, da das global-Universum ebenfalls als Universum akzeptiert werden muss.

Quellcode 5.2 ist der Abschnitt, der für das global-Universum neu dazu kam.
Quellcode 5.2: Die Grammatik in der Datei Mjc.g des MultiJava-Kompilier wird ergänzt durch die beiden Abschnitte für das global-Universum.

5.1.3 CUniverseGlobal.java

Die Klasse CUniverseGlobal.java repräsentiert das global-Universum. Sie implementiert das Singleton-Pattern; dies bleibt auch so, wenn später peer- und rep-Universen mal parametrisiert werden sollten, da es genau ein global-Universum gibt.

Ansonsten ist die Klasse analog zu CUniverseRep.java und CUniversePeer.java aufgebaut und implementiert die abstrakte Klasse CUniverse.java.

Die ganze Klasse CUniverseGlobal.java ist im Anhang im Quellcode A.4 zu finden.

5.1.4 CUniverse.java

In dieser Klasse musste die statische Methode public static CUniverse combine(CUniverse first, CUniverse second) angepasst werden. Darin wird der Typ-Kombinator (vgl Abschnitt 4.5) implementiert. Die entsprechenden Fälle mussten ergänzt werden.

Die Methode public static CUniverse combine(CUniverse first, CUniverse second) ist im Anhang im Quellcode A.5 zu finden.
Literaturverzeichnis


LITERATURVERZEICHNIS


Anhang A

Einige Details

A.1 grep java.util

Das Packet java.util habe ich schlussendlich mit folgendem Befehl durchsucht:
```
%> grep -r static . | grep -v final| grep -v native | grep -v class | grep 'java/util' | grep -v '(
```

- alle Zeilen mit 'static' (rekursiv aus dem aktuellen Verzeichnis).
- Zeilen mit 'final' interessieren mich nicht, das sind Konstanten.
- Zeilen mit 'native' interessieren mich nicht, da es nur eine Verlinkung zu Code einer anderen Programmiersprach sind.
- Klassendefinitionen interessieren mich auch nicht.
- Methodendefinitionen interessieren mich auch nicht, diese habe ich mit grep -v '(' ausgeschlossen.
A.2 Quellcode zu Beispiel in Abschnitt 4.4

Quellcode A.1, A.2 und A.3

class MyObject {
    global Object globalObject = null;
}

Quellcode A.1: Klasse MyObject

class Singleton {
    /** hold the one and only instance */
    static global Singleton instance = null;
    /** avoid instantiation from outside */
    protected Singleton () {} 
    /** provide the one and only instance */
    public static global Singleton getInstance () {
        if (singleton == null) instance = new Singleton();
        return instance;
    }
    global Object globalO = null;
    readonly Object readonlyO = null;
}

Quellcode A.2: Singleton-Implementation, wie in den Abbildungen zum Abschnitt 4.4 verwendet.

void main () {
    rep MyObject repO = new rep MyObject();
    repO.globalObject = Singleton.getInstance ();
    Singleton.getInstance ().readonlyObject = repO;

    global MyObject globalO = new global MyObject();
    globalO.globalObject = Singleton.getInstance ();
    Singleton.getInstance ().globalObject = globalO;
}

Quellcode A.3: main()-Programm zum Singleton in Quellcode A.2
A.3 Quellcode von CUniverseGlobal.java

Der Quellcode aus dem MultiJava-Kompiler (vgl Kapitel 5) ist in Quellcode A.4 aufgelistet.

A.4 Quellcode von Methode combine(..) aus CUniverse.java

Der Quellcode der Methode public static CUniverse combine(CUniverse first, CUniverse second) ist in Quellcode A.5 dargestellt. Diese Methode implementiert den Typ-Kombinator, der in Abschnitt 4.5 behandelt wird.
/** Combine two given universes and return the result universe.
 * @param first The first/left hand side universe.
 * @param second The second/right hand side universe.
 * @return The combined result universe.
 */
public static CUniverse combine(CUniverse first, CUniverse second) {
    if (first instanceof CUniverseReadOnly ||
        second instanceof CUniverseReadOnly ) {
        return CUniverseReadOnly.getUniverse();
    }
    else if ( (first instanceof CUniverseRep ||
               first instanceof CUniverseGlobal) && // TH
                second instanceof CUniverseRep ) {
        // rep * rep is not allowed!
        // global * rep either not.
        return null;
        // at one point we discussed that
        // rep * rep could be changed to readonly.
        // return CUniverseReadOnly.getUniverse();
        // but this would be rather un-intuitive for programmers imho.
    }
    // TH
    else if (first instanceof CUniverseGlobal ||
              second instanceof CUniverseGlobal ) {
        return CUniverseGlobal.getUniverse();
    }
    else if (first instanceof CUniverseRep ||
              second instanceof CUniverseRep ) {
        return CUniverseRep.getUniverse(null);
    }
    else if (first instanceof CUniversePeer &&
              second instanceof CUniversePeer ) {
        return CUniversePeer.getUniverse();
    } else {
        // nothing should really get here
        return null;
    }
}

Quellcode A.5: Methode `combine(..)` aus der Klasse `CUniverse` aus dem MultiJava-Kompilator implementiert den Typ-Kombinator (beschrieben in Abschnitt 4.5).