Universität Paderborn

Fakultät für Elektrotechnik, Informatik und Mathematik

Refactoring in eXtreme Programming, Komposition von Transformationen

Hermann Wessels

03.03.2003

Betreuer:
Prof. Dr. Uwe Kastens
Dipl. Inform. Jochen Kreimer

Alle zur Verbesserung der Struktur eines Softwareprojektes potentiell möglichen Transformationen, können durch Kombinationen einer begrenzten Anzahl von Basistransformationen ausgedrückt werden. Im Rahmen dieser Ausarbeitung wird eine Methodik vorgestellt, die auf Basis von Vor- und Nachbedingungen, sowie mit Hilfe statischer Analysen berechnet, ob eine Menge von Transformationen in einer bestimmten Reihenfolge durchgeführt werden kann.

Inhaltsverzeichnis

1. Einleitung

Software wird in einem evolutionären Prozess entwickelt. Dieser Prozess fängt immer seltener bei Null an. Viel mehr wird Software aufgrund von neu bekannt werdenden, sich verändernden oder sich als falsch bzw. unvollständig erweisenden Anforderungen weiterentwickelt.
Durch den gesamten Lebenszyklus einer Software zieht sich somit eine Kette von mehr oder weniger umfangreichen Anpassungen, aufgrund sich verändernder Anforderungen. Diese Anpassungen können dabei sowohl funktionaler, wie auch struktureller Natur sein. Häufig sind strukturelle Anpassungen Vorraussetzung um funktionale Anpassungen vornehmen zu können. Im Rahmen dieser Ausarbeitung beschränken wir uns auf die Betrachtung struktureller Anpassungen.
Strukturelle Anpassungen, wie beispielsweise das Umbenennen einer Methode oder das Ersetzen redundanter Codesegmente durch Methodenaufrufe, sind auch als *Refactorings* oder Restrukturierungen bekannt. Unter diesem Oberbegriff sind Verbesserungen der Programmstruktur zusammengefasst. Mithilfe von *Refactoring* kann beispielsweise eine bessere Les- und Wartbarkeit, aber auch eine einfachere Erweiterbarkeit des Softwareprojektes erreicht werden. Der Einsatz von *Refactorings* wird insbesondere innerhalb der leichtgewichtigen Softwareentwicklungsmethodik *eXtreme Programming* von Kent Beck [Beck99] zum Grundsatz erhoben.
Einen ganzen Katalog solcher *Refactorings* beschreibt Fowler in seinem Buch *Refactoring: Improving the Design of Existing Code* [Fowler99]. *Refactorings* sind bei ihm grundsätzlich in drei Teilschritte zerlegt. An die systematische Änderung des Quellcodes, schliessen sich bei Fowler Phasen ausführlicher Regressionstests und deren Auswertung an.
Fowler verzichtet darauf nachzuweisen, dass die von ihm vorgestellten *Refactorings* Semantik erhalten sind bzw. überlässt den Nachweis den Regressionstests. Da Regressionstests aber nur selten vollständig sind bzw. alle Aspekte der Semantik abdecken, muss davon ausgegangen werden das sich doch Änderungen einschleichen können.
William Obdyke hat in [Obdyke92] eine Reihe von *Refactorings* identifiziert, für die er die Eigenschaft der Semantikerhaltung formal nachweisen konnte. Allerdings werden die von Obdyke identifizierten Refactorings von ihm auch als Basistransformationen bezeichnet, da sie so klein sind, dass sie nicht weiter zerlegt werden können. Die von Fowler vorgestellten *Refactorings* werden von Obdyke entsprechend als eine Kompositionen von Basistransformationen aufgefasst.
Für die projektweiten Verbesserung der Struktur sind i. d. R. wiederum Anpassungen notwendig, die aus mehreren *Refactorings* bestehen. Es kann daher auf allen Detailebenen von Mengen von Transformationen gesprochen werden.
Da eine Menge von Transformationen unter Umständen untereinander Abhängigkeiten aufweisen, können die Transformationen aber nicht in jeder beliebigen Reihenfolge durchgeführt werden. Wie eine mögliche durchführbare Reihenfolge für eine Menge von Transformationen bestimmt werden kann, wird in dieser Ausarbeitung dargelegt.

2. Konzepte zur Beschreibung von Transformationen

In diesem Kapitel werden die grundlegenden Konzepte vorgestellt, auf denen die in Kapitel 3 vorgestellte Methodik beruht. Dazu bezieht sich diese Ausarbeitung im folgendem hauptsächlich auf die Doktorarbeit von William Obdyke [Obdyke92] und die darauf aufbauende Doktorarbeit von Donald Bradley Roberts [Roberts99].

2.1. Überblick

William Obdyke hat 23 Basistransformationen und drei einfache Kompositionen auf ihrer Grundlage identifiziert. Für die einzelnen Basistransformationen konnte Obdyke formal nachgeweisen, dass sie semantikerhaltend sind.
Da die Basistransformationen korrekt sind, sind auch Kombinationen dieser Basistransformationen korrekt. Somit können alle potentiell möglichen Transformationen als Kompositionen von Basistransformationen aufgefasst werden.
Die 23 Basistransformationen sind Verfeinerungen von acht *Basisrefactorings*. Diese acht *Basisrefactorings* hat Obdyke wiederum aufgrund einer Bestandsaufnahme verwandter Untersuchungen und der Langzeituntersuchung eines speziellen Projektes bezüglich struktureller Änderungen bestimmen können. Die acht *Basisrefactorings* sind in Anhang A.1 angeführt.
Die 23 bekannten Basistransformationen können in fünf Kategorien eingeteilt werden:

1. Die erste Kategorie fasst die Basistransformationen zum Erzeugen von Programmeigenschaften zusammen. Enthalten ist beispielsweise die Basistransformation zum Erzeugen einer leeren Klasse.

2. In der zweite Kategorie sind die Basistransformationen zum Löschen von Programmeigenschaften zusammengefasst. Hier sind die Basistransformationen zum Löschen unreferenzierter Klassen, Methoden und Variablen zu finden.

3. Die dritte Kategorie umfasst alle Basistransformationen zum Verändern von Programmeigenschaften, mit Ausnahme der Basistransformationen zum Verschieben von Variablen in der Klassenhierarchie. Basistransformationen zum Umbenennen von Klassen, Methoden und Variablen oder die Basistransformation zum erweitern einer Methodensignatur um einen neuen Parameter sind in dieser Kategorie enthalten.

4. Die Basistransformationen zum Verschieben von Variablen in der Klassenhierarchie ergeben die vierten Kategorie von Basistransformationen.

5. Die drei einfachen Kompositionen sind in der fünften Kategorie enthalten. Diese Kategorie ist so gesehen ein Sonderfall und enthält beispielsweise die Transformation zum Kapseln einer Variablen als Attribut mit den zugehörigen Zugriffsmethoden.

Die fünf Kategorien und alle in Ihnen enthaltenen Transformationen sind im Anhang A.2 noch einmal vollständig aufgelistet.

2.2. Vorbedingungen

Um die Korrektheit der von ihm identifizierten 23 Basistransformationen zu zeigen, hat William Obdyke Vorbedingungen für die Basistransformationen bestimmt. Sind die Vorbedingungen einer Transformation erfüllt, so kann diese durchgeführt werden ohne dass das Programmverhalten verändert wird.
Um die Vorbedingungen möglichst allgemein und sprachunabhängig halten zu können, hat Obdyke einige Einschränkungen bzgl. zulässiger Programmeigenschaften vorgenommen. So hat er z. B. Mehrfachvererbung oder das Überladen von Methodennamen von vorneherein ausgeschlossen, obwohl er mit C++ arbeitete. Die Einschränkungen sind in Anhang A.3 detailliert aufgelistet.
Die Gültigkeit von Vorbedingungen wird von Obdyke mit Hilfe statischer Analysen überprüft. Beispielweise wird durch die Analyse des Quelltextes die Menge der Klassen innerhalb eines Projektes erfasst. Im Rahmen dieser Ausarbeitung wird die Existenz entsprechender automatisierter Analyseverfahren unterstellt und nicht weiter betrachtet.
Anhand der Basistransformation *Erzeugen einer leeren Klasse* wird im folgendem exemplarisch der Sinn der einzelnen Vorbedingungen erläutert. Mit den Vorbedingungen der Basistransformation *Erzeugen einer leeren Klasse* wird sichergestellt, dass die Klasse syntaktisch korrekt erzeugt wird und der Quellcode übersetzbar bleibt.
Die Vorbedingung *istGültigerKlassenname(name)* stellt sicher, dass der für die neue Klasse gewählte Name syntaktisch korrekt ist. In der Programmiersprache Java würde mit dieser Vorbedingung z. B. sichergestellt, dass keine ungültigen Sonderzeichen wie das Euro Symbol im Namen enthalten sind.
Mit der Vorbedingungen $\neg$ *erzeugtNamenskonflikt(MOD,(kontext,name))* und $kontext \in Pakete \vee kontext \in \{KLASSEN\}$ wird sichergestellt, dass der angegebene Kontext, aber keine Klasse mit dem selben Namen im gewünschtem Kontext existiert.
Die letzte Vorbedingung ist eigentlich eine Fallunterscheidung, mit der geprüft wird, ob die gewünschten Zugriffsmodifizierer eine korrekte Kombination sind. Beispielsweise dürfen die Zugriffsmodifizierer *abstract* und *final* nicht in Kombination auf-

treten.
Die Vorbedingungen der Basistransformation *Erzeugen einer leeren Klasse* sind in Abbildung 2.4 Abschnitt 2.5 noch einmal vollständig abgebildet.
Das Programmverhalten und damit die Semantik bleibt bei Durchführung der Basistransformation *Erzeugen einer leeren Klasse* erhalten, da keinerlei semantische Beziehungen zwischen der Klasse und den Programmelementen des Projekt, mit Ausnahme einer evtl. Einbindung der Klasse in die Vererbungshierarchie, bestehen und der Quellcode weiterhin übersetzt werden kann.

2.3. Nachbedingungen

Die von Obdyke identifizierten Basistransformationen sind so elementar, dass sie isoliert nicht sinnvoll angewandt werden können. Allerdings können Basistransformationen zu grösseren Transformationen zusammengefasst werden.
In einer solchen Kombination von Basistransformationen bestehen zwischen einzelnen Basistransformationen i. d. R. Abhängigkeiten. So werden die Vorbedingungen bestimmter Basistransformationen erst mit der Durchführung anderer Basistransformationen erfüllt. In Abbildung 2.1 ist dies anschaulich durch die Pfeile dargestellt. Aufgrund solcher Abhängigkeiten, können Basistransformationen i. d. R. nicht in jeder beliebigen Reihenfolge ausgeführt werden.
Ein Beispiel für eine solche Abhängigkeit findet sich bei den Basistransformationen

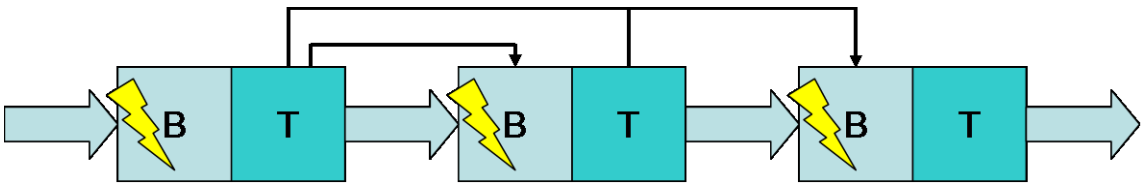

Abbildung 2.1: Abhängigkeiten in einer Sequenz

Erzeugen einer leeren Klasse und *Erzeugen einer Variablen* oder *Erzeugen einer Methode*, wenn die Variable oder Methode in der zu erzeugenden Klasse erzeugt werden soll. Ein weiteres Beispiel sind die Basistransformationen *Erzeugen einer Variablen* oder *Erzeugen einer Methode*, für den Fall, dass die zu erzeugende Methode als Zugriffsmethode für die Variable genutzt werden soll.
Vorbedingungen können, aufgrund von potentiellen Abhängigkeiten, erst direkt vor der Durchführung einer Basistransformation überprüft werden, vergleiche Abbildung 2.1. Auf diesem Wege kann die Durchführbarkeit daher selbst für kleinste Kombinationen nicht im Vorfeld überprüft werden.
Donald Bradley Roberts hat im Rahmen seiner Doktorarbeit [Roberts99] die Basistransformationen von Obdyke um Nachbedingungen erweitert. Roberts Ziel war es mit Hilfe von Nachbedingungen den Analyseaufwand für Kombinationen von Basistransformationen zu verringern. Roberts zeigt ebenfalls, dass es Dank dieser

Erweiterung möglich ist Kombinationen bereits im Vorfeld auf Durchführbarkeit zu überprüfen.
Analysen können durch die Einführung von Nachbedingungen zumindest teilweise eingespart werden. Durch die sequentielle Abarbeitung der Transformationen sind die Nachbedingungen einer Transformation zu Beginn der nachfolgenden Transformation immer noch gültig. Wie in Abbildung 2.2 veranschaulicht können einzelne Vorbedingungen anhand der Nachbedingungen der vorhergehenden Transformation anstatt durch Analysen überprüft werden, vorrausgesetzt es existieren Abhängigkeiten zwischen den beiden Transformationen.
Greifen wir wieder auf das Beispiel der Basistransformationen *Erzeugen einer leeren Klasse* und *Erzeugen einer Variablen* oder *Erzeugen einer Methode* zurück. Die Nachbedingungen der Basistransformation *Erzeugen einer leeren Klasse* sind in Abbildung 2.6 Abschnitt 2.5 vollständig abgebildet. Für das Beispiel sind nur die Nachbedingungen $\{Klassen\} = \{Klassen\} \cap neueKlasse$ und $neueKlasse.name = name$ wichtig. Vorbedingung der beiden Basistransformationen *Erzeugen einer Variablen* oder *Erzeugen einer Methode* ist unter anderem die Existenz der Zielklasse. Im Anschluss an jede Basistransformation, deren Nachbedingungen die Menge der im Projekt existierenden Klassen umfasst, kann nun mit Hilfe des Nachbedingungen anstelle von Analysen überprüft werden ob eine bestimmte Klasse innerhalb des Projektes existiert.

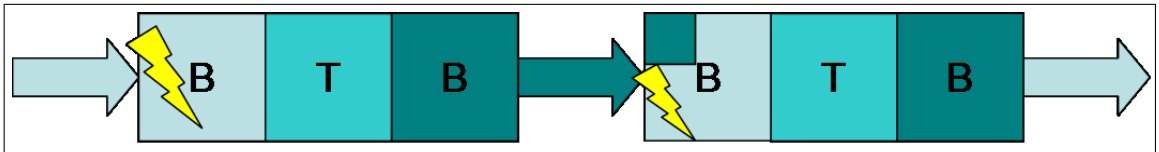

Abbildung 2.2: Analysen durch Nachbedingungen eingesparen

2.4. Abhängigkeiten

Wenn große Änderungen in sequentielle Teilschritte zerlegt werden, kann davon ausgegangen werden, dass in den Sequenzen Transformationen enthalten sind, die vollständig oder teilweise von einander unabhängig sind. Sollen beispielsweise mehrere Variablen einer Klasse verschoben werden, so ist die Reihenfolge in der dies geschieht irrelevant.
Die bisher immer als Sequenzen aufgefassten Transformationen können auch als Abhängigkeitsgraph dargestellt werden, siehe Abbildung 2.3. Durch die Überführung der Sequenzen in Abhängigkeitsgraphen werden neue Nutzungsmöglichkeiten der Abhängigkeiten erschlossen bzw. bestehende vereinfacht.
Im Folgenden werden einige potentielle Nutzungsmöglichkeiten aufgezeigt:

1. *Refactorings* sollten mit Hilfe einer *Undo*-Funktion problemlos wieder rückgängig gemacht werden können. Soweit eine *Undo*-Funktion überhaupt zur

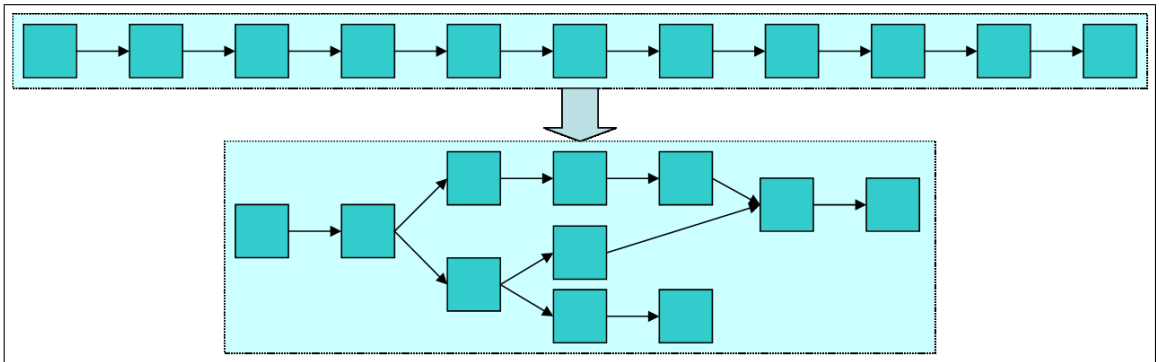

Abbildung 2.3: Sequenz als Abhängigkeitsgraph

Verfügung steht, werden dazu alle Änderungen protokolliert. Bisher werden im Fall eines *Undos* alle Änderungen bis zu einem gewünschtem Punkt zurückgesetzt. Dabei werden unter Umständen auch Änderungen zurückgesetzt die erhalten bleiben sollen. Nach dem *Undo* müssen diese Änderungen dann noch einmal durchgeführt werden. Mit Hilfe von Abhängigkeitsgraphen wäre es nun möglich nicht mehr alle, sondern nur noch die abhängigen Teile der Änderungen rückgängig zu machen.

2. *Refactorings* sollen automatisiert werden, da manuelle Anpassungen fehleranfällig und bei großen Änderungen nicht besonders effektiv sind. Mit Hilfe von Abhängigkeitsgraphen wäre es möglich große bzw. zahlenmäßig umfangreiche Änderungen durch eine parallele Verarbeitung noch effektiver durchzuführen.

3. Abhängigkeitsgraphen vereinfachen das Zusammenführen von Änderungen in Mehrbenutzerumgebungen. Durch die Automatisierung des *Refactorings* kommt es potentiell viel schneller zu Konflikten. Ohne Abhängigkeitsgraphen müssen alle Änderungen auf Konflikte überprüft und eingepflegt werden. Mit Hilfe von Abhängigkeitsgraphen wäre es möglich alle Änderungen die unabhängig von einander sind automatisch einzupflegen und nur noch die abhängigen Änderungen überprüfen zu müssen.

4. Durch Analysen von Abhängigkeitsgraphen ist es möglich besonders häufig wieder kehrende Sequenzen von Transformationen zu bestimmen. Wenn diese Sequenzen häufig genug auftauchen, kann es sinnvoll sein diese zu einer Komposition zusammenfassen und ihre Automatisierung zu optimieren.

5. In Sequenzen sind “überflüssige“ Transformationen kaum zu identifizieren. Die gewünschten Transformationen können von den “überflüssigen“ Transformationen nicht abhängig sein. Daher können “überflüssige“ Transformationen mit Hilfe von Analysen, ausgehend von den Endknoten der Abhängigkeitsgraphen, leicht identifiziert werden.

Donald Roberts geht im Rahmen seiner Doktorarbeit [Roberts99] auf die Ermittlung von Anhängigkeiten ein. Allerdings führt er zur Ermittlung von Abhängigkeiten nur das paarweise Vergleichen der Bedingungen der Transformationen als Methodik an. Zur Überprüfung überschaubarer Kombinationen reicht diese Methodik noch aus. Abhängigkeiten innerhalb umfangreicher Änderungen können auf diesem Wege aber nicht effizient ermittelt werden.
Erkenntnisse in diesem Bereich beschränken sich bisher auf Betrachtungen am Rande bzw. Ausblicke.

2.5. Basistransformation *Erzeugen einer leeren Klasse*

Wie in den Beispielen bereits angedeutet, sind die Vor- und Nachbedingungen in ihrem Umfang nicht auf einzelne triviale Bedingungen beschränkt. Am Beispiel der Basistransformation *Erzeugen einer leeren Klasse* soll im folgendem eine Basistransformation im vollem Umfang präsentiert werden.

Die Basistransformation *Erzeugen einer leeren Klasse* hat folgende Signatur:

erzeugeKlasse(MOD,(kontext,name)):Klasse

Die Vorbedingungen, in Abbildung 2.4 vollständig aufgeführt, stellen sicher, dass ein syntaktisch korrekter Namen gewählt wurde, kein Namenskonflikt besteht, der Kontext existiert und die Klasse syntaktisch korrekte Zugriffsmodifikatoren erhält.

$$
\begin{array}{l}
\text{istGültigerKlassenname}(name) \\
\qquad \wedge \\
\neg\ \text{erzeugtNamenskonflikt}(MOD,(kontext,name)) \\
\qquad \wedge \\
kontext \in \{\text{PAKETE}\} \vee kontext \in \{\text{KLASSEN}\} \\
\qquad \wedge \\
\begin{array}{lll}
(\ \text{„}public\text{“} \in MOD & \rightarrow & (kontext \in \{\text{PAKETE}\} \vee kontext \in \{\text{KLASSEN}\}) \\
\wedge\ \text{„}protected\text{“} \in MOD & \rightarrow & (kontext \in \{\text{KLASSEN}\} \wedge \text{„}public\text{“} \in MOD \\
 & & \wedge\ \text{„}private\text{“} \in MOD) \\
\wedge\ \text{„}private\text{“} \in MOD & \rightarrow & (kontext \in \{\text{KLASSEN}\} \wedge \text{„}public\text{“} \notin MOD \\
 & & \wedge\ \text{„}protected\text{“} \notin MOD) \\
\wedge\ \text{„}static\text{“} \in MOD & \rightarrow & (kontext \in \{\text{KLASSEN}\}) \\
\wedge\ \text{„}abstract\text{“} \in MOD & \rightarrow & ((kontext \in \{\text{PAKETE}\} \vee kontext \in \{\text{KLASSEN}\}) \\
 & & \wedge\ (\text{„}private\text{“} \notin MOD \wedge \text{„}final\text{“} \notin MOD)))
\end{array}
\end{array}
$$

Abbildung 2.4: Vorbedingungen der Basistransformation Erzeuge leere Klasse

Sind die Vorbedingungen erfüllt, wird mit der eigentlichen Transformation, die neue Klasse kontextabhängig erzeugt. Für den Fall das der Parameter *kontext* ein Paket referenziert, wird eine Klasse mit dem Parameter *name* als Namen und den Zugriffsmodifikatoren *MOD* erzeugt. Anderenfalls wird eine innere Klasse in der durch *kontext* referenzierten Klasse erzeugt, siehe dazu auch Abbildung 2.5.

kontext $\in$ {PAKETE}	$\rightarrow$	erzeuge neue Klasse *name* im Paket *kontext* mit den Zugriffsrechten *MOD*
$\vee$		
kontext $\in$ {KLASSEN}	$\rightarrow$	erzeuge innere Klasse *name* in der Klasse *kontext* mit den Zugriffsrechten *MOD*

Abbildung 2.5: Transformationen der Basistransformation Erzeuge leere Klasse

Nach dem die Transformation durchgeführt wurde, gelten die in Abbildung 2.6 aufgeführten Bedingungen. Die neu erzeugte Klasse ist Element der Menge der Klassen des Projektes, trägt den gewählten Namen *name* und ist mit den übergebenen Zugriffsmodifikatoren im gewünschten Kontext erzeugt worden. Weitere gültige Informationen sind, dass die Klasse leer und unreferenziert ist, d. h. es gibt keine Zugriffe auf die Klasse, die Klasse ist nicht in die Vererbungshierarchie eingebunden und enthält keine inneren Klassen. Die Klasse enthält keine Variablen oder Methoden und insbesondere auch keinen Konstruktor. Des Weiteren handelt es sich um eine Klasse und keine Schnittstelle.

$$
\begin{aligned}
&\{\text{Klassen}\} = \{\text{Klassen}\} \cup \text{neueKlasse} \\
&\qquad \wedge \\
&\text{neueKlasse.name} = \mathit{name} \\
&\qquad \wedge \\
&\text{neueKlasse.qualifizierterName} = (\mathit{kontext}, \mathit{name}) \\
&\qquad \wedge \\
&\text{MOD.neueKlasse} = \mathit{MOD} \\
&\qquad \wedge \\
&(\ \text{Klassenzugriffe.neueKlasse} = \emptyset \\
&\ \wedge\ \text{Superklassen.neueKlasse} = \emptyset \\
&\ \wedge\ \text{innereKlassen.neueKlasse} = \emptyset \\
&\ \wedge\ \text{Konstruktoren.neueKlasse} = \emptyset \\
&\ \wedge\ \text{Attribute.neueKlasse} = \emptyset \\
&\ \wedge\ \text{Methoden.neueKlasse} = \emptyset\ ) \\
&\qquad \wedge \\
&\text{istSchnittstelle(neueKlasse)} = \text{false}
\end{aligned}
$$

Abbildung 2.6: Nachbedingungen der Basistransformation Erzeuge leere Klasse

3. Komposition von Transformationen

Nachdem im vorherigen Kapitel die grundlegenden Konzepte vorgestellt wurden, wird in diesem Kapitel eine Methodik zur Überprüfung einer Sequenz von Transformationen auf ihre Durchführbarkeit vorgestellt.

3.1. Problematik

Das Bestimmen von Vor- und Nachbedingungen für beliebige Kompositionen von Basistransformationen ist nicht trivial. Simples Verknüpfen der Vorbedingungen der beteiligten Basistransformationen funktioniert i. d. R. schon bei einfachsten Beispielen nicht. Dies liegt an den potentiellen Abhängigkeiten zwischen den Basistransformationen.
Bemühen wir wieder das Beispiel vom Erzeugen einer Variablen in einer gerade erzeugten Klasse. Eine der Vorbedingungen für das Erzeugen der Variablen ist die Existenz der Zielklasse. Die Zielklasse existiert im Beispiel, aber erst nach der Durchführung der entsprechenden Basistransformation.
Innerhalb des Kontextes darf aber keine Klasse mit dem gewünschtem Namen existiert, sonst sind nicht alle Vorbedingungen der Basistransformation *Erzeugen einer leeren Klasse* erfüllt und diese darf nicht durchgeführt werden, siehe dazu auch Ab-

bildung 2.6. Werden die Vorbedingungen der beiden Basistransformationen mittels der UND-Verknüpfung verbunden, erzeugt dies einen Widerspruch.

3.2. Algorithmus zur Überprüfung von Sequenzen

Donald Roberts hat im Rahmen seiner Doktorarbeit [Roberts99] informal eine Methodik angeben, mit der überprüft werden kann, ob eine Sequenz von Transformationen durchgeführt werden kann. Alternativ kann die Methode auch dazu genutzt werden um Vor- und Nachbedingungen zusammengesetzter Transformationen zu berechnen.
Prinzipiell können beliebige Sequenzen von Transformationen auf ihre Durchführbarkeit überprüft werden, solange die Vor- und Nachbedingungen der einzelnen Transformationen bekannt sind.
Die Methodik startet mit der analytischen Überprüfung der Vorbedingungen der ersten Transformation. Ist nur eine nicht erfüllt, kann die Sequenz in keinem Fall durchgeführt werden. Sind alle Vorbedingungen erfüllt, werden die Analyseergebnisse gespeichert. Die Analyseergebnisse werden als Initialisierung einer Formel gebraucht, in der zu jedem Zeitpunkt alle bekannten gültigen Bedingungen gespeichert sind.
Seien die Vorbedingungen erfüllt, und die erste Transformation durchgeführt, dann werden anschließend die Nachbedingungen der ersten Transformation mit der Formel verknüpft.
Für die nachfolgende Transformation werden nun die Vorbedingungen der folgenden Transformation mit der Formel verglichen. Ist eine Vorbedingung oder eine Menge von Vorbedingungen nicht erfüllt so kann die Sequenz von Transformationen nicht durchgeführt werden. Sind Vorbedingungen anhand der Formel nicht überprüfbar, müssen sie auf analytischem Wege geprüft werden. Führen diese Analysen dazu, dass Vorbedingungen als nicht erfüllt bewertet werden müssen, haben wir wieder eine Sequenz die nicht durchgeführt werden kann. Andernfalls werden die Analyseergebnisse mit der Formel verknüpft und die Transformation durchgeführt. Anschließend an die Durchführung der Transformation werden die Nachbedingungen mit der Formel verknüpft.
Dies wiederholt man mit allen Transformationen der Sequenz bis eine Vorbedingung nicht erfüllt ist oder man die letzte Transformation durchgeführt hat.
In Abbildung 3.1 ist das beschriebene Verfahren zur Überprüfung einer Sequenz auf ihre Durchführbarkeit in formalisierter Form dargestellt.

1. Analysen der Vorbedingung(en) der ersten Basistransformation
   a) Ja: Initiale Formel = Analyseergebnisse
   b) Nein: Sequenz ist nicht durchführbar
2. Verknüpfung der Nachbedingung(en) mit der Formel
3. Überprüfung ob Vorbedingung(en) der folgenden Basistransformation wahr sind
   a) Ja: Wiederholen der Schritte zwei und drei
   b) Nein:
      i. Vorbedingung(en) falsch ⇒ Sequenz ist nicht durchführbar
      ii. Vorbedingung(en) „unbekannt“ ⇒ Analysen
         A. Vorbedingung wahr ⇒ Formel = Formel ∩ Analyseergebnisse, wiederholen der Schritte zwei und drei
         B. Vorbedingung falsch ⇒ Sequenz ist nicht durchführbar

Algorithmus 3.1: Methodik zur Überprüfung einer Sequenz auf ihre Durchführbarkeit

3.3. Überprüfung einer Beispielsequenz

Am Beispiel des Kapselns einer bestehenden Variablen als Attribut in einer neuen Klasse, soll im folgendem der Algorithmus einmal durchlaufen werden.
Dazu sind folgende Basistransformationen notwendig:

1. Erzeugen der (leeren) Klasse
2. Erzeugen der Variablen
3. Erzeugen der get-Methode
4. Hinzufügen des Methodenrumpfes zur get-Methode
5. Erzeugen der set-Methode
6. Hinzufügen des Methodenrumpfes zur set-Methode
7. Transformation aller Variablenreferenzen auf die Originalvariable in die neu erstellten Methodenaufrufe
8. Löschen der nun unreferenzierten Originalvariablen

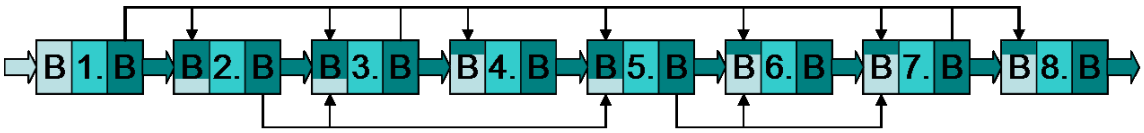

Abbildung 3.1: Die Beispielsequenz aus Abschnitt 3.3

In Abbildung 3.1 ist die Sequenz noch einmal graphisch dargestellt.
Die Schritte zwei bis sieben hat Obdyke als einfache Transformation zusammengefasst und könnten alternativ durch diese ersetzt werden, siehe dazu auch Anhang A.2 Unterabschnitt *Einfache Transformationen* Nummer 1.
In Abschnitt 2.5 ist die Basistransformation *Erzeugen einer leeren Klasse* erläutert worden. Da diese Basistransformation die erste in der zu überprüfenden Sequenz ist, werden ihre Vorbedingungen auf analytischem Wege geprüft. Seien die Vorbedingungen alle erfüllt, dann wird die Formel der gültigen Bedingungen mit den Analyseergebnissen initialisiert.
Im nächsten Schritt werden die Nachbedingungen der Basistransformation *Erzeugen einer leeren Klasse* mit der Formel der gültigen Bedingungen verknüpft. Aufgrund der Analysenergebnisse des ersten Schrittes enthält die Formel unter anderem die Menge der Klassen. Mit der Verknüpfung der Nachbedingungen und der Formel wird diese Menge nun um die erzeugte Klasse erweitert. Des Weiteren werden der Formel die Informationen hinzugefügt die ausdrücken das die Klasse leer ist.
Die Vorbedingungen der zweiten Basistransformation *Erzeugen einer Variablen* können nun als erstes gegen die Formel der gültigen Bedingungen geprüft werden. In Abbildung 3.1 ist dies farblich kodiert. Der dunkle Teil der Vorbedingungen kann anhand der Formel und der helle Teil muss auf analytischem Wege überprüft werden.
Die Existenz der Zielklasse kann anhand der Formel überprüft und bestätigt werden, da wir die Zielklasse im letzten Schritt erstellt haben. Dass die Erzeugung der Variablen keinen Namenskonflikt auslöst, können wir ebenfalls anhand der Formel überprüfen da die Menge der Variablen der Zielkasse enthalten und leer ist. Auf analytischem Wege muss noch geprüft werden, ob der gewählte Name syntaktisch korrekt ist. Anschliessend können dann die Nachbedingungen der Basistransformation *Erzeugen einer Variablen* mit der aktuellen Formel verknüpft werden.
Dabei wird in der Formel die Menge der Variablen in der Zielklasse um die neu erstellte Variable erweitert. Diese Information erfüllt eine der nun zu überprüfenden Vorbedingungen der Basistransformation *Erzeugen einer Methode*. Vorbedingungen dieser Basistransformation die anhand der Formel überprüft werden können, sind die Existenz der Zielklasse und das die Erzeugung der Methode keinen Namenskonflikt erzeugt. Ob der gewählte Name syntaktisch korrekt ist muss auch hier auf analytischem Wege geprüft werden.
An dieser Stelle brechen wir das Beispiel ab, da sich die Vorgänge innerhalb des Algorithmusses nun für jede Basistransformation wiederholen. Zu dem Beispiel sei noch angemerkt, dass es viele Abhängigkeiten zu den vorher gehenden Basistrans-

formationen gibt. Insofern kann von diesem Beispiel nicht auf beliebige Sequenzen geschlossen werden, was den Umfang der eingesparten Analysen betrifft.

3.4. Vor- und Nachbedingungen von Kompositionen

Der Algorithmus liefert uns zum einen die Information ob eine Sequenz durchführbar ist. Darüber hinaus entspricht die Formel am Ende des Algorithmus den Nachbedingungen der Komposition der Teiltransformationen.
Durch eine kleine Modifikation kann der Algorithmus so angepasst werden, dass er auch die Vorbedingungen einer solchen Komposition liefert. Dazu wird eine zweite Formel innerhalb des Algorithmus gehalten. Diese wird ebenfalls mit den Vorbedingungen der ersten Transformation initialisiert. Im Verlauf des Algorithmus werden dieser Formel alle Vorbedingungen hinzugefügt die auf analytischem Wege überprüft werden müssen.
Für das in Abschnitt 3.3 vorgestellte Beispiel, würden die Vorbedingungen der Komposition alle hellen Bereiche aller Vorbedingungen in der Sequenz, wie sie in Abbildung 3.1 dargestellt werden, umfassen.

4. Zusammenfassung

Hintergrund dieser Ausarbeitung ist die evolutionäre Entwicklung von Softwareprojekten. In vielen Phasen des Lebenszykluses eines Softwareprojektes werden Anpassungen, aufgrund sich ändernder Anforderungen, notwendig. Mit Hilfe von *Refactorings* können dabei strukturelle Änderungen vorgenommen werden, ohne das von außen beobachtbare Programmverhalten zu beeinflussen.
Ziel dieser Ausarbeitung ist es, dem Leser eine Methodik vorzustellen, mit der es möglich ist beliebige Sequenzen von Transformationen vor ihrer Ausführung auf ihre Durchführbarkeit zu überprüfen. Mit Hilfe der vorgestellten Methodik, kann nun sichergestellt werden, dass strukturelle Änderungen das Programmverhalten nicht beeinflussen.
Grundlage dafür sind die von Obdyke identifizierten Basistransformationen und ihre zugehörigen Vor- und Nachbedingungen. Die Eigenschaft das Programmverhalten nicht zu verändern, wurde von Obdyke für die Basistransformationen nachgewiesen und gilt auch für beliebige durchführbare Kombinationen von Basistransformationen. Da selbst umfangreiche Änderungen in Sequenzen von Basistransformationen zerlegt werden können, kann für diese nun formal nachgewiesen werden, dass das Programmverhalten durch sie nicht verändert wird.
Durch leichte Modifizierung der vorgestellten Methodik ist es auch möglich Vor- und Nachbedingungen zusammen gesetzter Transformationen zu ermitteln. Damit halten wir nun einen Teil der notwendigen Grundlage zur automatischen Durchführung von *Refactorings* in den Händen. Was fehlt sind Methoden zur Überprüfung des Umfangs der Zerlegung einer Transformation bzw. die umfangreicher Anpassungen. So sollte sicher gestellt sein, dass eine bestimmte Zerlegung vollständig ist, d. h. das alle notwendigen Schritte in der Zerlegung enthalten sind. Aus Effizienzgründen ist es darüber hinaus wünschenswert sicherstellen zu können, dass eine Zerlegung minimal ist. "Überflüssige" Schritte wie das Erzeugen und anschließende Löschen von Programmelementen könnten so eingespart werden.
Die Darstellung der Abhängigkeiten zwischen Transformationen in Form von Abhängigkeitsgraphen, wie sie in Abschnitt 2.4 vorgeschlagen wurden, könnte beispielsweise dazu genutzt werden um überflüssige Transformationen zu identifizieren und/oder um eine automatischen Durchführung der Anpassungen durch Parallelisierung effizienter auszuführen. Diese potentiellen Anwendungsmöglichkeiten sind bisher nur "angedacht" worden, so dass hier sicherlich eine fundierte Prüfung notwendig ist.

A. Anhang

A.1. Basisrefactorings nach Opdyke

1. Definieren einer abstrakten Superklasse für eine oder mehrere bestehende Klassen
2. Spezialisieren einer Klasse durch das definieren einer Unterklasse, und Nutzen von Unterklassen zum Vermeiden von Bedingungstests
3. Verändern von Beziehungen der Klassen eines Modells in Teilen oder im Ganzen, von einer Vererbungshierarchie zu einer Instanzhierarchie mit aggregierten Komponenten
4. Verschieben von Klassen innerhalb und zwischen Vererbungshierarchien
5. Verschieben von Variablen und Methoden
6. Ersetzen eines Codeabschnittes durch einen Methodenaufruf
7. Umbenennen von Klassen, Variablen und Methoden
8. Ersetzen eines unbeschränkten Zugriffs auf eine Variable, druch eine abstraktre Schnittstelle

A.2. Basistransformationen nach Opdyke

Erzeugen von Programmeigenschaften :

1. Erzeugen einer leeren Klasse
2. Erzeugen einer Variablen
3. Erzeugen einer Methode

Löschen von Programmeigenschaften :

1. Löschen einer unreferenzierten Klasse
2. Löschen einer unreferenzierten Variablen
3. Löschen einer unreferenzierten Methode

Verändern von Programmeigenschaften :

1. Umbenennen einer Klasse
2. Umbenennen einer Variablen
3. Umbenennen einer Methode
4. Verändern des Typs einer Variablen oder Funktion

5. Verändern der Zugriffsrechte
6. Erweitern einer Methode um einen Parameter
7. Löschen eines Methodenparameters
8. Neuordnen von Methodenparametern
9. Hinzufügen eines Methodenrumpfes
10. Löschen eines Methodenrumpfes
11. Transformation einer Instanzvariablen in eine Variable die eine Referenz einer Instanz hält
12. Transformation einer Variablenreferenz in einen Methodenaufruf
13. Ersetzen eines Codeabschnittes durch einen Methodenaufruf
14. Ersetzen eines Methodenaufrufs durch den entsprechenden Methodenrumpf
15. Anpassen der Superklassenbeziehung

Verschieben von Variablen :

1. Überführen einer Variablen in die Superklasse
2. Überführen einer Variablen in eine Unterklasse

Einfache Transformationen :

1. Transformation einer Variablen in ein Attribut mit Zugriffsmethoden
2. Transformation eines Codeabschnittes in eine neue Methode
3. Verschieben einer Klasse

A.3. Einschränkungen zulässiger Programmeigenschaften durch Opdyke

1. Eindeutige Superklasse
2. Eindeutige Klassennamen
3. Eindeutige Methoden- und Variablennamen
4. Geerbte Variablen werden nicht neu definiert
5. Identische Signaturen beim Überschreiben einer Methode
6. Typsichere Zuweisungen
7. Semantische Äquivalenz von Referenzen und Operatoren

Literatur

[Beck99] Kent Beck: Extreme Programming Explained: Embrace Change. Addison-Wesley, 1999. 3

[Cinneide] Mel O Cinneide, Paddy Nixon: Composite Refactorings for Java Programs

[Fowler99] Martin Fowler: Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999. 3

[Kuttruff02] Volker Kuttruff: Ein Modell für invasive Softwareadaption, Diplomarbeit, Universität Karlsruhe, 2002.

[Roberts99] Donald Bradley Roberts: Practical Analysis for Refactoring. Doktorarbeit, University of Illinois at Urbana-Champaign, 1999. 4, 6, 9, 12

[Obdyke92] William F. Obdyke: Refactoring Object-Oriented Frameworks. Doktorarbeit, University of Illinois at Urbana-Champaign, 1992. 3, 4