FACHBEREICH INFORMATIK
LEHRGEBIET PROGRAMMIERSYSTEME
Prof. Dr. Friedrich Steimann

Abschlussarbeit im Studiengang Informatik

Abschluss Bachelor

Kapselung von Objekten in objektorientierten Programmiersprachen

Vorgelegt von

Mario Bertschler
Steinteilweg 5
A-6800 Feldkirch
Matrikelnummer 6435653
mario.bertschler@gmx.at

Feldkirch, den 21. Dezember 2005

Zusammenfassung

Oft ist die objektorientierte Programmierung aus Sicherheitsgründen vermieden worden. Die Ursache der Schwachstellen eines in einer objektorientierten Sprache geschriebenen Programmes liegt oft in der unkontrollierten Verwendung von Aliasen eines Objekts, die in unterschiedlichen Programmteilen liegen. Dabei kommt es oft zu unerwarteten Seiteneffekten, der Überblick über das System geht verloren und externe Angreifer können diese Schwachstellen ausnutzen. Umso erstaunlicher ist es, dass es bei allen bekannten objektorientierten Programmiersprachen keine direkte Implementierung für Kapselungsmechanismen gibt, die das Alias-Problem lösen könnten. Dabei gab es schon seit Anfang der 90er Jahre Lösungsansätze, die bis heute weiterentwickelt wurden und immer weiter ausreifen.

In dieser Arbeit werden zunächst die vom Alias-Problem ausgehenden Gefahren aufgezeigt. Weiters wird die Zugriffskontrolle auf Objekte genauer untersucht. Es wird dabei gezeigt, dass übliche Sichtbarkeitsmechanismen bei weitem nicht ausreichen, um Objekte vor unerlaubten Objektzugriff zu schützen. Außerdem werden wichtige Kapselungsansätze diskutiert und untereinander verglichen. Dabei dient die jeweilige Kapselungsfunktion dazu, den Wirkungsgrad dieser Mechanismen abzuschätzen. Schließlich wird versucht zu eruieren, in welcher Form Kapselungsmechanismen in Zukunft eingesetzt werden könnten.

Danksagung

Mein besonderer Dank gilt Herrn Prof. Dr. Friedrich Steimann, der mir die Möglichkeit gab, meine Bachelor-Abschlussarbeit über ein sehr spannendes Thema in der Programmierung zu schreiben. Seine Ratschläge über empfehlenswerte Literatur, Aufbau der Arbeit und Querverweise auf ähnliche Inhalte in anderen Themenbereichen waren sehr wertvoll, um die unterschiedlichen Aspekte dieser Arbeit in einen Kontext bringen zu können.

Außerdem möchte ich Frau Dr. Daniela Keller für ihre große Hilfestellung in der Gestaltung etlicher Textformulierungen und für ihre mathematische Unterstützung herzlich danken.

Inhaltsverzeichnis

1 Einführung

Zu den wichtigen Konzepten in der objektorientierten Programmierung gehört das Konzept der Kapselung. Die Vorteile der Kapselung von Programmteilen sind bessere Änderbarkeit, Wiederverwendbarkeit und Übersichtlichkeit des Quellcodes und bessere Testbarkeit des entwickelten Programmes. Eine andere Frage ist die der *Sicherheit* des während der Laufzeit erzeugten Codes. In dieser Arbeit werden daher die derzeitigen Kapselungsmechanismen diesbezüglich untersucht.

Unter Sicherheit wird dabei verstanden, wie die Zugriffbeschränkungen auf Objekte kontrolliert werden. Herkömmliche Zugriffsmechanismen benützen dazu meistens Sichtbarkeitsrestriktionen für Objektnamen, wie `private` oder `protected`, um einen Namensraum festzulegen, indem das Objekt sichtbar ist. Außerhalb dieses Namensraums ist das Objekt versteckt. Das wird in der Programmierung als Information Hiding bezeichnet.

Mit Mitteln des Information Hiding kann somit eine begrenzte Kapselung erreicht werden. Die reine Geheimhaltung des Namens eines Objektes ist aber keine sichere Methode, um den Zugriff auf ein Objekt tatsächlich zu verhindern, da Aliase bestehen können, die den Zugriff über eine Hintertür offen halten.

1.1 Das Alias Problem

> ...for many centuries astronomers used the distinct terms “evening star” and “morning star” without realizing that both referred to one object, the planet Venus.[HLW+92]

Eines der größten Probleme in der objektorientierten Programmierwelt ist, wann ein Objekt ein Alias besitzen darf und wann nicht, bzw., wann ein Objekt eine Referenz übergeben sollte und wann eine Kopie des Objekts. Immer dann, wenn es von außen eine Referenz auf ein Objekt gibt, besteht die potenzielle Gefahr, dass dieser Zugang zum Objekt in einer gefährlichen Art angewendet wird.

Kapselungsmechanismen bieten Sicherheit im Sinne von Schutz gegen *Unachtsamkeiten* des Programmierers, die Abhängigkeiten im Programm entstehen lassen, die unerwünschte Seiteneffekte erzeugen können, und Sicherheit im Sinne von Schutz gegen mutmaßliche *Angreifer von außen*, die Schwachstellen im System suchen, um Zugang zu vertraulichen Informationen zu bekommen. Da aktuelle objektorientierte Programmiersprachen nur geringe bis gar keine Mechanismen zur Kapselung bieten, liegt es meist am Programmierer, Gefahren zu vermeiden.

1.1.1 Gefahren durch Angreifer von außen

Ein gutes und viel zitiertes Beispiel hierfür ist die Sicherheitslücke in Java 1.1.1 [GJS96]. Durch die Übergabe eines Aliases der Liste der digitalen Unterschriften von Sun's Java Applets war es möglich, jedes Applet von außen als vertrauenswürdig zu deklarieren [AKC02], [PH02]. Die nächste Version von Java korrigierte den Fehler und übergab anstatt einer Referenz eine Kopie der Liste und somit war die Sicherheitslücke geschlossen.

```
1  public final class Class ... {
2    private Identity[] signers;
3    ...
4    public Identity[] getSigners() {
5      return signers; // Rückgabe einer Referenz
6                      // --> Sicherheitslücke
7    }
8  }
```

Die Rückgabe eines Arrays ist in Java im Gegensatz zu primitiven Datentypen immer eine Referenz zum Original und keine Kopie des Originals. Über die Funktion `System.arraycopy(...)` kann der Inhalt eines ganzen Arrays in einen neuen Array kopiert werden - damit wäre diese Sicherheitslücke geschlossen gewesen.

1.1.2 Gefahren durch Unachtsamkeit

Während der Programmentwicklung ist es für den Programmierer oft nützlich, viele Referenzen auf ein Objekt zu legen und sie zu benutzen. Sofern nicht syntaktisch eine Kapselung angelegt wurde (durch explizite Schnittstellendefinition), existiert semantisch jedoch immer eine solche, die die eigentliche Funktionalität des Objektes (auch Repräsentation eines Objekts genannt) von der Schnittstelle trennt. Leider geschieht es oft, dass solche nur semantisch festgelegten Invarianten vom Programmierer übersehen werden und z.B. interne Objekte herausgegeben werden, die von außen manipuliert werden können, oder beim Importieren von Objekten Abhängigkeiten entstehen. Dazu eine kleine Beispielklasse, welche den Import eines Motors in ein Auto erlaubt, ohne Maßnamen vorzunehmen, die sicherstellen, dass der Motor keine schon vorhandenen Aliase beibehält:

```
1  class Auto {
2    private Motor motor;
3    Auto(Motor m) { this.motor = m; }
4    Motor gibMotor(Motor m) { return motor; }
5  }
```

Der Motor ist Teil der Repräsentation eines Autos[1] und ist damit ein zu schützendes Objekt. Obwohl der Motor als `private` deklariert ist, stellt die Referenzübertragung von einem Parameter im Konstruktor zu einem Instanzobjekt, wie sie in der Zeile 3 durchgeführt wird, eine Gefahr dar. Ein unachtsamer Programmierer könnte nun die `Auto`-Klasse in einer Weise, wie in den nächsten Zeilen zu sehen ist, anwenden:

```
6   class Programm {
7     public static void main(String[] args) {
8       Motor m1 = new Motor();
9       Auto a1 = new Auto(m1);
10      Auto a2 = new Auto(m1);
11      a2.gibMotor().manipuliere();
12        // Motor in a1 wird ungewollt mitmanipuliert
13    }
14  }
```

Sofern der Motor eines Autos Referenzen von außen besitzt, kann er auch, ohne das Auto zu informieren, den Motor manipulieren. Dies wird durch `a2.gibMotor().manipuliere()` im obigen Beispiel in der Zeile 11 demonstriert. Da der Motor in der `Programm`-Klasse erzeugt wurde und der Motor ohne eine Kopie zu erstellen (weder in der `Programm`-Klasse noch in der `Auto`-Klasse) in das Auto importiert wird, besteht das `Motor`-Objekt weiterhin in der `Programm`-Klasse und kann Aliase erzeugen.

Desweiteren teilen sich zwei Objekte unkontrolliert ein drittes Objekt: Derselbe Motor `m1` wird in zwei unterschiedlichen Autos `a1` und `a2` importiert. Der fiktive Programmierer hat wahrscheinlich gedacht, dass die Motoren identisch sind und deshalb dasselbe Objekt benutzt werden kann. In dem Moment aber, wo der Motor in einem Auto manipuliert wird (`m1.manipuliere()`), wird automatisch und meistens ungewollt der Motor im anderen Auto mitmanipuliert. Wenn so eine gefährliche Referenz von mehreren Objekte geteilt wird, ist es schwierig, diesen Fehler bei der Fehlersuche zu finden.

Dieses kleine Beispiel ist relativ einfach zu überschauen, aber bei großen Projekten finden sehr komplexe Vorgänge statt, die den Überblick über die Referenzen in einem großen Maße erschweren. So kann es vorkommen, dass die unerwünschten Seiteneffekte erst bei einem dritten Objekt, das eigentlich keinen Kontakt mit dem eigentlichen Verursacher hat, auftreten. Das geschieht, wenn dieses dritte Objekt Abhängigkeiten zur zu schützenden Referenz enthält. In diesem Fall dürfte die Fehlersuche recht aufwändig sein.

[1] Eselsbrücke für die Erkennung von Repräsentationsobjekten: Funktioniert ein Objekt X ohne Unterobjekt Y? Ein Auto funktioniert *nicht* ohne Motor!

1.2 Schutz eines Objekts

Information Hiding, in Deutsch als „Geheimnisprinzip“ bezeichnet, wird oft mit Kapselung gleichgesetzt bzw. verwechselt. Das Geheimnisprinzip besitzt einen starken Bezug zu einem *Modul* im üblichen Sinne, das ein Behälter für Objekte ist, der aus einem *sichtbaren* und einem *unsichtbaren* Teil besteht. Es geht also um die Sichtbarkeit der Objekte einer Komponente, nicht aber um die grundsätzlichen Zugriffsrechte von anderen äußeren Objekten auf die Objekte einer Komponente. Kapselung unterliegt einer viel *schärferen Interpretation* und erfordert zusätzlich zur Geheimhaltung gewisser Informationen eine Festlegung von Zugriffsrechten auf Objekte innerhalb und außerhalb eines festgelegten Rahmens (Kapsel). Dabei sollte das Zugriffsrecht nicht nur für den Namen des Objekts gelten (*name protection*), sondern vielmehr für das Objekt selbst (*object protection*). In dieser Arbeit wird nur der Begriff der gekapselten Komponente benutzt. Eine gekapselte Komponente kontrolliert jeden Objektzugriff.

1.2.1 Name Protection

Die Festlegung von Zugriffsrechten für Objektnamen durch Zugriffsmodi wie `protected` ist die häufigste Art um Objektnamen zu schützen (*name protection*). Selbst moderne objektorientierte Programmiersprachen wie z.B. Java ([GJS96]) benutzen diesen Mechanismus. Es wird angenommen, dass auf ein Objekt immer über seinen Namen zugegriffen wird. Wenn nun der Zugriff über den Namen des Objekts verwehrt wird, dann sollte das Objekt automatisch bestmöglichst mitgeschützt sein.

Wie wir in den Beispielen im vorherigen Abschnitt gesehen haben, stimmt dies leider nur, wenn der Programmierer der Schnittstellenbildung höchste Aufmerksamkeit widmet. Tatsächlich gibt es immer Situationen, in denen ein Objekt übergeben wird, welches schon Aliase besitzt, und erst im Objekt, in welches das Objekt übergeben wird, der Name geschützt wird. Es kann zwar dann von dort aus kein Alias mehr erstellt werden und der Zugriff wird Außenstehenden nicht gestattet, aber die schon vorhandenen Aliase haben trotzdem vollen Zugriff und weitere Aliase können über diese erstellt werden.

Außerdem sollte angemerkt werden, dass die meisten Programmiersprachen den Objektnamen nicht einmal in Bezug zu anderen Objekten schützen, sondern in Bezug zu anderen Klassen [NVP98]. Das heißt, ein Objekt o_1 aus Klasse K mit einem privat geschützten Unterobjekt p bietet absolut keinen Schutz vor einem Objekt o_2 aus der gleichen Klasse K wie o_1. Der Namensschutz von Objekten ist demnach klassenbasiert und nicht objektbasiert. Selbst *innere Klassen* haben in Programmiersprachen wie Java immer vollständigen Zugriff auf die privaten Unterobjekte der Klasse, in der sie deklariert wurden.

1.2.2 Innere Klassen

Innere Klassen werden oft verwendet, um Unterobjekte besser zu schützen[2], indem diese Objekte nicht über eine Variable gespeichert, sondern in eine innere Klasse eingebettet werden. Auch die innere Klasse bietet keinen 100%-igen Schutz des Objekts; der Schutz gilt auch hier nur für den Namen der Instanz der inneren Klasse und kann unter gewissen Umständen durch Programmiertechniken umgangen werden [NVP98]. Da innere Klassen normalerweise mit allen Instanzen der gleichen Klasse geteilt werden und die Klasse, in der die innere Klasse deklariert wurde, oft von offenen, als `public` deklarierten, Programmbibliotheksklassen erben müssen, kann *dynamische Typkonvertierung* den Zugriff auf private innere Klassen gestatten, siehe dazu das nächste Beispiel.

```
1  public class StringBox {
2    private String value = "xyz";
3    public void setValue(String s) { value = s; }
4    public String getValue() { return value; }
5  }
6  class Abc {
7    SafeBox safebox;
8    private class SafeBox extends StringBox {
9        SafeBox() { setValue("Versteckter Inhalt?"); }
10   }
11 }
```

Die Klasse `Abc` besitzt eine private innere Klasse `SafeBox`, die von `StringBox` erbt. Es macht den Anschein, dass über die Deklarierung `private` der Klasse `SafeBox` die Werte innerhalb einer `SafeBox`-Instanz versteckt sind. Es ist jedoch über dynamische Typkonvertierung möglich, auf manche Werte zuzugreifen. Dies wird von folgender Programmklasse demonstriert:

```
12 class Program {
13   Abc abc = new Abc();
14   // abc.safebox.getValue(); --> Übersetzungsfehler!
15   String value =
16     ((StringBox) abc.safebox).getValue(); // Erlaubt!
17   System.out.println(value);
18     // Ausgabe: "Versteckter Inhalt?"
19 }
```

Da die private innere Klasse `SafeBox` von der als `public` deklarierten Klasse `StringBox` erbt und Zugriff auf `safebox` erlaubt ist (jedoch keinen Zugriff auf

[2]Hier wird der zusätzliche Programmieraufwand nicht berücksichtigt, kann aber bei vielen zu schützenden Unterobjekten beträchtlich sein.

Objekte und Methoden von `safebox`), sind nicht alle Objekte innerhalb von `safebox` tatsächlich versteckt. In der Zeile 16 wird in der Klasse `Program` dynamische Typkonvertierung der privaten inneren Klasse zur öffentlichen Klasse `StringBox` durchgeführt; dies bewirkt, dass in der folgenden Zeile 17 ein eigentlich versteckter String `value` ausgegeben werden kann.

1.2.3 Object Protection

Von *object protection*, auch „object control“ und Referenzkontrolle genannt, wird gesprochen, wenn ein Objekt nicht nur über seinen Namen geschützt wird, sondern wenn alle Zugriffe auf dieses Objekt explizit überprüft werden. Das kann statisch während der Übersetzungszeit, wie auch dynamisch während der Laufzeit geschehen. Hier wird kein Namensraum für eine Variable festgelegt, sondern ein Zugriffsrecht bzw. Zugriffsverbot für verschiedene Klassen von Objekten. Die Klassifizierung erfolgt dabei meistens über das Typsystem der jeweiligen Programmiersprache. Im einfachsten Fall gibt es für jedes Objekt zwei Klassen: Die Klasse der Objekte, die Zugriff haben und die Klasse der Objekte, die keinen Zugriff haben.

Bei statischen alias-control-Systemen muss der Übersetzer bei einem Objektzugriff überprüfen, von welchem Objekt aus der Aufruf stattfindet und ob dieses Objekt zur Klasse der Objekte, die Zugriff haben, gehört. Ist das Aufrufobjekt in der Klasse mit Zugriffserlaubnis, dann ist der Zugriff erlaubt und der Übersetzungsvorgang geht weiter; falls nicht, dann folgt ein Übersetzungsfehler (statische Objektkontrolle).

Bei dynamischen alias-control-Systemen muss die Laufzeit-Umgebung zu jedem Objekt zusätzliche Informationen speichern. Nur über diese Zusatzinformationen kann entschieden werden, ob der Objektzugriff erlaubt ist oder nicht. Diese sogenannte dynamische Objektkontrolle muss z.B. dann angewendet werden, wenn ein Objekt einem anderen gehört und sich diese Zugehörigkeit während der Laufzeit ändern kann. Leider erleidet hier die Performanz zur Ausführungszeit erhebliche Einbußen. Das ist auch der Grund, warum die meisten Lösungsansätze ein statisches alias-control-System anstreben. Die in dieser Arbeit vorgestellten Ansätze haben alle vom Konzept her statische Objektkontrolle.

1.3 Lösungswege

Hier werden kurz unterschiedliche Strategien vorgestellt, die Lösungsansätze zum Alias-Problem anwenden. Der Umgang mit Aliasen kann in folgende vier Kategorien[3] eingeteilt werden [HLW+92]:

[3] Im Originalartikel wurden für die Kategorisierung die englischen Bezeichnung „detection“, „advertisement“, „prevention“ und „control“ benutzt.

Erkennung Unter *Erkennung* wird die statische (zur Übersetzungszeit) oder dynamische (zur Laufzeit) Erfassung von potenziellen und aktuellen Gefahren von Aliasen verstanden. Diese Diagnose erfolgt, nachdem Aliase erstellt wurden (*post hoc*).

Diese Strategie wird vor allem dort angewandt, wo über den Quellcode hinaus keine Informationen über Abhängigkeiten *a priori* vorhanden sind. Übersetzer können somit effizienteren Code erstellen.

Aufforderung Wenn eine Programmiersprache Notationen anwendet, um die Übergabe von Werten oder Referenzen zu erkennen, wird von der Strategie der *Aufforderung* gesprochen.

Bei den meisten Methoden hat der Programmierer eine Vorstellung darüber, wie die Übergabe einer Variable stattfindet (z.B. über Wertübergabe oder Referenzübergabe), aber manchmal trifft leider genau das Nichterwartete zu. Sowohl Programmierer als auch Übersetzer profitieren von zusätzlichen Notationen, die einen besseren Überblick über Abhängigkeiten zwischen Objekten zulassen.

Vorbeugung Die *Vorbeugung* wird von Systemen angewandt, die über statische Überprüfung des Quelltextes Aliase verbieten können.

Dabei werden unterschiedliche Kontexte und die aktuellen Zustände der zu schützenden Objekte berücksichtigt. Die große Herausforderung ist, dem Übersetzer genau die Informationen zu übergeben, die er braucht, um entscheiden zu können, ob eine Referenz eine potenzielle Gefahr bildet, oder nicht. Außerdem sollte der Programmierer in der Programmierung nicht zu sehr eingeschränkt werden. Dies ist die Stategie der meisten Ansätze, die in Abschnitt 4 vorgestellt werden.

Seiteneffekt-Kontrolle Wenn zugesichert wird, dass Aliase garantiert keine unerwünschten Seiteneffekte erzeugen, wird der Mechanismus als *Seiteneffekt-Kontrolle* klassifiziert.

Idealerweise existiert ein Verfahren, das zusichern kann, dass auch während der Laufzeit keine unerwünschten Zustände von Objekten vorkommen. Bisherige Seiteneffekt-Kontroll-Systeme arbeiten sehr ineffizient und haben eine geringe Anwendbarkeit. Ein Beispiel für diese Strategie ist eine Erweiterung der SPOOL-Sprache in [Ad90].

Nur über den Einsatz von Lösungsansätzen des Alias-Problems kann Kapselung von Objekten erreicht werden. Die Lösung ist Aliase zu *erkennen*, wo sie vorkommen, *aufzufordern*, wo es möglich ist, *vorzubeugen*, wo Aliase ungewollt sind und Seiteneffekte von Aliase zu *kontrollieren*, wo es nötig ist [HLW+92].

1.4 Aufbau

Ziel dieser Arbeit ist, dem Leser deutlich zu machen, dass es zu den herkömmlichen Kapselungsmechanismen noch weitere Mechanismen gibt, die zum Alias-Problem viel bessere Lösungsansätze bieten. Das gelingt meines Erachtens über das genaue Betrachten und Vergleichen von *mehreren* unterschiedlichen Kapselungsstrategien. Dabei hilft uns eine Kapselungsfunktion, die für jeden Ansatz neu definiert und angepasst werden kann, den Wirkungsgrad eines Kapselungsmechanismus besser einzuschätzen. Zugriffe zwischen den Objekten, die eine Komponente ergeben und hoffentlich vor anderen Objekten kapseln, können durch einen Objektgraph graphisch dargestellt werden. Das hilft dem Leser, das Resultat eines relativ komplexen Vorgangs im Typsystem bildlich zusammenzufassen und schließlich besser zu verstehen.

Diese Arbeit will ein möglichst breites Spektrum an unterschiedlichen Kapselungsmechanismen vorstellen, die seit Anfang der 90er Jahre entwickelt wurden. In der Literatur-Recherche sind über das Thema Objektkapselung dutzende Artikel zu finden, die direkt das Thema aufgreifen [4]. Natürlich war da die Selektion der hier vorzustellenden Ansätze nicht einfach. Bei der Selektion wurden folgende Punkte berücksichtigt:

- Wie hoch ist der Wirkungsgrad des Mechanismus einzuschätzen?
- In wieweit hat der Ansatz die darauf folgenden Ansätze beeinflusst?
- Wie oft wird auf den Ansatz von den folgenden Ansätzen referenziert?
- Ist das präsentierte System verständlich oder sehr komplex?
- Gibt es eine existierende Implementierung?
- Passt der Ansatz inhaltlich zu den anderen vorzustellenden Ansätze?
- In wieweit finde ich subjektiv den Ansatz überzeugend oder stehe ihm ablehnend gegenüber?

Das Ergebnis sind acht sehr unterschiedliche Ansätze, die im Kapitel 4 chronologisch, mit Unterstützung von Beispielen und der Kapselungsfunktion, vorgestellt werden.

Die Beschreibung und Diskussion dieser Ansätze soll es allen Leserinnen und Lesern ermöglichen, sich eine eigene Meinung zu bilden, in wieweit andere als gewohnte Kapselungsmechanismen für gewisse Einsatzgebiete hilfreich oder eher störend sein können.

Die nächsten Kapitel dieser Arbeit sind wie folgt aufgebaut:

[4] Meine persönliche Sammlung beläuft sich zur Zeit auf etwa 40 Artikel.

In Kapitel 2 werden wichtige Begriffe eingeführt, Kapselung und eine gekapselte Komponente definiert und schließlich die Eigenschaften einer gekapselten Komponente vorgestellt.

In Kapitel 3 wird eine Kapselungsfunktion definiert, über die unterschiedliche Ansätze später in Hinsicht auf die Wirkungsgrade der Kapselungsmechanismen verglichen werden können. Hier werden dazu die Notationen der Funktion, als auch die Notation der graphischen Darstellung einer Kapsel erklärt.

In Kapitel 4 werden 8 wichtige Kapselungsansätze chronologisch vorgestellt. Zusätzlich wird jeweils eine angepasste Kapselungsfunktion definiert und die Kapsel graphisch dargestellt.

Abschließend werden in Kapitel 5 die existierenden Kapselungsmechanismen kritisch betrachtet und untereinander verglichen. Schließlich wird auch ein Ausblick auf zukünftige Anwendung von Kapselungsmechanismen gemacht.

2 Eine gekapselte Komponente

2.1 Begriffserklärungen

Wie wir schon bei der Einführung gesehen haben, haben Kapselungsmechanismen einen sehr breiten Anwendungsbereich, der über die Kontrolle über Aliase hinausgeht. Mit Hilfe von Kapselung versucht man hauptsächlich zu strukturieren, um Wartbarkeit und Wiederverwendbarkeit von Modulen zu verbessern. Dabei werden Implementierungsteile versteckt und Schnittstellen für kontrollierten Zugriff erstellt. Bei objektorientierten Programmiersprachen müssen Objekte erkannt werden, die geschützt werden sollen. Dabei haben wir gesehen, dass hauptsächlich Gefahren von bestehenden Aliasen eines Repräsentationsobjekts von außen oder nach außen ausgehen. Wann sprechen wir nun von einer Referenz, wann von einem Alias?

In dieser Arbeit werden Begriffe wie *Referenz*, *Alias*, *Zeiger* und *Variable* wie folgt verwendet:

- Ein *Zeiger* ist eine Variable, die auf eine beliebigen Adresse verweist. Dabei ist irrelevant, ob sich die Adresse im selben Programm bzw. Speicherbereich befindet oder auf der Festplatte oder auf einem anderen Rechner.

- Eine *Referenz* ist ein Zeiger, der auf ein Objekt verweist. Deshalb besitzt eine Referenz einen viel engeren Rahmen, meist innerhalb eines Programmes (Speicherbereich). Objektorientierte Programmiersprachen benutzen Referenzen, weil dadurch unterschiedliche Programmteile sich ein Objekt teilen können: es entstehen Aliase. Die Anwendung von Referenzen unterstützt Datenkonsistenz und Anwendungsperfomanz.

- Es wird von einem *Alias* gesprochen, wenn eine Referenz dupliziert werden kann. Es kann auf ein nur einmal real im Speicher existierendes Objekt über mehrere Aliase im Quelltext zugegriffen werden.

- Der Begriff der *Variable* wird hier oft synonym zum Begriff der Referenz benutzt. Im Allgemeinen besitzt eine Variable immer einen konkreten Namen; eine Referenz kann auch anonym sein. Außerdem besitzen die meisten objektorientierten Sprachen das Konstrukt eines primitiven Datentyps. Primitive Datentypen sind keine Objektinstanz einer Klasse. Über Variablen kann auf diese Werte zugegriffen werden.

- Die Implementierung der Funktionalität eines Objekts wird in dieser Arbeit als *Repräsentation* oder *interne Repräsentation* bezeichnet. Die Repräsentation besteht aus den Unterobjekten eines Objekts, die die Speicherung von Daten und das Innenleben eines Objekts durchführen. Beispiele: Ein Motor ist Teil der Repräsentation eines Autos; eine Menüleiste ist Teil der Repräsentation eines Fensters; die Repräsentation einer Tabelle besteht zum einen aus den Daten der Tabelle und zum anderen

aus Zellen, untergliedert in Zeilen und Spalten. In UML-Notation wird die Kompositum-Beziehung dazu benutzt, um die Repräsentation eines Objekts graphisch darzustellen.

Alias control bzw. *alias protection* sollte flexibel gestaltet werden, sodass die Vorteile von Aliasen (Konsistenz, Performanz) dort zur Geltung gebracht werden kann, wo es keine Sicherheitsgefahren gibt, und vice versa, wo Aliase gefährlich werden können, müssen Schutzmaßnahmen durch die Kontrolle über Aliase erfolgen, was bedeutet, manchmal Aliase *nicht* zu erlauben.

2.2 Definition der Kapselung

Definition 1 *Kapselung ist eine Technik zur Strukturierung des Zustandraumes ablaufender Programme. Ziel ist es, durch Bildung von Kapseln mit klar definierten Zugriffsschnittstellen die Daten- und Strukturkonsistenz zu gewährleisten.* [PH02]

Eine Kapsel muss nicht ausschließlich eine Klasse oder ein Objekt umschließen. Sie kann

1. ein einzelnes Objekt
2. mehrere Objekte aus ev. unterschiedlichen Klassen
3. eine Klasse (mit all ihren Unterobjekten)
4. alle Klassen in einer Vererbungshierarchie
5. ein Paket (mit allen Klassen und deren Instanzen)
6. mehrere Pakete

enthalten.

Wichtig ist, dass eine klar definierte Grenze der Kapsel festgelegt wurde und eine Schnittstelle zwischen der Außenwelt und der Kapsel besteht.

Ein Beispiel einer Komponente, die aus mehreren Objekten unterschiedlicher Klassen besteht, ist die Anwendung des bekannten *MVC-Prinzips*. MVC steht für die Aufteilung von Aufgaben einer Komponente in *Model*, *View* und *Controller*. Die Daten einer Komponente werden im Modell verwaltet, somit hängt der Zustand einer Komponente nur von dessen Modell ab. Der View präsentiert die graphische Darstellung meistens über die Visualisierung der Komponente auf dem Bildschirm (übernimmt das Zeichnen). Es können mehrere unterschiedliche Views für eine Komponente erstellt werden. Die Interaktion zwischen Anwender und Komponente regelt der Controller, indem er die Eingaben empfängt und die Veränderungen im Modell und im View veranlasst. Die Entkopplung zwischen Darstellung und Daten wird damit stärker unterstützt.

Diese drei Teile bilden einzeln keine eigenständige Komponente, der Kontext der Komponente wird nur über die Verbindung von Model, View und Controller erreicht. Beispielsweise besteht eine übliche Implementierung einer Tabelle über das MVC-Prinzip über die Deklarierung von drei Klassen: `TableModel`, `TableView` und `TableController`. Die Komponente `Table` umfasst mehrere Objekte unterschiedlicher Klassen.

Um den Punkt 4 der obigen Liste zu veranschaulichen, ist beispielsweise eine Hierarchie von Dokumentenstrukturen zu nennen. Ein Dokument besteht aus einer Struktur, die den Inhalt in Dokumentanfang und -ende, Titel, Abschnitte, Tabellen usw. aufteilt. Oft ist es notwendig, mit einer abstrakten Dokumentenstruktur zu arbeiten, die unterschiedliche Realisierungen haben kann. Ein Programm benutzt die abstrakten Methoden der abstrakten Dokumentenstruktur und kümmert sich nicht weiter über Details des Dokuments. Es können z.B. ein HTML-Dokument, ein RTF-Dokument, ein TeX-Dokument und ein Text-Dokument alle von einer gemeinsamen abstrakten Dokumentenklasse erben (vergleiche Abbildung 1). Weiters kann ein HTML-Dokument verfeinert werden, indem es Frames verwenden darf. Ein TeX-Dokument könnte mit der Erweiterung von LaTeX arbeiten. Die Komponente eines Dokuments benötigt demnach eine ganze Hierarchie von Dokumentenstrukturen, da das Programm, in dem das Dokument benutzt wird, nur mit einer abstrakten Klasse arbeitet, die erst in den Spezialisierungen in der Vererbungshierarchie weiter unten liegenden Klassen realisiert wird.

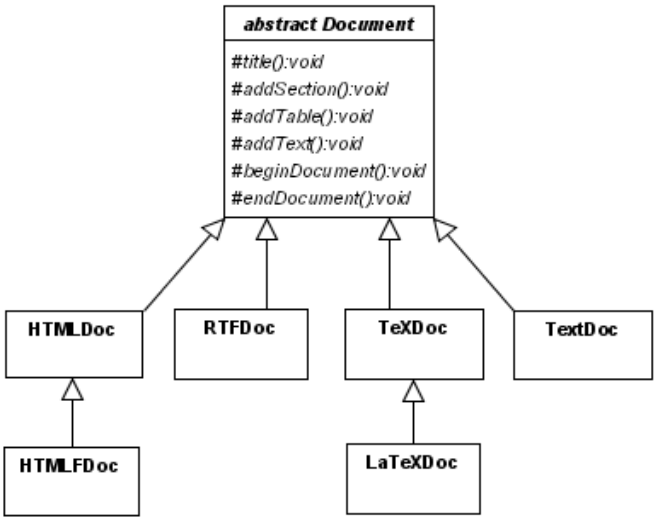

Abbildung 1: Komponente eines Dokuments bestehend aus einer Hierarchie von Dokumentenstrukturen

2.3 Definition einer gekapselten Komponente

In typischen objektorientierten Programmiersprachen besitzen Objekte *Zeiger* auf die Unterobjekte. Ein *Zugriff* auf ein Objekt o_2 von einem Objekt o_1 kann somit wie folgt definiert werden:

Definition 2 *Ein Objekt o_1 hat Zugriff auf ein Objekt o_2 genau dann, wenn o_1 einen Zeiger auf o_2 hat (o_2 ist ein Unterobjekt von o_1) oder wenn o_1 eine Methode besitzt, die einen Eingabeparameter hat, der auf o_2 zeigt. [Boy04]*

Viele Kapselungssysteme unterscheiden zusätzlich zwischen Lese- und Schreibzugriffe. In Abschnitt 4 sind dies das Insel-Schema (4.2), Flexible-Alias-Protection (4.4) und das Universum-Typsystem (4.6).

Wie ein solcher Zugriff in der Programmierung aussieht, wird hier kurz demonstriert. Greifen wir nochmals das Beispiel eines Autos und dessen Motor auf. Wenn im Rumpf der `Auto`-Klasse ein Motor deklariert wird mit `Motor motor = new Motor()`, dann besitzt die Referenz `motor` einen Zeiger auf ein `Motor`-Objekt. Über diesen Zeiger vom `Auto`-Objekt zum `Motor`-Objekt kann das Auto auf den Motor *zugreifen*. Außerdem wäre denkbar, dass die `Auto`-Klasse den Herstellernamen als String speichert. Im Rumpf einer Methode `setzeHerstellername(Hersteller h)` kann nun der Herstellername mit `herstellername = h.name;` gesetzt werden. Das `Auto`-Objekt besitzt also eine Methode, dessen Eingabeparameter `h` auf den Hersteller zeigt und bietet somit *Zugriff* auf ein `Hersteller`-Objekt.

Über den Begriff des Zugriffs auf ein Objekt kann die Kapselung eines *Objekts* wie folgt definiert werden:

Definition 3 *Ein Objekt o_1 kapselt ein Unterobjekt o_2 bezüglich eines Objektes p genau dann, wenn p Zugriff auf o_1 haben kann, aber niemals Zugriff auf o_2 hat. [Boy04]*

Um den Begriff der Kapselung einer ganzen Komponente zu definieren, fehlt noch die Definition der Abhängigkeit zwischen einem Objekt o_1 und einem Unterobjekt o_2, da nicht alle Unterobjekte notwendigerweise gekapselt werden müssen.

Definition 4 *Ein Objekt o_1 besitzt Abhängikeiten zu einem Unterobjekt o_2 genau dann, wenn o_1 Lese- oder Schreiboperationen auf Objekten von o_2 ausführt oder Methoden von o_2 aufruft und desweiteren diese Operationen und Methodenaufrufe den Zustand von o_2 verändern, welcher Einfluss auf die Invarianten von o_1 besitzt und damit auch den Zustand von o_1 ändert. [Boy04]*

Abhängigkeiten haben eine besonders große Bedeutung in Verbindung mit Besitzverhältnissen, siehe dazu vor allem Abschnitt 4.7, aber auch die Abschnitte 4.5, 4.6, 4.8 und 4.9.

Abhängigkeiten bestehen also nur, wenn das veränderbare Verhalten[5] des Unterobjekts Einfluss auf den inneren Zustand und Verhalten der Repräsentation des Objekts hat. Andernfalls besteht nur Zugriff auf äußere Objekte bzw. Zugriff von außen auf Unterobjekte des Objekts, welcher aber keine Gefahren mit sich bringt und deshalb nicht als Abhängigkeit klassifiziert wird.

Auch dazu wird ein kleines Beispiel vorgestellt: Es ist anzunehmen, dass die Implementierung eines Motors ein Attribut `maxGeschwindigkeit` besitzt, welches festlegt, mit welcher maximalen Geschwindigkeit der Motor noch ordnungsgemäß arbeiten kann. Weiters könnte die `Auto`-Klasse eine Methode `beschleunige()` besitzen, die seinen Motor dazu veranlasst, das Auto zu beschleunigen. Natürlich gilt die Invariante, dass das Auto nur so schnell fahren darf, wie der Motor ordnungsgemäß arbeiten kann. Das Auto wird nur dann tatsächlich beschleunigt, wenn die maximale Geschwindigkeit des Motors nicht überschritten wird. Es besteht also eine Abhängigkeit zwischen Auto und Motor. Demgegenüber gilt wahrscheinlich keine Abhängigkeit des Autos zu seinem Hersteller, da nur die Funktionalitäten des Autos Auswirkungen auf sein Verhalten haben.

Eine gekapselte Komponente kann daher in Verbindung mit der Definition der Abhängigkeit wie folgt definiert werden:

Definition 5 *Eine gekapselte Komponente kapselt mindestens* alle *Unterobjekte, zu denen Abhängigkeiten bestehen. [Boy04]*

2.4 Eigenschaften einer gekapselten Komponente

Eine saubere Kapselung soll die fehlerhafte oder böswillige Benutzung einer Komponente verhindern [PHMM+03]. Dies wird ermöglicht, indem sichergestellt wird, dass es von außerhalb der gekapselten Komponente keine direkten, unerwünschten Referenzen auf Objekte, die die Funktionalität des Objekts implementiert (Repräsentation), gibt. Dieser Vorgang wird im englischen als *alias control* oder *alias protection* bezeichnet.

Es dürfen keine Abhängigkeiten von Objekten bestehen, die außerhalb einer Komponente liegen, zu Objekten, die sich in der (internen) Repräsentation der Komponente befinden. Es sollten weder Objekte, die außerhalb der Komponente erzeugt wurden, in die Repräsentation eingebaut werden, noch Referenzen von Objekte, die Teil der Repräsentation sind, an Programmteile außerhalb der Komponente herausgegeben werden.

Außerdem sollte der Zugriff auf die Objekte der internen Repräsentation einer Komponente, die nicht Teil der Komponentenschnittstelle oder deren Repräsentation selbst sind, verwehrt werden.

Eine gekapselte Komponente hat folglich mindestens zwei Eigenschaften (angelehnt an [NVP98]):

[5] Im Englischen als „mutable behaviour“ genannt.

1. E_1 `Keine Herausgabe von Unterobjekten` Objekte der internen Repräsentation dürfen nicht an Objekten außerhalb der gekapselten Komponente herausgegeben werden. Sie sollten also *nicht* in der Schnittstelle der Komponente vorkommen. Sofern die Information eines Objekts der Repräsentation auch für äußere Objekte interessant ist, sollte keine Referenz des Objektes herausgegeben werden, sondern eine Kopie des Objekts.

2. E_2 `Keine Abhängigkeiten zu äußeren Objekten` Sofern der innere Zustand einer Repräsentation von außen geändert werden sollte, darf keine Abhängigkeit zum eingehenden Objekt bestehen. Das heißt, das eingehende Objekt sollte entweder unveränderbar (*immutable*) und somit bestmöglich geschützt sein oder es sollte eine Kopie des Objektes als neues Objekt für die interne Repräsentation der Komponente benutzt werden, sodass keine Abhängigkeiten nach außen bestehen können.

3 Formalisierung einer gekapselten Komponente

In [NCP+03] wurde ein Modell vorgestellt, das versucht, die Kapselung in objektorientierten Programmiersprachen zu formalisieren. Dabei wird ein Zugriffsgraph für die Modellierung der Topologie eines Systems verwendet. Der Zugriffsgraph besteht einerseits aus *Knoten*, welche die Objekte darstellen, andererseits aus gerichteten *Kanten*, welche die Zugriffsrechte zwischen den Objekten definieren. Eine *gekapselte Komponente* muss sich übrigens nicht auf einen Knoten beschränken, sondern kann durchaus auch aus mehreren bestehen. Es wird deshalb zwischen einer Komponente $c \in \mathcal{C}$ und einem Knoten $o \in \mathcal{S}$ unterschieden.

3.1 Notationen und Invarianten

Es werden folgende Bezeichnungen, angelehnt an [NCP+03], verwendet[6]:

$\mathcal{S}$	Alle Knoten im Typsystem
$\mathcal{C} = \mathcal{P}(\mathcal{S})$	Menge aller Teilmengen von $\mathcal{S}$
$o, p, q \in \mathcal{S}$	Bezeichner für konkrete Knoten
u, v, w	Bezeichner für konkrete Kanten
$c \in \mathcal{C}$	Bezeichner für eine konkrete Komponente
$\{o\}\triangleright$	Knoten, die eine eingehende Kante von o haben
$\triangleright\{o\}$	Knoten, die eine ausgehende Kante nach o haben
$\{o\}\twoheadrightarrow$	Knoten, die von o aus erreichbar sind
$\twoheadrightarrow\{o\}$	Knoten, die o erreichen
$o \longrightarrow p$	Es gibt eine Kante von o nach p
$o \leq p$	o dominiert p - Pfade nach p müssen durch o gehen

Tabelle 1: Notationen zur Formalisierung einer gekapselten Komponente

Das *System* $\mathcal{S}$ kann in vier Objektmengen aufgeteilt werden. Eine gekapselte Komponente beinhaltet den *Bund* $\mathcal{B}$, die *interne Repräsentation* $\mathcal{I}$ und die *äußeren Referenzen* $\mathcal{R}$.

Der Bund beschreibt die Menge der Knoten, die die Schnittstelle zu den äußeren Objekten im System definieren. Diese Knoten besitzen eine eingehende

[6] Achtung: Manche Bezeichnungen stimmen *nicht* mit denen aus [NCP+03] überein.

Kante (die Spitze zeigt auf das Objekt im Bund) ausgehend von einem Knoten im System, welcher sich außerhalb der Kapsel befindet.

Die interne Repräsentation einer Komponente besteht aus den Objekten, die den gekapselten Teil der Komponente repräsentieren. Sie kann nur über die Schnittstelle (Bund) von außen erreicht werden.

Die äußeren Referenzen beschreiben all diejenigen Knoten der Komponente, auf die vom Bund oder von der internen Repräsentation zugegriffen wird, die jedoch dort nicht enthalten sind.

Dementsprechend wird das *Äußere* $\mathcal{O}$ einer gekapselten Komponente definiert als die Menge derjenigen Knoten im System, die weder in der internen Repräsentation $\mathcal{I}$ noch im Bund $\mathcal{B}$ beinhaltet sind. Das bedeutet, dass die äußeren Referenzen $\mathcal{R}$ eine Teilmenge des Äußeren $\mathcal{O}$ einer gekapselten Komponente bilden.

Damit gelten die Invarianten [NCP+03]:

$$\begin{aligned} \mathcal{O} &= S - (I \cup B) \\ \mathcal{R} &\subseteq O \end{aligned}$$

Der Bund, die interne Repräsentation und die äußeren Referenzen sind zueinander disjunkt. Dies kann wie folgt dargestellt werden:

$$\mathcal{B} \cap \mathcal{I} = \mathcal{B} \cap \mathcal{R} = \mathcal{I} \cap \mathcal{R} = \{\}$$

Es muss definiert werden, wie der Zugriff auf Knoten der jeweiligen Teile der Komponente stattfindet. Damit die Komponente gültig gekapselt ist, darf die interne Repräsentation nur über ihren Bund Zugriff gestatten. Jede Kante, die in der internen Repräsentation endet, darf nur von dort selbst ausgehen oder vom Bund:

$$\triangleright\mathcal{I} \subseteq (\mathcal{B} \cup \mathcal{I})$$

Desweiteren müssen alle Objekte bzw. Knoten im System, die von der Komponente aus direkt erreicht werden können, aber weder in der interner Repräsentation, noch im Bund enthalten sind, in die Menge der äußeren Referenzen aufgenommen werden:

$$(\mathcal{B} \cup \mathcal{I})\triangleright \subseteq (\mathcal{B} \cup \mathcal{I} \cup \mathcal{R})$$

In Abbildung 2 ist eine Kapsel mit ihren Knoten grafisch dargestellt. Die Kapsel (der ovale Kreis) stellt eine Komponente dar. Sie besteht aus drei Abschnitten. Links liegt der Bund (gekennzeichnet mit **B**), in der Mitte die interne Repräsentation (**I**) und rechts die äußeren Referenzen (**R**). Wie man sieht, ist die Linie der Kapsel rechts (abgrenzend zu **R**) strichliert und nicht durchgezogen. Das weist darauf hin, dass die äußeren Referenzen $\mathcal{R}$ eine Teilmenge vom

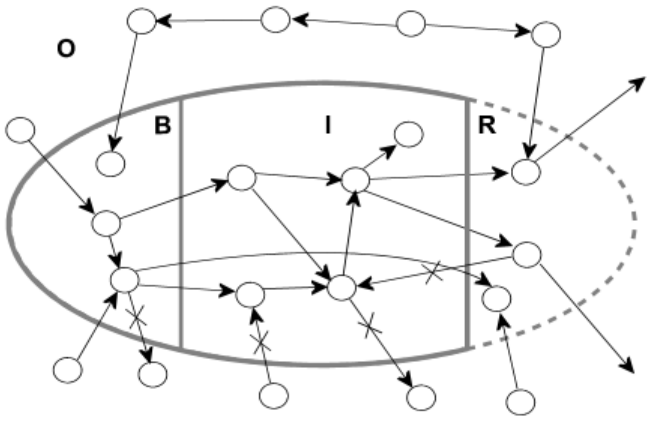

Abbildung 2: Eine gekapselte Komponente (modifiziert aus [NCP+03])

Äußeren $\mathcal{O}$ sind. Der Bereich **O** (im Bild links oben) besteht aus allen Knoten im System, außer denen, die in der Kapsel enthalten sind. Ein Knoten wird als Kreis dargestellt. Normalerweise entspricht ein Knoten einem Objekt in der Programmierung. Da eine Komponente meistens aus mehreren Objekten besteht, besitzt eine Kapsel auch mehrere Knoten, aufgeteilt in die drei Teile **B**, **I** und **R**. Ein Zugriff auf ein Objekt wird in dieser grafischen Notation als gerichtete Kante dargestellt. Die Spitze des Pfeils zeigt auf den Knoten (das Objekt), auf den zugegriffen wird. Der Schaft des Pfeils zeigt auf den Knoten, von dem aus zugegriffen wird. Eine normale Kante zwischen zwei Knoten besagt, dass dieser Zugriff gestattet ist. Kanten, die ein **X** in der Mitte haben, deuten deutlich darauf hin, dass dieser Zugriff *nicht* erlaubt ist. Normalerweise kann man aber davon ausgehen, dass, wenn eine Kante nicht explizit gezeichnet wurde, dann auch dieser Zugriff nicht erlaubt ist. Ausnahmen gelten z.B. bei trivialen Zugriffen zwischen äußeren Knoten oder von **R** nach **O**.

3.2 Definition der Kapselungsfunktion

Eine Kapselungsfunktion **k** weist jeder Komponente $c \in \mathcal{C}$ den zugehörigen Bund $\mathcal{B}$, die zugehörige internen Repräsentation $\mathcal{I}$ und die zugehörigen äußeren Referenzen $\mathcal{R}$ zu.

Sei $c \in C = \mathcal{P}(\mathcal{S})$ und $\mathcal{B}_c, \mathcal{I}_c, \mathcal{R}_c \subseteq S$, dann hat die Kapselungsfunktion k die Gestalt:

$$\begin{aligned} \mathbf{k}: \quad \mathcal{C} &\longrightarrow \mathcal{C} \times \mathcal{C} \times \mathcal{C} \\ c &\longrightarrow \begin{pmatrix} \mathcal{B}_c \\ \mathcal{I}_c \\ \mathcal{R}_c \end{pmatrix} \quad \text{mit } c = \mathcal{B}_c \cup \mathcal{I}_c \cup \mathcal{R}_c \end{aligned}$$

3.3 Eine kleine Beispielkapsel

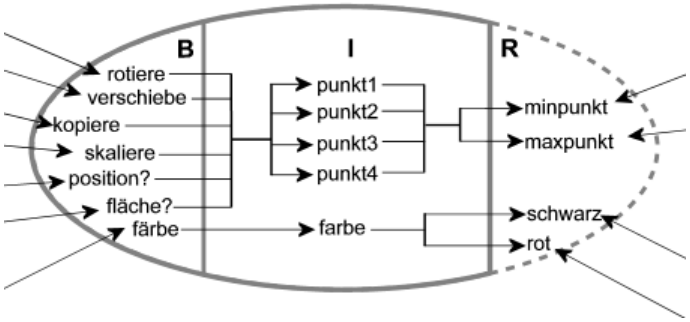

Abbildung 3: Die Kapsel eines `FarbViereck`-Objekts

In Abbildung 3 ist eine Kapsel eines Vierecks zu sehen. Die Kapselgrenzen würden in einer objektorientierten Sprache wie Java nur *ein* Objekt mit dessen Unterobjekten umfassen. Zugegebenermaßen ist ein *FarbViereck*-Objekt vielleicht nicht unbedingt das heikelste Thema in Hinsicht auf das Alias-Problem. Es zeigt aber gut, wie eine Kapsel (in diesem Falle ein Objekt) in die drei Teile Schnittstelle, Repräsentation und äußere Referenzen aufgeteilt werden kann.

Ein Viereck besteht natürlich aus vier Punkten; sie befinden sich in der internen Repräsentation $\mathcal{I}$, da sie das Viereck selbst repräsentieren (fehlt ein Punkt, wäre es kein Viereck, sondern ein Dreieck). Ein Punkt besteht normalerweise aus den zwei Werten `x` und `y`, auf die wir hier aber nicht näher eingehen werden. Klar sollte nur sein, dass ein Punkt kein primitiver Datentyp wie ein Integer ist. Unser `FarbViereck` besitzt außerdem eine Farbe, die genauso in $\mathcal{I}$ enthalten ist.

Die Komponente darf nur Zugriff von außen über die Schnittstelle erlauben. Die Schnittstelle besteht aus den meisten Funktionen der Objekte, wie dem Verschieben, dem Rotieren, dem Skalieren, dem Färben oder Kopieren. Alle diese Funktionen besitzen normalerweise einen oder mehrere Parameter. Sofern die Parameterübergabe über call-by-reference[7], also Referenzübergabe stattfindet, ist das Objekt, auf das referenziert wird, ein Objekt in der Schnittstelle $\mathcal{B}$. Auch wenn der übergebene Wert ein primitiver Datentyp ist, ist er Teil der Schnittstelle. Es können aber in der internen Repräsentation keine Aliase zu dem übergebenen Wert entstehen.

Es wurde mit Absicht keine Funktion festgelegt, die einen Punkt setzen würde. Damit wäre zunächst unklar, wie dies zu implementieren ist. Würde die Wertübergabe über primitive Datentypen stattfinden in der Art

[7] call-by-reference steht im Gegensatz zu call-by-value, der Übergabe des Wertes. In Java wird call-by-value für die primitiven Datentypen angewandt. Bei Übergabe des Wertes treten keine Alias-Probleme auf.

`setzePunkt1(int x, int y)`, würden (zumindest in Java) keine Alias-Probleme entstehen. Über die intuitivere Art `setzePunkt1(Punkt p)` entstehen jedoch Abhängigkeiten nach außen, wenn die Zuweisung von `p` über `this.p1 = p` erfolgen würde. Werden die Werte `x` und `y` kopiert (`this.p1.x = p.x` und `this.p1.y = p.y`), dann entstehen natürlich auch keine Abhängigkeit nach außen.

Im Beispiel befinden sich zwei Punkte `minpunkt` und `maxpunkt` in den äußeren Referenzen $\mathcal{R}$. Diese zwei Punkte können in einem anderen Programmteil erzeugt worden sein und setzen z.B. die Grenzen eines Zeichenbereichs. Um eine Funktion `verschieben` durchzuführen, muss aller Wahrscheinlichkeit nach geprüft werden, ob die durch die Operation hervorgeruften Änderungen keine unerlaubten Zustände, wie eine Zeichenrahmenüberschreitung, hervorrufen. Zugriffe von der internen Repräsentation zu äußeren Objekten müssen also erlaubt sein. Alle diese Referenzen nach außen werden in die Menge der äußeren Referenzen $\mathcal{R}$ aufgenommen. Im Beispiel sind dies außerdem die Farben `rot` und `schwarz`. Sie könnten zum Beispiel über eine Methode mit einem Parameter `istRot` mit Typ `boolean` gebraucht werden. Im Falle `true` wird die vordefinierte Farbe `rot` gesetzt, sonst die Farbe `schwarz`.

4 Kapselungsansätze (chronologisch)

Um die unterschiedlichen Kapselungssysteme des letzten Jahrzehnts bis heute für objektorientierte Programmiersprachen zu untersuchen und untereinander zu vergleichen, werden hier chronologisch die wichtigsten Ansätze grob erläutert, die Kapselungsfunktion definiert und die wichtigsten Eigenschaften diskutiert. Eine streng chronologische Auflistung ist dabei nicht möglich, da die unterschiedlichen Ansätze nicht immer aufeinander aufbauen und meist parallel zu anderen Systemen entwickelt wurden. Die verschiedenen Verfahren unterscheiden sich nicht nur in syntaktischen Differenzen und Implementierungstechniken, sondern vor allem auch in der semantischen Abstraktion einer Kapsel. Außerdem wurden viele Techniken nicht nur für *Information Hiding*, *Dependeny Control*, *Alias Control* entwickelt, sondern finden auch andere Anwendungen wie z.B. Architekturkontrolle.

Um die Quelltexte der unterschiedlichen Beispiele der folgenden Ansätze besser vergleichen zu können, wird eine einheitliche Notation, orientiert an Java [GJS96], verwendet. Außerdem wird als einheitliches Beispiel ein Stack benutzt. Eine Klasse `Stack` besteht aus einer Referenz zu einer Instanz der Klasse `Link`, dem obersten Eintrag des Stapels `top`. Einträge haben jeweils eine Referenz `next` mit Typ `Link`. Operationen eines Stacks sind u.a. (Java Notation):

void push(Object item) Legt einen neuen Eintrag auf den Stapel.

Link pop() Entnimmt den obersten Eintrag vom Stapel und gibt ihn aus.

Link peek() Gibt den obersten Eintrag vom Stapel aus ohne ihn zu löschen.

void put(Stack s) Legt einen Stapel von Einträgen auf einen anderen.

boolean isEmpty() Gibt aus, ob der Stack leer ist, also keinen Eintrag enthält.

int size() Gibt die Größe des Stacks aus.

Um die jeweiligen Quelltexte nicht überlaufen zu lassen, werden nur bestimmte Methoden, und diese nicht vollständig, implementiert. In allen Beispielen werden jedoch mindestens die Methoden `push(Object item)` und `pop()` implementiert, da sie das Importieren in bzw. Exportieren aus einer Komponente vorzeigen.

4.1 Kurzvorstellung der Ansätze

Hier werden die acht Ansätze, die in diesem Kapitel behandelt werden, kurz vorgestellt und erklärt, warum sie für das Verständnis der Entwicklung der Kapselungssysteme wichtig sind.

Insel-Schema (1991) Das Insel-Schema war der erste Schritt in Richtung Kapselung von Objekten. Das in Abschnitt 4.2 vorgestellte System hat zum Ziel, komplexe Objekte vollständig von anderen Objekten zu kapseln, sodass keine Aliase entstehen können. Leider ist das Insel-Schema etwas starr und für manche Programmiertechniken unbrauchbar, es hatte aber großen Einfluss auf die folgenden Ansätze.

Ballon-Typen (1997) Dieser in Abschnitt 4.3 vorgestellte Mechanismus verfeinert das Insel-Schema, was einen etwas erleichterten Umgang mit dem System für den Programmierer erbringt. Dieses Schema besitzt zwei unterschiedliche Grade des Kapselungseingriffs. Es wird aber auch hier vollständige Kapselung ausgeübt, was in vielen Fällen mehr als benötigt ist.

Das Insel-Schema und Ballon-Typen sind beides Systeme mit Konstrukten, die stark in die Freiheiten des Programmierens eingreifen. Sie stellen aber die ersten Ansätze und Ideen für Objektkapselung dar und sind somit Ausgangspunkt von fast allen weiteren Ansätzen.

Flexible Alias Protection (1998) FAP (Abschnitt 4.4) ist ein ziemlich komplexes Typsystem, das versucht, einen flexiblen, aber sicheren Umgang mit Aliasen zu erreichen. Nachdem FAP vorgestellt wurde, folgten eine ganze Reihe von unterschiedlichen Ansätzen; es scheint, als ob dieses System ausschlaggebend für viele Ideen der darauffolgenden Kapselungssysteme war. Das Ziel, Kapselung zu erreichen und trotzdem flexibel mit Aliasen umzugehen, wurde erreicht und daher sollte das System unbedingt genauer betrachtet werden. Das System leidet jedoch unter einer meines Erachtens mäßigen Anwendbarkeit für den Programmierer.

Einfache Besitzverhältnisse (1998) Sehr wichtig war die Einführung von Besitzverhältnissen zwischen Objekten. Obwohl dieses hier in Abschnitt 4.5 vorgestellte System recht einschränkend für den Programmierer ist (ganz im Gegensatz zur Ursprungsidee, FAP zu vereinfachen), war es der erste Schritt in diese Richtung und beinahe alle folgenden Systeme beinhalten Besitzverhältnisse in einer ähnlichen Weise im Programmiersystem.

Universum-Typsystem (1999-2001) Das Universum-Typsystem (Abschnitt 4.6) ist ein relativ ausgeklügeltes System, das Besitzverhältnisse und read-only Referenzen kombiniert. Die Einführung von read-only Referenzen ist meines Erachtens grundsätzlich sehr hilfreich und sollte auf jeden Fall behandelt werden. Es gibt eine Implementierung, und mehrere Arbeiten vor allem von Studenten beschäftigen sich mit dem Universum-Typsystem.

Tiefe Besitzverhältnisse (2003) Im Bereich der sogenannten tiefen Besitzverhältnissen, die sich speziell auf die Verbesserung der Besitzerverwaltung konzentrieren, war die Auswahl des hier vorzustellenden Systems nicht einfach. Es gibt mehrere Ansätze, die mindestens ebenbürtig zu dem Ansatz sind, der in Abschnitt 4.7 vorgestellt wird. Der relativ einfache Umgang mit Besitzern hat mich persönlich aber überzeugt. Außerdem bildet dieser Ansatz die Vorstufe des folgenden Ansatzes.

Externe Einzigartigkeit (2003) Einzigartigkeit ist ein wohlbekanntes Konstrukt, das in den meisten Programmiersprachen für primitive Datentypen angewendet wird. Für komplexe Objekte versuchten dies auch das Insel-Schema und Ballon-Typen. Das Ergebnis war aber für die praktische Programmierung zu sehr einschränkend. Imponierend fand ich deshalb das hier in Abschnitt 4.8 vorgestellte System, das Einzigartigkeit genauer untersuchte und zeigte, dass auch dieser Mechanismus verfeinert werden kann und somit ein flexiblerer Umgang erreicht wird. Externe Einzigartigkeit benutzt zusätzlich Besitzverhältnisse, die Objektkapselung unterstützen.

FGJ+c (2004-2005) Dieses in Abschnitt 4.9 vorgestellte System bietet den meines Erachtens bisher modernsten Kapselungsmechanismus, der aufbauend auf den besten Ideen aller Ansätze eine gute Kombination gefunden hat. Ein entscheidender Faktor, warum ich dieses System auch persönlich bevorzugen würde, ist, dass hier auch Rücksicht auf die Weiterentwicklung von anderen Aspekten in der objektorientierten Programmierung als der Kapselung genommen wurde. Wichtig war mir auch, dass das Confinement-Konstrukt mit einbezogen ist, um auch auf diese Art von Kapselung eingehen zu können.

Jede Vorstellung eines Ansatzes gibt zuerst einen groben Überblick über das System, erklärt dann die Besonderheiten und Neuerungen und versucht, diese an einem Beispiel zu zeigen. Schließlich wird am Ende jedes Ansatzes mithilfe der Vorarbeiten der Autoren in [NCP+03] die jeweilige Kapselungsfunktion(en) vereinfacht oder detailliert definiert. Diese Kapselungsfunktionen bedürfen außerdem einer genaueren Erklärung, die zusammen mit der jeweiligen Definition erfolgt. Manchmal unterscheidet sich die hier benutzte Notation von [NCP+03], manchmal auch die jeweilige Interpretation gewisser Detailaspekte.

4.2 Insel-Schema

John Hogg hat in [Hog91] 1991 das erste Konzept zur Kapselung von Objekten für objektorientierte Programmiersprachen entwickelt. Er definierte den Begriff der *Insel*[8] als eine vollständige Kapselung von (eventuell) mehreren Klassen eines Programms.

4.2.1 Modell

Der erste Vorschlag, um das Konstrukt einer Insel zu implementieren, ist die Unterscheidung einer Methode in eine *Funktion* oder eine *Prozedur*[9]. Dabei wird in Prozeduren der interne Zustand des Objekts geändert, Prozeduren haben aber keinen Rückgabewert. Funktionen hingegen garantieren, dass es zu keinen Änderungen des internen Zustands des Objekts kommt, geben aber einen Rückgabewert aus. Es liegt nahe, diese Unterscheidung auch syntaktisch zu unterstützen. In diesem Artikel wird dazu für Prozeduren das Wort `procedure` vorangesetzt[10]. Wozu diese Unterscheidung benötigt wird, wird im Regelsystem des Insel-Schemas deutlich.

Weiters wird zwischen statischen und dynamischen Referenzen unterschieden. Ein statischer Alias ist eine Referenz auf ein langlebiges Objekt (meist Objektinstanzen oder globale Variablen), welches schon beim Programmstart existiert. Demgegenüber ist ein dynamischer Alias eine Referenz auf ein kurzlebiges Objekt, welches während der Ausführung des Programms erstellt und auch meist vor Programmende wieder zerstört wird. Normalerweise entsteht und stirbt eine dynamische Referenz innerhalb eines Methodenblocks.

Die Strategie im Insel-Schema ist, keine statische Referenz von einem Objekt außerhalb der Insel zu einem Objekt innerhalb der Insel zuzulassen. Eine Ausnahme bilden Referenzen über eine sogenannte *Brücke*. Die Brücke ist eine Instanz einer Brückenklasse[11]; diese definiert Methoden, über die äußere Objekte mit der Insel interagieren können. Es wird damit zugesichert, dass zwischen zwei Methodenaufrufe einer Brückenklasse der innere Zustand einer Insel unverändert bleibt [Hog91].

Über dynamische Referenzen können Nachrichten zu Objekte innerhalb der Insel gesendet werden. Eine Nachricht erfolgt über ein Objekt als Parameter einer Methode der Brückenklasse. Obwohl diese Interaktion mit einer Insel den Zustand der Insel verändern kann, ist sie trotzdem kontrolliert, da sie nur über die Brückenklasse erfolgen kann. [Hog91].

[8] Im Englischen als *Island* bezeichnet

[9] In der imperativen Programmierung ist die Unterscheidung von Funktionen und Prezeduren wohlbekannt, wurde aber nicht in objektorientierte Sprachen übernommen.

[10] Im Originalartikel wurde vorgeschlagen, dass Funktionen im Gegensatz zu Prozeduren mit einem Großbuchstaben beginnen.

[11] Im Englischen als *bridge classes* bezeichnet

4.2.2 Implementierung

Die Implementierung einer Insel wird durch zusätzliche Zugriffsmodi für Variablen erreicht. Ein Zugriffsmodus ist eine statisch überprüfbare Einschränkung auf Operationen einer Variable [Hog91]. Es werden zu dem in imperativen Programmiersprachen bekannten `read`-Modus[12] noch zwei neue Modi vorgestellt: `unique` und `free`:

read Eine Variable, die als `read` deklariert wird, darf gelesen werden, gestattet jedoch keinen Zugriff, der Änderungen des Werts der Variable hervorrufen würde. Es werden folgende syntaktische Regelungen vorgeschlagen [Hog91]:

1. Es dürfen keine Zuweisungen auf `read`-Variablen durchgeführt werden.
2. Ein `read`-Ausdruck darf nur als `read`-Variable über einen Aufruf eines Parameters einer Methode oder über den Rückgabewert einer Funktion herausgegeben werden.
3. Ein `read`-Ausdruck darf nicht auf der rechten Seite einer Zuweisung vorkommen.
4. In einer Funktion sind alle Parameter und selbst die Referenz auf die eigene Instanz `this` implizit als `read` deklariert.
5. Wenn innerhalb einer Prozedur auf `this` nur `read`-Zugriff gestattet ist, so haben auch alle Instanz-Variablen implizit Zugriffsmodus `read`.

unique Das Objekt (oder der Wert) einer als `unique` gekennzeichneter Variable, auf das referenziert wird, hat nur eine einzige Referenz. Über diese Variablen können keine Aliase erstellt werden.

free Ein `free`-Ausdruck besitzt hingegen überhaupt nur eine Referenz auf die *Variable*, d.h. im Quelltext kommt eine `free`-Variable nur einmal vor. Free-Ausdrücke sind für Instanzvariablen natürlich ungeeignet und werden nur zur Übertragung einer Variable über Parameter einer Methode oder als Rückgabewert (anonym) benutzt.

Es wird außerdem eine neue Zugriffsoperation benötigt, der sogenannte destruktive Lesezugriff[13] [Hog91]. Er liest den Wert der Variable, übergibt ihn und setzt anschließend die Referenz der Ursprungsvariable auf `null` (*Nullifizierung*). Hier wird dazu ein Rufezeichen vor die Variable gesetzt[14]. Beispiel: `a`

[12] Dieser Modus ist z.B. in Pascal der Standardmodus.
[13] Im Englischen als *destructive read* bezeichnet.
[14] Im Originalartikel wurde dies durch einen nach unten zeigenden Pfeil dargestellt, dazu fehlt aber leider eine Taste auf der Tastatur.

`= !b` definiert einen destruktiven Lesezugriff auf `b`; `a` bekommt den Wert von `b` und `b` wird auf `null` gesetzt.

Über den destruktiven Lesezugriff können folgende Regeln für die Modi `unique` und `free` festgelegt werden [Hog91]:

1. Eine `free`-Variable erlaubt nur destruktiven Lesezugriff.
2. Eine Zuweisung auf eine Variable mit Modus `unique` darf nur durch die Übertragung eines `free`-Ausdrucks erfolgen.
3. Eine Variable mit Modus `unique` darf nur als `unique` herausgegeben werden.
4. Wenn der Empfänger eines Methodenaufrufs `unique` ist, müssen alle Parameter und der Rückgabewert der Methode entweder `read`, `unique` oder `free` sein.

Ein destruktiver Lesezugriff einer Instanzvariable mit Modus `unique` oder `free` erzeugt ein Ergebnis mit Modus `free`. Außerdem kann ein `free`-Objekt (z.B. im Methodenrumpf) auf eine Instanzvariable mit Modus `unique` zugewiesen werden. Man erkennt den starken Zusammenhang zwischen `unique`- und `free`-Ausdrücke. Ein üblicher Vorgang ist, dass Parameter-Objekte Modus `free` haben, um Instanzvariablen mit Modus `unique` zu setzen. Die umgekehrte Richtung funktioniert genauso: Eine Instanzvariable mit Modus `unique` wird über den destruktiven Lesezugriff zu einem `free`-Ausdruck, der als Rückgabewert einer Funktion dienen kann[15].

Wenn jede Methode einer Klasse die Eigenschaft hat, dass alle Parameter und alle Rückgabewerte `read`, `unique` und `free` sind, dann ist jede Instanz dieser Klasse eine *Brücke*. [Hog91]

Wie man sieht, ist das Regelsystem recht komplex. Mein persönlicher Eindruck ist jedoch, dass, sobald man sich das System angeeignet hat, dieses Typsystem einen relativ intuitiven Umgang mit eingeschränkten Referenzen erlaubt (was nicht heißt, dass der Umgang mit Aliasen flexibel ist). Ein `free`-Ausdruck bildet eine Art Konstruktor für Variablen, die nur eine einzige Referenz besitzen (`unique`). Leider leidet dieses System darunter, dass der Programmierer dabei nicht über die Benutzung des Schemas *aufgefordert* wird, eine Insel zu konstruieren. Es wird die Strategie der Aufforderung (Abschnitt 1.3) angewandt (der Programmierer muss selber die richtige Notation anwenden). Es können sich meines Erachtens leicht Fehler in die Programmierung einschleichen, die die Herausgabe von zu schützenden Werten mit sich bringen. Außerdem macht die Deklarierung von Variablen als `unique` viele Programmiertechniken unmöglich,

[15] Die Frage bleibt aber dann, ob der destruktive Lesezugriff den inneren Zustand der Insel beeinflussen kann. Das würde der Invariante einer Funktion widersprechen.

wie z.B. einen Iterator, wo von Natur aus mindestens zwei Variablen auf einen Eintrag existieren müssen (eine Referenz aus der Datenstruktur und eine Referenz aus dem Iterator).

4.2.3 Beispiel: Stack im Insel-Schema

In diesem Abschnitt wird ein Stack im Insel-Schema implementiert. Dazu wird zuerst die Klasse `Link` deklariert, deren Instanzen das Daten-Element speichern und die Verbindungen zum nächsten (im Stapel darunterliegenden) Eintrag bilden.

```
1  class Link {
2    Link next;
3    unique Object data;
4    free Link(free Object item) {
5      this.data = !item;
6    }
7  }
```

Das Element `data` in der Zeile 3 hat Modus `unique`. Das bedeutet, dass dieses Objekt nur eine einzige Referenz besitzt. Es existiert im ganzen System keine andere Variable als `data`, die auf das gleiche Objekt zeigt. Außerdem wurde ein Konstruktor für die `Link`-Klasse entworfen, welcher einen Parameter besitzt, mit dem man das Daten-Objekt sofort setzen kann[16]. Konstruktoren erzeugen `free`-Ausdrücke, da naturgemäß bei der Konstruktion eines Objekts nur eine einzige Referenz über nur *eine*, die gerade erzeugte Variable, auf das Objekt existiert und außerdem dieser Ausdruck in keinem anderen Teil im ganzen Quellprogramm benutzt wird.

Wie wir sehen, kann ein `free`-Ausdruck nur dazu benutzt werden, um über einen destruktiven Lesezugriff eine Variable mit Modus `unique` zu setzen. Dieser Mechanismus wird auch in der Zeile 5 angewandt. Der Parameter `item` im Konstruktor hat Modus `free` und wird im Rumpf über einen destruktiven Lesezugriff an die Instanzvariable `data`, die als `unique` gekennzeichnet ist, übergeben.

Ein Stack bildet eine Insel, seine Einträge befinden sich im Inneren der Insel. Der Stack bildet somit die Brückenklasse und kann wie folgt implementiert werden:

```
8  class Stack {
9    Link top;
10   procedure push(free Object item) {
11     Link newTop = new Link(!item);
```

[16] Als der Originalartikel 1991 veröffentlicht wurde, war der Mechanismus des Konstruktors einer Klasse noch nicht bekannt.

```
12      newTop.next = top;
13      top = newTop;
14    }
15    procedure Object pop() {
16      top = top.next;
17    }
18    read Object peek() {
19      return top.data;
20    }
21    read boolean isEmpty() { return (top == null); }
22  }
```

Der Stack besitzt die üblichen Methoden `push`, `pop` und `peek`, wobei `push` und `pop` mit `procudure` gekennzeichnet sind. Die Prozedur `push` legt einen neuen Eintrag auf den Stapel und `pop` entnimmt den obersten Eintrag vom Stapel; beide verändern somit den Zustand der Insel. Unüblich ist hier, dass `pop` keinen Rückgabewert besitzt, dies erfordert die Definition der Prozedur. Um einen üblichen `pop`-Aufruf zu simulieren, wird vor `pop` zuerst `peek` aufgerufen.

Der Eintrag `item` in der Prozedur `push` mit Modus `free` kann dem Konstruktor der `Link`-Klasse über einen destruktiven Lesezugriff übergeben werden, da dort ein `free`-Objekt erwartet wird (Zeile 9). Die Funktion `peek` gibt den obersten Eintrag des Stapels mit Modus `read` (nur Lesezugriff) aus.

Da weiters alle Prozeduren von `Stack` nur Parameter mit Zugriffsmodi `free` und `unique` haben, und alle Funktionen `read` Rückgabewerte liefern, bildet die Klasse `Stack` eine Brückenklasse. Obwohl die Einträge (Instanzen der `Link`-Klasse) in einer anderen Klasse deklariert werden, gehören sie zur Insel des Stacks dazu.

4.2.4 Funktion für vollständige Kapselung

Vollständige Kapselung hat den Vorteil, dass die Komplexität der Spracherweiterung sowohl für den Sprachdesigner, als auch für den Programmierer sehr einfach gehalten werden kann. Dazu definieren wir die Kapselungsfunktion einer Insel durch:

Sei $o \in \mathcal{S}$ und o eine Brückenklasse. Dann gilt

$$\mathbf{k}_{\text{insel}}(o) = \begin{pmatrix} \mathcal{B} = \{o\} \\ \mathcal{I} = \{o\} \gg \\ \mathcal{R} = \{\} \end{pmatrix}$$

(vergl. [NCP+03])

Sofern das Objekt o als Brückenklasse identifiziert wurde, besteht die Schnittstelle (Bund $\mathcal{B}$) also aus genau diesem und nur diesem Knoten. Die interne

Repräsentation $\mathcal{I}$ wird definiert als diejenigen Knoten, die von o aus erreichbar sind, und da Objekte einer Insel keine Abhängigkeiten nach außen besitzen dürfen, hat eine Insel keine äußeren Referenzen. Die Menge der äußeren Referenzen $\mathcal{R}$ ist demnach die leere Menge.

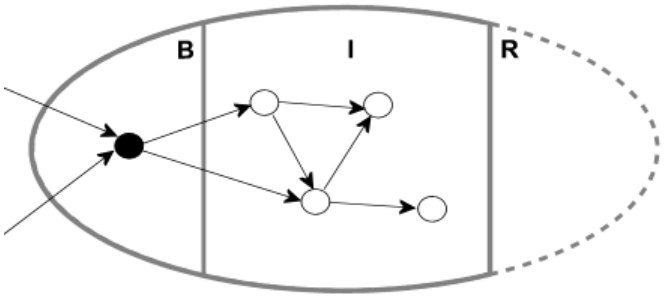

Abbildung 4: Eine Insel: vollständige Kapselung (modifiziert aus [NCP+03])

In Abbildung 4 ist eine Insel als Kapsel mit ihren Knoten grafisch dargestellt. Der schwarze Knoten wurde als Brückenklasse identifiziert, nur über ihn sind alle Knoten der internen Repräsentation zugänglich. Es gibt keine Kante aus der Komponente nach außen (totale Kapselung). Innerhalb der internen Repräsentation $\mathcal{I}$ sind beliebige Zugriffe zwischen den Knoten gestattet.

4.3 Ballon-Typen

1997 hat Paulo Sergio Almeida in [Alm97] das Insel-Schema um ein paar weitere Merkmale erweitert. Er stellte in diesem Artikel den Typ *Ballon*[17] vor, der wie eine Insel totale Kapselung bereitstellt, jedoch feinere Differenzierungen vornimmt.

Analogien zwischen Ballon-Schema und Insel-Schema sind:

- Eine Insel als Komponente im Insel-Schema entspricht in ihren Eigenschaften im wesentlichen der Eigenschaften eines Ballons als Komponente im Ballon-Schema.
- Die Instanz einer Brückenklasse im Insel-Schema wird als Ballonobjekt bezeichnet und besitzt dieselbe Eigenschaft der Schnittstellenbildung.
- Sofern Objekte geschützt werden sollen, wird vollständige Kapselung angewendet.
- Es wird zwischen dynamischen und statischen Referenzen unterschieden.

4.3.1 Invarianten

Um die Invarianten der Laufzeit-Umgebung für Ballons zu beschreiben, ist noch der Begriff eines Clusters zu definieren. Sei G der Graph mit Objekten als Knoten und mit den Referenzen zwischen Nicht-Ballonobjekten und von Ballonobjekten zu Nicht-Ballonobjekten als Kanten in einem Systems $\mathcal{S}$. Ein *Cluster* ist ein maximaler zusammenhängender Untergraph von G [PH02]. Hiermit kann auch definiert werden, ob sich ein Objekt *innerhalb* oder *außerhalb* eines Ballons befindet:

Ein Knoten $o \in \mathcal{S}$ befindet sich *innerhalb* eines Ballons B, wenn mindestens eine der folgenden Bedingungen gilt:

- o ist kein Ballonobjekt, aber befindet sich im gleichen Cluster wie B
- o ist ein Ballonobjekt mit einer Referenz von B selbst oder einem anderen Objekt (kein Ballonobjekt) im gleichen Cluster
- Es existiert ein Ballon $B^{'}$ innerhalb von B und o befindet sich innerhalb von $B^{'}$

Ein Knoten $o \in \mathcal{S}$ befindet sich *außerhalb* eines Ballons B, wenn er weder das Ballonobjekt B selbst ist, noch sich innerhalb von B befindet.

Die Ballon-Invarianten können somit wie folgt definiert werden:

I_{B_1} : Es gibt höchstens eine Referenz zu B

[17] Im Englischen als *Balloon* bezeichnet

I_{B_2} : Sofern diese Referenz exisitiert, hat sie ihren Ursprung im Äußeren $\mathcal{O}$

I_{B_3} : Kein Objekt innerhalb von B besitzt Referenzen zum Äußeren $\mathcal{O}$

I_{B_4} : Von allen Objekten, die ein *Cluster* bilden, ist höchsten eines ein *Ballonobjekt*

Die Invarianten erlauben es, dass lokale Variablen Objekte innerhalb eines Ballons (zumeist nur temporär) referenzieren, die nicht unbedingt auch umgekehrt Zugang vom Objekt zur Variable benötigen. Deshalb kann man einen Ballon auch nicht als die Menge der von einem Ballon erreichbaren Objekte definieren.

In Abbildung 5 sind die Zugriffe auf Objekte zwischen ineinander geschachtelte Ballons graphisch notiert[18]. Objekte werden als Feld von kleinen Quadraten dargestellt, die jeweils auf einen Behälter eines Zeiger innerhalb des Objekts hinweisen. Ein Ballon ist als Kreis gezeichnet. Wie man sieht, dient das Ballonobjekt (mit **B** beschriftet) als Schnittstelle der Objekte außerhalb des Ballons (**external Objects**) zu den Objekten innerhalb des Ballons (**internal Objects**). Es gibt nur eine eingehende Kante zum Ballonobjekt. Von ihm aus kann aber auf beliebig viele Objekte, die innerhalb des Ballons sind, zugegriffen werden. Diese haben jedoch keinen Zugriff auf Objekte, die außerhalb des Ballons sind. Das Cluster zu **B** ist durch die Objekte, die grau schraffiert sind, dargestellt. Es besteht aus jenen Objekten innerhalb des Ballons, die nicht selbst Ballonobjekt eines neuen innerer Ballons sind. Dass lokale Variablen auch Teil eines Ballons sein können, ohne jedoch vom Ballonobjekt erreichbar zu sein, ist durch das **x** und dessen Zugriff auf ein internes Objekt des äußersten Ballons dargestellt.

4.3.2 Erlaubter Zugriff und Kopierzuweisungen

Wie wir gesehen haben, ist der Zugriff von außen auf Unterobjekte des Ballonobjekts nicht gestattet. Das würde keine Interaktion von außerhalb des Ballons nach innen erlauben. Darum wurde im Ballon-Schema folgende Regel zur Entschärfung des Systems bestimmt:

Die einzige Möglichkeit, wie Programmteile Zugriff auf den Inhalt von Objekten innerhalb des Ballons haben, ist die Übertragung der Referenz über eine Funktion des Ballonobjekts. [Alm97]

Ein Beispiel: Sei unser Stack ein Ballonobjekt, die Einträge befinden sich innerhalb des Ballons. In ihm ist dann die Funktion `pop()`, die den obersten Eintrag des Stapels übergibt, erlaubt! Direkter Zugriff auf die Referenz `top` wäre nicht erlaubt, da der Zugriff auf die Referenz nicht über eine Funktion geschieht, vergl. dazu das Beispiel auf Seite 39.

[18] Die Notation weicht hier von den anderen Grafiken ab, da das Bild aus dem Originalartikel stammt. Außerdem unterstützt die rechteckige Darstellung von Objekten (mit Feld von Zeiger) die Unterscheidung zu einem Ballon als Kreis.

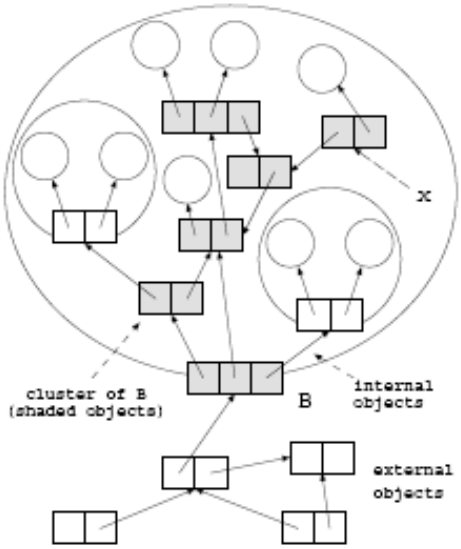

Abbildung 5: Objekte innerhalb und außerhalb eines Ballons (aus [Alm97])

Eine Frage blieb bisher noch offen: Wie wird die Invariante I_{B_1}, dass es nur eine Referenz auf das Ballonobjekt geben darf, im Typsystem definiert und damit die Prüfung dem Übersetzer übergeben? Es muss eine Zuweisung definiert werden, die zusichert, dass dies die einzige Referenz auf das Objekt ist. Im Ballon-Schema wurde deshalb die *Kopierzuweisung*[19] vorgestellt.

Die Kopierzuweisung erfolgt mit dem Zuweisungoperator **:=**. Sie veranlasst den Übersetzer zu prüfen, ob die Quellreferenz *einzig*[20] ist, vergleiche destruktiver Lesezugriff im Insel-Schema. Das wird von Konstruktoren geboten, kann aber auch über den Rückgabewert einer Funktion erreicht werden. In Java könnte man eine solche einzige Referenz überall dort finden, wo über das Schlüsselwort **new** ein neues Objekt erstellt wird.

Sei v eine Variable und b ein Ballonobjekt, dann ist

- $v = b$ nicht erlaubt, da die Referenz übertragen wird!
- $v := b$ erlaubt, da eine Kopierzuweisung erfolgt!

4.3.3 Unterscheidung transparenter und opaquer Ballons

Um eine noch stärkere Einschränkung für dynamische Entwicklungen während des Programmablaufs zu erreichen, hat Almeida den Typ des *opaquen Ballons*

[19] Im Originalartikel mit der englischen Bezeichnung *copy assignment* bezeichnet.

[20] Die englische Bezeichnung für eine einzige Referenz ist *unique reference*. Oft ist das deutsche Wort „einzig“ mehrdeutig: „die einzige Referenz“ kann auf eine nur einmal vorkommende Referenz, aber auch auf eine als „einzig“ gekennzeichnete Referenz hinweisen. Darum wird manchmal die englische Bezeichnung *unique* in Klammern hinzugeschrieben, um besser zu verdeutlichen, dass zweiteres gemeint ist.

vorgestellt, der zusätzlich zum normalen Mechanismus des Ballons (*transparenter Ballon*) die Bedingung hat, dass die Objekte innerhalb des Ballons keine Referenzen nach außen herausgeben dürfen [Alm97]. Das gilt auch für nur zeitbegrenzten Zugriff einer Variable in einer Funktion. Ein opaquer Ballon ist von außen wie eine Konstante zu sehen. Sofern mit ihm gearbeitet werden muss, kann dies nur über eine Kopie erfolgen und hat somit keine Auswirkungen auf die Objekte des opaquen Ballons.

Das verhindert die Bildung von Aliase, das Alias-Problem ist komplett gelöst. Leider bewirkt die Einzigartigkeit der Referenz große Einbußen hinsichtlich der Performanz, da eine Übergabe eines opaquen Ballonobjekts als Parameter für ein anderes Objekt nur über die komplette Kopie erfolgen kann. Um ein Array von mehreren tausend Einträgen zu kopieren, muss man jeden einzelnen Eintrag neu erstellen und in einen neuen Array einfügen. Opaque Ballons eignen sich also nur für nicht-komplexe Objekte, wie z.B. ein Zeit-Datum-Objekt oder ein Name-Anschrift-Objekt.

Opaque Ballons eignen sich für Kollektionsklassen wie einen Stack prinzipiell nicht gut, da die Einzigartigkeit das Muster eines *Iterators* verhindert. Über einen Iterator kann der Programmierer jeden einzelnen Eintrag der Reihe nach durchlaufen. Dabei werden nur die jeweiligen Referenzen der Einträge übergeben, da meistens ja nur auf ganz wenigen Einträgen tatsächlich Operationen angewendet werden. Diese Referenzübergabe ist aber mit opaquen Ballons nicht erlaubt.

4.3.4 Beispiel: Ballon eines Stacks

Im folgenden Beispiel wurde versucht, einen Stack im Ballon-Schema zu implementieren. Das Ergebnis ist aber meines Erachtens nicht wirklich zufriedenstellend, da es nicht gelungen ist[21], beim Einfügen eines neuen Eintrags auf den Stack ein Objekt zu übergeben, ohne die Ballon-Invarianten zu verletzen. Stattdessen muss jeder einzelne Wert des Objekts einzeln über die Parameter der Funktion übergeben werden. Zuerst aber der Rumpf der Klasse `Stack`.

```
1  balloon Stack {
2    ... // Deklaration der Funktionen
3    non-balloon Item {
4      int val1, val2;
5      Item(int val1, int val2) {
6        this.val1 = val1;
7        this.val2 = val2;
8      }
9    non-balloon Link {
10     Item data;
```

[21] Zumindest ist es mir persönlich nicht gelungen.

```
11      Link next;
12      Link(Item item) { this.data = item; }
13    }
14  }
```

Die Klasse `Stack` ist in der Zeile 1 als Ballon deklariert. Eine Instanz des Stacks dient dann als Ballonobjekt. Alle Objekte, die innerhalb dieser Klasse definiert werden, gehören automatisch zum Ballon des Stacks dazu. Die folgende Implementierung der Funktionen des Stacks ist hier nur mit drei Punkten angedeutet. Es ist zu sehen, dass zwei innere Klassen innerhalb der `Stack`-Klasse deklariert sind. In der Zeile 3 ist eine `Item`-Klasse zu sehen, die benötigt wird, da dieser Stack keine ganzen Objekte importieren kann, sondern nur ihre Teilwerte. Dieses beispielhafte `Item` besteht aus zwei Werten: `val1` und `val2`. In der Zeile 9 ist ein `Link` als innere Klasse deklariert. Er besteht wie üblich aus einem Daten-Objekt (hier `Item`) und der Referenz `next`. Auch `Link` ist nicht als Ballon klassifiziert. Nichtsdestotrotz gehören alle Instanzen beider inneren Klassen zum Ballon des gesamten Stacks.

Die Implementierung der Funktionen eines `Stack`-Ballons könnte wie folgt aussehen:

```
15  balloon Stack extends Collection {
16    Link top;
17    Stack() { top = null; }
18    Stack(Collection c) {
19      top = c.top;
20    }
21    void push(int val1, int val2) {
22      Link newTop = new Link(new Item(val1, val2));
23      newTop.next = top;
24      top = newTop;
25    }
26    Object pop() {
27      Object item = top.data;
28      top = top.next;
29      return item;
30    }
31    Object peek() {
32      return top.data;
33    }
34    ... // Deklaration der inneren Klassen
35  }
```

Auf den ersten Blick erkennt man keine durchgreifenden Änderungen im Vergleich zu üblichen Programmcodes eines Stacks. In der Tat übernimmt der

Übersetzer auch praktisch einen großen Teil der Aufgabe. In der Zeile 19 ist zu sehen, wie im Konstruktor die Referenz *top* frei ohne Einschränkungen verwendet werden darf. Das liegt daran, dass die Referenz vom Typ einer Klasse (innere Klasse) ist, die innerhalb des Ballons deklariert wurde.

In der Zeile 26 ist zu sehen, wie weiter oben schon beschrieben wurde, dass die Herausgabe eines Objekts (`Object`) über eine explizite Funktion des Ballonobjekts (`pop()`) erlaubt ist. Über die Funktion `push(...)` kann der Programmierer einen neuen Eintrag auf den Stack legen (Zeile 21). Wie man sieht, muss er jedoch die Werte des Eintrags einzeln übergeben, da sonst eine Abhängigkeit von außerhalb des Ballons nach innen entstehen würde. In der Zeile 22 wird nun ein `Item`-Objekt aus den Werten erstellt und über den Konstruktor der `Link`-Klasse ein `Link`-Objekt erstellt, welches in den nächsten Zeilen auf den Stack gelegt wird.

4.3.5 Kapselungsfunktion eines opaquen Ballons

Die Kapselungsfunktion für einen opaquen Ballon hat zusätzlich zu den Eigenschaften von Hogg's Inseln die Bedingung der Einzigartigkeit des Ballonobjekts (*Uniqueness*). Das bedeutet, dass eine Ballon-Kapselungsfunktion durch zwei verschiedene Funktionen, eine innere *vollständige* Kapselung und eine äußere *einzige* Kapselung (nur eine eingehende Kante), definiert werden kann. Naturgemäß gleicht die innere Kapselungsfunktion der Kapselungsfunktion einer Insel und die äußere Kapselungsfunktion der eines einzigen (unique) Objekts, wie wir in Kapitel 4.8 noch genauer sehen werden.

Es ist aber auch möglich, beide Funktionen miteinander zu kombinieren, sodass sowohl das Ballonobjekt, als auch die Knoten innerhalb des Ballons mit einbezogen werden.

Sei $o \in \mathcal{S}$, o ist Ballonobjekt, $|\triangleright\{o\}| = 1$, $H := \{o\}^{\triangleright\triangleright}$
und $H_p := \{p \in K \mid p \text{ hat Zugriff auf irgendein } h \in H\}$. Dann gilt

$$\mathbf{k}_{\text{ballon}}(o) = \begin{pmatrix} \mathcal{B} = \triangleright\{o\} \\ \mathcal{I} = \{o\} \cup H \cup H_p \\ \mathcal{R} = \{\} \end{pmatrix}$$

(vergl. [NCP+03])

Die interne Repräsentation $\mathcal{I}$ beinhaltet alle Knoten, die vom Ballonobjekt o aus erreichbar sind, auch all die Knoten, die Zugriff auf einen vom Ballonobjekt aus erreichbaren Knoten haben[22], sowie das Ballonobjekt selbst.

[22] Diese hier beschriebene Kapselfunktion eines Ballons unterscheidet sich von der Funktion in [NCP+03], da die Knoten, die zwar nicht vom Ballonobjekt aus erreichbar sind, jedoch selbst Zugriff auf ein Objekt im Ballon haben, nach meiner Ansicht auch in der Kapselungsfunktion berücksichtigt werden sollten.

Analog zu Hogg's Inseln bestehen der Bund $\mathcal{B}$ aus den Knoten, die direkten Zugriff auf das Ballonobjekt haben. Die äußeren Referenzen $\mathcal{R}$ bilden die leere Menge.

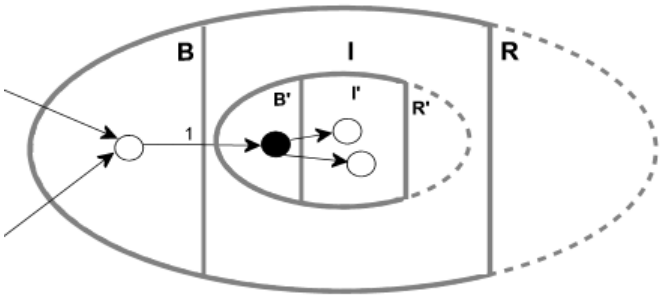

Abbildung 6: Kapselung eines opaquen Ballons (modifiziert aus [NCP+03])

In Abbildung 6 ist die Kapsel eines opaquen Ballons dargestellt. Er besteht aus einer inneren und einer äußeren Kapsel. Die äußere Kapsel beinhaltet die üblichen drei Teile $\mathcal{B}, \mathcal{I}$ und $\mathcal{R}$, die innere Kapsel ist in $\mathcal{B}', \mathcal{I}'$ und $\mathcal{R}'$ aufgeteilt. Der schwarze Knoten ist das Ballonobjekt. Es befindet sich in der internen Repräsention des äußeren Ballons und ist Ausgangspunkt (einziges Objekt im Bund) für den inneren Ballon zu den Knoten in $\mathcal{I}'$. Da keine Kante von $\mathcal{I}$ nach außen geht, bietet das Ballon-Schema totale Kapselung. Außerdem, da zusätzlich über die äußere Kapselung nur eine einzige Referenz auf das Ballonobjekt eingeht (in der Abbildung ist dies über die Beschriftung der Kante mit **1** gekennzeichnet), ist Einzigartigkeit (Uniqueness) eines Ballons gegeben.

4.4 Flexible Alias Protection (FAP)

Da totale statische Kapselung, wie es bei Inseln und Ballons der Fall ist, keinen Schutz für dynamische Referenzen bietet und andererseits keine äußeren Referenzen erlaubt waren, haben James Noble, Jan Vitek und John Potter in [NVP98] ein System mit verschiedenen Alias-Modi entwickelt, welches ein Objekt in zwei Hauptkategorien unterteilt und damit versucht, einen flexiblen Umgang mit der Kapselung zu erreichen. Zum einen enthält jedes Objekt eine *Repräsentation*, die aus einer Ansammlung von Objekten besteht, welche geschützt werden sollen. Zum anderen besteht ein Objekt aus *Argumente*[23], die mit anderen Objekten geteilt werden können und somit eine schwächere bis gar keine Kapselung verlangen. Um einen feineren und flexibleren Umgang mit den Unterobjekte eines Objekts zu ermöglichen, existieren in [NVP98] 5 verschiedene Modi für Objekte.

4.4.1 Invarianten in FAP

Flexible Alias Protection macht zusätzlich zu den Eigenschaften E_1 und E_2 von Abschnitt 2.4 eine dritte Eigenschaft für eine gekapselte Komponente erforderlich, welche zusichert, dass zwei ähnliche Variablen mit den gleichen Modi, aber unterschiedlichen Rollen nicht untereinander geteilt werden. Die Invarianten[24] einer Komponente in FAP sind demnach:

- I_{FAP_1} `no representation exposure` Die Repräsentation einer Komponente darf nicht herausgegeben werden (siehe E_1, Abschnitt 2.4).
- I_{FAP_2} `no argument dependence` Es dürfen keine Abhängigkeiten der Repräsentation zu äußeren Objekten bestehen (siehe E_2, Abschnitt 2.4).
- I_{FAP_3} `no role confusion` Unterschiedliche Rollen müssen von unterschiedlichen Variablen erfüllt werden (siehe Modi `arg` und `var`).

Das Rollenkonzept ist ein wesentlicher Bestandteil in FAP, da der Übersetzer eine Übergabe eines Objekts gleichen Typs und gleichen Modus wie die Zielvariable, jedoch mit unterschiedlichen Rollen, nicht akzeptiert. Dieses Kontstrukt sollte den Programmierer dabei unterstützen zu hinterfragen, in welchem Zusammenhang eine Variable oder Referenz benutzt wird. Bevor wir jedoch ein Beispiel geben, sollten wir die verschiedenen Modi in FAP kennenlernen.

[23] Hier hat ein Argument eine spezielle Bedeutung, deshalb wird in dieser Arbeit in den anderen Ansätzen auch nur von *Unterobjekten* eines Objekts geredet, um die Unterscheidung zu diesen Argumenten zu verdeutlichen.

[24] Hier wird von Invarianten und nicht von Eigenschaften gesprochen, da in [NVP98] *alias mode checking* vorgestellt wird, das tatsächlich diese drei Invarianten auf ihre Gültigkeit prüft.

4.4.2 Modi in Flexible Alias Protection

In FAP werden fünf Zugriffsmodi für Variablen eingeführt [NVP98]:

1. `rep` Der *rep*-Ausdruck bezieht sich auf ein Objekt, das Teil einer internen Repräsentation eines anderen Objekts (Besitzer) ist. Er kann nur von der internen Repräsentation selbst erzeugt, verändert und zerstört werden und hat auch nur Zugang zu Modifizierungen von Objekten derselben Repräsentation. Er darf niemals für Objekte freigegeben werden, die nicht Teil des Besitzers sind.

2. `arg` $\mathcal{R}$ Der *arg*-Ausdruck bezieht sich auf ein Argument eines gekapselten Objekts. Diese Argumente können herausgegeben werden, können dort nach Belieben vervielfacht (referenziert) und benutzt werden, haben aber nur Zugriff auf die unveränderbare (*immutable*) Schnittstelle des referenzierten Objekts und können deshalb keine Änderungen der internen Repräsentation dieses Objekts hervorrufen. Dies ist also eine Art *read-only* Referenz, die zwar kein *Information Hiding* ausübt, Kapselung also nur in geringem Maße unterstützt, aber den gefährlichen Manipulationszugriff von außen verwehrt.[25] Ein Ausdruck vom Typ *arg* besitzt optional eine Bezeichnung der *Variablenrolle*, um ähnliche Modi mit unterschiedlichen Rollen zu unterscheiden.

3. `free` Ein *free*-Ausdruck bezieht sich auf die einzige Referenz eines Objekts, ein Objekt also, welches kein Alias enthält und somit *einzig* (unique) ist. Free-Ausdrücke kommen zum Beispiel als Rückgabewerte von Konstruktoren vor. Sie sind destruktiv, d.h., die übergebene Referenz wird nach der Zuweisung auf `null` gesetzt. Sie entsprechen dem `free`-Modus im Insel-Schema und in ihrer Funktion der *Kopierzuweisung* im Ballon-Schema.

4. `val` Eine Variable mit Modus *val* dient zur einfachen Wertübertragung. Semantisch gesehen hat ein *val*-Argument dieselbe Bedeutung wie ein Argument mit Modus *arg*. Der *val*-Ausdruck wurde jedoch hinzugefügt, um nicht eine separate Rolle des Arguments *arg* definieren zu müssen.

5. `var` $\mathcal{R}$ Ein Argument mit Modus *var* kann nach Belieben modifiziert werden. Außerdem dürfen Aliase auf das Objekt verwendet werden. Es stellt also den typischen ungesicherten, aber freien Objekttyp dar, der Aliase behält und keinen Zugriffseinschränkungen unterliegt. Auch der *var*-Modus kann optional eine Rollenbeschreibung besitzen.

[25] Ein trifftiger Grund, warum FAP nie erfolgreich in eine Programmiersprache eingebunden worden ist, könnte meines Erachtens in der komplizierten Implementierung des `arg`- bzw. `val`-Modus in einer objektorientierte Sprache liegen.

4.4.3 Rollen einer Variable

Die Modi *arg* und *var* haben beide optional die Möglichkeit, zusätzlich eine Rolle $\mathcal{R}$ der Variable zu definieren. Das legt fest, in welchem Zusammenhang der Programmierer diese Variable sieht. Den Sinn und die Anwendung von Variablen-Rollen kann man am besten über ein kleines Beispiel zeigen.

Bei einer *Hashtable* wird normalerweise über die Methode `put(...)` ein neues Paar in die Tabelle aufgenommen. Dabei benutzt sie zwei Parameter, Schlüssel und Wert, die beide vom gleichen Typ sein können.

Eine sehr vereinfachte Hashtable [NVP98]:

```
1  class Hashtable {
2    ...
3    public void put(arg k Object key, arg v Object value) {
4      ...
5      hash(value); // Fehler zur Übersetzungszeit!
6      hash(key);   // OK, da gleiche Rolle
7    }
8    private val int hash(arg k key) {
9      // Berechne den Hash-Wert
10   }
11 }
```

Die Hashtabelle `Hashtable` benutzt eine Methode `put(...)` mit den zwei Argumenten `key` und `value`, um ein neues Schlüssel-Wert-Paar in die Tabelle aufzunehmen. Die Rolle des Schlüssels `key` ist `k`, die Rolle des Wertes `value` ist als `v` bezeichnet. Wie zu sehen ist, gäbe es hier in der Zeile 5 einen Fehler zur Übersetzungszeit. Das wäre in einem System ohne Rollenkonzept nicht der Fall, da Typ und Modus von `value` genau der Parameterdeklaration der Hashfunktion `hash(...)` in der Zeile 8 entsprechen, nur eben die Rollen nicht. In der Zeile 6 stimmen auch die Rollen überein und der Übersetzer erkennt den Funktionsaufruf als korrekt an.

4.4.4 Beispiel: Stack in FAP

Hier wird die Klasse `Stack` in Java-Notation angegeben[26], die die Schlüsselwörter von Flexible Alias Protection miteinbezieht. Das Beispiel wurde sehr einfach gehalten und enthält keine Implementierungsaspekte, sondern zeigt nur die Verwendung der unterschiedlichen Modi in FAP.

```
1  class Link extends Object {
2    arg connect Link next = null;
```

[26] Die Notation weicht hier stark von der ursprünglichen Notation in [NVP98] ab, um den Vergleich zu den anderen Ansätzen zu erleichtern.

```
3     arg elem Object data;
4     free Link(arg elem Object item) {
5       data = item; // item hat gleiche Rolle wie data
6     }
7     void setNext(arg connect Link newnext) {
8       next = newnext; // newnext hat gleiche Rolle wie next
9     }
10  }
```

Ein `data`-Objekt eines Eintrags (`Link`) in FAP wird als `arg` deklariert (Zeile 3) und hat zusätzlich eine Rollenbeschreibung mit `elem`. Dies legt fest, dass ein Objekt mit Modus `arg` nur dann als Datenobjekt gesetzt werden darf (Zeile 5), wenn es auch als Element (`elem`) fungiert. Das gleiche gilt für die Referenzen auf den vorherigen und den folgenden Eintrag `prev` und `next`. Auch sie haben Modus `arg` und dürfen somit nur eine *read-only* Referenz herausgeben. Die Rollenbeschreibung `connect` sichert analog zu `elem`, dass die Variable nur dann gesetzt werden darf, wenn der Argument-Parameter auch `connect` als Rolle hat.

Der Stack kann dann in FAP wie folgt implementiert werden:

```
11  class Stack {
12    rep Link top;
13      // Links bilden die Repräsentation des Stacks
14    free Stack() { top = null; }
15      // Modus für Rückgabewert von Konstruktoren ist free
16    free Stack(arg Collection c) {
17      rep Link tmpTop = c.top;
18      top = topTemp;
19    }
20    void push(arg connect Object item) {
21        // Modus für Eingabewerte ist arg
22      rep Link newTop = new rep Link();
23      newTop.setData(item); // item hat Rolle connect
24      newTop.setNext(top);
25      top = newTop; // OK, da newTop Modus rep hat
26    }
27    arg elem Object pop() {
28      arg elem Object item = top.data;
29      rep Link newTop = new rep Link();
30      newTop.setData(top.data);
31      newTop.setNext(top);
32      top = newTop;
33      return item;
34    }
```

```
35     arg elem Object peek() {
36       return top.data;
37     }
38     val int size();
39       // unveränderbare Nicht-Argument-Werte val
40     val boolean isEmpty() { return (top == null); }
41   }
```

Die Repräsentation eines Stacks besteht aus allen Links. Die Referenzen `top` und `tail` werden deshalb als `rep` gekennzeichnet (Zeile 12). Konstruktoren besitzen einen Rückgabewert mit Modus `free` (Zeile 14). Parameterwerte und Methoden-Rückgabewerte, die als Argumente der Ein- und Ausgabe der Repräsentation dienen, haben Modus `arg` (Zeile 20). Sie können nicht verändert werden (*immutable*), ebensowenig wie Modus `val` für einfache Wertübergabe (Zeile 38).

Schön zu sehen ist in der Methode `push(...)`, wie ein Argument-Objekt in eine Repräsentation importiert werden kann. Es muss eine Kopie vom Objekt erstellt werden, welches jedoch Modus `rep` hat. Diese Kopie stimmt nun deklarativ mit der Repräsentationsobjekt überein und kann übergeben werden (Zeile 25).

Für die Methode `push` ist der Argument-Parameter hier erlaubt, da `newTop` zuvor als Argument-Objekt mit der Rolle *connect* deklariert wurde und dies den Anforderungen von `setNext(...)` in der Klasse `Link` in der Zeile 7 genügt.

4.4.5 Kapselungsfunktion von FAP

Zur Vereinfachung der Kapselungsfunktion werden die vier Argumentmodi `arg`, `free`, `val` und `var` nicht unterschieden und zusammen in die Menge der äußeren Referenzen aufgenommen. Das bedeutet aber nicht, dass alle Objekte im System außer `rep` automatisch als äußere Referenzen vorkommen *müssen*, sondern, dass sie als Referenz vorkommen *dürfen*.

Sei $c \in \mathcal{C}$,
$rep(c) := \{\ k \in c \mid$ Modus von k ist `rep` $\}$,
$arg(c) := \{\ k \in c \mid$ Modus von k ist `arg` $\}$,
$free(c) := \{\ k \in c \mid$ Modus von k ist `free` $\}$,
$val(c) := \{\ k \in c \mid$ Modus von k ist `var` $\}$,
$var(c) := \{\ k \in c \mid$ Modus von k ist `val` $\}$, dann gilt

$$\mathbf{k}_{\text{fap}}(c) = \left(\begin{array}{l} \mathcal{B} = \triangleright c \\ \mathcal{I} = rep(c) \\ \mathcal{R} = arg(c) \cup free(c) \cup val(c) \cup var(c) \end{array} \right)$$

für $immutable(arg(c) \cup val(c))$

(vergl. [NCP+03])

Der Bund $\mathcal{B}$ in Flexible Alias Protection besteht aus den Knoten, die direkten Zugriff auf irgendeinen Knoten der Komponente haben, die interne Repräsentation $\mathcal{I}$ aus den Knoten, die als `rep` identifiziert wurden und die äußeren Referenzen $\mathcal{R}$, wie schon erläutert, aus den Knoten mit den restlichen, schwächer[27] bis gar nicht eingeschränkten (`var`) Modi.

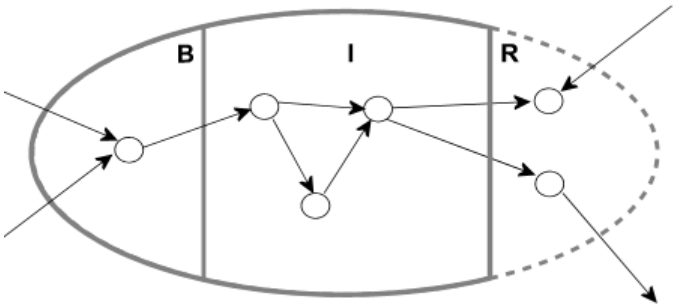

Abbildung 7: FAP-Topologie (modifiziert aus [NCP+03])

Die Zugriffsinteraktion einer Kapsel in FAP ist in Abbildung 7 dargestellt. FAP bietet einen guten Kapselungsmechanismus. Knoten, die eine eingehende Kante haben, werden im Bund aufgenommen. Sie besitzen eine Kante auf einen Knoten in der internen Repräsentation $\mathcal{I}$. Innerhalb von $\mathcal{I}$ können die Knoten freien Zugriff auf andere Knoten haben. Kanten nach außen sind erlaubt, jedoch weder vom Äußeren noch von den äußeren Refernzen darf eine Kante nach $\mathcal{I}$ gehen.

[27] Die Kapselungsfunktion unterscheidet sich von [NCP+03] in der Definition der äußeren Referenzen $\mathcal{R}$, weil die 4 unterschiedlichen Argumentmodi `free`, `arg`, `val` und `var` einzeln behandelt werden. Außerdem sind die Bedingungsfunktionen für `free`, `arg` und `val` angegeben.

4.5 Einführung von Besitzverhältnissen

Noch im gleichen Jahr haben die Autoren von FAP im Artikel „Ownership Types for Flexible Alias Protection“ [CPN98] erstmals *Besitzrelationen* zwischen den Objekten eingeführt. Das heißt, Objekte können Besitzer haben und können selbst Besitzer von anderen Objekten sein. So entsteht ein Graph von Besitzverhältnissen, der die Zugriffseinschränkungen definiert.

Die Grundideen von Besitzverhältnissen sind [PH02]:

- Es wird eine Besitzrelation zwischen einem Objekt o_1 und einem Objekt o_2 eingeführt.
- Diese Besitzrelation wird dazu benutzt, um Kapselung zu realisieren, indem Zugriff auf Objekt o_2 nur über den Besitzer o_1 gestattet ist.
- Diese Besitzrelationen werden mittels erweiterter Typinformationen deklariert.

4.5.1 Kontext-Unterscheidung eines Objekts

Das hier vorgestellte System unterscheidet im wesentlichen zwischen drei Kontexten eines Objekts, nämlich `rep`, `norep` und `owner` [CPN98].

- `rep` Ein Objekt p, das Teil der Repräsentation eines Objektes o ist, hat Kontext *rep*. Es wird damit die Besitzrelation zwischen o und p definiert durch p **ownedby** o. Nur o hat das Zugriffsrecht auf p, da o Besitzer von p ist.
- `norep` Ein Objekt p mit Kontext *norep* unterliegt keiner Besitzerrelation. Das Schlüsselwort *norep* steht für „nobody's representation“. Auf diese Objekte kann also global über das gesamte System zugegriffen werden und sie können nach Belieben referenziert werden.
- `owner` Der Kontext *owner* legt fest, dass der Besitzer eines Objektes p nicht das Objekt o ist, das p erstellt (p wird in der Klasse von o deklariert), sondern der Besitzer von o ist (p **ownedby** $owner(o)$). Der Besitzer eines Objekts kann somit auch das Besitzerobjekt eine Stufe höher in der Hierarchie des Besitzverhältnisbaums sein, aber *nicht* ein beliebiges Objekt.

Ein Programm startet vom Wurzelknoten aus. Der Wurzelknoten ist ein Objekt vom Typ `norep`, er hat also keinen Besitzer. Dies macht Sinn, da er das globale System darstellt, über das verschiedene Programmteile sich global Objekte und Datentypen (*value types*) teilen können. Das Wurzelobjekt kann aber Besitzer von anderen Objekten werden, indem Objekte innerhalb der Repräsentation des Wurzelobjekts mit dem Kontext `rep` bezeichnet werden und damit festgelegt wird, dass sie Teil des Wurzelobjekts sind und ihm gehören.

Die Besitzerrelation p **ownedby** o besagt, dass o Besitzer von Objekt p ist und somit *alleinige* Kontrolle hat. D.h., es ist u.a. zuständig für die Erzeugung und das Zerstören von p. Das hat große Ähnlichkeiten zur Definition der *Komposition* in UML [UML], die eine Verstärkung der *Aggregation* ist. Es besteht ein *Teil-Ganzes-Verhältnis* zwischen o und p. Dieses Verbindung entspricht somit einer Aggregation. Die Komposition verlangt zusätzlich, dass p kein Teil eines anderen Ganzen ist, was übertragen in die Besitzverhältnisse bedeutet, dass jedes Objekt maximal einen Besitzer haben kann. Außerdem verlangt die Komposition, dass bei der Zerstörung des Ganzes-Objekts automatisch alle seine Teilobjekte kaskadierend mitzerstört werden. Dies wird durch die Zugriffseinschränkungen auf p erfüllt.

In [CPN98] dürfen Objekte nur direkten Zugriff auf Objekte haben, die sie besitzen. Auf Objekte, die in der Besitzerhierarchie zwei oder mehr Stufen weiter unten liegen, ist kein direkter Zugriff erlaubt; dies müssten die Objekte dazwischen explizit mit einer Zugangsmethode erlauben. Das hat sehr große Ähnlichkeiten mit dem Prinzip des „schüchternem Code“.

4.5.2 Vergleich mit Law-of-Demeter

1987 wurde an der Northeastern University in Boston, Massachusetts, das Prinzip vom *schüchternen Code* vorgestellt. Das sogenannte *Law-of-Demeter* besteht aus Entwurfsrichtlinien, die das Ziel haben, eine möglichst geringe Kopplung zwischen Objekten zu erreichen. Umgangssprachlich wird dabei oft folgende Aussage verwendet:

> „Sprich nur zu deinen nächsten Freunden.“

Formal ausgedrückt soll eine Methode `m()` einer Klasse `K` nur Methoden der folgenden Klassen verwenden [Wik05]:

1. Methoden von `K` selbst
2. Methoden der Parametertypen von `m()`
3. Methoden der mit `K` assoziierten Klassen
4. Methoden von Klassen-Instanzen, die `m()` erzeugt

Abgesehen von den Nachteilen, wie mehr Codeaufwand und Performanzeinbußen, sind die Richtlinien des Law-of-Demeter (LoD) ein wirksamer Weg um sicheren Code zu erstellen. Bei den meisten objektorientierten Programmiersprachen gibt es keine explizite Unterstützung von LoD. Das Typsystem in [CPN98] hat jedoch die Funktionalität von LoD fest vorgegeben, da ein Objekt nicht Zugriff auf ein beliebiges Objekt gewährt, sondern nur auf die Objekte, die es selbst und nur es alleine besitzt. Hier kann man aber nicht mehr von Richtlinien sprechen, dies ist die Einschränkung des Typsystems selbst (vergleiche die Anmerkung in Abschnitt 4.5.6).

4.5.3 Beispiel: Einschränkungsschwierigkeiten

Hier wird ein kleines Beispiel angegeben, das die Schwierigkeiten beim Einschränken von einfachen Besitzverhältnissen zeigt (entnommen aus [CPN98]). Ein Motor ist Teil der Repräsentation eines Autos. Der Fahrer hat normalerweise keinen Zugriff auf den Motor, sondern kann ihn nur über die Funktionen des Autos starten und stoppen. In Abbildung 8 ist dieser Umstand in UML-Notation skizziert. Auto und Motor bilden eine Komposition, zwischen Fahrer und Auto herrscht eine Benutzungsbeziehung und zwischen Motor und Fahrer darf keine Verbindung bestehen.

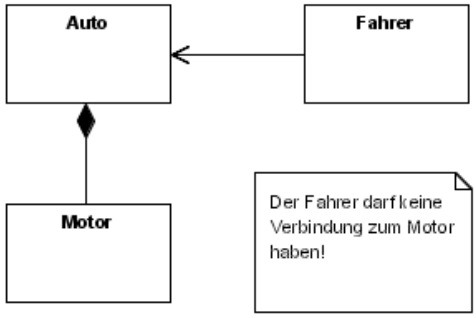

Abbildung 8: Beziehungen zw. Auto, Motor und Fahrer (UML-Notation)

Im folgenden Beispiel sind nun die Besitzverhältnissen zwischen den Klassen `Auto`, `Motor` und `Fahrer` angegeben.

```
1   class Motor {
2     void start() { ... }
3     void stop() { ... }
4   }
5   class Fahrer { ... }
6   class Auto {
7     rep Motor motor; // Teil der Repräsentation
8     Fahrer fahrer;   // kein Teil der Repräsentation
9     Auto() { motor = new rep Motor(); }
10    rep Motor gibMotor() { return motor; }
11    void setzeMotor(rep Motor m) { motor = m; }
12    void losgehts() {
13      if(fahrer != null) motor.start(); }
14  }
```

Der Motor ist Teil eines Autos (Zeile 7), nur das Auto selbst hat das Recht ihn zu starten (Zeile 12).

```
15  class Programm {
```

```
16    void main(String[] args) {
17      Fahrer franz = new Fahrer(); // kein Besitzer
18      Auto audi = new Auto();      // kein Besitzer
19      audi.fahrer = franz;
20      audi.motor.start();          // unzulässig
21      audi.losgehts();             // OK
22      audi.motor.stop();           // unzulässig
23      audi.gibMotor().stop();      // unzulässig
24      rep Motor m = new rep Motor();
25      audi.setzeMotor(m);          // unzulässig
26    }
27  }
```

Der Fahrer kann also nur über die Methode `losgehts()` den Motor zum Starten bringen, jedoch nicht direkt über `audi.motor.start()` (Zeile 20). Analog gilt dies für Zugriffe wie den Motor stoppen (Zeilen 22-23).

Der Motor kann aber leider auch nicht vom Fahrer ausgewechselt werden, weil der Fahrer dann Besitzer des Motors wäre und nicht das Auto und die Besitzverhältnisse nicht während der Programmausführung geändert werden können (Zeilen 24-25). Dieses Beispiel zeigt also auch die großen Schwierigkeiten dieser simplen Form von Einschränkungen bezüglich Besitzverhältnissen. Immer dann, wenn Objekte in die Repräsentation eines Objektes importiert werden sollten, gibt es *keine* Möglichkeit dazu.

4.5.4 Evaluierung von ownership-as-dominator

Diese Form von Besitzverhältnissen, auch als ownership-as-dominator[28] bezeichnet, ist die einfachste, aber auch die unflexibelste. Allgemeine Schwächen der einfachsten Form von Besitzverhältnissen sind:

- Es sind keine Vererbungshierarchien möglich.
- Objekte können *nicht* mehrere Besitzer haben.
- Die Besitzverhältnisse können während der Programmausführung nicht verändert werden.
- Diese Art der Zugriffseinschränkung ist zu stark für viele Entwurfsmuster der Programmierung.

Sie hat jedoch den Vorteil, dass die Einhaltung der Invarianten einfach zu beweisen ist. Es war der erste Schritt in diese Richtung, darauf aufbauend

[28] Die englische Bezeichnung „dominator“ bedeutet soviel wie „Herrscher“. Die Bezeichnung kommt daher, dass der Besitzer alleinige „Herrschaft“, in diesem Sinne Zugriff, auf das Objekt hat, vergleiche dazu LoD (Abschnitt 4.5.2).

konnten viele weitere Systeme, die weitaus flexiblere und ausgereiftere Mechanismen benutzen, entwickelt werden. Darunter sind u.a. zu nennen: [CPN99], [CPN01], [Cla02], [CD02], [BLS03]. Letzteres werden wir in Kapitel 4.7 noch etwas genauer betrachten.

4.5.5 Beispiel: Stack mit Besitzverhältnissen

Über das hier vorgestellte System könnte die Implementierung eines Stacks mit einfachen Besitzverhältnissen wie folgt erfolgen. Das Beispiel lehnt sich dem Beispiel in [CPN98] an.

Zuerst wird wie üblich die Klasse `Link` definiert:

```
1   class Link<n> {
2     owner Link<n> next;
3       // Besitzer werden übergeben
4      n Object data;
5      Link(n Object data) {
6        next = null;
7        this.data = data; }
8   }
```

Die Klasse `Stack` wird in Verbindung mit der Klasse `Link` wie folgt implementiert:

```
9   class Stack<m> {
10    rep Link<m> top; // Repräsentation
11    Stack() { top = null; }
12    void push(m Object data) {
13      rep Link<m> newTop = new rep Link<m>(data);
14      newTop.next = top;
15      top = newTop;  // gültig, da newTop Modus rep hat
16    }
17    m Object pop() {
18      m Object item = top.data;
19      top = top.next;
20      return top.data;
21    }
22    m Object peek() {
23      return top.data;
24    }
25    boolean isEmpty() { return (top == null); }
26  }
```

In diesem Beispiel ist zu sehen, dass die `Link`-Referenz `top` mit demselben Besitzer `m` wie der Stack als `rep` deklariert wurde (Zeile 10) und somit Teil

der Repräsentation des Stacks ist. Dadurch, dass `m` auch Besitzer von `top` ist, und der Besitzer von `next` in `top top` selbst ist (Schlüsselwort `owner` in der Zeile 2), gehören alle Links des Stacks dem Besitzer der Stacks. Analog gilt dies für die Daten der Stacks, da der Besitzer von `data` der Besitzer des Links ist (Zeile 4). Im Rumpf der Methode `push(...)` ist in der Zeile 13 zu sehen, wie der Import in die Repräsentation in diesem System funktioniert. Es muss eine neue Variable deklariert werden (`newtop`), die Kontext `rep` hat.

4.5.6 Kapselungsfunktion von Besitzverhältnissen

Sei $o \in \mathcal{S}$, dann gilt

$$\mathbf{k}_{\mathbf{besitz}}(o) = \begin{pmatrix} \mathcal{B} = \{o\} \\ \mathcal{I} = \{p \in \mathcal{S} \mid p \textbf{ ownedby } o\} \\ \mathcal{R} = \{q \in \mathcal{S} \mid o \textbf{ ownedby } \mathit{owner}(q)\} \end{pmatrix}$$

(vergl. [NCP+03])

Die Kapselungsfunktion von Besitzverhältnissen zeigt die starke Einschränkung des Zugriffs auf Objekte auf. Eine Kapsel stellt einen Teilgraph des Besitzverhältnisbaums dar. Die Schnittstelle (Bund $\mathcal{B}$) besteht aus dem Objekt o selbst, da *nur* dieses Objekt Zugriff auf die eigenen Objekte hat. Die interne Repräsentation $\mathcal{I}$ besteht also aus genau den Objekten, welche als Besitzer o haben. Die Menge der äußeren Referenzen $\mathcal{R}$ besteht aus all jenen Objekten, deren Besitzer das gleiche Objekt ist wie der Besitzer von o.

Anmerkung: *Der Zugriff ist nur für den* direkten *Besitzer erlaubt (ownership-as-dominator). Das bedeutet, dass auch die Besitzer des Besitzers Zugriff haben, jedoch nur über dessen besitzendes Teilobjekt. Es erfordert also zusätzlichen Programmieraufwand, um den Zugriff auf die Teilobjekte über eine Schnittstellenmethode zu erlauben, der meines Erachtens nicht unerheblich sein kann. Außerdem stört dieser Eingriff die Philosophie von guter objektorientierter Programmierung, da Objekte mehr Wissen über ihre Teilobjekte besitzen müssen. Es wäre also eine Abschwächung der Definition des Bundes sinnvoll, die direkten Zugriff für den Besitzer und alle folgenden in der Hierarchie des Besitzerverhältnissbaums höher gelegenen Besitzer erlaubt. Vergleiche dazu die Diskussion der Law-of-Demeter-Richtlinien in Abschnitt 4.5.2.*

In Abbildung 9 sind die Knoten innerhalb einer Kapsel für einfache Besitzverhältnisse zu sehen. Der schwarze Knoten im Bund ist der Ausgangsknoten, er stellt das Objekt dar, das andere Unterobjekte besitzt. Diese Unterobjekte befinden sich in der internen Repräsentation. In der Abbildung entspricht ein Pfeil vom schwarzen Punkt (nennen wir ihn o) auf einen Knoten in $\mathcal{I}$ (z.B. als p bezeichnet) der Funktion: p `ownedby` o. Auf jedes Unterobjekt, auf das zugegriffen werden kann, gibt es automatisch über den Aufruf `owner` Zugriff auf den schwarzen Knoten im Bund (dargestellt als strichlierte Pfeile von $\mathcal{I}$ nach $\mathcal{B}$), den Besitzer. Zugriffe von außerhalb der Kapsel auf Knoten in $\mathcal{I}$ sind nicht

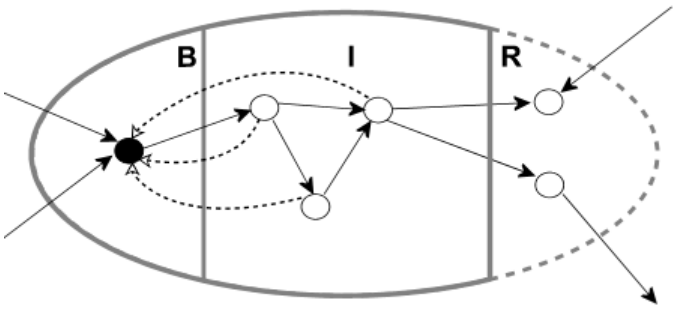

Abbildung 9: Ownership Types (modifiziert aus [NCP+03])

erlaubt. Die äußeren Referenzen $\mathcal{R}$ einer Kapsel sind hier diejenigen Unterobjekte, die sich den gleichen Besitzer teilen. Auf sie darf von $\mathcal{I}$ aus zugegriffen werden, jedoch nach $\mathcal{I}$ haben sie keinen Zugriff.

4.6 Universum-Typsystem

Universa[29] wurden in den Jahren 1999, 2000 und 2001 in den Arbeiten von Peter Müller und Arnd Poetzsch-Heffter [MPH99], [MPH00], [MPH01] vorgestellt. Sie haben einen engen Bezug zu Vorarbeiten wie Inseln (Abschnitt 4.2), Ballons (Abschnitt 4.3) und Besitzverhältnissen (Abschnitt 4.5).

- Jede Komponente im System bekommt eine Verbindung zu einem *Universum*.
- Ein Universum kann andere Universa enthalten. Ein Universum U_1 umschließt ein anderes Universum U_2 oder aber U_1 und U_2 sind disjunkt.
- Das Wurzel-Universum umschließt alle anderen Universa.
- Es entsteht ein Hierarchiebaum von Universa.
- Bezug zu Besitzverhältnissen: Ein Objekt o gehört dem untersten o umschließenden Universum im Hierarchiebaum.

Ein Universum kapselt die *Repräsentation*, nicht die Schnittstelle, und kann andere Universa enthalten. Die Objekte der Schnittstelle im Universum U, als *Besitzerobjekte*[30] bezeichnet, sind Teil des direkten Universums von U - im Hierarchiebaum das um eine Ebene höher gelegenen Universums.

4.6.1 Kontrolle über Abhängigkeiten

Besitzerobjekte - Objekte, die die Schnittstelle einer Komponente bilden - dürfen nur Referenzen auf Objekte haben, die sich im selben Universum befinden. Alle anderen Zugriffe außerhalb des eigenen Universums werden verwehrt, auch in Kind- bzw. Vateruniversa. Dies sichert zu, dass die folgenden zwei Invarianten nicht verletzt werden können (vergleiche dazu Eigenschaft E_2 in Abschnitt 2.4):

1. I_{U_1} Objekte außerhalb eines Universums dürfen keine Referenz nach innen haben, da diese Referenz dazu benutzt werden könnte, um den inneren Zustand der Komponente (die interne Repräsentation) zu verändern.

2. I_{U_2} Objekte innerhalb eines Universums dürfen keine Referenz nach außen haben, da diese Referenz eine Abhängigkeit nach außen bedeutet, die Änderungen des inneren Zustands bewirken kann, ohne diese über die Schnittstelle hervorzurufen.

[29] Im Originalartikel mit dem englischen Synonym „universes“ bezeichnet.
[30] Im Originalartikel mit dem englischen Synonym „owner objects“ bezeichnet.

Dieses System würde Objekt-Referenzen innerhalb eines Universum und außerhalb eines Universums erlauben, aber keine Referenz weder von außen nach innen, noch von innen nach außen, was viel zu sehr einschränkend ist. Um mehr Flexibilität und Freiheit zu erreichen, wurden in [MPH99] *read-only Referenzen* vorgestellt.

4.6.2 Read-Only Referenzen

Die *read-only Referenzen* bieten nur Lesezugriff auf eine Variable. Der Wert der Variable kann von einer read-only Referenz nicht verändert werden, da jeder Schreibzugriff schon zur Übersetzungszeit erkannt wird und nicht erlaubt wird. Genauso dürfen auch keine Methoden eines read-only Objekts aufgerufen werden[31].

Read-only Referenzen haben folgende Eigenschaften [MPH01]:

1. Sie sind Supertypen der korrespondierenden read-write Typen. Das bedeutet, dass `read-only o` Supertyp von `read-write o` ist.

2. Ein Ausdruck eines read-only Typs kann nicht Ziel eines Schreibzugriffs auf ein Objekt oder Aufruf einer nicht-funktionalen Methode sein.

3. Objekte der Repräsentation und funktionale Methoden mit Repräsentationsobjekten als Rückgabewert dürfen Zugriff auf read-only Referenzen gestatten, ohne die Eigenschaft E_1 zu verletzen.

4. *Downcasts*, die Konvertierung von `read-only` zu `read-write`, ist für den Besitzer eines Universums U und die Objekte, die U besitzt, möglich, wenn die read-only Referenz auf U zeigt. Das erfordert natürlich dynamisches Binden und somit Speicherung von Besitzerinformationen in jedem Objekt zur Laufzeit[32].

4.6.3 Objekt-Universum und Typ-Universum

Das Modell in [MPH99] besitzt eine einfache Komponentenaufteilung von Universa. Dabei wird zwischen zwei unterschiedlichen Universum-Typen unterschieden. Das *Objekt-Universum*[33] umschließt genau *ein* Objekt; das *Typ-Universum*[34] hat einen breiteren Kapselungskreis:

[31] Die Autoren von [MPH99] haben darauf hingewiesen, dass sogenannte *read-only Methoden* eingeführt werden könnten, die Lesezugriff über Methoden eines read-only Objekts gestatten könnten. Dies wurde aber aus Gründen der Vereinfachung des System in dieser Arbeit nicht behandelt.

[32] In [MPH01] wird darauf hingewiesen, dass dynamischen Binden nicht notwendig ist, wenn statische Analyse oder Verifikationtechniken angewandt werden.

[33] Im Originalartikel mit dem englischen Synonym „object universe“ bezeichnet.

[34] Im Originalartikel mit dem englischen Synonym „type universe“ bezeichnet.

1. Jedes Objekt besitzt eine Assoziation zu seinem eigenen *Objekt-Universum*. In anderen Worten: Es gibt für jedes Objekt ein eigenes Universum. Das heißt u. a., dass jedes Objekt eine eigene Schnittstelle besitzt, die eventuell aber eine leere interne Repräsentation hat [MPH01]. Nur das Objekt selbst ist Besitzer des „eignenen“ Objekt-Universums.

2. Jede Klasse besitzt eine Assoziation zu einem eigenem *Typ-Universum*, die alle Objektinstanzen des gleichen Typs umschließt [MPH01]. Das heißt, dass alle Instanzen einer Klasse sich innerhalb des Typ-Universums (der Klasse) befinden. Jede dieser Objektinstanzen ist ein Besitzerobjekt des Typ-Universums. Das Typ-Universum hat demnach i.A. mehrerer Besitzer.

Der zweite Punkt hat einen sehr großen Einfluss auf das Programmiermodell, da die Zugriffskontrolle sehr vereinfacht werden kann. Solange man sich innerhalb eines Moduls bewegt (Typ-Universum), ist die Anzahl an Abhängigkeiten überschaubar und bringt damit keine großen Gefahren mit sich. Dies hat große Ähnlichkeiten mit dem Prinzip des Confinement, wie in Abschnitt 4.9.1 weiter unten beschrieben wird.

4.6.4 Beispiel: Stack im Universum-Typsystem

In diesem Abschnitt wird ein `Stack` im Universum-Typsystem implementiert. Die Funktionsweise der Implementierung ist in Abbildung 10 zu sehen. Die Einträge der Liste bilden das Universum. Das Universum umschließt genau die interne Repräsentation $\mathcal{I}$, dargestellt als grauer Kreis um $\mathcal{I}$. Die Daten eines Eintrags befinden sich nicht im gleichen Universum und liegen deshalb in den äußeren Referenz $\mathcal{R}$. Auf sie haben die Link-Instanzen nur `read-only` Zugriff (dargestellt als strichlierte Pfeile).

Weiters werden zwei Iterator-Objekte in Bezug zum Stack gebracht. Sie liegen im Äußeren (außerhalb der Kapsel), da angenommen wurde, dass der Stack nichts von einem Iterator weiß. Wäre eine Funktion `getIterator()` innerhalb des Stacks implementiert, würde der Stack in den äußeren Referenzen liegen; das hätte aber keine Auswirkungen auf die Interaktion zwischen dem Iterator und dem Stack.

Ein Iterator hat Zugriff auf den Stack und eine *read-only* Referenz (position) auf einen Link des Stacks. Der Iterator erlaubt also Lesezugriff auf alle Einträge, kann aber den inneren Zustand des Stacks nicht verändern.

Wie wir gesehen haben, wird das `data`-Objekt in `Link` als `readonly` gekennzeichnet. Die Referenz auf `next` hat keine Einschränkungen, da sie das Universum so nicht verlassen kann.

```
1  class Link {
2    public readonly Object data;
```

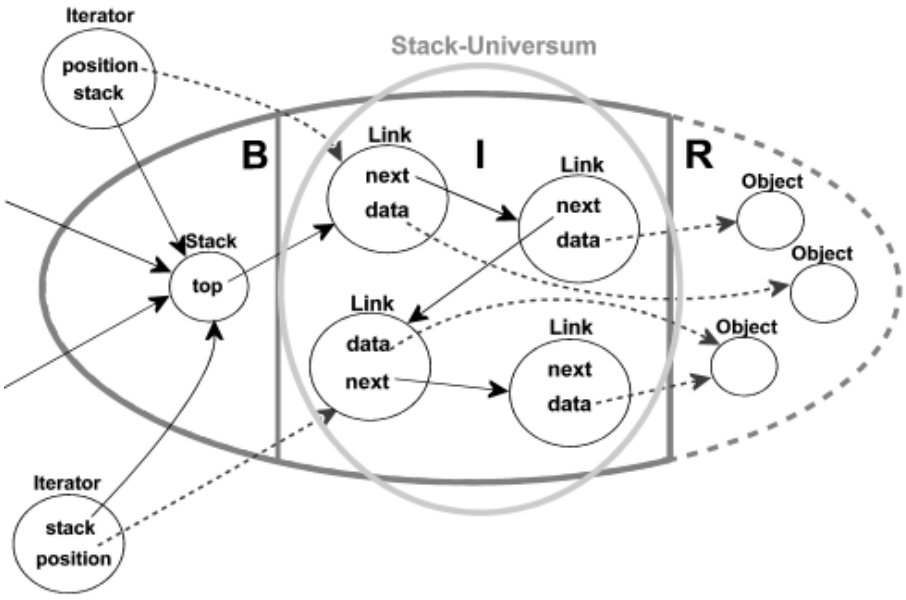

Abbildung 10: Universum eines Stacks

```
3     public Link next;
4     Link(readonly Object item) {
5       this.data = item;
6     }
7   }
```

Unser `Stack` hat wie üblich die Referenz `top`; sie wird als `rep` gekennzeichnet. Die Methode `push(...)` besitzt einen *read-only* Parameter, um den `Object`-Eintrag zu übergeben. Die Idee ist, dass die Daten der Einträge von einem anderen Universum verwalten werden. Ein Stack bildet das Universum der Einträge, wobei der Zugriff auf die Daten sowohl vom Stack, als auch von den Links, mit *read-only* beschränkt ist.

```
8   class Stack {
9     protected rep Link top;
10    public void push(readonly Object data) {
11      Link newTop = new Link(data);
12      newTop.next = top;
13      top = newTop;
14    }
15    Object pop() {
16      Object item = top.data;
17      top = top.next;
18      return item;
19    }
```

```
20     readonly Object peek() {
21       return top.data;
22     }
23   }
```

Ein Iterator für einen Stack könnte in etwa so implementiert werden (entnommen und leicht modifiziert aus [MPH01]):

```
24   public class Iterator {
25     protected Stack stack;
26     protected readonly Link position;
27     Iterator(Stack s) {
28       this.stack = s;
29       readonly Stack ros = s;
30       position = ros.top; }
31     public readonly Object next() {
32       readonly Object result = position.data;
33       position = position.next;
34       return result; }
35     public void remove() { stack.remove(position); } ...
36   }
```

Wie schon aus der Grafik aus Abbildung 10 zu sehen war, hat ein `Iterator` vollen Zugriff auf den Stack, jedoch nur eine *read-only* Referenz auf einen Eintrag (`Link`) des Stacks, die die aktuelle Position des Iterators repräsentiert (Zeile 26). Im Konstruktor wird dann aus dem `Stack`-Objekt ein *read-only* Stack-Objekt erstellt (Zeile 29), um eine read-only Referenz des obersten Eintrags des Stapels zu bekommen. Diese Referenz fungiert folglich als `position`. Wie zu erwarten übergibt die Methode `next()` das aktuelle Datum des Links der aktuellen Position als read-only und die Referenz `position` wird angepasst (Zeilen 31-34).

4.6.5 Kapselungsfunktion

Um die Kapselungsfunktion für Universa einfach zu halten, wird nur die Funktion für ein Objekt-Universum definiert. Analog könnte die Funktion für das Typ-Universum erstellt werden, zu beachten ist dabei nur, dass sich mehrere Objekte ein Typ-Universum teilen. Es werden folglich nur *read-write* Zugriffe berücksichtigt.

Sei $c \in C$, dann ist

$$\mathbf{k}_{\text{objuniv}}(c) = \begin{pmatrix} \mathcal{B} = \{o \in \mathcal{S} \mid \text{ das Objekt-Universum von } o \text{ ist } c\} \\ \mathcal{I} = \{p \in \mathcal{S} \mid \ p \text{ gehört zum Universum } c\} \\ \mathcal{R} = \mathcal{O} \end{pmatrix}$$

(vergl. [NCP+03])

Die größte Schwierigkeit beim Verständnis der Kapselungsfunktion für Universa ist meines Erachtens, dass ein Objekt-Universum bildlich gesehen kleiner ist als das Objekt selbst. Das Universum (die Kapsel) umschließt nur die interne Repräsentation $cal I$.

Der Bund $\mathcal{B}$ einer Komonente c besteht somit aus allen Unterobjekten eines Objekts, nennen wir es X, die das gleiche Objekt-Universum wie o haben (nicht bilden, denn sie sind ja Ausgangspunkt eines neuen Unteruniversums!). Sie bilden die Besitzerobjekte für das Universum c. Die interne Repräsentation $\mathcal{I}$ besteht aus allen Unterobjekten von X, die als Repräsentation deklariert wurden (über Modus `rep`). Diese Unterobjekte gehören zu c.

Eine Komponente besteht zum einen aus Besitzerobjekte (die Schnittstelle), denen gemeinsam das Objekt-Universum gehört. Zum anderen besteht sie aus den Unterobjekten, die das zugehörigen Objekt-Universum bilden (die Repräsentation).

Nach dieser strengen Einteilung in entweder Schnittstelle oder Repräsentation bleibt die Menge der äußeren Referenzen $\mathcal{R}$ klein, kann aber als Teilmenge des Äußeren $\mathcal{O}$ gesetzt werden[35].

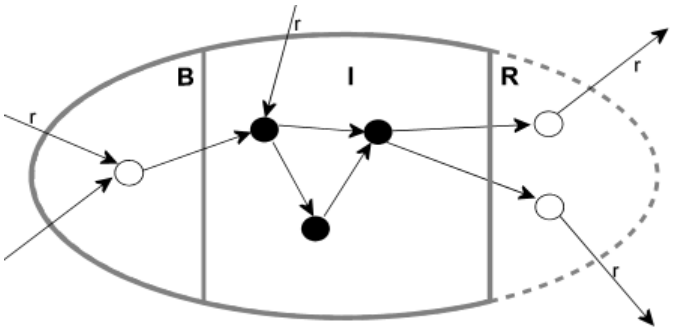

Abbildung 11: Zugriff auf ein Universum (modifiziert aus [NCP+03])

In Abbildung 11 ist eine Universum-Kapsel mit ihren Knoten und Zugriffskanten abgebildet. Ein Objekt o befindet sich im Bund $\mathcal{B}$ und kapselt alle Unterobjekte, die es besitzt (schwarze Knoten in der internen Repräsentation $\mathcal{I}$). Zu beachten sind die Pfeile mit Beschriftung **r**. Sie sind *read-only* Zugriffe und können daher keine Änderungen vornehmen, sondern nur Informationen lesen. Read-only Zugriffe dürfen auch auf Knoten in der internen Repräsentation stattfinden.

[35] Im Artikel [NCP+03] wurde die Menge der äußeren Referenzen $\mathcal{R}$ als leere Menge definiert; es ist meines Erachtens aber nicht nachvollziehbar warum. Ein einfacher Umgang mit z.B. einem String-Objekt in einer Klasse kann zwar als Schnittstelle oder als Repräsentation festgelegt werden, ist aber nicht zwingend notwendig. Auch im Universum-System ist eine einfache Benutzungsbeziehung zwischen zwei Objekten erlaubt.

4.7 Tiefe Besitzverhältnisse

Wie wir in Abschnitt 4.5 gesehen haben, ist die einfache Form von Besitzverhältnissen (ownership-as-dominator) für viele Programmiertechniken zu stark eingeschränkend. Dies liegt hauptsächlich daran, dass der Bund (Schnittstelle) auf nur ein Objekt (den direkten Besitzer) beschränkt ist. Die Systeme wurden aber darauf aufbauend weiterentwickelt und mit der Zeit immer ausgereifter. Im Jahr 2003 haben dann Boyapati et. al. in „Ownership Types for Object Encapsulation“ ([BLS03]) ein System vorgestellt, welches mehrere Objekte im Bund der Kapsel erlaubt[36]

4.7.1 Eigenschaften

Das System von Liskov und Boyapati hat sehr viele Ähnlichkeiten zum System von Noble, Clarke und Potter. Dies ist auch aus den allgemeinen Eigenschaften zu erkennen, welche sind:

1. O_1 : Jedes Objekt hat einen Besitzer.
2. O_2 : Der Besitzer kann entweder ein anderes Objekt sein oder `world`.
3. O_3 : Der Besitzer eines Objekts ändert sich nicht während der Laufzeit eines Programmes.
4. O_4 : Die Besitzverhältnisse formen einen Baum mit Wurzelknoten `world`.

Eine Änderung gibt es durch die Eigenschaft O_1, dass jedes Objekt einen Besitzer hat. Dies war in [CPN98] nicht der Fall, da es den Kontext `norep` gab. Hier wird das Objekt `world` vorgestellt, das als Wurzel des gesamten Baums der Besitzverhältnisse zu sehen ist (O_4). Dies ist in Abbildung 12 dargestellt. Es ist als Besitzer für alle Objekte gedacht, die globalen Zugriff gestatten sollen. Aus jedem beliebigen Programmteil darf auf ein Objekt mit Besitzer `world` zugegriffen werden. Außerdem, analog zu [CPN98], können alle Objekte Besitzer von anderen Objekten sein (O_2) und diese Besitzverhältnisse ändern sich nicht während der Laufzeit (O_3).

4.7.2 Zugriffsrechte

Eine der Zentralideen in [BLS03] ist die Überlegung, dass Kapselung nur dann notwendig ist, wenn zu Unterobjekten *Abhängigkeiten* bestehen.

1. Ein Objekt o besitzt Abhängigkeiten zu einem Unterobjekt p, wenn o Methoden von p aufruft und dieser Methodenaufruf in p den Zustand der Repräsentation von o auf irgendeine Art beeinflussen kann (vergleiche dazu Definitionen 2 und 4 in Abschnitt 2.3).

[36] Das genannte System ist sehr umfangreich. Deshalb können hier manche Aspekte und Konstruktionen nicht vorgestellt werden.

2. Ein Objekt o sollte alle Unterobjekte p_i besitzen, für die eine vom Objekt o ausgehende Abhängigkeit besteht.

Dadurch wird Kapselung erreicht: Wenn Objekt p zur internen Repräsentation von Objekt q gehört und o außerhalb ist, dann hat o keinen Zugriff auf q. In Abbildung 12 liegt o_7 in der internen Repräsentation von o_6 und o_1 befindet sich außerhalb der Kapsel von o_6. o_1 hat keinen Zugriff auf o_7 und o_3, aber auf alle anderen Objekte. Der Zugriff auf o_3 ist für o_1 nicht gestattet, da auch in diesem System der Zugriff nur auf die „eigenen“ Kindobjekte erlaubt ist. Da der Besitzer eines Objekts sich während der Laufzeit nicht ändern darf (O_3), kann sich die Baumstruktur natürlich auch nicht ändern. Es ist z.B. nicht möglich o_1 die Besitzerrechte von o_3 zu übergeben.

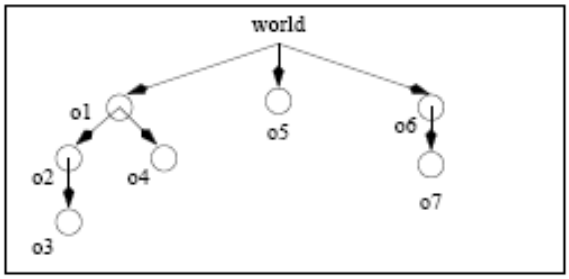

Abbildung 12: Baum der Besitzverhältnisse (aus [BLS03])

Objektzugriff ist nur erlaubt auf:

1. das eigene Objekt (`this`)
2. die Kindobjekte (Vererbungshierarchie eine Ebene weiter unten)
3. die Objekte, die das Objekt besitzt
4. globale Objekte (Objekte mit Besitzer `world`)

4.7.3 Programmstruktur

Das System in [BLS03] benutzt eine *Parametrisierung* der Klassen, welches dem parametrischen Polymorphismus ähnlich ist (siehe Abschnitt 4.9.3), nur dass die Parameter keine Typen sind, sondern *Besitzer*. Dabei ist der erste Parameter von spezieller Bedeutung: er identifiziert den Besitzer des korrespondierenden Objektes. Die anderen Parameter werden dazu benutzt, um zusätzliche Informationen über Besitzverhältnisse zu übergeben. Diese Art der Parametrisierung erlaubt die Implementierung von *generischen Klassen*, deren Objekte unterschiedliche Besitzer haben können.

Die Schlüsselwörter für Besitzer sind `this` und `world`, wobei ersteres auf ein gekapseltes Objekt hinweist, auf welches von außerhalb nicht zugegriffen werden

darf. `World` weist, wie wir schon gesehen haben, auf ein globales Objekt hin, auf das von überall, auch von außen, zugegriffen werden darf. Es gibt also keine Zugriffseinschränkungen auf diese Objekte. Bei der Deklaration einer Klasse werden Variablen für die Besitzer verwendet, und nicht die Schlüsselwörter, da nur so die Parametrisierung auch die Funktion von Parametern übernimmt (siehe folgendes Beispiel).

4.7.4 Beispiel: Besitzer-parametrisierte Klasse Auto

Eine Klassendeklaration für ein Auto, dessen CD-Sammlung nicht vom Autobesitzer stammen muss, könnte also z.B. wie folgt aussehen:

```
1  class Auto<autoBesitzer, cdSammlungBesitzer> { ... }
```

Über diese Deklaration der Klasse `Auto` kann die Klasse `AutoFahrer` die Parameter `autoBesitzer` und `cdSammlungBesitzer` verwenden, um die Besitzer des jeweiligen Objekts zu definieren.

```
2  class AutoFahrer<meinBesitzer, cdSBesitzer> {
3    ... // Ein Autofahrer besitzt ein Auto!
4    Auto<this, cdSBesitzer> meinAuto =
5      new Auto<this, cdSBesitzer>;
6    ...
7  }
8
9  class Programm {
10   void main(String args[]) {
11     AutoFahrer<this, world> franz =
12       new AutoFahrer<this, world>();
13     // Franz kann CDs von allen akzeptieren
14     AutoFahrer<this, this>  klaus =
15       new AutoFahrer<this, this>();
16       // Klaus darf nur seine eigene CD-Sammlung
17   }   // oder eine vom Programm benutzen
18 }
```

In den Zeilen 12 und 15 ist zu sehen, wie die Schlüsselwörter `this` und `world` eingesetzt werden können, um den Besitzer eines Objektes zu definieren. *Franz* hat als Besitzer den Besitzertyp `world` übergeben bekommen und kann somit CD-Sammlungen von einem beliebigen Objekt akzeptieren. Die CD-Sammlung von *Klaus* hingegen muss das Programm als Besitzer haben. Nur das Programm und Klaus selbst haben Zugriff auf die CD-Sammlung, sofern kein Unterobjekt existiert, das von Klaus geerbt hat.

Anmerkung: *Nach meiner Ansicht wäre es schön, wenn es ein Schlüsselwort wie* `yours` *geben würde, welches das Besitztum an das jeweilige Objekt übergeben würde,*

ohne Anspruch auf eigenen Besitz zu haben. Das Objekt selbst hat zwar sowieso Zugriff auf das Objekt, aber wie im Beispiel zu sehen ist, kann nicht definiert werden, dass die CD-Sammlung nur Klaus gehört, jedoch nicht dem Programm. Da das Programm Besitzer von Klaus ist, mag dies keine große Rolle spielen, für die semantische Interpretation wäre es aber sicher näher an der Realität gelegen.

4.7.5 Beispiel: Stack mit Besitzer-parametrisierten Klassen

Im folgenden Beispiel ist die verkettete Liste mit den parametrisierten Klassen implementiert.

```
1  class Link<itemOwner, dataOwner> {
2    Link<itemOwner, dataOwner> next;
3    Object<dataOwner> data;
4    Link(Object<dataOwner> data,
5      Link<itemOwner,dataOwner> next) {
6      this.next = next;
7      this.data = data; }
8  }
```

Die Parameter der Klasse `Link` (siehe Zeile 1) sind `itemOwner`, der Besitzer der Link-Instanz, und `dataOwner`, der Besitzer der Daten-Elemente. Die Referenzen zu den Links `prev` und `next` haben die gleiche Parametrisierung, um sich jeweils die gleichen Besitzer zu teilen. Nur das `data`-Objekt hat eindeutig den zweiten Parameter `dataOwner` als Besitzer festgelegt (Zeile 3). Dem Konstruktor (Zeile 5) müssen die jeweiligen Besitzer natürlich für jeden Parameter explizit angegeben werden.

```
9  class Stack<listOwner, dataOwner> {
10   Link<this, dataOwner> top;
11   Stack() { top = null; }
12   void push(Object<dataOwner> data) {
13     Link<this, dataOwner> newTop =
14       new Link<this, dataOwner> data;
15     newTop.next = top;
16     top = newTop;
17   }
18   Object<dataOwner> pop() {
19     Object<dataOwner> item = top.data;
20     top = top.next;
21     return item;
22   }
23   Object<dataOwner> peek() {
24     return top.data;
```

```
25      }
26      boolean isEmpty() { return (top == null); }
27  }
```

Die Klasse `Stack` besitzt genauso zwei Parameter. Der erste besagt, wem der Stack gehört, der zweite, wem die Daten des Stacks gehören. Letzteres wird für die Deklaration von `top` (Zeile 10) gebraucht, und natürlich auch für die meisten Funktionszugriffe vom Stack aus auf die Links wie zum Beispiel in `push` und `pop` (14 und 19).

4.7.6 Kapselungsfunktion

Sei $o \in \mathcal{S}$ und P die Menge von Instanzen einer nicht-statischen inneren Klasse, die innerhalb der Klasse, die o deklariert, eingebettet ist. Dann gilt

$$\mathbf{k}_{\text{tiefe\_besitzer}}(o) = \begin{pmatrix} \mathcal{B} = \{o\} \cup P \\ \mathcal{I} = \{q \in \mathcal{S} \mid q \textbf{ ownedby } o\} \\ \mathcal{R} = \{q \in \mathcal{S} \mid o \textbf{ ownedby } \mathit{owner}(q)\} \end{pmatrix}$$

(vergl. [NCP+03])

Die Kapselungsfunktion ist bis auf den Bund $\mathcal{B}$ mit der Kapselungsfunktion in Kapitel 4.5.6 identisch. Der Bund besteht zusätzlich zum Objekt o auch aus den *inneren Klassen*[37] von o. Es können also mehrere Objekte die Schnittstelle einer Kapsel bilden, nichtsdestotrotz ist auch dies immer noch eine sehr eingeschränkte Version (siehe Anmerkung in Kapitel 4.5.6).

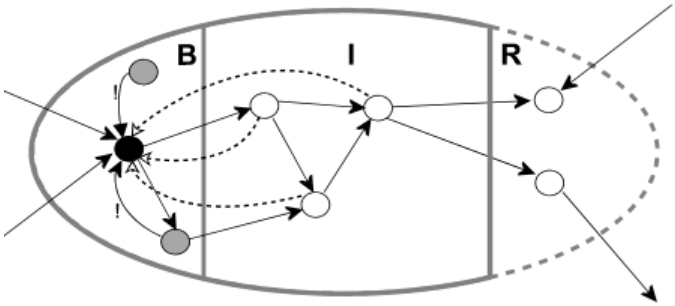

Abbildung 13: Bund mit inneren Klassen (modifiziert aus [NCP+03])

In Abbildung 13 sind zwei Objekte einer inneren Klasse zu sehen, sie befinden sich im Bund und sind grau schraffiert. Die Einbettung der Klasse ist durch die mit einem Ausrufezeichen markierten Pfeile dargestellt. Auch die Instanzen der inneren Klassen haben Zugriff auf die Objekte der internen Repräsentation $\mathcal{I}$, die natürlich aus den Objekten besteht, die das Objekt o besitzt. Ansonsten ist wie üblich $\mathcal{I}$ von außen nur über die Schnittstelle erreichbar.

[37] Natürlich wird hier von nicht-statischen inneren Klassen ausgegangen, da ja statische Klassen keinen Objekzugriff erlauben.

4.8 Externe Einzigartigkeit

Externe Einzigartigkeit kombiniert eine verfeinerte Variante von Einzigartigkeit und Besitzverhältnissen zwischen Objekten. Es besteht eine von den Autoren des Originalartikels „External Uniqueness is Unique enough“ [CW03b] entwickelte Java-Spracherweiterung `Joline`, die externe Einzigartigkeit unterstützt.

Externe Einzigartigkeit benutzt den Mechanismus der *tiefen Besitzverhältnisse*, der ähnlich zu dem System in Abschnitt 4.7 ist. Die Wurzel aller Besitzer ist ein spezieller Besitzer `world`; Besitzer werden über die Parameter einer Klasse übergeben. Näher wird auf die Besitzerverwaltung dieses Systems nicht eingegangen, da die Einführung von tiefen Besitzverhältnissen im vorherigen Abschnitt zum Verständnis der Erweiterung von externer Einzigartigkeit ausreicht.

Der Mechanismus der Einzigartigkeit wurde für dieses System genauer untersucht. Bevor wir uns den eigentlichen Mechanismus des Systems näher anschauen, sollte der Begriff der Einzigartigkeit eines Objekts verständlich gemacht werden.

4.8.1 Einfache Einzigartigkeit

Einzigartigkeit (Uniqueness) basiert auf der simplen Idee, dass Objekte, die als *unique*, oder einem Synonym davon, gekennzeichnet sind, nur eine einzige Referenz besitzen dürfen oder aber auf `null` zeigen [CW03b] [CW03a]. Aliase können somit über diese Variablen nicht erstellt werden. Diese strikte Einschränkung des Zugriffs ist, wie wir schon in Almeida's Ballons ([Alm97]) in Abschnitt 4.3 gesehen haben, oft zu hinderlich und mehr als tatsächlich benötigt. Außerdem sollte hier nochmals angemerkt werden, dass reine *Einzigartigkeit* zwar die Eigenschaft E_2 in Abschnitt 2.4 für eine gekapselte Komponente besitzt, Eigenschaft E_1 aber nur bedingt. Es wird dort verlangt, dass nur Informationen sichtbar sein dürfen, die *nicht* Teil der internen Repräsentation sind. Das heißt, einzige, als *unique* gekennzeichnete, Objekte sind zwar von Aliasen geschützt, es können keine Abhängigkeiten weder von außen noch nach außen bestehen, aber es wird keinerlei *Information Hiding* ausgeübt (vergleiche Abschnitte 4.2 und 4.3).

Anwendung fand diese Art von *Einzigartigkeit* schon lange in primitiven Datentypen wie z.B. `int`, `float`, `double` in Java[38] [GJS96]. Es gibt keine Möglichkeit, Aliase in Form von Zeigern oder anderer Form direkt[39] mit diesen Datentypen zu erzeugen.

[38] In C++ sind Zeiger auf diese Datentypen erlaubt. Dies hat aber zur Folge, dass Programme schwieriger zu verstehen sind und öfters Fehler auftreten können. Deshalb hat die Java-Spezifikation dies aus Sicherheitsgründen vermieden.

[39] Indirekt ist dies über Wrapperklassen möglich, die den Wert des Datentyps intern speichern.

Für komplexere Objekte fand Einzigartigkeit bis jetzt aber keine große Anwendung, da sie, wie schon weiter oben erwähnt, zu strikt und zu starr ist. Eine mögliche Lösung der Starrheit dieses Systems haben vielleicht David Clarke und Tobias Wrigstad in dem Artikel [CW03b] gefunden. Ziel war es, einer nach außen zu schützenden Referenz (Unterobjekt) innerhalb des Objekts, auf das das Unterobjekt referenziert, Freiheiten zu geben. Nach außen wird das Unterobjekt geschützt. Das System differenziert dazu zwischen *internen* und *externen* Referenzen.

4.8.2 Zur Unterscheidung von internen und externen Referenzen

Eine traditionelle objektorientierte Programmiersprache kann nicht zwischen Referenzen, die nur innerhalb der Datenstruktur existieren, und Referenzen, die von außen (extern zur Datenstruktur) in die Datenstruktur eintreten, unterscheiden. Wie wir weiter unten sehen werden, ist diese Unterscheidung in Verbindung mit Besitzverhältnissen (Abschnitt 4.5) jedoch für den Übersetzer keine große Schwierigkeit.

- Interne Referenzen zeigen auf ein Objekt oder einen Typen, die sich innerhalb der Datenstruktur befinden. Die interne Datenstruktur kann dabei im Allgemeinen mehr als nur das eigene Objekt (`this`) umfassen.

- Externe Referenzen zeigen auf ein Objekt oder einen Typen, die sich außerhalb der Datenstruktur der Referenz befinden.

- Die Unterscheidung von internen und externen Referenzen übernimmt nicht der Programmierer, sondern der Übersetzer über Besitzverhältnissen zwischen den Objekten.

Wenn ein Objekt o als `unique` gekennzeichnet ist, es die Datenstruktur verlässt, demnach eine extern-einzige Referenz u ist, ist die Kante zu diesem Objekt auch wirklich die einzige Kante mit Ursprung außerhalb der Datenstruktur, die auf das Objekt zeigt. Weitere Kanten *intern* zur Datenstruktur dürfen auf das Objekt bestehen (siehe Abbildung 14).

In Abbildung 14 ist der Zugriff mit *externe Einzigartigkeit* dargestellt. Dabei ist die gestrichelte Kante u zum grauen Knoten die extern-einzige Kante, die von einem Knoten außerhalb der Besitzverhältnissen des grauen Knotens ausgeht. Von dieser externen Sicht bezüglich des Besitztums des grauen Knotens (dargestellt als abgerundetes Rechteck) aus gesehen, ist sie die extern-einzige Kante u, die alleinige Kante zum grauen Knoten. Innerhalb der Datenstruktur (innerhalb des Besitztums des grauen Knoten) dürfen Zugriffe (über die Kanten) auf den grauen Knoten bestehen (interne Kante i). Externe Zugriffe f und f' sind nicht erlaubt.

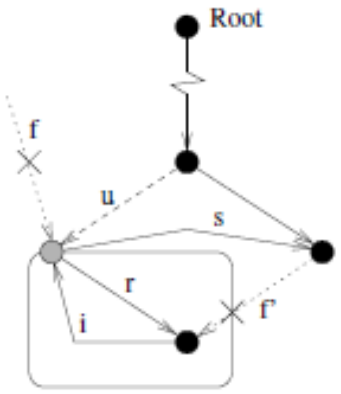

Abbildung 14: Zugriff mit externer Einzigartigkeit (aus [CW03b])

4.8.3 Operationen mit extern-einzigen Referenzen

In [CW03b] werden zwei Operationen mit externer Einzigartigkeit vorgestellt: *bewegen*[40] und *ausleihen*[41].

- `bewegen` Das Bewegen einer extern-einzigen Referenz bewirkt, dass die einzige Referenz von einer Datenstruktur zu einer anderen transportiert wird. Die Einzigartigkeit der Referenz kann dabei verloren gehen. Um dies zu verhindern, muss natürlich der Zugriff auf die Referenz nach der Bewegung aus der Quell-Datenstruktur verwehrt werden. Dies kann entweder dadurch erreicht werden, dass die Referenz der Quell-Datenstruktur gelöscht wird (auf `null` zeigt, vergleiche destruktiver Lesezugriff im Insel-Schema in Abschnitt 4.2) oder aber durch Anwendung einer Technik wie „Alias Burying“, welche statisch versichert, dass keine Aliase auf ein Objekt mehr bestehen, sobald eine einzige (unique) Variable gelesen wurde (siehe dazu [Boy01]). Hier wird ersteres angewendet; die Notation der *Nullifizierung* einer Referenz `ref` ist `!ref`[42]. Diese sogenannten *destruktiven Leseoperationen* haben also die Syntax:

$$Besitzer : Typ\ x = !y$$

- `ausleihen` Das Ausleihen bewirkt eine zeitbegrenzte Zugriffserlaubnis von mehreren Variablen auf eine als unique gekennzeichnete Variable, jedoch nur in einem bestimmten Rahmen. Ausleihen erzeugt eine nicht-einzige Referenz y mit temporären Besitzer *by* auf die auszuleihende Va-

[40] Im Originalartikel mit dem englischen Begriff „movement“ bezeichnet.
[41] Im Originalartikel mit dem englischen Begriff „borrowing“ bezeichnet.
[42] Diese Notation unterscheidet sich von der Notation in `Uniqueness`, da die ursprüngliche Notation `ref--` meiner Ansicht zu sehr einer gewöhnlichen Dekrementierung eines primitiven Datentyps in Java ähnelt. Außerdem bewirkt ein Ausrufezeichen mehr Aufmerksamkeit.

riable x mit Zugriffserlaubnis im Rahmen r.

`borrow` x `as <` by `>` y { r }

Innerhalb des Rahmens r kann nun auf x über y mit Besitzer by zugegriffen werden. Das ist auch die Notation, die von den Autoren in [CW03b] verwendet wird. Für das einfachere Verständnis und den schnelleren Umgang mit diesem Konstrukt kann die Notation stark vereinfacht werden. Da die Namensgebung innerhalb des Rahmens keine große Rolle spielt und die Referenzen den Rahmen nicht verlassen können, ist die Umbenennung im Quelltext nicht unbedingt notwendig. Diese Aufgabe kann dem Übersetzer übergeben werden oder aber eine Konvention spezifiziert werden. Deshalb wird in dieser Arbeit die folgende Notation vorgeschlagen:

`scopefor` x { r }

Im Rahmen r kann x frei dupliziert werden (zusätzliche Referenzen auf x) und über x`Owner` (zu x angehängtes „Owner“) kann auf den Besitzer von x zugegriffen werden. Die erzeugten Referenzen können aber den Rahmen r nie verlassen.

Dieses Konstrukt wird vor allem eingesetzt, um dem Übersetzer *Type Checking* zu erleichtern. Es wird ein Rahmen festgelegt, indem eine eigentlich einzige (unique) Referenz als nicht-einzige Referenz behandelt werden kann.

4.8.4 Beispiel: Stack mit externer Einzigartigkeit

Hier wird unser kleines Beispiel eines Stacks mit externer Einzigartigkeit implementiert. Zuerst die Klasse `Link`:

```
28  class Link<dataOwner> {
29    dataOwner:Object data;
30    owner:Link<dataOwner> next;
31  }
```

Sie hat einen Besitzer-Parameter `dataOwner`, der den Besitzer für das `data`-Objekt in der Zeile 29 festlegt. Dieses Objekt gehört also nicht dem `Link`-Objekt, sondern einem speziell definierten Besitzer. In der Zeile 30 wird die Referenz `next` mit dem Schlüsselwort `owner` gekennzeichnet. Auch sie gehören nicht dem *Link*-Objekt selbst, sondern dem Besitzer des *Link*-Objekts.

Die Klasse `Stack` kann nun wie folgt implementiert werden:

```
32  class Stack<dataOwner> {
33    unique:Link<dataOwner> top;
34    void push(Object<dataOwner> data) {
35      Link<dataOwner> newTop = new Link<dataOwner> data;
36      newTop.next = top;
37      top = !newTop;
38        // newTop zeigt nach der Operation auf null
39    }
40    Object<dataOwner> pop() {
41     Object<dataOwner> item = top.data;
42     unique:tmpTop = !top;
43     top = tmpTop.next;
44     return item;
45    }
46    Object peek() {
47     return Object<dataOwner> top.data;
48    }
49    void put(owner:Stack<dataOwner> otherStack) {
50      scopefor top {
51        owner[top]:Link<data> otherTop = !otherStack.top;
52        if (otherTop != null) {
53          owner[top]:Link<dataOwner> tmpTop = otherTop;
54          while (tmpTop.next != null) {
55            tmpTop = tmpTop.next;
56          }
57          tmpTop.next = top;
58          top = otherTop;
59  } } } }
```

Die Klasse `Stack` hat genauso einen Besitzer-Parameter `dataOwner` (Zeile 32), da es dadurch möglich ist, die Daten nicht an die Liste zu koppeln. Sie können einem anderen Besitzer gehören und bilden eine eigene Kapsel. Die Links werden nun als `unique` gekennzeichnet (Zeile 33). Das bedeutet, dass diese Referenzen generell *nicht* dupliziert werden dürfen, was sich ausgezeichnet für die Entkopplung zu anderen Objekte eignet. Da dies aber eine zu große Einschränkung für viele Programmiertechniken ist, wird hier auch die Technik des *Ausleihens* in Verbindung mit einer *Bewegung* über die Methode `put(...)` vorgestellt.

In der Zeile 50 ist die Kopfzeile der Rahmendefinition für `top` (Technik des Ausleihens) in der Klasse `Stack` zu sehen. Innerhalb des Rahmens (Scope) kann nun `top` frei dupliziert werden. Auf den Besitzer von `top` kann über die implizite Bezeichnung `topOwner` zugegriffen werden. In der Zeile 51 wird dann eine Bewegung durchgeführt. Der Stapel `otherStack`, der über die Methode `put(Stack otherStack)` auf den eigenen Stapel gelegt wird, überträgt die Re-

ferenz `otherStack.top` auf eine im Rahmen erzeugte Variable `otherTop` mit demselben Besitzer wie dem obersten Eintrags des eigenen Stacks `topOwner`. Die destruktive Leseoperation sichert zu, dass vorhandene Referenzen auf die Variable `otherTop` gelöscht werden. In der Zeile 58 kann schließlich eine Operation durchgeführt werden, die außerhalb des Rahmens verboten wäre. Die `unique`-Referenz des obersten Eintrags des eingehenden Stapels verliert den Bezug zum ursprünglichen Stapel.

4.8.5 Kapselungsfunktion von einfacher Einzigartigkeit

Bevor wir zur Definition der Kapselungsfunktion von externer Einzigartigkeit kommen, zuerst die Kapselungsfunktion von einfacher Einzigartigkeit, die schon in Abschnitt 4.3 indirekt zur Geltung kam.

Sei $o \in \mathcal{S}$, dann gilt

$$\mathbf{k}_{\text{unique}}(o) = \begin{pmatrix} \mathcal{B} = \triangleright\{o\} \\ \mathcal{I} = \{o\} \\ \mathcal{R} = \mathcal{O} \end{pmatrix}$$

$$\text{für } |\triangleright\{o\}| = 1$$

(vergl. [NCP+03])

Der Bund $\mathcal{B}$ besteht aus einem einzigen beliebigen Knoten im System, der eine eingehende Kante zu o hat; die Anzahl der eingehenden Kanten auf o ist auf 1 beschränkt. Dies ist auch in Abbildung 15 zu sehen: die einzige eingehen Kante nach $\mathcal{I}$ geht vom Knoten im Bund (mit **1** beschriftet) aus. Die interne Repräsentation $\mathcal{I}$ besteht aus dem eigenen Knoten o. In der Abbildung ist er schwarz gekennzeichnet. Die äußeren Referenzen eines normalen einzigen Objekts o sind unbeschränkt auf das Äußere $\mathcal{O}$ gesetzt. Das heißt, ein Objekt mit Einzigartigkeit hat nur eine einzige eingehende Kante, aber beliebig viele ausgehende Kanten nach außen (vom schwarzen Punkt in $\mathcal{I}$ gehen beliebig viele Kanten nach $\mathcal{R}$).

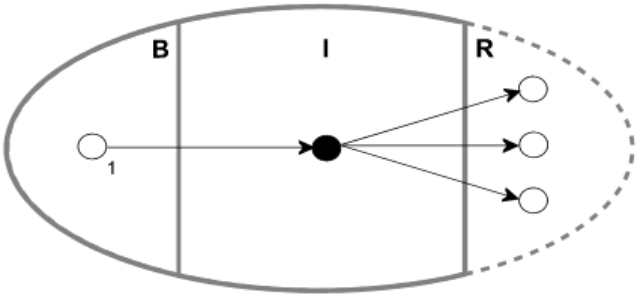

Abbildung 15: Ein Objekt mit einfacher Einzigartigkeit. (modifiziert aus [NCP+03])

4.8.6 Kapselungsfunktion von externer Einzigartigkeit

Die Kapselungsfunktion von externer Einzigartigkeit enthält im Vergleich zu normaler Einzigartigkeit Änderungen im Bund und in der internen Repräsentation.

Sei $o \in \mathcal{S}$, dann gilt

$$\mathbf{k}_{\text{ext-einzig}}(u) = \begin{pmatrix} \mathcal{B} = \{v\} \\ \mathcal{I} = \{o\} \cup \{p \in \mathcal{S} \mid p \ \leq \ o\} \\ \mathcal{R} = \{\mathcal{O}\} \end{pmatrix}$$

$$\text{für } v = (\triangleright\{o\} - \mathcal{I}) \text{ und } |v| = 1$$

(vergl. [NCP+03])

Der Bund $\mathcal{B}$ hat zusätzlich die Bedingung, dass der Knoten mit der *einzigen* eingehenden Kante zu o sich nicht in der interne Repräsentation $\mathcal{I}$ befinden darf. Wichtig ist, wie in Abbildung 16 auch zu sehen ist, dass von der internen Repräsentation eingehende Kanten nicht zählen. Die Einzigartigkeit von o wird nicht verletzt; es zählen nur eingehende Kanten von Knoten im Bund.

Die interne Repräsentation $\mathcal{I}$ besteht analog zur einfachen Einzigartigkeit aus dem Knoten o selbst, sowie allen Knoten, die o dominieren. Eine Dominierungsbeziehung $o \leq p$ (siehe Abschnitt 3.1) besteht genau dann, wenn alle Pfade, die p erreichen, irgendwann durch o gehen. Dies kann z.B. durch Zugriffsrechte mit Besitzverhältnissen in der Programmentwicklung festgelegt werden.

Die äußeren Referenzen $\mathcal{O}$ sind unverändert *alle* Knoten im Äußeren $\mathcal{O}$, obwohl diese natürlich *nicht* vorkommen müssen.

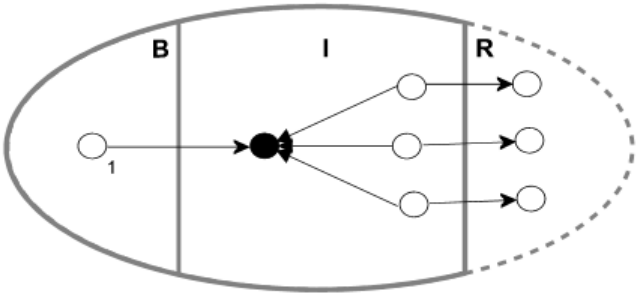

Abbildung 16: Externe Einzigartigkeit (modifiziert aus [NCP+03])

In Abbildung 16 ist grafisch dargestellt, welche Zugriffe hier im Vergleich zur einfachen Einzigartigkeit erlaubt sind. Der schwarze Knoten in der internen Repräsentation $\mathcal{I}$ ist der als *unique* gekennzeichnete Knoten. Er ist aber *nicht*

der einzige Knoten in $\mathcal{I}$; es dürfen weitere Knoten in $\mathcal{I}$ existieren und diese haben Zugriff auf Knoten im Äußeren und den einzigen (uniquen) Knoten (schwarze Knoten in $\mathcal{I}$). Dieser hat jedoch auf keinen Knoten Zugriff, auch nicht auf Knoten in $\mathcal{I}$.

4.9 Featherweight Generic Confinement

Featherweight Generic Confinement [PNCB05] ist der zur Zeit aktuellste Ansatz, um Kapselung in modernen objektorientierten Programmiersprachen zu realisieren. Dazu wurde die Sprache Featherweight Java [IAP+01], eine Variante von Generic Java (GJ), erweitert (vergleiche auch Java 1.5 [Jav05]). Die Idee ist, Besitzerverhältnisse in ein System einzubauen, das generische Klassen zulässt, somit die Syntax dieser Sprache zu definieren, sodass die Parameter für Besitzer (*owner parameter*) und Daten (*generic parameter*) sich gegenseitig nicht beeinflussen und das Ganze in Verbindung mit *Confinement* zu bringen. Confinement ist eine Kapselungstechnik, die *paketbasiert* Objekte innerhalb eines Paketes bestmöglich von Objekten außerhalb des Pakets schützt.

4.9.1 Confinement

Eine Kapsel in Confinement ist nicht auf ein Objekt oder ein paar speziell zusammengefasste Objekten begrenzt, sondern ist meist ein ganzes Paket (`package`). Auf eine Klasse, die als `public` bezeichnet wurde, kann von jeder anderen Klasse, im Paket und auch von außen, zugegriffen werden. Klassen, die als `confined`[43] deklariert wurden, bieten nur Zugriff auf Klassen, die sich im selben Paket befinden. Dieser Ansatz wurde zum Beispiel von Vitek und Bokowski in [VB01] für Java vorgestellt.

Ähnlich wie in den Ansätzen für Inseln (4.2) und Ballons (4.3) wird ein Bereich festgelegt, der geschützt ist. In diesem Ansatz ist der Bereich normalerweise ein ganzes Paket von Klassen. In Java wurden Pakete hauptsächlich für die Strukturierung von Klassen benutzt. Sie bieten auch Strukturen, um die Objektnamen innerhalb eines Paketes zu schützen (vergleiche Abschnitt 1.2)). Da ein Java-Paket aber keine Einteilung in Schnittstelle und Repräsentation vornimmt, können Abhängigkeiten *zwischen* Paketen nur sehr schwierig kontrolliert werden.

Die Erfahrung zeigt, dass ein Paket von Klassen von wenigen Programmierer entwickelt wird und deshalb die Gefahren von Abhängigkeiten innerhalb eines Paketes überschaubar bleiben und wichtige Entwurfsmuster wegen der internen Flexibilität viel einfacher anzuwenden sind. Das Paket enthält eine Schnittstelle, die *indirekt* Zugriff auf ausgewählte Objekte der Repräsentationen gewährt - direkter Zugriff ist nicht erlaubt.

Damit sollte folgende charakteristische Eigenschaft für einen geschützten Bereich in Confinement gelten:

- Der Quelltext eines geschützten Bereiches (Paket) sollte nach der Ausführung niemals *direkt* auf ein Repräsentations-Objekt eines anderen geschützten Bereiches zeigen.

[43] Das englische Wort „confined“ bedeutet soviel wie „gefangen“.

4.9.2 Kapselungsfunktion für Confinement

Seien $o \in \mathcal{S}$, $p \in \mathcal{P}(\mathcal{S})$,
$package(x)$ gebe das Paket (Confinementbereich) von x aus und $confined(x)$ definiere, dass x als confined deklariert wird. Dann gilt

$$\mathbf{k}_{\mathbf{confined}}(p) = \begin{pmatrix} \mathcal{B} = \{o \mid package(o) = p \wedge \neg confined(o)\} \\ \mathcal{I} = \{o \mid package(o) = p \wedge confined(o)\} \\ \mathcal{R} = \mathcal{O} \end{pmatrix}$$

(vergl. [NCP+03])

In Confinement ist die Kapselungsfunktion relativ einfach zu definieren. Es wird ein Paket im Wesentlich in zwei Teile geteilt. Der Bund $\mathcal{B}$ besteht aus den sich im gleichen Paket p befindenden Objekten, die nicht als *confined* deklariert wurden. Die interne Repräsentation $\mathcal{I}$ besteht aus den als *confined* deklarierten Objekten im selben Paket p. Die äußeren Referenzen $\mathcal{R}$ sind unbeschränkt auf das Äußere $\mathcal{O}$ festgelegt.

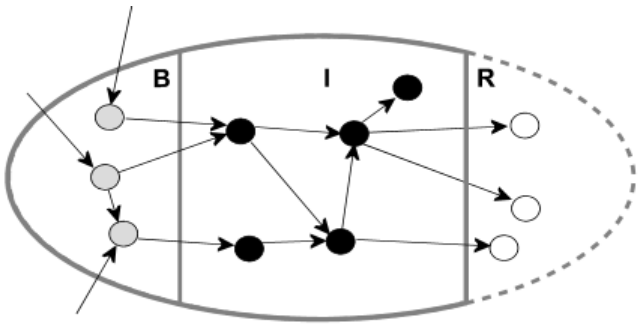

Abbildung 17: Indirekter Zugriff in Confinement (modifiziert aus [NCP+03])

Wie der Zugriff in Confinement funktioniert, kann in Abbildung 17 gesehen werden. Die Knoten der Schnittstelle eines Paketes sind als graue Knoten im Bund dargestellt, die Knoten der Repräsentation als schwarze Knoten in $\mathcal{I}$ und die äußeren Referenzen als weiße Knoten. Jeder Zugriff von außen auf entweder die Repräsentation oder auf die äußeren Refernzen geht immer über einen Knoten des Bundes (Schnittstelle). Zugriff auf Knoten im Bund ist beliebig; auf die Repräsentation des Paketes (schwarze Konten der Kapsel) nur über den Bund.

4.9.3 Generisches Java

Die Idee generischer-/parametrischer Typen ist, *Typvariablen* zu benutzen. Der Typ wird dabei nicht auf eine bestimmte Klasse festgelegt, sondern kann als Parameter bei der Erzeugung einer neuen Instanz deklariert werden. Dadurch ergeben sich parametrisierte Klassen- und Schnittstellendeklarationen.

Die Typvariablen müssen beim deklarierten Typ vereinbart werden [PH02]. Den größten Anwendungsbereich von parametrisierten Klassen bieten sicher Kollektionsklassen wie ein Stack:

```
1  class Stack<A> implements Collection<A> {
2    protected class Link {
3      A data;
4      Link next = null;
5      Link (A data) {
6        this.data = data;
7      }
8    }
9    protected Link top = null;
10   public Stack() { }
11   public void push(A data) {
12     Link newTop = new Link(data);
13     newTop.next = top;
14     top = newTop;
15   }
16 }
```

Seit der Version 1.5 sind parametrisierte Klassen in Sun's Java Development Kit [Jav05] schon implementiert[44]. Neue Kapselungssysteme sollten also darauf achten, dass die Implementierung von parametrisierten Klassen möglich ist.

4.9.4 Programmstruktur

Die Idee des generischen Confinement ist, Besitzerverhältnissen, wie sie in Abschnitt 4.5 vorgestellt wurden, zusammen mit Typinformation, wie sie in normalen parametrisierten Klassen vorkommt, über Parameter einer Klasse zu übertragen. Dabei benutzen die Autoren von [PNCB05] die letzte Stelle in der Parameterliste einer Klasse als Besitzerparameter. Das von ihnen entwickelte System wird FGJ+c (*Featherweight Generic Java + confinement*) genannt[45].

Vererbung und Subtypisierung wird zusammengefasst als *Subclassing* bezeichnet. Dies wird in Java mit dem Wort `extends` eingeleitet. Damit Subclassing in FGJ+c funktioniert, wurde folgende Programmstruktur angewandt, siehe dazu Abbildung 18:

- *Pure* FGJ+c Klassen besitzen einen expliziten Besitzer-Parameter am Ende der Parameterliste der Klassendeklaration.

[44] In C++ und anderen erfolgreichen objektorientieren Programmiersprachen wurden parametrisierte Klassen schon viel früher implementiert.

[45] Generic Java (GJ) ist einer Erweiterung zu pure Java, Featherweight Generic Java (FGJ) ist eine Erweiterung von GJ und FGJ+c ist eine Erweiterung von FGJ.

- *Manifestierte* FGJ+c Klassen erben von einer puren FGJ+c Klasse (Subclassing).
- Eine manifestierte FGJ+c Klasse besitzt einen impliziten unveränderbaren Besitzer, übernommen von der puren FGJ+c Superklasse.
- Besitzerklassen liegen außerhalb der Klassenhierarchie, weil sie von einem FGJ+c Programm nicht instanziert werden können.
- Besitzerklassen dienen nur als Besitzer für pure FGJ+c Klassen.
- In der Hierarchie der Besitzerklassen ist die vordefinierte Klasse `world`[46] ganz oben. Das Pendant zu `world` für Besitzerklassen ist `CObject<O>` in puren FGJ+c Klassen. In manifestierten FGJ+c Klassen ist dies obsolet.
- Manifestierte FGJ+c, die über einen speziellen Besitzer (nicht `world`) geschützt (*confined*) sind, nennt man auch FGJ Klassen.

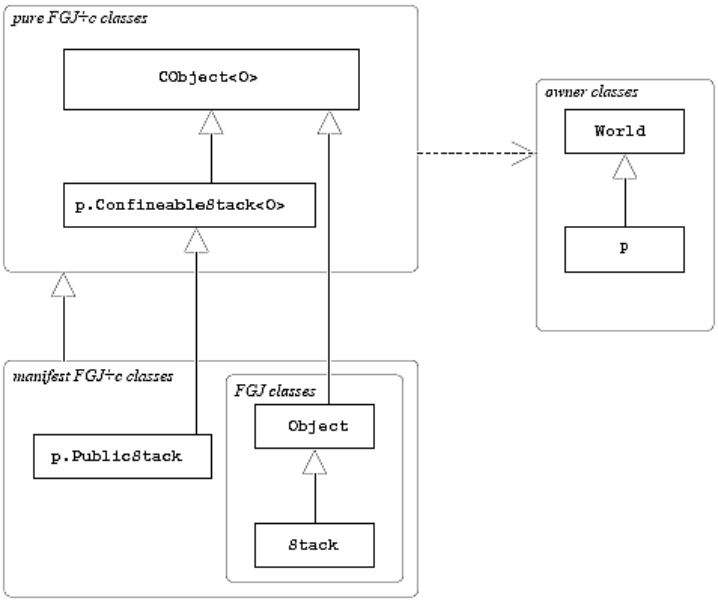

Abbildung 18: Subclassing in FGJ+c (aus [PNCB05])

In Abbildung 18 ist die Topologie in FGJ+c graphisch in UML-Notation dargestellt. Pure FGJ+c Klassen (*pure FGJ+c classes*) erben von `CObject<O>` (siehe Generalisierungsbeziehungen), so auch `p.ConfinableStack<O>`. Sie benutzen die Besitzerklassen (*owner classes*), um den Besitzer der Klasse festzulegen; dies wird mit dem gestrichelten Pfeil angedeutet (Abhängigkeitsbeziehung). Die manifestierten FGJ+c Klassen (*manifest FGJ+c classes*) haben keinen Zugang zu den Besitzerklassen. Sie erben von den puren FGJ+c Klassen und

[46] Ähnlich dem Prinzip der Klasse `Object` in einem normalen Java Programm.

haben deshalb auch denselben Besitzer wie die Superklasse. Eine Klasse `Object` und `Stack`, die über einen Besitzer (außer `world`) geschützt wird, befindet sich im Bereich der FGJ Klassen; ein öffentlicher Stack mit Besitzer `world` nicht.

4.9.5 Confinement Invariante in FGJ+c

Confinement wird in FGJ+c damit erreicht, dass jede konkrete Besitzerklasse außer `world` nur in Klassen desselben Paketes vorkommen darf[47].

Eine Besitzerklasse M im Paket m darf nur im Paket m vorkommen. Diese Restriktion verbietet jedoch nicht, dass eine FGJ+c Klasse in einem anderen Paket vorkommen darf, wie z.B. der Aufruf `m.Main()` im Paket p, weil die Klassennamen an sich nicht geschützt werden, nur die Besitzerstruktur.

```
1  class Objekt extends CObject<World>  ...
```

Mit dieser Objektdeklaration haben `Objekt` und alle weiteren FGJ Klassen darunter einen Standardbesitzer `world`. Somit sind alle diese Klassen automatisch für alle Klassen - auch außerhalb des Pakets - zugänglich.

```
2  class l.Link<Item> extends CObject<L>  ...
```

FGJ+c erfordert, dass `L` Besitzer ist von allen `Link`-Instanzen und dass `L` nur im Paket `l` sichtbar ist [PNCB05]. Das sichert zu, dass Instanzen der Klasse `Link` im Paket `l` gefangen bleiben und deshalb geschützt sind.

4.9.6 Beispiel: Stack in FGJ+c

Im folgenden Beispiel ist ein Stack in FGJ+c-Notation skizziert. Wie schon erklärt, ist die letzte Stelle der Klassen-Parameterliste für die Deklaration des Besitzers reserviert. Die Klassen `Stack` und `Link` sind im *Confinement*-Bereich (Paket) `p` deklariert. Dabei erbt `Link` von der puren FGJ+c-Klasse `CObject<P>`, wobei `P` eine nicht näher spezifizierte Owner-Klasse ist. `A` ist der Parameter, der den Besitzer den Daten festlegt.

```
1  class p.Link<A> extends CObject<P> {
2    Link<P> next;
3    A data;
4  }
```

Die Klasse `ConfineableStack` hat zwei Parameter, `A` und `M`, wobei `M` der Besitzer des Stacks ist und `M extends Owner` festlegt, dass `M` vom Besitzer `Owner` erbt. In der folgenden Klasse `Main` wird `Owner` als Sub-Besitzer von `world` deklariert. Würde anstatt `Owner` das Schlüsselwort `World` hier eingesetzt, dann gehörte der Stack allen. `A` ist auch hier der Besitzer für `Link`.

[47] Das ermöglicht eine statische Confinement-Kontrolle.

```
5   class p.ConfineableStack<A, M extends Owner> {
6     Link<A> top;
7     void push(Link<A> data) {
8       Link<A> newTop = new Link<A>data);
9       newTop.next = top;
10      top = newTop;
11    }
12    A pop() {
13      A item = top.data;
14      top = top.next;
15      return item;
16    }
17    A peek() {
18      return top.data;
19    }
20    boolean isEmpty() { return (top == null); }
21  }
```

Die Klasse `Main` befindet sich nun in einem anderen Confinement-Bereich `m` (Zeile 22). Hier wird nun der Besitzer `Owner` als Sub-Besitzer von `World` deklariert. Außerdem ist Main auch eine pure FGJ+c-Klasse, deklariert durch `... extends CObject<Owner>`.

Die zwei Methoden
`p.ConfineableList<...> publicList() ...`
und
`p.ConfineableList<...> confinedList() ...`
innerhalb der Klasse `Main` erzeugen nun je eine Liste, die jeweils aber unterschiedlich gekapselt sind. Erstere (Zeilen 26-27) erzeugt eine öffentliche Liste, da der Rückgabewert eine `ConfineableList` ist mit Besitzer-Parameter `World`. Die zweite Methode (Zeilen 29-30) erzeugt eine gekapselte[48] Methode, da der Rückgabewert als Daten-Besitzer die Klasse `Main` übergibt und als Listen-Besitzer den Confinement-Bereich (Paket) `M`. Obwohl die Methoden in der Klasse `m.Main` deklariert sind, können sie die Konstruktoren der `ConfineableList` in Paket `p` aufrufen. Es werden nur die Inhalte der Liste gekapselt.

```
22  class m.Main<Owner extends World> extends CObject<Owner> {
23    m.Main() {
24      super();
25    }
26    p.ConfineableStack<CObject<World>, World> publicList1() {
27      return new p.ConfineableList<CObject<World>, World>;
```

[48] Genauer wäre die englische Bezeichnung „confined“ mit der deutschen Übersetzung „gefangen“.

```
28    }
29    p.ConfineableStack<m.Main<World>, M> confinedList() {
30      return new p.ConfineableStack<m.Main<World>, M>;
31    }
32  }
```

5 Schlussfolgerungen und Ausblick

5.1 Kritik

Die vielen Vorteile der Kapselung wurden vor allem in den Kapiteln 1 und 2 diskutiert. Meine persönliche Einschätzung bezüglich der Anwendung der hier vorgestellten Kapselungsansätze für objektorientierte Programmiersprachen ist jedoch, dass es noch ein paar Jahre dauern wird, bis große Programmiersprachen - wie heute Java - einen solchen Mechanismus anbieten. Das hat folgende Gründe:

1. Die Anwendbarkeit (*Usability*) der unterschiedlichen Ansätze ist bei praktisch allen System ziemlich gering, da entweder viele Entwurfsmuster in der Programmierung nicht möglich sind (z.B. Iterator bei Kollektionsklassen) oder aber Flexibilität nur durch zu komplizierte Mode-Systeme erreicht wird, die Programmierer abschrecken.

2. Alle Ansätze haben ein gemeinsames großes Problem: das Fehlen von breit angelegten *Usabilitytests*. Meistens ist die Ursache die, dass ein Implementierungsprojekt eines Systems gar nicht erst begonnen wurde oder zumindest nicht mal einen Beta-Status erreichte. Auch wenn eine neue Sprache entworfen wurde, die größtenteils funktionsfähig implementiert wurde (z.B. Java-Erweiterung), gibt es immer noch das Problem viele Anwender zu finden, die Zeit und Geld investieren und dabei eventuell damit rechnen müssen, dass das Ergebnis hinsichtlich Anwendbarkeit des Systems und/oder des übersetzten Endprogramms nicht genügend ist.

3. Die *Komplexität* von solchen Programmiersystemen ist bei einer durchschnittlichen Programmierausbildung zu hoch. Auch wenn das System flexibel und sicher ist, braucht der Programmierer viel *Erfahrung* in der Entwicklung von Schnittstellen. Es muss permanent klar sein, wann eine Variable öffentlich sein darf und wann nicht, da es nicht mehr nur um die Sichtbarkeit von Objekten geht, sondern um grundsätzliche Zugriffsrecht und -verbot in der Interaktion zwischen Objekten. Zu erwarten ist, dass gute Kapselungssysteme viel eingeschränkter sind als herkömmliches Information Hiding und trotzdem Mechanismen bieten, um Zugriff explizit zu erlauben.

4. Projektleiter haben wenig Erfahrung mit und/oder *wenig Vertrauen* in neue Programmiersysteme. Das liegt vor allem an Punkt 1 und Punkt 2, hat aber auch den Ursprung im menschlichen psychologischen Verhalten. Vor allem wenn es um Status und viel Geld geht, wie es bei einer Leiterposition meistens der Fall ist, wird normalweise auf wohlbekannte und schon oft angewandte Systeme gebaut, da diese sicherer zu sein scheinen und Gefahren besser einzuschätzen sind.

5. Stichwort *Sicherheit*: Vielen Programmierern bzw. Projektleitern ist die große Gefahr des leichtsinnigen Umgangs mit dem Alias-Problem nicht vollständig bewusst. Andere wissen zwar über die Gefahr Bescheid, haben aber gelernt, mehr oder weniger damit umzugehen, und schauen sich nicht weiter um, ob es schon Lösungen dazu gibt.

6. *Abwärtskompatibilität* ist nur sehr schwierig zu erreichen. Kapselungssysteme schränken den Handlungsspielraum des Programmierers meistens sehr stark ein, der Quelltext mit z.B. Besitzerverhältnissen ändert sich im Vergleich zu normalen Subtyping ohne Besitzverhältnisse erheblich. Der Übersetzer müsste automatisch erkennen, wann Kapselung angewendet wird und wann nicht.

7. *Geschwindigkeit*: Die Dauer der Übersetzungszeit ist nicht so sehr relevant wie die Geschwindigkeit der Ausführung. Bei manchen Systemen muss die Laufzeit-Umgebung laufend Informationen von Objekten speichern. Bei z.B. dynamischen Besitzerverhältnissen zwischen Objekten (der Besitzer eines Objekts kann während der Laufzeit geändert werden) muss vor dem Zugriff immer geprüft werden, ob der Zugriff erlaubt ist. Dies übernimmt die Laufzeit-Umgebung und beansprucht deshalb auch während der Ausführung Prozessorzeit.

8. Aus meiner Sicht ist der Hauptgrund, warum Kapselungssysteme in bekannten Programmiersprachen noch nicht implementiert wurden, der *Reifungsgrad* der Systeme. Alle Systeme weisen noch gewisse Mängel im Typsystem oder in den Implementierungsmöglichkeiten des Übersetzers auf. Ich denke, Firmen wie *Sun Microsystems* arbeiten schon lange an der Entwicklung von Kapselungssystemen und haben ein wachsames Auge auf die hier vorgestellten Systeme, sehen aber, dass es noch Zeit braucht, bis solche wirklich mit Erfolg angewandt werden können.

5.2 Vergleich der unterschiedlichen Ansätze

In Tabelle 2 werden alle hier vorgeführten Ansätze und deren manchmal (teilweise) existierende Pendants der Implementierung kurz mit ihren Kernmerkmalen aufgelistet. Außerdem wird der direkte Vergleich mit Java und C++ aufgezeigt und ein Überblick über die Systeme hinsichtlich des Mechanismus der Kapselung und deren Methoden gegeben.

Es werden folgende Fragen beantwortet:

1. Wird *name protection* (Schutz von Objektnamen) geboten?
2. Wird *object protection* (Referenzkontrolle) geboten?
3. Welcher grundlegende *Mechanismus* steckt dahinter?
4. Wird die Überprüfung *statisch* oder *dynamisch* hinsichtlich der Programmausführung durchgeführt?
5. Gibt es Mechanismen, um die Anzahl von Aliasen auf 1 zu reduzieren, sprich *Einzigartigkeit*?
6. Ist *Subtyping* möglich?
7. Ist *Polymorphismus* möglich? Die Frage bezieht sich auf typisierte Klassen, nicht allein auf parametrisierte Klassen.
8. Ist während der Ausführung die *Laufzeitumgebung* mit Zugriffskontrolle beschäftigt?
9. Wie *tief* ist Kapselung mit dieser Technik erreichbar?

Folgende objektorientierte Sprachen wurden während des Entwicklungsprozesses der unterschiedlichen Ansätze mitentwickelt und werden hier kurz präsentiert:

Joe Im Artikel „Ownership, Encapsulation and the Disjointness of Type and Effect“ [CD02] wurde die Sprache `JOE` vorgestellt und steht für „Java + Ownership + Effect“. Wie der Name schon andeutet, ist sie eine Java-Erweiterung, die Besitzverhältnisse und Alias-Notation verbindet.

Joline Wurde zumindest während der Ausarbeitung des Artikels „External Uniqueness is Unique Enough“ [CW03b] (Abschnitt 4.8) für die Konkretisierung entwickelt und angewandt. Es konnte aber nicht herausgefunden werden, ob tatsächlich eine Implementierung der Sprache `Joline` existiert oder eventuell weiter daran gearbeitet wird.

		inf-hiding		alias control			feature			
		NP	OP	Kaps-Mech		Uni	Sub	PM	LZ	St	Tiefe
4.2	Inseln	×	√	mode	s	√	×	×	×	a	full
4.3	Ballons	×	√	mode	s	√	×	×	×	v	full
4.4	FAP	×	√	mode	d	free	×	×	×	v	deep
4.5	OFAP	×	√	oad	d	free	√	×	√	v	deep
4.6	Universes	√	√	o+u	d	√	√	×	√	v	deep
4.8	ExtUnique	√	√	extu	d	√	√	×	×	v	deep
4.7	OTypes	√	√	o+u	d	×	√	×	×	v	deep
4.9	FGJ+c	√	√	o+c	d	pd	√	√	×	v	deep
	Java 1.2	√	×	×	s	pd	√	×	×	×	none
	Java 1.5	√	×	×	s	pd	√	√	×	×	none
	C++	×	×	×	s	×	√	√	×	×	none
	Joe	√	√	oad	d	pd	√	×	×	v	deep
	Joline	√	√	o+u	d	pd	√	×	×	v	deep
	MultiJava	√	√	o+u	d	√	√	×	×	v	deep
	AliasJava	√	√	o+u	d	pd	√	×	×	v	deep
	OGJ	√	√	o+c	d	pd	√	√	×	v	deep

√	vorhanden
×	nicht vorhanden
a	Strategie der Aufforderung
d	dynamischer Alias-Schutz
deep	unterstützt tiefe flexible Kapselung
extu	über externe Einzigartigkeit
free	für Konstruktoren
full	totale unflexible Kapselung bereitgestellt
mode	über Mode-System
none	praktisch keine Kapselung unterstützt
oad	über ownership-as-dominator Prinzip
o+c	ownership + confinement
o+u	ownership + uniqueness
pd	für primitive Datentypen
po	Kapselungsmechanismus über Besitzerparameter
s	statischer Alias-Schutz
v	Strategie der Vorbeugung
FAP	Flexible Alias Protection [NVP98]
FGJ+c	Featherweight Generic Java plus Confinement [PNCB05]
LZ	Zugriffskontrolle über die Laufzeit-Umgebung
NP	Name protection (Schutz von Objektnamen)
OFAP	Ownership Types for Flexible Alias Protection [CPN98]
OP	Object protection (Schutz des Objektes)
PM	Polymorphismus möglich
St	Strategie im Umgang mit Aliasen
Sub	Subtyping möglich
Uni	Mechanismus für Einzigartigkeit (Uniqueness) vorhanden

Tabelle 2: Vergleich unterschiedlicher Kapselungsansätze und Programmiersprachen

MultiJava Die SCT Group der ETH Zürich hat das Universe Type System 4.6 in MultiJava [Mul], einer Java-Erweiterung, implementiert. Das Projekt ist im Internet öffentlich zugänglich und scheint funktionsfähig zu sein. Die Schwerpunkte liegen jedoch nicht nur in einem besseren Kapselungssystem, sondern in vielen weiteren Erweiterungen der Java-Sprache.

AliasJava Wurde als „alias annotation system“ bezeichnet, wo Alias-Muster explizit markiert werden. In [AKC02] vorgestellt, wurde es im ArchJava-Projekt [ACN02] als integriertes Modul implementiert und erfolgreich eingesetzt. ArchJava ist, wie der eigenen Homepage zu entnehmen, das bisher einzige System, das verifizieren kann, dass ein Programm einer Software-Architektur konform bleibt. Dabei wird die AliasJava-Erweiterung dazu benutzt, um den Datenfluss zwischen Komponenten besser zu kontrollieren, siehe dazu Abschnitt 4.7.

OGJ Eine Java-Erweiterung für die Version 1.5, also ganz aktuell. Steht für „Oh! Gee! Java!“, dem meiner Ansicht nach jedoch „Ownership Generic Java“ besser entsprechen würde. Sie ist die Implementierung des FGJ+c (Abschnitt 4.9) und vereint demnach Besitzverhältnisse, Confinement und Generisches Java in einer Sprache. Sie ist leider öffentlich im Internet nicht zugänglich. In wieweit sie tatsächlich funktionsfähig ist, konnte ich auch nicht ausfindig machen.

5.3 Ein Blick in die Zukunft

Wenn wir Java als Beispiel nehmen, hat man gesehen, dass große Änderungen an der Sprache selbst nur sehr wenig und vor allem selten vorgenommen wurden. Es kamen zur Spezifikation der ersten Version [GJS96] seit 1995 nur das Konstrukt der inneren Klassen mit Version 1.2 und einige neue Konstrukte wie parametrisierte Klassen, Autoboxing, explizite Enumerationen, usw. mit der aktuellen Version 1.5 [Jav05] hinzu. Jeglicher hier vorgestellter Mechanismus, um Kapselung in die Programmiersprache einzubauen, hat weitreichende Auswirkungen auf den Quelltext. Abwärtskompatibilität ist meiner Meinung nach nicht zu ermöglichen, was - wie schon erwähnt - ein entscheidender Faktor sein kann. Für die nächsten zwei, drei Java-Versionen sehe ich also keinen Schritt in diese Richtung.

Eher wahrscheinlich halte ich eine komplett neue objektorientierte Programmiersprache, die sich speziell an stark sicherheitrelevanten Aufgaben orientiert. Aktuelle objektorientierte Sprachen sind für solche Aufgaben oft gemieden worden, weil das Alias-Problem bekannt war und deshalb lieber auf bewährte imperative Programmiersprachen gesetzt wurde. Dort können zwar ganz ähnliche Probleme auftreten, aber die objektorientierte Programmierung verleitet viel mehr dazu, Aliase zu benutzen.

Von den hier vorgestellten Systemen halte ich die Kombination von Confinement und Besitzverhältnissen und die Miteinbeziehung von generischen Klassen (Abschnitt 4.7) für den wohl bis jetzt ausgereiftesten Ansatz. Trotzdem sind die sehr starken Einschränkungen aufgrund des „Law-of-Demeter"-Prinzips innerhalb der Besitzverhältnissen vielleicht für die Programmierung nicht optimal, da der Besitzerbaum sehr komplex werden kann und die Laufzeit erheblich beeinträchtig wird. Deshalb, und vor allem vor dem Hintergrund der Abwärtskompatibilität von schon vorhandenen Programmiersystemen und der eher geringeren Komplexität der Programmierung, wird *Confinement*, paketbasierte Kapselung, die vielleicht erste breit angewandte Kapselungsmaßnahme sein, die über Information Hiding hinaus geht. Ob Confinement aber wirklich die richtige Antwort auf das Alias-Problem ist, wird sich erst dann herausstellen.

Abbildungsverzeichnis

Tabellenverzeichnis

Literatur

[ACN02] Jonathan Aldrich, Craig Chambers, and David Notkin. Archjava: Connecting software architecture to implementation (http://www.archjava.com). *Proceeding International Conference on Software Engineering, Orlando, Florida, USA*, Mai 2002.

[Ad90] Pierre America and Frank deBoer. A sound and complete proof system for spool. *Technical Report 505, Philips Research Laboratories*, Mai 1990.

[AKC02] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias annotations for program understanding. *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 2002.

[Alm97] Paulo Sérgio Almeida. Balloon types: Controlling sharing of state in data types. *ECOOP Proceedings*, Juni 1997.

[BLS03] Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. *Ownership Types for Object Encapsulation*. http://citeseer.ist.psu.edu/boyapati03ownership.html, Jänner 2003.

[Boy01] John Boyland. Alias burying: Unique variables without destructive reads. *Software - Practice and Experience*, Mai 2001.

[Boy04] Chandrashekar Boyapati. Safejava : A unified type system for safe programming, Februar 2004.

[CD02] David Clarke and Sophia Drossopoulou. *Ownership, encapsulation, and the disjointness of type and effect*. http://citeseer.ist.psu.edu/730535.html, 2002.

[Cla02] David Clarke. Object ownership and containment, 2002.

[CPN98] David G. Clarke, John M. Potter, and James Noble. *Ownership Types for Flexible Alias Protection*, volume 33:10 of *ACM SIGPLAN Notices*. ACM Press New York, http://citeseer.ist.psu.edu/clarke98ownership.html, Oktober 1998.

[CPN99] David Clarke, John Potter, and James Nobel. Object ownership for dynamic alias protection. 1999.

[CPN01] David Clarke, John Potter, and James Nobel. Simple ownership types for object containment. *European Conference for Object-Oriented Programming*, Juni 2001.

[CW03a] David Clarke and Tobias Wrigstad. External uniqueness. *Workshop*, 2003.

[CW03b] David Clarke and Tobias Wrigstad. *External uniqueness is unique enough*. 2003.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, http://java.sun.com, 1996.

[HLW+92] John Hogg, Doug Lea, Alan Wills, Dennis deChampeaux, and Richard Holt. *The Geneva Convention on the Treatment of Object Aliasing*, volume 3. http://citeseer.ist.psu.edu/hogg92geneva.html, 1992.

[Hog91] John Hogg. Islands: Aliasing protection in object-oriented languages. *OOPSLA Proceedings*, November 1991.

[IAP+01] Igarashi, Atsushi, Pierce, Benjamin, Wadler, and Philip. Featherweight java: A minimal core calculus for java and gj. acm transactions on programming languages and systems. *ACM Transactions on Programming Languages and Systems (TOPLAS), 23(3), 396450*, 2001.

[Jav05] *Java Development Kit*. Sun Microsystems, http://java.sun.com, 2005.

[MPH99] Peter Müller and Arnd Poetzsch-Heffter. Universes - a type system for controlling representation exposure. Technical report, Fernuniversität Hagen, November 1999.

[MPH00] Peter Müller and Arnd Poetzsch-Heffter. A type system for controlling representation exposure in java. ECOOP Workshop on Formal Techniques for Java Programs Nummer TR 269, Fernuniversität Hagen, 2000.

[MPH01] Peter Müller and Arnd Poetzsch-Heffter. Universes - a type system for alias and dependency control. Technical report, Fernuniversität Hagen, 2001.

[Mul] *The MultiJava Project*. http://multijava.sourceforge.net.

[NCP+03] James Noble, David Clarke, Alex Potanin, Robert Biddle, and Ewan Tempero. *Towards a Model of Encapsulation*. http://citeseer.ist.psu.edu/637430.html, 2003.

[NVP98] James Noble, Jan Vitek, and John Potter. *Flexible Alias Protection*, volume 1445. http://citeseer.ist.psu.edu/47376.html, 1998.

[PH02] Arnd Poetzsch-Heffter. Fortgeschrittene aspekte objektorientierter programmierung, 2002.

[PHMM+03] Arnd Poetzsch-Heffter, J. Meyer, Peter Müller, Jens Knoop, Markus Müller-Olm, and Ursula Scheben. *Einführung in die objektorientierte Programmierung*. Fernuniversität Hagen, 2003.

[PNCB05] Alex Potanin, James Noble, David Clarke, and Robert Biddle. Featherweight generic confinement. *Foundations of Object-Oriented Languages (FOOL11)*, 2005.

[UML] *UML Version 1.4. 2001. Unified Modeling Language. The Object Management Group (OMG).* http://www.omg.org.

[VB01] Jan Vitek and Boris Bokowski. Confined types in java. *Software Pratice & Experience 31(6), 507-532*, 2001.

[Wik05] *Wikipedia.* http://de.wikipedia.org/wiki, Juni 2005.

A Erklärung

Ich erkläre, dass ich die Arbeit selbständig angefertigt und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Feldkirch, den 21. Dezember 2005 Mario Bertschler