Quelltext-Transformation für Restrukturierungen anhand expliziter Intentionen

Klaus Meffert[1]
Ilka Philippow[2]

Technische Universität Ilmenau
98693 Ilmenau
PF 100565
meffert@rz.tu-ilmenau.de[1], ilka.philippow@tu-ilmenau.de[2]

Abstract: Die Restrukturierung von Programmcode, das Refactoring, verfolgt das Ziel, die Wartbarkeit eines Softwaresystems zu verbessern. Dabei werden häufig Entwurfs- oder die komplexeren Architekturmuster eingesetzt. Um diese Muster anwenden zu können, muss der gegebene Quelltext transformiert werden. Diese Arbeit zeigt Möglichkeiten, um Transformationen von Quelltext zu definieren, deren Anwendbarkeit auf unbekannte Quelltexte zu erkennen und sie zu einem gewissen Grad automatisiert auszuführen. Die Basis dieser Prozesse ist ein Programmverständnis, das für eine Maschine durch das Dokumentieren von Intentionen im Quelltext zu einem hohen Grade ermöglicht werden soll.

1 Einführung

Die Position der Autoren ist, dass der Entwickler bei komplizierteren Refactoring-Operationen, wie der Selektion und Anwendung von Entwurfs- oder Architekturmustern, durch Werkzeuge nicht genügend unterstützt wird. Eine weitere Position ist, dass derartige Operationen wichtig sind, um die Wartbarkeit und auch die Verständlichkeit eines Programms aufrechtzuerhalten. Als drittes besteht die durch Beobachtung vieler Projekte gereifte Position, dass die meisten Software-Projekte heutzutage hauptsächlich quelltextorientiert sind, das bedeutet, der Entwickler arbeitet überwiegend mit dem Quelltext und weniger mit Modellierungswerkzeugen, Code-Generatoren oder einer modellgetriebenen Methodik.

Diese Arbeit thematisiert Transformationen von Quelltext, um Refactoring-Operationen, zu denen auch die Anwendung von Entwurfsmustern gezählt wird, so automatisiert wie möglich ablaufen zu lassen. Um dies zu ermöglichen muss eine durchzuführende Quelltext-Transformation von der Maschine „verstanden“ werden.

Das folgende Kapitel beschreibt die der Arbeit zugrunde liegende Problematik bei quelltextbasierter Software-Entwicklung. Kapitel 3 stellt den Ansatz dieser Arbeit vor, der die Problematik reduzieren soll. Kapitel 4 bietet eine Zusammenfassung und einen Ausblick.

2 Problembeschreibung

Software wird aufgrund der technologischen Entwicklungen und gesteigerten Anwenderbedürfnisse immer komplexer. Deren Wartbarkeit ist sehr wichtig, um einerseits schnell und kostengünstig sowie andererseits mit möglichst wenigen Fehlern Erweiterungen und Korrekturen vornehmen zu können.

Eine weit verbreitete Möglichkeit zur Erhöhung der Wartbarkeit und Erweiterbarkeit einer Software sind Refactoring-Operationen. Typische Refactoring-Operationen können sehr einfach sein, z.B. *Encapsulate Field* (vgl. [Fo99]), aber auch sehr komplex, wie beispielsweise die Anwendung von Entwurfsmustern (z.B. *Composite*, vgl. [Ga95]). Aufgrund der Vielzahl an dokumentierten Mustern ist es für eine Person kaum möglich, für ein gegebenes Architekturproblem das optimale Muster auswendig oder überhaupt zu kennen oder ein geeignetes Muster schnell zu identifizieren. Die falsche Anwendung eines Musters wird mit steigender Komplexität zudem wahrscheinlicher.

In einer von akutem Druck getriebenen Branche kommen Aktivitäten, die zur Qualitätssicherung der Architektur langlebiger Software notwendig sind, aus Zeitgründen oft zu kurz. Dagegen können gut strukturierte und verständliche Programme Aufwand und Kosten für die Wartung beachtlich reduzieren und sich langfristig auszahlen. Aus Erfahrung können die Autoren bestätigen, dass in vielen kommerziellen Software-Projekten von den meisten Entwicklern keine komplexeren Muster angewandt werden und wenn, dann oft mit negativem Effekt.

Die automatisierte Erkennung und Ausführung von Refactoring-Operationen zur Milderung des Problems mangelnder Architekturqualität wird allerdings von verfügbaren Entwicklungswerkzeugen nur in begrenztem Maße unterstützt. So werden nur sehr einfach erkennbare Refactorings erkannt und nur relativ triviale Transformationen vorgenommen. Beispielsweise wird der Entwickler bei der Anwendung eines Entwurfsmusters von der IDE objectiF [Ob06] lediglich dadurch unterstützt, dass die festen Bestandteile (wie Schnittstellen) angelegt und variable Teile (wie Methodenrümpfe) leer und mit einem Kommentar wie „*Add your code here*“ erzeugt werden.

3 Vorgeschlagene Lösung

Die vorgeschlagene Lösung konzentriert sich auf ein großes Problem beim Refactoring, der fehlenden Werkzeugunterstützung für die Bestimmung sinnvoller Operationen und ihrer Anwendung. Eine These dieses Papiers ist, dass eine Maschine ohne explizit verfügbar gemachte Informationsbasis unbekannte Quelltexte nicht ausreichend „verstehen“ kann, um die Notwendigkeit komplexerer Transformationen erkennen zu können. Die Anwendung einer Transformation erfordert zusätzliche Informationen. Eine weitere These ist, dass für einen unbekannten Quelltext A, der ähnlich einem bekannten Quelltext B ist, die Transformationsschritte automatisiert von B analog auf A analog anwendbar sind. Diese Annahme orientiert sich daran, dass ein Entwickler sowohl

ähnliche Programmtexte erkennen als auch unbekannte Programme in Ihrer Bedeutung durch Vergleich mit bisher Erfahrenem erfassen kann.

Das Paradigma des Ansatzes basiert darauf, bei der beispielhaften Transformation von Quelltext A nach B so viele Informationen bereitzustellen, damit jeder Transformationsschritt maschinell nachvollzogen werden kann. Nicht automatisch erkannte Schritte müssen während dieses Schrittes explizit definiert werden (etwa durch Hinzufügen weiterer Dokumentationen), was mit höherer Anzahl an Definitionen immer unwahrscheinlicher wird. Im Anschluss müssen die Angaben auf ihre Generalisierbarkeit untersucht werden, indem als Eingabe ein unbekannter Quelltext gewählt wird. Im Zuge dessen kann es erforderlich werden, die Transformationsdefinitionen (iterativ) zu adaptieren. Die Errichtung einer Hierarchie von Dokumentationen samt zugehörigem Quelltext geht hiermit einher. Als Endergebnis sollen Definitionen vorliegen, die möglichst generell auf unbekannte Quelltexte anwendbar sind.

Die vorgestellte Lösung basiert auf mehreren Schritten zur Erreichung des Ziels, Quelltexttransformationen definieren, deren Anwendbarkeit erkennen und sie zu einem gewissen Grad automatisiert ausführen zu können:

1. Exemplarisches Durchführen einer Transformation von Quelltext A nach B.
2. Festhalten der Intention von Quelltextbestandteilen durch Hinzufügen von maschinenverarbeitbaren Dokumentationen zu A und B.
3. Erkennen der Anwendbarkeit einer Transformation von Quelltext C durch Auswerten der Informationen aus den vorigen Schritten, durch Hinzufügen von Dokumentationen analog zu Schritt 2 und durch Einführen von Isomorphen.
4. Durchführen einer (möglicherweise unreifen) Transformation durch Auswerten der verfügbaren Informationen.

Die folgenden Abschnitte beschreiben die eben genannten vier Schritte und dokumentieren die weiterverwertbaren Informationen, die bei der Analyse der vorliegenden Daten anfallen.

3.1 Exemplarisches Durchführen einer Transformation

Dieser Prozess hat das Ziel, eine Transformation eines Quelltextes A in einen Quelltext B beispielhaft durchzuführen und die dabei anfallenden Informationen zu speichern. Quelltext A ist so gewählt, dass auf ihn eine komplexe Refactoring-Operation, etwa ein bekanntes Entwurfsmuster, sinnvoll anwendbar ist. B beinhaltet dann das angewandte Muster und behält den Rest von A bei. Dieses Vorgehen ist vergleichbar mit dem in [Li01] bezeichneten *Programming By Example*. Geeignete Ausgangsquelltexte können entweder konstruiert, in Projekten mit offen gelegtem Quelltext gesucht oder der Literatur entnommen werden.

Damit die Transformationsschritte von A nach B nachvollzogen werden können, wird jeder Schritt durch Anbringung wohldefinierter Kommentare in A und B dokumentiert (in [Me06] werden Annotationen als wohldefinierte Kommentare eingeführt, in [JSR175] als vollwertige Sprachkonstrukte). Um eine Verknüpfung eines Teils aus A mit

einem Teil aus B feststellen zu können, wird für einen korrespondierenden Teil in A und B jeweils derselbe Kommentar verwendet, der innerhalb dieses Prozesses zudem eindeutig ist. Der folgende Quelltext zeigt ein Beispiel für die Dokumentation eines Quelltextes (Annotation als Dokumentation jeweils fett gedruckt):

```
public void setCelsius(double value) {
  /*@@intent store celsius*/
  celsius = value; // Instanzvariable vom Typ double
  /*@@intent update display*/
  setDisplay("" + celsius);
}
```

Wenn der obige Quelltext beispielhaft als Teil von Quelltext A angesehen werden kann, so könnte ein durch das *MVC*-Muster [Bu96] zugehöriger transformierter Quelltextteil B so aussehen:

```
/*@@intent update display*/
public void update(Observable t) {
  setDisplay("" + model().getCelsius());
}
```

Einige weitere Quelltextteile sind aus Platzgründen hier nicht angegeben (so etwa die Deklaration des MVC-Modells für Quelltext B). Eine Verknüpfung zwischen A und B kann nun über die bei beiden vorliegende Annotation "*intent update display*“ hergestellt werden. Die nur in A abgebildete Annotation „*intent store celsius*“ ist in B an anderer Stelle ebenfalls vorhanden. Ein ableitbarer Transformationsschritt ist letztlich, dass die Instanzvariable *celsius* aus A nicht in der ursprünglichen Klasse verbleibt, sondern in eine neue Klasse, die die Rolle *Modell* innerhalb des *MVC*-Musters spielt, verschoben wird (wo das Äquivalent zur Variablen *celsius* aus A existiert und vor Aufruf von Methode *update* gesetzt wird).

Die Semantik der Dokumentation über Annotationen ist für die Definition einer Transformation nicht vorrangig. Wichtiger ist hier ihre Eindeutigkeit und ihr Vorhandensein sowohl in A als auch in B. Für den Entwickler, der die Definition vornimmt ist es allerdings ein Vorteil ein aussagekräftiges Konstrukt verwenden zu können, um das Geschriebene später leicht nachvollziehen zu können. Das ist analog zu einem obfuskierten Quelltext, der zwar dasselbe wie der Ausgangsquelltext tut, aber in seiner Bedeutung viel schwieriger vom Entwickler erfasst werden kann.

Die Gestaltung der Dokumentation muss lediglich eine Maschinenverarbeitung zulassen und eine gute Lesbarkeit für einen Entwickler realisieren. Weitere Ausführungen hierzu sind in [Me06] zu finden, wo eine zusammengesetzte Annotation vorgeschlagen wird. Sie enthält einen maschinenlesbaren und einen natürlichsprachlichen Teil.

Es gibt mehrere Fälle bei Transformation von A nach B zu unterscheiden, die unterschiedliche Anforderungen an eine Dokumentation mit Annotationen haben, nämlich:

1. Teil aus A wird ohne Veränderung in B übernommen → Keine Dokumentation notwendig.
2. Teil aus A wird an anderer Stelle (etwa anderer Klasse) in B unverändert eingesetzt (Verschiebung) → Dokumentation des Teils in A und B
3. Teil aus A wird restlos entfernt → Dokumentation des Teils in A
4. Teil wird neu zu B hinzugefügt → Dokumentation des Teils in B
5. Teil wird von A nach B modifiziert → Dokumentation des Teils in A und B

Nach Analyse von A und B anhand der Untersuchung der Annotationen, des Quelltextes und der Abstrakten Syntaxbäume (ASTs) kann eine Reihe von Informationen ermittelt werden:

- Transformationsanweisungen für alle Teile aus A nach B anhand der Annotationen (welche Teile wurden kopiert, entfernt, neu hinzugefügt, modifiziert, isomorph modifiziert (siehe Abschnitt 3.3), verschoben?)
- "Bedeutung" der annotierten Teile: Einem Programmstück ist eine eindeutige Annotation zugeordnet, die dessen Bedeutung im Rahmen der Transaktion ausdrückt
- Beziehung der annotierten Teile durch AST-Analyse (wer ruft wen, wer referenziert wen, wer wird von wem referenziert?)
- Beziehung der Annotationen durch AST-Analyse über Beziehung der annotierten Teile (welche Annotation ist einer anderen untergeordnet?)
- Sprechende Namen, Namensräume (etwa: Klasse *java.util.List* stellt eine geordnete Liste dar)
- Erfüllen A und B die dazu definierten Vorbedingungen? Wenn nein, ist die Definition fehlerhaft (zudem sind weitere Konsistenzprüfungen möglich, etwa ob zwei ungleiche Teile in A und B annotiert sind oder nicht)

Damit Transformationsschritte maschinell nachvollziehbar sind, werden bisher bekannte Annotationsdefinitionen sowie die vorliegenden Informationen mit einander abgeglichen (Vergleich von Annotationen unter Berücksichtigung definierter Hierarchien, Vergleich von Quelltextstücken inklusive isomorpher Betrachtung, Vergleich von Strukturen über Aufrufer einer Methode und Referenzierer einer Klasse oder Variablen). Ein Transformationsschritt kann etwa sein: Ersetze den Methodenrumpf X aus A, der mit Annotation Y versehen ist, durch die Logik Z aus B (in Z sind auch Platzhalter möglich).

3.2 Dokumentieren von Intentionen im Quelltext

Nachdem im vorigen Abschnitt bereits Intentionen dokumentiert wurden, müssen noch solche dokumentiert werden, die nicht automatisch folgerbar sind, auch wenn sie nicht direkt mit der Transformation in Beziehung stehen. Das betrifft insbesondere dynamische Methodenaufrufe, wie sie in Java etwa über *Reflection* möglich sind.

Das in Abschnitt 3.1 beschriebene Vorgehen beinhaltet bereits die Dokumentation von Intentionen im Quelltext mithilfe von wohldefinierten Kommentaren. Erfasst sind aber nur Quelltextteile, die bei Transformation von A nach B irgendwie verändert wurden.

Für die Erkennung der Anwendbarkeit einer Transformationen für einen unbekannten Quelltext kann es hilfreich und notwendig sein, dass auch die in A oder B von einer annotierten Stelle heraus gerufenen oder referenzierten Teile (Methodenaufruf, Dereferenzierung einer Variable o.ä.) annotiert werden. Das bedeutet letztendlich, dass die Annotation weiterer Programmstücke die Wahrscheinlichkeit erhöht, die Anwendbarkeit einer Transformation zu erkennen.

Da Intentionen unterschiedlich abstrakt ausgedrückt werden können, wird vorgeschlagen, eine Hierarchie von Annotationen zu definieren. So können auch auf Ebene der Annotationen quasi-äquivalente Ausprägungen erkannt werden. Beispiel für eine Hierarchie von Annotationen:

`increase celsius` → `increase value` → `change value` → `change state of class` → `change state of application`

Es ist außerdem vorstellbar, eine Ähnlichkeitsmatrix für Annotationen aufzustellen, um das Ergebnis eines Vergleichs gewichten zu können.

3.3 Erkennen der Anwendbarkeit einer Transformation

Um für einen bisher unbekannten Quelltext C die Anwendbarkeit einer wie oben beschrieben definierten Transaktion erkennen zu können, muss C mit A und B verglichen werden. Ein solcher Vergleich bedeutet, die für die Transformationen wesentlichen Eigenschaften in C wiederzufinden und Vorbedingungen für C zu prüfen, die für A und B definiert worden sind. Da C ungleich A ist (sonst wäre das Problem keines) müssen Analogien zwischen C und A erkannt werden können. Um eine Analogie zwischen Quelltextteilen zu ermöglichen, wird vorgeschlagen Isomorphe zu definieren (vgl. auch [Me06]). Ein Isomorph ist hier ein Quelltext, der zwar nicht gleich einem anderen ist, aber in einem gegebenen Kontext (hier: im Rahmen einer Transformation) eine äquivalente Bedeutung besitzt. Beispielsweise macht es in Java keinen Unterschied, ob eine Anweisung *x++* lautet oder *x + 1* (mit x als ganze Zahl). Es ist in vielen Fällen (bei Transformationen) auch egal, ob Elemente in einer geordneten Liste gehalten werden oder einer ungeordneten, auf Hashwerten beruhenden (etwa für das Muster *Observer* [Ga95]).

Für C könnten, ebenfalls wie für A und B, Intentionen dokumentiert sein, etwa manuell oder durch Werkzeuge. Werkzeuge können Annotationen hinzufügen, indem sie Quelltextstücke von C isomorph mit annotierten Teilen aus A und B vergleichen und die in Abschnitt 3.1 beschriebenen Informationen ausbeuten.

Quelltext C kann mit A und B auf mehrere Arten verglichen werden:

1. Vergleich der Quelltexte in A und B mit C unter Berücksichtigung von Isomorphen → Identifizieren äquivalenter Quelltextteile
2. Vergleich von Annotationen aus A und B mit C → Erkennen von gleichbedeutenden Quelltextteilen

3. Vergleich von nicht annotierten Teilen aus C mit annotierten Teilen aus A und B → Hinzufügen von Annotationen zu C möglich
4. Vergleich der ASTs von A und B mit dem von C → Identifizieren von gleichen Strukturen

3.4 Durchführen einer Transformation

Nachdem der Vergleich von C mit A und B wie eben erfolgt ist, kann aus dem Grad der Übereinstimmung einer der folgenden Fälle erkannt werden:

1. Genügend Informationen in C vorhanden (über Annotationen, durch Erkennen isomorpher Teile oder gleicher Strukturen):
   1.1. Widersprüche zwischen C und A oder B entdeckt → Transformation wahrscheinlich ungeeignet
   1.2. Keine Widersprüche entdeckt: Transformation wahrscheinlich geeignet
2. Nicht genügend Informationen in C vorhanden:
   2.1. Widersprüche zwischen C und A oder B entdeckt → Transformation wahrscheinlich ungeeignet
   2.2. Keine Widersprüche entdeckt: Anwender des Verfahrens nach weiteren Informationen fragen

Ein Widerspruch ergibt sich etwa, wenn eine in C vorhandene Annotation nicht in A und B vorliegt, der zugehörige Quelltext nicht als isomorph erkannt wurde oder Vorbedingungen nicht erfüllt sind.
Fall 2.2 erfordert, dass der das Werkzeug benutzende Anwender konsultiert wird. Die noch nicht vorliegenden Informationen betreffen Teile aus C, für die weder eine Übereinstimmung in A oder B bezüglich Annotationen gefunden wurde noch ein isomorpher Vergleich erfolgreich war. Insgesamt kann ein solcher Fall nur auftreten, wenn die Wissensbasis nicht vollständig ist. Das wird allerdings lange der Fall sein, da das Verfahren auf der Definition bisher unbekannter Strukturen basiert (siehe Abschnitt 3.1). Das bedeutet, mit der Anzahl von Definitionen steigt die Wahrscheinlichkeit, innerhalb eines bisher unbekannten Quelltextes bekannte Konstrukte in ausreichender Zahl identifizieren zu können.
Da eine Maschine nach Erfahrung der Verfasser nicht in der Lage ist, einen Quelltext zu verstehen vergleichbar dem wie ein Programmierer dies kann, ist es möglich, bei der Transformation unreife Resultate zu erhalten, die Nacharbeit durch den Anwender des Verfahrens bedeuten.

4 Zusammenfassung und Ausblick

Es wurde ein Verfahren skizziert, mit dem eine Quelltexttransformation über die Dokumentation von Intentionen im Quelltext definiert werden kann. Die sich daraus ergebenden Informationen und die Definition von Isomorphen sind nach Ansicht der Autoren geeignet, Analogien beim Vergleich mit anderen Quelltexten zu erkennen. Diese Analogien können dann zur Erkennung der Anwendbarkeit einer Transformation

für einen Quelltext dienen. Eine Durchführung der Transformation selbst ist nach Meinung der Autoren möglich, wenngleich diese unreif sein kann.

Das Verfahren basiert darauf, dass eine sehr gute Transformationsdefinition vorliegt. Das wird initial nicht ohne weiteres gelingen. Iterative Verbesserungen sind notwendig. Dies kann kompensiert werden, indem nach Transformationsdefinition eine Reihe anderer geeigneter Ausgangsquelltexte durchgespielt wird, um Verfeinerungen vorzunehmen. Analog zu [JSR250], wenngleich in größerem Rahmen, ist die Bereitstellung einer Basismenge von Annotationen Ziel laufender Bemühungen.

Die Festlegung von Isomorphen ist zeitaufwendig, aber nach Meinung der Autoren notwendig. Die mangelnde maschinelle Intelligenz wird durch Massendaten quasi kompensiert. Zur Rechtfertigung des hohen Aufwands, der bei der Definition der Transformationen, der Code-Intentionen sowie der Isomorphe auftritt, sei auf den Aufwand verwiesen der nötig ist, damit ein Mensch nach Geburt eine Sprache erlernt. Geeignete Sensoren wie sie der Mensch besitzt (und die für die Erlernung einer Sprache durch den Menschen wichtig sind) stehen der Maschine nicht zur Verfügung und anstelle der Eltern eines lernenden Kindes treten die erwähnten Definitionen und Massendaten.

Zur Zeit entwickelt einer der Autoren einen komplexen Prototypen für die Sprache Java, der das in diesem Papier beschriebene Verfahren umsetzt. Der Prototyp beherrscht bereits umfangreiche Quelltextanalysen und soll durch Analyse mehrerer Open-Source Projekte auf seine Tauglichkeit hin getestet werden, Analogien zu erkennen und Transformationen durchzuführen. Das bedeutet in Konsequenz, für einzelne Quelltextteile Voraussagen über deren Bedeutung zu machen. Eine Vision dieser Arbeit ist es, für einen beliebigen, unannotierten Quelltext die Anwendbarkeit zuvor definierter Transformation feststellen und diese Transformationen so weit wie möglich zu automatisieren.

Literaturverzeichnis

[Bu96] Buschmann, F. et al.: Pattern Oriented Software Architecture. Wiley, 1996.

[Fo99] Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, 1999.

[Ga95] Gamma, E. et al.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[JSR175] Java Specification Request 175: A Metadata Facility for the Java™ Programming Language. http://www.jcp.org/en/jsr/detail?id=175.

[JSR250] Java Specification Request 250: Common Annotations for the JavaTM Platform. http://www.jcp.org/en/jsr/detail?id=250.

[Li01] Lieberman, H.: Your Wish Is My Command: Programming By Example. Morgan Kaufman, 2001.

[Me06] Meffert, K.: Supporting Design Patterns with Annotations. *ecbs*, pp. 437-445, 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems (ECBS'06), 2006.

[Ob06] objectiF: http://www.microtool.de/objectif/de/index.asp.