Generic Refactoring

Björn Hagemeier <bjoernh@uni-paderborn.de>

3. März 2003

Seminararbeit im Rahmen des Seminars *Refactoring in eXtreme Programming*
Wintersemester 2002/2003
Prof. Dr. Uwe Kastens
Fakultät EIM
Universität Paderborn
Betreuer: Jochen Kreimer <jotte@uni-paderborn.de>

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Refaktorisieren ist besonders im Bereich der objektorientierten Programmierung bekannt geworden. Nicht ohne Grund stellt Martin Fowler in [Fow00] Refaktorisierungen in ausschließlich in Java vor. Dass Refaktorisierungen auch in anderen Programmierparadigmen als der objektorientierten Programmierung angewendet werden können ist lange bekannt.

Ralf Lämmel und Joost Visser gehen in [Läm02b] und [LV03] noch weiter und stellen generische Refaktorisierungen vor, welche für die unterschiedlichsten Paradigmen, Sprachen und betrachteten Codefragmente anwendbar sind. Sie zeigen dies an Hand der Refaktorisierung Methoden-Extraktion. Eine Voraussetzung für die Mächtigkeit des generischen Refaktorisierens ist eine hinreichen allgemeine Beschreibung der zu Grunde liegenden Algorithmen.

Welcher Werkzeuge es zum generischen Refaktorisieren bedarf und in wie weit man im Falle einer konkreten Implementierung für eine zu behandelnde Sprache sprachspezifischen Code schreiben muss, soll in dieser Arbeit geklärt werden.

2 Motivation

Durchgehendes Beispiel in dieser Arbeit soll die Methoden-Extraktion oder auch Abstraktion sein. Eine Abstraktion fasst Codefragmente, die möglicher Weise mehrfach im Code vorkommen oder ein bestimmtes Ziel verfolgen, unter einem Namen zusammen. Die abstrahierten Codefragmente werden dann durch die Anwendung (z. B. ein Methodenaufruf) ihrer Abstraktion ersetzt. Dadurch kann der Code übersichtlicher und leichter verständlich werden, weil mehrere Statements durch die Anwendung ihrer Abstraktion ersetzt werden können und Methoden dadurch kürzer werden. Zusätzlich kann mehrfach in einem Programm vorkommender Code immer durch eine Anwendung seiner Abstrakti-

```
void printOwning(double amount) {
      printBanner();
      // print details
      System.out.println(‘‘name:’’ + _name);
      System.out.println(‘‘amount’’ + amount);
}

                              ⇓

void printDetails(double amount) {
      System.out.println(‘‘name:’’ + _name);
      System.out.println(‘‘amount’’ + amount);
}
void printOwning(double amount) {
      printBanner();
      printDetails(amount);
}
```

Abbildung 1: Extraktion einer Java-Methode

on ersetzt werden. Ein Problem ergibt sich dabei durch lokale Variable. Diese können unter der Voraussetzung, dass keine Werte an diese zugewiesen werden, als Parameter übergeben werden. Redundanter Code kann also auf diese Weise in vielen Fällen verhindert werden, was sich unter verschiedenen Aspekten positiv auf den Code auswirkt. Jede Änderung kann an einer zentralen Stelle, nämlich in der Abstraktion, vorgenommen werden. Die Abstraktion sollte unmittelbar verständlich werden, nach Möglichkeit schon durch ihren Namen.

Ich möchte zunächst an zwei Beispielen erläutern, dass man für Refaktorisierungen grundlegende Algorithmen finden kann, die auf Verschiedene Programmierparadigmen bzw. Sprachen anwendbar sind. Das erste Beispiel in Abbildung 1 zeigt ein Java Codefragment, das abstrahiert werden soll. Das zweite Beispiel in Abbildung 2 zeigt Haskell Typ-Definitionen, von denen für Teile einer Definition eine Abstraktion erstellt werden soll. Beide Beispiele werden zu diesem Zweck auch in [Läm02b] verwendet. Das Beispiel in Abbildung 1 stammt ursprünglich aus [Fow00].

2.1 Java-Extraktion

In Abbildung 1 wird eine Methoden-Extraktion dargestellt. Die Statements der Zeilen 4 und 5 aus dem oberen Ausschnitt sollen eine Methode als Abstraktion bekommen, die aufgerufen werden kann. Ein solcher Aufruf wird allgemein auch als Anwendung einer Abstraktion bezeichnet. Später werden wir sehen, dass eine Anwendung nicht immer ein Methodenaufruf sein muss, sondern u. A. abhängig vom Programmierparadigma ganz anders aussehen kann.

Originalprogramm mit Focus

data*Prog* = *ProgProgName* [Dec][Stat]
data*Dec* = *VDecIdType*|...
data*Stat* = *AssignIdExpr*|*IfExprStatStat*|...

Verändertes Programm

data*Prog* = *ProgProgNameBlock*
data*Block* = *Block*[*Dec*][*Stat*]
data*Dec* = ...
data*Stat* = ...|*BlockStatBlock*
...

Abbildung 2: Extraktion von Haskell Datentypen

Paradigma	Fokus	Abstraktion
OO Programmierung	Statements	Methode
OO Programmierung	Eigenschaften	Klasse
Funktionale Programmierung	Ausdruck	Funktion
Funktionale Programmierung	Typausdruck	Datentyp
Funktionale Programmierung	Funktionen	Typklasse
Logische Programmierung	Literal	Prädikat

Abbildung 3: Extraktion in verschiedenen Paradigmen

2.2 Haskell-Extraktion

In Abbildung 2 werden einige Typdefinitionen in Haskell gezeigt. Teile der der Definition eines Programmblocks sollen einen abstrahierenden Typ bekommen. Das Fragment [*Dec*][*Stat*] bekommt eine eigene Abstraktion in der Typdefinition eines *Block* . Eine Anwendung der Abstraktion *Block* wird an Stelle des ursprünglichen Fragmentes eingefügt.

In diesem Beispiel wird auch deutlich, dass die Anwendung einer Abstraktion nicht unbedingt ein Methodenaufruf sein muss, sondern vom Programmierparadigma und dem betrachteten Fokus abhängig ist.

2.3 Verschiedene Paradigmen

Abbildung 3 zeigt, welche Bedeutung Extraktion für verschiedene Paradigmen abhängig vom betrachteten Codefragment haben kann. Ein betrachtetes Codefragement wird auch als Fokus bezeichnet.

Das in Abbildung 1 behandelte Java-Beispiel war also eine Instanz des Problems aus der ersten Zeile der Tabelle. Der betrachtete Fokus, Java-Statements, wurde zu einer Methode als Abstraktion zusammengefasst.

Das Beispiel aus Abschnitt 2.2 war ein Beispiel dafür, Typausdrücke in funktionaler Programmierung als eigenen Datentyp zu abstrahieren.

Den vorangehenden Beispielen gemein war, dass ein Codeblock zusammengefasst und diesem als Abstraktion ein Name gegeben wird. Das ursprüngliche Codefragment wird durch eine Anwendung der erzeugten Abstraktion ersetzt. In einer imperativen Programmiersprache entspricht diese Anwendung einem Methoden- oder Funktionsaufruf.

3 Generisches Refaktorisieren

An dieser Stelle möchte ich eine Vorstellung über generisches Refaktorisieren vermitteln. Es soll die Frage geklärt werden, wie weit der generische Ansatz gehen kann und welche Anteile sprachabhängig definiert werden müssen. Danach wird ein Framework vorgestellt, dass die notwendigen Fähigkeiten bietet, um generische Refaktorisierungen zu implementieren.

3.1 Allgemeines

Eelco Visser stellt in [Vis00] fest, dass den meisten Refaktorisierungen die selben Algorithmen zu Grunde liegen, auch wenn sie auf verschiedene Sprachen oder Paradigmen angewendet werden.

Generisches Refaktorisieren bedeutet, dass Refaktorisierungen unabhängig von der verwendeten Programmiersprache oder dem Programmierparadigma definiert werden können. Dazu bedarf es einer Abstraktionsebene, die ausreichend mächtig ist, um Konstrukte möglichst vieler Paradigmen und Sprachen zu beschreiben. Gleichzeitig bedarf es einer gewissen Flexibilität einzelner Abstraktionen, damit die Schnittstelle nicht unnötig groß wird und man die Vorteile des generischen Ansatzes, nämlich die Allgemeingültigkeit der Refaktorisierungs-Algorithmen, voll ausnutzen kann.

3.2 Strafunski

Strafunski ist ein allgemeines funktionales Framework aus dem Bereich 'language processing'. Es ist sehr flexibel gestaltet, so dass ohne großen Aufwand Funktionalität für beliebige Sprachen hinzugefügt werden kann. Genauer gesagt gehört es zum Konzept des Framework, dass sprachspezifische Anteile den zentralen Komponenten als Parameter übergeben werden.

Lämmel stellt die Struktur des Framework in [Läm02b] aus den folgenden Teilen bestehend dar:

1. Generische Algorithmen für einfache Analyse und Transformationen, die für Refaktorisierungen notwendig sind. Diese bekommen als Parameter Hilfsfunktionen übergeben, die die Konstrukte der Programmiersprache auf syntaktischer Ebene analysieren und solche auch konstruieren können.

2. Ein Abstraktions-Interface, dass die Fähigkeit besitzt, die für die spezielle Refaktorisierung notwendigen Konstrukte zu erzeugen und zu analysieren. Das Abstraktions-Interface gehört zu den Hilfsfunktionen, die den generischen Algorithmen übergeben werden.

3. Generische Refaktorisierungen werden als generische Algorithmen definiert, die mit Hilfe der Hilfsfunktionen und des Abstraktions-Interface arbeiten.

Lämmel stellt zusätzlich noch die Behauptung auf, dass generische Ansätze nicht nur für Extraktion, sondern für fast alle Refaktorisierungen Sinn machen. Den Beweis oder eine Begründung dieser Aussage bleibt er jedoch schuldig.

3.3 Funktionale Strategien

Funktionale Strategien sind eine bestimmte Art von Funktionen, die für die Analyse der Programmtexte beim Refaktorisieren gebraucht werden. Sie haben die Eigenschaft, dass sie sowohl auf Terme jeden Typs angewendet werden können, sondern auch in die Unter-Terme absteigen können.
Definition aus [LV01]:

> typeful generic functions that not only can be applied to terms of any type, but which also allow generic traversal into subterms.

In Strafunski werden funktionale Strategien verwendet, um die Traversierung der Syntax-Bäume zu realisieren und beim Besuch der Knoten Transformationen oder andere Funktionen auszuführen. Im Gegensatz zum Stratego-Ansatz aus [Vis00], wo generische Algorithmen nur zum Traversieren verwendet werden, werden in Strafunski auch die Transformationen der Terme generisch beschrieben. Ein Abstraktions-Interface sorgt für die Übersetzung der Refaktorisierungen in die behandelte Programmiersprache. Weitere sprachspezifische Funktionen werden den generischen Algorithmen als Parameter übergeben. Diese können z. B. Variablendeklarationen und -definitionen aus dem Quelltext analysieren und Programmkonstrukte der behandelten Sprache erzeugen. Diese sprachspezifischen Funktionen können, z. B. im Fall von *check* im später aufgeführten Beispiel, wieder generische Funktionen benutzen, um ihre Aufgaben zu erfüllen. Im genannten Fall wird ein generischer Algorithmus zum Finden freier Variabler wiederverwendet. Freie variable sind solche, die in einer Funktion oder Methode referenziert, aber nicht als Parameter übergeben werden.

3.4 Kombinatoren für Strategien

In Abbildung 4 sind die grundlegenden Kombinatoren für Strategien aufgelistet. Kombinatoren sind Funktionen ohne freie Variablen, d. h. alle referenzierten Variablen sind solche, die als Parameter an sie übergeben wurden. Die Abbildung zeigt bei den Traversierungs-Kombinatoren nur solche, die eine Ebene eines Syntaxbaumes traversieren können. Kombinatoren, die ganze Bäume traversieren, können rekursiv aus diesen erzeugt werden, wie in Abbildung 6 zu sehen ist.

Kombinatoren für Strategie-Anwendung sind solche, die eine Funktion bzw. Strategie auf einen bestimmten Term anwenden.

Mit Kompositions-Kombinatoren kann man Strategien miteinander verknüpfen. *seq* ist dabei die sequentielle Verknüpfung von Strategien. Von zwei Strategien, die an *seq* als Parameter übergeben werden, wird zunächst die erste auf einen Term angewendet und dann die zweite.

Strategie-Anwendung		
$applyTP :: (Monad\ m, Term\ t)$	$\Rightarrow$	$TP\ m \to t \to m\ t$
$applyTU :: (Monad\ m, Term\ t)$	$\Rightarrow$	$TU\ m\ a \to t \to m\ a$
Komposition		
$seq :: Monad\ m$	$\Rightarrow$	$TP\ m \to MG\ \kappa\ m \to MG\ \kappa\ m$
$pass :: Monad\ m$	$\Rightarrow$	$TU\ u\ m \to (u \to MG\ \kappa\ m) \to MG\ \kappa\ m$
$choice :: MonadPlus\ m$	$\Rightarrow$	$MG\ \kappa\ m \to MG\ \kappa\ m \to MG\ \kappa\ m$
Ein-Ebenen-Traversierung		
$all :: Monad\ m$	$\Rightarrow$	$MG\ \kappa\ m \to MG\ \kappa\ m$
$one :: MonadPlus\ m$	$\Rightarrow$	$MG\ \kappa\ m \to MG\ \kappa\ m$

Abbildung 4: Liste der Strategie-Kombinatoren (aus [LV01])

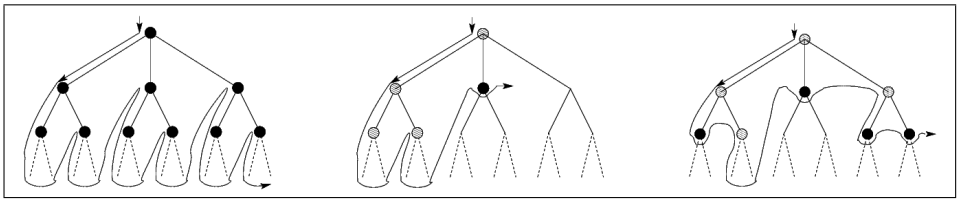

Abbildung 5: Arten der Traversierung (aus [LV03]).

pass verhält sich ähnlich, übergibt jedoch das Ergebnis der Anwendung der ersten Strategie als zusätzliches Argument an die zweite.

Der Kombinator *all* realisiert die Traversierung aller direkten Unter-Terme eines Terms in einem Syntaxbaum während *one* den ersten Unter-Term bearbeitet, der einen Erfolg der Strategie erlaubt.

3.5 Generische Traversierungen

Mit Traversierung wird das Absteigen in einem Syntax-Baum bezeichnet. Das heißt, dass zunächst die gegebene Strategie auf einen gesamten Term angewendet wird und dann, je nach Art der Traversierung, auf alle Sub-Terme. Dabei gibt es die Möglichkeit, sowohl Top-Down als auch Bottom-Up zu traversieren. Abbildung 5 zeigt die Traversierungen *fulltd*, *oncetd* und *stoptd*. Alle drei verarbeiten Knoten in den Syntaxbäumen von oben nach unten und von links nach rechts. *fulltd* bearbeitet alle Knoten im gesamten Baum, während bei *oncetd* und *stoptd* die Bearbeitung der Knoten davon abhängt, ob die Strategie für die darüberliegenden Knoten erfolgreich war oder nicht. Wie in Abbildung 6 zu sehen, bekommen die Traversierungen die zu verwendende Strategie als Parameter übergeben (Parameter *s*). Es wird auch deutlich, dass letztere rekursiv aus den Basis-Kombinierern *all* und *one* erzeugt werden.

Diese Auflistung soll nur einen Überblick geben. Technische Details findet man am besten in [Läm02b].

$allrec\ op\ s$	$=$	$s`op`all(allrec\ op\ s)$	– Rekursive Version von *all*
$onerec\ op\ s$	$=$	$s`op`one(onerec\ op\ s)$	– Rekursive Version von *one*
$oncetd\ s$	$=$	$onerec\ choice\ s$	– Top-down Traversierung mit einem Treffer
$oncebu\ s$	$=$	$oncerec\ (flip\ choice)\ s$	– Bottom-up Traversierung mit einem Treffer
$oncepe\ s\ u\ e$	$=$	$(s\ e)`choice`(u\ e`pass`(\lambda e' \rightarrow one(oncepe\ s\ u\ e')))$	– Traversierung mit Propagierung der Umgebung
$s`above`p$	$=$	$oncebu(oncetd(one\ p)`pass`\lambda() \rightarrow s)$	– Einen Knoten über einem anderen verarbeiten

Abbildung 6: Wiederverwendbare Traversierungen (aus [Läm02b])

$$
\begin{aligned}
&typed\_free\_vars :: (MonadPlus\ m, Eq\ v) \\
&\qquad \Rightarrow [(v,t)] \rightarrow TU\ m\ [v] \rightarrow TU\ m\ [(v,t)] \rightarrow TU\ m\ [(v,t)] \\
&typed\_free\_vars\ env\ getvars\ declvars \\
&= afterTU\ (flip\ appendMap\ env)\ (tryTU\ declvars)`letTU`\ \lambda\ env' \rightarrow \\
&choiceTU\ (afterTU\ (flip\ selectMap\ env')\ getvars) \\
&(comb\ diffMap\ (allTU\ (typed\_free\_vars\ env'\ getvars\ declvars)) \\
&(tryTU\ declvars))
\end{aligned}
$$

Abbildung 7: Extraktion freier Variabler (aus [LV01])

3.6 Transformationen

In Strafunski werden die Transformationen des Prgrammcodes generisch beschrieben. Die Instanz der Klasse *Abstraction*, die den generischen Algorithmen übergeben wird, 'weiß' wie z. B. eine Abstraktion und deren Anwendung in Java, Haskell etc. aussieht und kann diese entsprechend erzeugen bzw. analysieren. Dafür sind die in 8 deklarierten Funktionen zuständig.

Abbildung 7 zeigt eine generische Funktion, die im gegebenen Programmtext absteigt und auf dem Weg alle freien Variablen sammelt. Die Parameter *getVars* und *declVars* sind sprachspezifische Funktionen.

3.7 Abstraktion auf Modell-Ebene

Die Abstraktion erlaubt es uns, Konstrukte verschiedener Sprachen und Paradigem gleich z. B. handeln. Es gibt eine Klasse *Abstraction*, von der für die z. B. handelnde Sprache eine Instanz vorhanden sein muss, die die Übersetzung vornimmt. Im Falle des Refaktorisierung Extraktion sieht diese wie in Abbil-

```
class
        (
         Term abstr,          –Termtyp der Abstraktion
         Eq name,             –Namen von Abstraktionen
         Term [abstr],        –Listen von Abstraktionen
         Term apply           –Term-Typ für Anwendungen
        )
         ⇒ Abstraction abstr name tpe apply
Abhängigkeiten zwischen den syntaktischen Domänen
         — abstr → name,
         abstr → tpe,
         abstr → apply,
         apply → name,
         apply → abstr

where
Observer
 getAbstrName        :: abstr → Maybe name
 getAbstrParas       :: abstr → Maybe [(name,tpe)]
 getAbstrBody        :: abstr → Maybe apply
 getApplyName        :: apply → Maybe name
 getApplyParas       :: apply → Maybe [(name,tpe)]
Konstruktoren
 constrAbstr         :: name → [(name,tpe)] → apply → Maybe abstr
 constrApply         :: name → [(name,tpe)] → Maybe apply
```

Abbildung 8: Abstraktion

dung 8 aus. Möchte man andere Refaktorisierungen implementieren, kann man sich noch weitere Funktionalitäten vorstellen, die Andere Abstraktionen von der Programmiersprache auf die Modell-Ebene vornehmen.

Im Folgenden sieht man, wie die Funktionen des Abstraktions-Interface von den generischen Refaktorisierungen verwendet werden. Im späteren Beispiel wird gezeigt, wie das Abstraktions-Interface instanziiert und diese Instanz an die generischen Refaktorisierungen übergeben wird.

3.8 Beispiele generischer Refaktorisierungen

In diesem Abschnitt werden die beiden generischen Refaktorisierungen Extraktion und Einfügen vorgestellt. Diese werden auch in einem späteren Beispiel eines konkreten Anwendungsfalls gebraucht, wo Sie mit Hilfe einer Instanz des Abstraktionsinterface für die Sprache JOOS implementiert werden.

3.8.1 Generische Extraktion

Die Refaktorisierung Extraktion benutzt als Teil seiner Implementierung die Refaktorisierung Einfügen. Einfügen wurde separat definiert, weil es auch für andere Refaktorisierungen als Teil der Implementierung sinnvoll zu gebrauchen

ist. Als Beispielt nennt [Läm02b] an dieser Stelle Code-Motion.

Die Parameter *declared*,*referenced*, *unwrap*, *wrap*, *unwrap'* und *check* sind Funktionen, die aus einer Instanz der Klasse *Abstraction* stammen. Sie sind also spezifisch für die behandelte Programmiersprache. Wie diese aussehen können, sehen wir später.

Der Algorithmus zur generischen Extraktion führt folgende Schritte aus:

1. Absteigen im gegebenen Programmtext (*program*), um das fokussierte Fragment zu suchen. Auf dem Weg werden gebundene (*bound*) Namen gesammelt.

2. Bestimmen der im fokussierten Fragment freien (*free*) Namen.

3. Aufrufen der *check* Funktion zum Prüfen der Sprachspezifischen Bedingungen.

4. Die Abstraktion des Programmfragmentes wird aus dem benutzerdefinierten Namen mit den freien Variablen als Parameter und dem fokussierten Fragment als Body erzeugt.

5. Die für die Abstraktion relevante Liste von Abstraktionen wir markiert (*markHost*).

6. Aufruf der Refaktorisierung *introduce* zum Einfügen der Abstraktion.

7. Eine Anwendung der neuen Abstraktion (z. B. Methodenaufruf) wird mit den freien Namen als Parameter erzeugt.

8. Das fokussierte Fragment wird durch die erzeugte Anwendung ersetzt (*replaceFocus*).

3.8.2 Generisches Einfügen

Auch hier werden wieder Funktionen des Abstraktions-Interface an die generische Refaktorisierungs-Funktion übergeben. In diesem Fall sind es *declared*, *referenced* und *unwrap*.

Das generische Einfügen (Abbildung 10) besteht aus folgenden Schritten:

1. Den Namen (*name*) der gegebenen Abstraktion bestimmen

2. Freie Namen (*free*) im Fokus bestimmen

3. Definierte Namen (*defined*) im bestimmen

4. Überprüfen, ob die neue Abstraktion mit einer bereits vorhandenen kollidieren könnte

5. Erweiterte Liste der Abstraktionen zurückgeben (*return*)

```
extract:: (Term prog, Abstraction abstr name tpe apply)
      ⇒ TU [(name, tpe)] Identity                    –Deklarationen finden
      → TU [name] Identity                           –Referenzen finden
      → (apply → Maybe apply)                        –Fokus entpacken
      → ([abstr] → [abstr])                          –Host packen
      → ([abstr] → Maybe [abstr])                    –Host entpacken
      → ([(name, tpe)] → apply → Bool)               –Fokus überprüfen
      → name                                         –Name der Abstraktion
      → prog                                         –Ursprünglicher Programmtext
      → Maybe prog                                   –Veränderter Programmtext

extract declared referenced unwrap wrap unwrap' check name prog
 =do
                                                     –Arbeiten auf dem Fokus
      (bound, focus)      ← boundTypedNames declared unwrap prog
      free                ← return (freeTypedNames declared referenced bound focus)
      guard (check bound focus)


                          –Abstraktion erstellen
      abstr               ← constrAbstr name free focus


                          –Abstraktion einfügen
      prog'               ← markHost (maybe False (const True) ∘ unwrap) wrap prog
      prog"               ← introduce declared referenced unwrap' abstr prog'


                          –Anwendung konstruieren
      apply               ← constrApply name free


                          –Fokus durch Anwendung der Abstraktion ersetzen
      replaceFocus (maybe Nothing (const (Just apply)) ∘ unwrap) prog"
```

Abbildung 9: Generische Extraktion

```
introduce:: (Term prog, Abstraction abstr name tpe apply)
        ⇒ TU [(name,tpe)] Identity          –Deklarationen finden
        → TU [name] Identity                –Referenzen finden
        → ([abstr] → Maybe [abstr])         –Scope mit Abstraktionen entpacken
        → abstr                             –Einzufügende Abstraktion
        → prog                              –Ursprüngliches Programm
        → Maybe prog                        –Verändertes Programm

introduce declared referenced unwrap abstr
    = replaceFocus(λ abstrlist → do
        abstrlist' ← unwrap abstrlist
        name       ← getAbstrName abstr
        free       ← return (freeNames declared reference abstrlist')
        def        ← mapM getAbstrName abstrlist'
        guard (and[¬ (elem name free), ¬ (elem name def)])
        return (abst : abstrlist'))
```

Abbildung 10: Generisches Einfügen

4 Beispiel

Das im Folgenden aufgeführte Beispiel soll verdeutlichen, wie nun ein konkreter Anwendungsfall des generischen Refaktorisierens aussieht. Zunächst werden einige Hilfsfunktionen vorgestellt, die verschiedene sprachabhängige Aufgaben erfüllen, z. B. das erkennen von bestimmten Sprach-Konstrukten.

4.1 Namensanalyse

Abbildung 11 zeigt sprachabhängige Funktionen aus einer Implementierung des Framework für die Sprache Joos. Die dargestellten Funktionen sind für die Namensanalyse verantwortlich. Die Namensanalyse stellt fest, wie Bezeichner in einem Kontext an Programmobjekte gebunden werden. Hier sind die guten Pattern-Matching-Eigenschaften der funktionalen Programmierung sehr nützlich. Unwichtige Parameter können einfach ignoriert werden, indem an ihrer Stelle lediglich anonyme Platzhalter gesetzt werden.

4.2 Abstraktions-Interface

Wie zuvor beschrieben definiert eine Instanz der Klasse *Abstraction* Funktionen, die z. B. Abstraktionen und deren Anwendungen erzeugen können. Beispielhaft sei hier die Funktion *constrApply n l* angegeben, die eine Anwendung einer Abstraktion, in JOOS also einen Methodenaufruf, konstruiert.

4.3 Refaktorisierung ausführen

JOOS ist ein Subset von Java und wird wg. einer einfacheren Syntax häufig zu Lehrzwecken eingesetzt.

```
data TypeJoos=ExprType Type
declaredJoos::TU[(Identifier, TypeJoos)] Identity
declaredJoos=adhocTU            (adhocTU(constTU[])
                                (Identity ∘ declaredBlock))
                                (Identity ∘ declaredMeth)
    where
        declaredBlock(Block vds _)
        = map(λ(VarDecl t i)→ (i,ExprType t)) vds
        declaredMeth(MethodDecl _ _  (Formals fps) _ )
        = map(λ(Formal t i)→ (i, ExprType t)) fps

definedJoos::TU[Identifier] Identity
definedJoos=adhocTU             (constTU[])
                                (Identity∘ definedAssignment)
    where
        definedAssignment(Assignment i _ )=[i]

usedJoos::TU[Identifier] Identity
usedJoos=adhocTU                (constTU[])
                                (Identity∘ usedExpression)
    where
        usedExpression(Identifier i)=[i]
        usedExpression _ = []

referencedJoos::TU[Identifier] Identity
referencedJoos=liftop2(++) definedJoos usedJoos
```

Abbildung 11: JOOS Namensanalyse

```
constrApply n l
    =maybe                      Nothing
        (λ aps→ Just            (MethodInvocationStat
                                (ExpressionInvocation This n
                                (Arguments aps))))
        (mapM toActual l)
where
    toActual(i,tpe)
        = case                  tpe of
                                ExprType t → Just(Identifier i)
                                _ → Nothing
```

Abbildung 12: Anwendung einer Abstraktion erzeugen

```
type TrafoJoos = Program → Maybe Program
extractJoos::Identifier → TrafoJoos
extractJoos = extract          declaredJoos
                               referencedJoos
                               unwrapStatement
                               wrapMethods
                               unwrapMethods
                               check
      where
            check_ f = and [noReturns f, noFrees f]
            noReturns = maybe True (const False)∘
                               applyTU (oncetd(adhocTU fail
                               (λ s   → case s of
                               ReturnStat _ → Just()
                               _ → Nothing)))
            noFrees=(≡) [] ∘ freeNames declaredJoos definedJoos

introduceJoos::MethodDecl → TrafoJoos
introduceJoos= introduce       declaredJoos
                               referencedJoos
                               unwrapMethods
```

Abbildung 13: Spezialisierte Extraktion für JOOS

In diesem Abschnitt soll gezeigt werden, wie das zuvor erwähnte Abstraktions-Interface, die Klasse *Abstraction* instanziiert und mit Sprachabhängiger Funktionalität versehen wird. Es werden die konkreten Ausprägungen von *declared*, *referenced* ... als *declaredJOOS*,*referencedJOOS*... definiert. Diese haben die für die Übergabe an die generischen Funktionen passende Signatur, z. B. *declaredJOOS::TU [(Identifier, TypeJoos)] Identity* (vgl. Abbildung 9). In Abbildung 13 sieht man, wie die einzelnen Elemente zusammen arbeiten. Die zuvor speziell für JOOS definierten Funktionen werden an den generischen Algorithmus *extract*übergeben. Damit wird die eigentliche Refaktorisierung ausgeführt.

In Abbildung 14 ist ein stark vereinfachter Aufrufgraph dargestellt, der skizzieren soll, wie die einzelnen Komponenten des Framework ineinander greifen. Der Pfeil $extractJoos \rightarrow extract$ bedeutet hier, dass die sprachabhängige Funktion *extractJoos* die generische Funktion *extract* aufruft.

Die sprachspezifische Funktion *extractJoos* ruft also die generische Funktion *extract* auf und übergibt ihr die für diese Refaktorisierung relevanten Funktionen als Parameter. *extract* wiederum bedient sich sowohl sprachspezifischer (*constrApply* und *constrAbstr*) z. B. zum Erzeugen der Abstraktion und ihrer Anwendung. *extract* benutzt aber auch generische Funktionen zur Namensanalyse, welche dann ihrerseits auf sprachabhängige Funktionen zugreifen, um die erforderlichen Teile aus dem Quelltext zu lesen.

Geht man eine Ebene weiter und betrachtet die Funktion *check*, so ruft diese die generische Funktione *freeNames* auf (Siehe Abbildung 13).

Dieser Graph ist nicht vollständig und soll nur einen Eindruck davon vermitteln, wie die sprachabhängigen und generischen Funktionen zusammen arbeiten. Ein

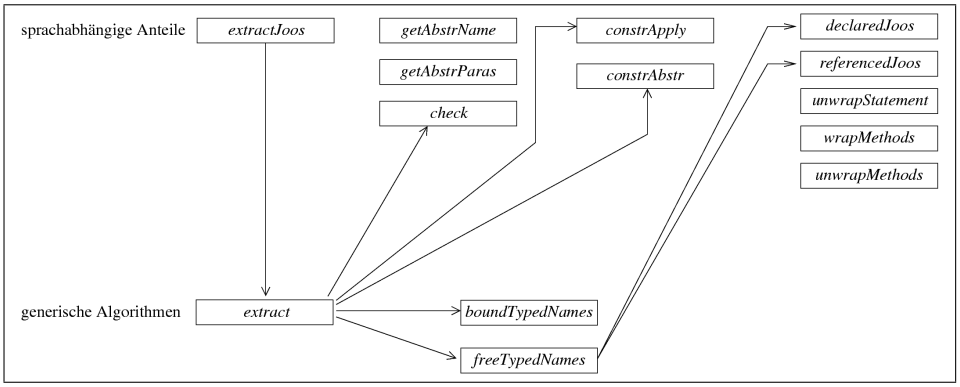

Abbildung 14: Funktionsaufrufe im Framework

vollständiger Graph würde schnell sehr unübersichtlich.

5 Zusammenfassung

In den vorangehenden Abschnitten wurde eine Motivation gegeben, um dem Leser zu verdeutlichen, dass ein generischer Ansatz des Refaktorisierens überhaupt Sinn macht. Gerade weil es wie später gezeigt beim generischen Refaktorisieren um verschiedene Programmierparadigmen geht, spielt nicht nur die Syntax der behandelten Sprachen eine Rolle. Anhand des Beispiels *Extraktion* wird deutlich gemacht, dass der Ansatz funktionieren kann.

Das vorgestellte Framework ist so flexibel implementiert, dass der Code, der für eine konkrete Implementierung benötigt wird, sich ausschließlich darauf beschränken kann, die Sprachkonstrukte auf einer höheren Ebene darzustellen. Die eigentlichen Refaktorisierungs-Algorithmen arbeiten auf dieser höheren Abstraktionsebene und müssen somit nur einmal definiert werden.

Sowohl die Traversierungen über den Syntaxbäumen, als auch die Transformationen werden generisch implementiert. Die eingesetzten Algorithmen bekommen als Parameter sprachabhängige Funktionen mit definierten Schnittstellen übergeben, die Teilaufgaben wie z. B. das Erstellen der Anwendung einer Abstraktion oder das Erkennen von Bezeichnern implementieren.

In [Läm02b] wird ausschließlich die Refaktorisierung *Extraktion* und als Teil davon das *Einfügen* behandelt. Lämmel stellt die These auf, dass generisches Refaktorisieren für fast alle Programmierparadigmen und -Sprachen, sowie für unterschiedlichste Refaktorisierungen möglich und sinnvoll ist. Diese Behauptung bleibt jedoch unbewiesen.

Es wird auch gesagt, dass es bisher nur erste Ansätze zum generischen Refaktorisieren gibt, weitere Untersuchungen, z. B. in Bezug auf andere Refaktorisierungen als die behandelte *Extraktion*, stehen noch aus. Lämmel schlägt vor, die Refaktorisierungen der Kataloge in [Fow00] und [Opd92] nach generischen Algorithmen zu untersuchen, die auch in anderen Programmierparadigmen Aussagekraft haben. Die den Refaktorisierungen zu Grunde liegenden generischen Algorithmen müssten extrahiert und in einem Framework zur Verfügung gestellt werden.

Literatur

[Fow00] Martin Fowler. *Refactoring: Wie Sie das Design vorhandener Software verbessern.* Addison-Wesley Verlag, 2000.

[Läm02a] Ralf Lämmel. The Sketch of a Polymorphic Symphony. In Bernhard Gramlich and Salvador Lucas, editors, *Proc. of International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2002)*, volume 70 of *ENTCS*. Elsevier Science, 2002. 21 pages.

[Läm02b] Ralf Lämmel. Towards Generic Refactoring. In *Proc. of Third ACM SIGPLAN Workshop on Rule-Based Programming RULE'02*, Pittsburgh, USA, October5 2002. ACM Press. 14 pages.

[LV01] R. Lämmel and J. Visser. Typed Combinators for Generic Traversal. Technical Report SEN-R0124, Centrum voor Wiskunde en Informatica, August 2001. 34 pages.

[LV03] R. Lämmel and J. Visser. A Strafunski Application Letter. In V. Dahl and P. Wadler, editors, *Proc. of Practical Aspects of Declarative Programming 2003 (PADL'03)*, LNCS. Springer-Verlag, January 2003. To appear; 18 pages.

[Opd92] William F. Opdyke. *Refactoring Object-Oriented Frameworks.* PhD thesis, Urbana-Champaign, IL, USA, 1992.

[Vis00] E. Visser. Language independent traversals for program transformation, May 2000.