Pattern/Refactoring

1 Towards Generic Refactoring

Ralf Lämmel
Vrije Universiteit, De Boelelaan 1081a, NL-1081 HV Amsterdam, CWI, Kruislaan 413, NL-1098 SJ Amsterdam, Ralf.Laemmel@cwi.nl

We define a challenging and meaningful benchmark for genericity in language processing, namely the notion of generic program refactoring. We provide the first implementation of the benchmark based on functional strategic programming in Haskell. We use the basic refactoring of abstraction extraction as the running example. Our implementation comes as a functional programming framework with hot spots for the language-specific ingredients for refactoring, e.g., means for abstraction construction and destruction, and recognisers for name analysis. The language-parametric framework can be instantiated for various, rather different languages, e.g., Java, Prolog, Haskell, or XML schema.
The full paper appeared in [1].

Bibliography

[1] R. Lämmel. Towards Generic Refactoring. In *Proc. of Third ACM SIGPLAN Workshop on Rule-Based Programming RULE'02*, pages 15–28, Pittsburgh, USA, 5 Oct. 2002. ACM Press. Paper obtainable from the ACM Digital Library.

2 Werkzeuggestützte Problemidentifikation und -behebung

Volker Kuttruff, Thomas Genßler, Markus Bauer, Olaf Seng
Forschungszentrum Informatik Karlsruhe (FZI), Haid-und-Neu-Straße 10-14, 76131 Karlsruhe, {kuttruff|genssler|bauer|seng}@fzi.de

2.1 Einleitung und Ziel

Zur Evolution von Software werden mit wachsender Größe der Software in zunehmenden Maße unterstützende Werkzeuge verwendet. Die Anwendungsgebiete solcher Werkzeuge umfassen die Analyse und Visualisierung von Software, die Identifikation von Problemstellen und die automatische Reorganisation von Software durch Refaktorisierungsoperationen. Während für jedes dieser Gebiete mehr oder weniger brauchbare Insellösungen existieren, fehlt derzeit jedoch eine geschlossene Methoden- und Werkzeugunterstützung von der automatisierten Problemerkennung über die Auswahl (oder Unterstützung bei der Auswahl) von Refaktorisierungsoperationen bis hin zur automatischen Durchführung dieser Refaktorisierungen. In [BGKS02] wurden erste Ansätze zur Verzahnung der oben genannten Techniken vorgestellt. Dieser Beitrag vertieft dieses Thema und berichtet über erste Fortschritte. Das Ziel unserer Arbeit ist die Entwicklung einer geschlossenen Methoden- und Werkzeugkette für die Evolution von Software. Dies umfasst:

- Die Verzahnung von Problemerkennung und Problembehebung auf technischer Ebene zur Beschreibung von Problemmustern und Lösungen mit lokaler Kenntnis des Systems.
- Die Integration der technischen Realisierung der Problemidentifikation und -behebung mit Qualitätsmodellen und einem Expertensystem zur teilautomatischen Auswahl von Problemmustern und möglicher Lösungsstrategien anhand vom Nutzer zu bestimmender und zu gewichtender Qualitätsmerkmale.

In diesem Beitrag konzentrieren wir uns auf den ersten Problembereich.

2.2 Ansatz

Metamodell Um die werkzeuggestützte Problemidentifikation und -behebung geschlossen durchführen zu können, benötigen die daran beteiligten Werkzeuge ein gemeinsames Metamodell. Ausgehend von den am FZI entwickelten Werkzeugen zur Analyse (jGoose [BSLM03])

und Transformation (Inject/J [GK03], Recoder [Rec02]) vereinheitlichten wir daher deren bisher unabhängig voneinander entstandenen Metamodelle. Dieses vereinheitlichte Metamodell besitzt im Wesentlichen zwei Stoßrichtungen: Zum einen ist dies Sprachunabhängigkeit, zum anderen ein dem jeweiligen Anwendungszweck angepasster Detaillierungsgrad. Sprachunabhängigkeit bezieht sich in unserem Kontext auf die Eignung des Modells für ausdrucksbasierte, statisch typisierte, objektorientierte Sprachen. Dies wird durch einen sprachunabhängigen Kern (Klassen, Methoden, Attribute etc.) des Metamodells mit jeweils sprachabhängigen Erweiterungen sichergestellt. Zur Zeit existiert eine detaillierte Beschreibung einer solchen Erweiterung nur für die Sprache Java. Die Abbildung auf C++, C# und Delphi werden derzeit nachgezogen. Der Kern stellt den niedrigsten Detaillierungsgrad des Metamodells dar. Die im Kern vorhandenen Informationen sind im Allgemeinen ausreichend, um Problemstellen in einem System zu identifizieren. So stellt der Kern zum Beispiel bereits eine Reihe von Basismetriken (z.B. McCabe-Komplexität von Methoden) zur Verfügung. Weiterhin existiert eine Erweiterung des Kerns mit höherem Detaillierungsgrad, welche feingranularere Informationen sowie Basistransformationen zur Verfügung stellt.

Aufbauend auf diesem Metamodell existiert eine Skriptsprache [GK01], die sowohl für die Beschreibung der Problemstellen als auch für die Spezifikation der das Problem lösenden Transformationen geeignet ist.

Erkennungsmuster Die geschlossene Beschreibung der problematischen Stellen eines Systems erfolgt in unserer Sprache mit Hilfe so genannter *Erkennungsmuster (detection patterns)*. Dies sind im Wesentlichen deklarative Beschreibungen von nicht notwendigerweise zusammenhängenden Graphmustern, welche die Problemstellen eines Systems charakterisieren. Erkennungsmuster beschränken sich dabei nicht auf die Beschreibung rein struktureller Eigenschaften, sondern erlauben auch die Nutzung darüber hinausgehender Kontextinformationen wie zum Beispiel Metriken. Um diese über strukturelle Muster hinausgehenden Kontextinformationen zu nutzen, haben Erkennungsmuster Zugriff auf die gesamten Informationen, welche das Metamodell zur Verfügung stellt, also insbesondere auch auf Basismetriken, Typ- und Querverweisinformationen etc. Wie bereits erwähnt, geschieht die Spezifikation der zu suchenden Problemmuster deklarativ. Werden von einem Erkennungsmuster allerdings Informationen benötigt, die komplexere Berechnungen zur Folge haben (z.B. komplexe Metriken), so lassen sich diese – im Gegensatz zu reinen musterbasierten Analyse- und Transformationssystemen – imperativ angeben. Dieses Vorgehen wurde gewählt, da sich Berechnungen im Allgemeinen einfacher imperativ als deklarativ beschreiben lassen, insbesondere dann, wenn eine große Menge an Kontextinformationen benötigt wird. Die Suche nach potentiellen Problemstellen, die den in den Erkennungsmustern angegebenen Bedingungen genügen, kann nun mit dem entsprechenden Werkzeug Inject/J automatisiert werden.
Die den Bedingungen der Erkennungsmuster genügenden Graphmuster werden als Instanzen eines Erkennungsmusters bezeichnet. Diese können im Weiteren als Einheit betrachtet werden, auch wenn die zum erkannten Graphmuster beitragenden Strukturelemente über weite Teile des Systems verteilt sind.

Transformation Auf Basis der gefundenen Erkennungsmuster-Instanzen können im nächsten Schritt die notwendigen Transformationen angegeben werden. Dies geschieht ebenfalls mit der durch Inject/J bereitgestellten Skriptsprache. Die Spezifikation der Transformationen erfolgt dabei imperativ, da dies meist einfacher anzugeben ist, insbesondere falls zahlreiche Kontextinformationen beachtet werden müssen. Die teilweise komplexen und umfangreichen Transformationen werden in Termen sogenannter *Basistransformationen* ausgedrückt, welche durch die entsprechenden Erweiterungen unseres Metamodells bereitgestellt werden. Die Basistransformationen lassen sich dabei mit nur lokaler Kenntnis der zu transformierenden Stellen durchführen. Dies bedeutet, dass eine Basistransformation in eine primäre Transformation und weitere sekundäre bzw. abhängige Transformationen aufgeteilt werden kann. Die primäre Transformation führt die eigentlich gewünschte Änderung durch (zum Beispiel das Umbenennen eines Attributs), die sekundären Transformation ändern automatisch alle abhängigen Stellen (zum Beispiel die Benutzungsstellen des Attributs) oder führen notwendige Kontextanpassungen durch (zum Beispiel das Ausrollen von Ausdrücken). Zusammen mit einer Reihe von Vorbedingungen, welche für jede Basistransformation durch das Transformationsmodell gegeben sind, wird die Übersetzbarkeit des transformierten Systems durch unser Werkzeug Inject/J garantiert. Darüberhinaus lassen sich in Inject/J noch nutzerspezifische Vorbedingungen für komplexe Transformationen angegeben, um weitere Eigenschaften wie zum Beispiel Verhaltensbewahrung zuzusichern.

Werkzeug Das Werkzeug Inject/J erlaubt es nun, automatisch nach möglichen Problemstellen im System zu suchen. Ist eine solche Problemstelle durch ein Erkennungsmuster gefunden und eventuell durch den Benutzer bestätigt worden, so lassen sich unter Zuhilfenahme der in der Erkennungsmuster-Instanz gekapselten Informationen entsprechende Transformationen durchführen.

2.3 Zusammenfassung

Im vorliegenden Beitrag haben wir ein Verfahren zur Integration von Techniken der Problemerkennung und automatisierten Softwaretransformation skizziert. Die Hauptidee besteht darin, die Transformationen direkt mit der Beschreibung von Probleminstanzen zu verbinden und dadurch zielgerichtet durchzuführen. Die Struktur von Probleminstanzen wird mit Hilfe von Erkennungsmustern spezifiziert. Erkennungsmuster fassen Strukturelemente,

die im Strukturgraphen des Systems möglicherweise verteilt sind, anhand ihrer Eigenschaften (Beziehungen im Strukturgraphen, Basismetriken etc.) zu neuen, 'virtuellen' Einheiten zusammen und erlauben die geschlossene Transformation dieser Einheiten. Die Transformation selbst wird durch eine Kombination von Basistransformationen beschrieben, welche die Eigenschaft haben, notwendige Sekundäroperationen zur Behebung nicht-lokaler Effekte der Primärtransformation automatisch durchzuführen. Dadurch wird die aktuelle Lücke zwischen Problemerkennung und Behebung zumindest auf technischer Ebene teilweise geschlossen.

Literaturverzeichnis

[BGKS02] M. Bauer, T. Genßler, V. Kuttruff, and O. Seng. Werkzeugunterstützung für evolutionär Softwareentwicklung. In *Proceedings of the 4th German Workshop on Software-Reengineering*, July 2002.

[BSLM03] M. Bauer, O. Seng, S. Luzar, and T. Marz. jGoose Echidna WWW Page. `http://jgoose.sf.net/`, 2003.

[GK01] T. Genßler and V. Kuttruff. Werkzeugunterstützte Softwareadaption mit Inject/J. In *Proceedings of the 3th German Workshop on Software-Reengineering*, July 2001.

[GK03] T. Genssler and V. Kuttruff. Inject/J WWW Page. `http://injectj.sf.net/`, 2003.

[Rec02] The RECODER/Java homepage. `http://recoder.sf.net`, 2002.

---

3 Vergleich von Klonerkennungstechniken

Stefan Bellon, Daniel Simon
Universität Stuttgart, Institut für Softwaretechnologie, Universitätsstraße 38, D-70569 Stuttgart,
{bellon,simon}@informatik.uni-stuttgart.de

3.1 Einleitung

„Copy & Paste“ ist noch immer das vorherrschende Programmierparadigma, wenn es um Wiederverwendung von Code geht. Durch häufiges „Copy & Paste“ leidet jedoch die Wartbarkeit des Systems, da ein kopierter Fehler eventuell an vielen Stellen korrigiert werden muss. Allerdings ist in den seltensten Fällen dokumentiert, wohin ein Stück Code kopiert wurde. In der Literatur wurden eine Reihe von Techniken zur Entdeckung so genannter Klone (also Code-Stücke, die sich aus „Copy & Paste“ ergaben) vorgeschlagen. Jedoch ist bis dato unklar, welche der Techniken unter welchen Umständen die bessere ist.

Um dies herauszufinden, haben sich die Wissenschaftler Baker [1], Baxter [2], Kamiya [5], Krinke [7], Merlo [6] und Rieger [4] zusammengetan, um die verschiedenen Techniken quantitativ und qualitativ zu vergleichen.

Es wurde ein Vorgehen erarbeitet, welches den quantitativen Vergleich der Ergebnisse ermöglicht. Die Techniken wurden auf Java- und C-Systeme angewendet, die mit versteckten Klonen präpariert wurden, um die Aussagekraft der Ergebnisse einordnen zu können. Die Analyse und Auswertung der Ergebnisse soll vorgestellt und die Stärken und Schwächen der einzelnen Techniken herausgearbeitet werden.

3.2 Vergleichsmethode

Um die Techniken der sechs genannten Wissenschaftler miteinander vergleichen zu können, musste zuerst ein gemeinsames Verständnis von „Klon“ gefunden werden. Die Teilnehmer konnten sich alle darauf einigen, dass ein Paar von aufeinander folgenden, ununterbrochenen Sourcecode-Zeilen, so genannten „Codefragmenten“, ein „Klonpaar“ ergibt und dies als Vergleichsgröße aller Werkzeuge und Techniken dient. Techniken, die mit „Klonklassen“ arbeiten, können diese als Menge von Klonpaaren darstellen und können somit auch eingebunden werden.

Drei Typen von Klonen wurden definiert:

Typ 1: Exakte Kopie (keinerlei Veränderung bis auf Whitespace und Kommentare, z. B. Inlining von Hand)

Typ 2: Parametrisierte Übereinstimmung (Bezeichner werden in der Kopie umbenannt, z. B. „Wiederverwendung“ einer Funktion)

Typ 3: Kopie mit weiteren Modifikationen (Code der Kopie wird abgeändert, nicht nur Bezeichner, z. B. „Erweiterung“ einer Funktion)

Nach Test-Läufen mit kleineren Systemen wurden im Hauptteil des Experiments insgesamt acht Systeme ausgewählt, darunter vier in C und vier in Java. Die Größen der Systeme variierten von 30 KLOC bis über 200 KLOC, um eventuell Aussagen über das Auftreten von Klonen im Verhältnis zur Systemgröße treffen zu können.

Um die von den Teilnehmern des Experiments eingesen-