Code Refactoring

Zusammenfassung erstellt für das Seminar „Softwareentwicklung in der Wissenschaft“
im Sommersemester 2015 an der Universität Hamburg

Hauke Schulz

Betreuer: Julian Kunkel

20. Juli 2015

Zusammenfassung

Code Refactoring wird benutzt um die Wartung von Programmtexten zu minimieren und so die Produktivität zu steigern. Dabei kann die Produktivitätssteigerung anhand von Metriken quantifiziert werden.

Die untersuchten Studien offenbaren, dass das Code Refactoring durchweg zu positiven Ergebnissen führt, die mit Auto-tuning-Techniken sogar im rechenintensiven Hochleistungsbereich erreicht werden können.

1 Einführung

In der Wissenschaft gibt es meist keinen Entwicklungsplan für Programme, sondern die Entwicklung richtet sich nach der aktuellen Fragestellung. Zumeist wird die Entwicklung von Informatik-fremden Wissenschaftlern und nicht durch Softwareentwickler, die bestimmte Entwicklungsstrukturen kennen, vorgenommen, sodass Quelltexte schnell zu unübersichtlichen und schwer wartbaren Programmabschnitten verfallen. Durch Code Refactoring soll dem entgegengewirkt werden. Es bezeichnet die Optimierung eines Programmes ohne Veränderung seines Verhaltens. Dabei bezieht sich die Optimierung in erster Linie auf eine erhöhte Les- und Anpassbarkeit, nicht jedoch der Leistungsoptimierung.

Um die Wartbarkeit, aber auch aus Gründen anderer speziell in der Wissenschaft existierenden praktischen Aspekten, sollten Programme von Zeit zu Zeit neu strukturiert werden. Die Vorteile werden im Einzelnen in Kapitel 2 diskutiert. In Kapitel 3 wird der Frage nachgegangen, wie optimaler Code aussieht und anhand eines Beispiels illustriert. Dieses Beispiel wird in Kapitel 4 wieder aufgegriffen, um die gängigsten Tools zum Refactoring vorzustellen. Das Kapitel 5 widmet sich der Quantifizierung im Allgemeinen und im Hochleistungsrechnen im Speziellen.

2 Vorteile

Da unstrukturierter Code schwer zu warten ist, erhöht die gezielte Umstrukturierung die Lesbarkeit und Wartbarkeit. Mit der erneuten Beschäftigung des Quelltextes ist die Wahrscheinlichkeit erhöht, dass alte Fehler gefunden werden und das Verständnis über die Programmstrukturen klarer wird. Da zum Code Refactoring auch anschließende Tests gehören und vieles mit Tools (siehe 4) erledigt werden kann, ist das Einprogrammieren von neuen Fehlern minimiert. Doch nicht nur der einzelne Wissenschaftler profitiert von der Umstrukturierung, sondern die gesamte Wissenschaftsgemeinde. Strukturierte Programme werden eher bereitgestellt und können von anderen Mitgliedern schnell nachvollzogen werden, wodurch eine Produktivitätssteigerung der Wissenschaft wahrscheinlich ist. Nicht zuletzt lässt sich so das Interesse am jeweiligen Projekt steigern und neue wissenschaftliche Fragestellungen einfacher in den alten Code implementieren. Der ständige Wechsel von Fragestellungen und Neuorientierungen machen die Programmentwicklung in der Wissenschaft höchst dynamisch und somit auch schwerer organisierbar als in anderen Bereichen der Informatik. Hier sei auf die agile Softwareentwicklung verwiesen, die durch wechselseitiges Programmieren und Testen, eine bessere Vorangehensweise wäre. So ist das Refactoring auch fester Bestandteil dieses Entwicklungstyps. Neben der humanen Seite, bringt uns das Refactoring auch auf der technischen Seite einige Vorteile, die zu hohem Zeitersparnis führen. So kann bei richtiger Umstrukturierung, der allgemeinste Zustand des Programmes hergestellt werden, der je nach Architektur, Kompiler und weiteren systembedingten Randwerten, mit bestimmten Parametern oder Tools spezialisiert werden kann.

3 Optimaler Code

Wie sieht nun optimaler Code aus? In Tabelle 1 ist ein simples Beispiel gezeigt, welches die intuitiven Optimierungen veranschaulicht. Neben diesen intuitiven Änderungen, wie Einrückungen, Großschreibung von Parametern und Auslagerung von Berechnungen in bezeichnende Funktionen existieren auch komplexere Richtlinien, zu denen unter anderem die SOLID-Prinzipien gehören, welche im folgenden näher erläutert werden sollen. Dabei sei darauf hingewiesen, dass diese für die objektorientierte Programmierung entwickelt wurden, die unter den Autoren bevorzugt Verwendung findet.

SOLID ist ein Akronym für fünf Prinzipien die von Michael Feathers in den frühen 2000ern eingeführt wurden.

Ursprünglicher Quelltext

```
program main
dimension c(4,4,4)
n=4
DO i=1,n
DO j=1,n
DO k=1,n
c(i,j,k) = i+j+k
c(i,j,k) = c(i,j,k)/3
END DO
END DO
END DO
print*,c
end program
```

Umstrukturierter Quelltext

```
program main
 implicit none
 integer :: i,j,k,MAX=4
 integer :: c(MAX,MAX,MAX)
 DO i=1,MAX
  DO j=1,MAX
   DO k=1,MAX
    call mean(c,i,j,k,MAX)
   END DO
  END DO
 END DO
 print*,c
contains
 subroutine mean(c,i,j,k,MAX)
  implicit none
  integer :: i,j,k,MAX
  integer :: c(MAX,MAX,MAX)
  c(i,j,k) = (i+j+k)/3
 end subroutine mean
end program main
```

Tabelle 1: Exemplarischer Vergleich zwischen ursprünglichem Quelltext und jenem nach dem Refactoring

Single responsibility beschreibt den Grundsatz, dass jede Klasse nur eine einzige Verantwortung haben soll

Open-closed werden Software-Einheiten genannt, die offen für Erweiterungen sind, ihr Verhalten jedoch bestehen bleibt. Beispiele für Software-Einheiten sind Module und Klassen.

Liskov substitution besteht, wenn ein Programm, das Objekte einer Basisklasse verwendet, auch mit den Objekten der davon abgeleiteten Klasse korrekt funktioniert. Hier sei folgendes Beispiel angeführt: Es gäbe eine Klasse "Grafisches Element"mit der Funktion *zeichne*. Sei nun *Ellipse* eine Unterklasse von *Grafisches Element* mit den Methoden *Skaliere X* und *Skaliere Y*. Eine von der *Ellipse* abgeleitete Klasse könnte der *Kreis* sein, welcher alle Methoden erben würde. Würde man nun die Methode *Skaliere X* verwenden, so würde dies den Kreis zu einer Ellipse deformieren und so die Liskov substitution verletzen. Siehe hierzu auch Grafik 1

Interface segregation soll gewähren, dass Interfaces überschaubar sind und nicht zu komplex werden. Die Komplexität sollte an den Benutzer angepasst sein.

Dependency inversion formuliert das Prinzip, dass Abhängigkeiten immer von konkreten Modulen niedriger Ebenen zu abstrakten Modulen höherer Ebenen gerichtet sind

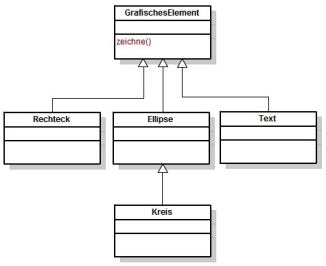

Abbildung 1: Beispiel für das Liskovsche Substitutionsprinzip (Quelle: www.wikipedia.de)

4 Tools

Wie wird nun das Refactoring im Einzelnen durchgeführt? Die Schritte, die unabhängig von der Methode durchgeführt werden müssen, sind nach [Mens and Tourwé, 2004] wie folgt :

1. Identifikation von Stellen, die Refactoring benötigen
2. Entscheidung über die Art des Refactorings
3. Durchführung von Verhaltens- und Unit-Tests
4. Durchführung des Refactorings
5. Durchführung von Qualitätstests
6. Wartung von möglicherweise abhängigen Programmen und Produkten (Dokumentation, Voraussetzungen,...)

Beim manuellen Refactoring müssen diese Schritte jeweils von Hand vorgenommen werden, wodurch bei größerem Quelltext, wie er zum Beispiel in den Klimawissenschaften existiert, eine hohe zeitliche Belastung herrscht. Außerdem ist diese Methode durch die menschliche Komponente sehr fehleranfällig. Um diese Komponente zu verkleinern und die Wiederholung von Schritten zu minimieren, gibt es semi-automatische Tools, die die Durchführung des Refactorings übernehmen und aufgrund der implementierten und getesteten Refactoring-Mechanismen die Durchführung von Verhaltenstest überflüssig machen, da die Methoden bereits auf Konsistenz geprüft wurden. Voll-automatische Tools hingegen übernehmen auch die ersten beiden Schritte der Refactoring-Kette. Im Gegenteil zu semi-automatischen Tools, ist bei ihnen die Gefahr, dass zu viele Änderungen vorgenommen werden und der Code komplexer wird als zuvor. Menschliches Wissen ist erforderlich um zum Beispiel beschreibende Modulnamen zu erzeugen.

In der Praxis kommen diese Tools hauptsächlich in Spezialgebieten, wie der Compiler-optimierung, zum Einsatz. Für alle anderen Bereiche wird auf semi-automatische Tools gesetzt. Eines der bekanntesten und verbreitetsten ist neben CamFort, SPAG und dem Clone analyst tool Photran, welches die meisten Refactoring-Methoden beherrscht und als Erweiterung für die Entwicklungsumgebung Eclipse installiert werden kann. Neben dem Entfernen von veralteten Sprachfragmenten wie z.B. der GOTO-Funktion, der Ausgliederung von Codezeilen in Prozeduren und Subroutinen, beherrscht es auch Fähigkeiten zur Leistungsoptimierung, wie das Loop-Unrolling. Das eingangs erwähnte Beispiel wurde mithilfe von Photran erstellt und ist in Abbildung 2 dargestellt. Schön ist vorallem, dass vor dem Übernehmen der Veränderungen, die entsprechenden Unterschiede hervorgehoben werden.

Für einen kompletten Überlick über die Funktionen sei an dieser Stelle auf die Doku-

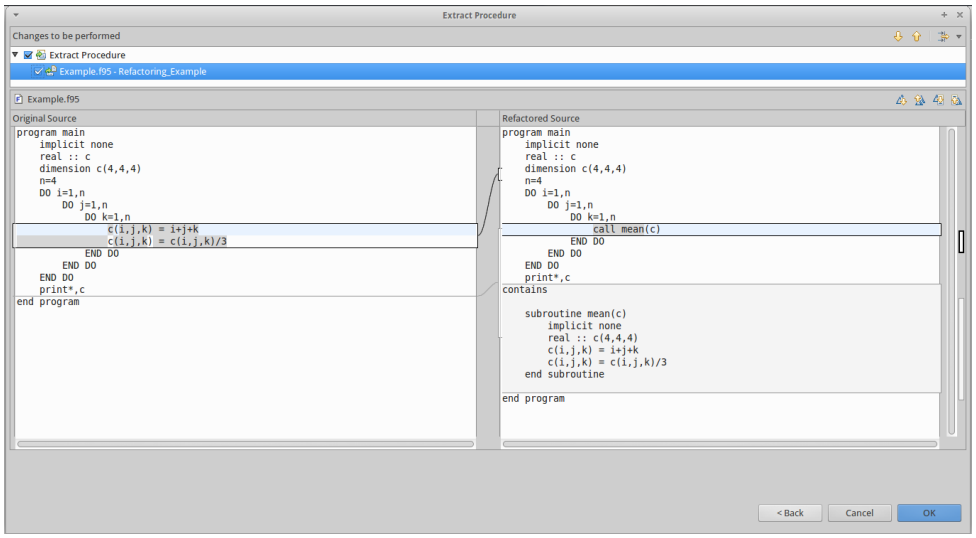

Abbildung 2: Vorschaufenster des Refactoring-Tools Photran

mentation von Photran verwiesen. Es sollte jedoch beachtet werden, dass bei dem hier gezeigten Beispiel, das Photran Werkzeug die ausgewählte Berechnung des Mittelwertes zwar in eine Funktion mit beschreibenden Namen auslagert, doch weiterhin limitiert auf drei Werte ist. Ein menschlicher Programmierer könnte und sollte im Rahmen des Refaktorings diese Funktion dynamischer machen und auch für andere Anzahlen von Werten funktionsfähig erhalten, ganz nach den SOLID-Prinzipien. Diese Intelligenz besitzen die hier erwähnten Tools noch nicht.

5 Quantifizierung

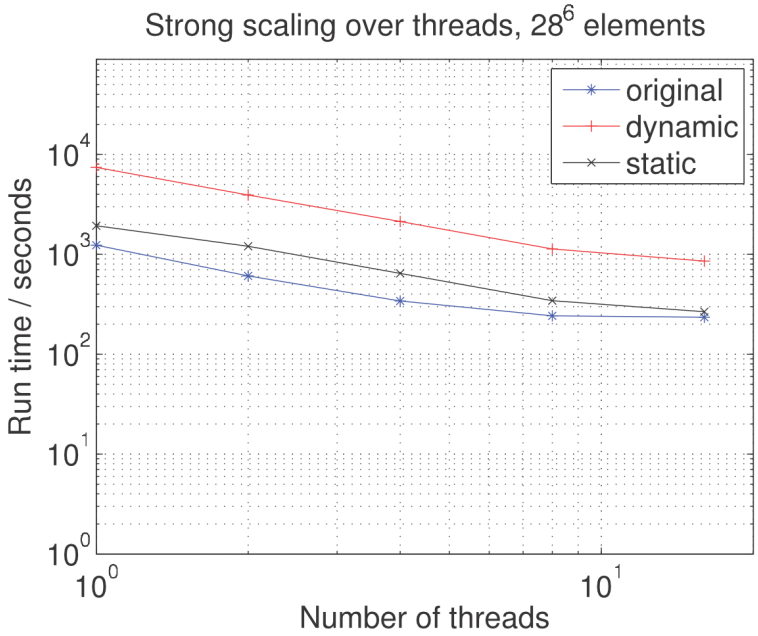

Abbildung 3: Abhängigkeit der Ausführungszeit von der Struktur des Programmcodes und der Anzahl der Threads. Abbildung aus [Kallen et al., 2014]

Um eine quantitative Aussage über das Ergebnis des Refactoring zu machen, gibt es unterschiedlichste Metriken, von einer einfachen Zählung der Codezeilen vor und nach dem Refactoring (Lines of Code) bis hin zur Kohäsion, die ein Maß für die Kapselung einer Methode ist (Lack of Cohesion in Methods). Dabei lässt sich bereits erkennen, dass auch diese Metriken durchaus kritisch zu betrachten sind. In der Arbeit von [Kallen et al., 2014] wird die Reduktion der Zeilenanzahl für ein erfolgreiches Refactoring bezeichnet, während es durchaus auch anders vorstellbar ist. Nichtsdestotrotz hat [Kallen et al., 2014] mit seiner Studie gezeigt, dass beträchtliche Vereinfachungen nach Maßgabe der Metriken erreicht werden können. Dies ist natürlich abhängig vom Anfangszustand des Codes und von dem Problem. Wobei letzteres, in dieser Studie ein iterativer Löser für komplexe, lineare Partialgleichungen in 2D, ein in den Naturwissenschaften durchaus häufig anzutreffender Fall sein dürfte. Für eine umfangreiche Beschreibung der Metriken sei auf [Chidamber and Kemerer, 1994] verwiesen.
Quantitativ führte die Umstrukturierung jedoch zu einer leichten Leistungseinbuße wie aus Abbildung 3 ersichtlich ist. Erinnern wir uns an die anfangs erwähnte Bedingung, dass beim Refactoring das Verhalten des Programmes erhalten bleiben soll, so verletzt die

Laufzeitveränderung diese. Je nach Software müssen neben der Funktion des Programms auch

- Laufzeit,
- Stromverbrauch,
- Speicherbedarf,
- Sicherheit

berücksichtigt werden. Allgemein muss daher abgewägt werden für welchen Bereich optimiert werden soll. Würde man zusätzlich die Kosten berücksichtigen, erhält man folgende (unvollständige) Bilanzgleichung:

$$\begin{aligned} \text{Kosten} &= \text{Kosten}_{\text{Entwicklung}} + \text{Kosten}_{\text{Wartung}} + \text{Kosten}_{\text{Betrieb}} + \dots \\ \text{wobei} \qquad \text{Kosten}_{\text{Wartung}} &= \text{Kosten}_{\text{Verständnis}} + \text{Kosten}_{\text{Testen}} + \text{Kosten}_{\text{Implementation}} \end{aligned} \tag{1}$$

Gesetz den Fall, dass die Kosten für die Wartung eines unübersichtlichen Programmes geringer ausfällt, als die Betriebskosten bei übersichtlichem Code, könnte es so schnell bei einem unleserlichen Code bleiben.

6 Refactoring im Hochleistungsrechnen

Gerade im Hochleistungsrechnen sind Quelltexte durch die Anpassung an bestimmte Architekturen höchst unflexibel und unleserlich. Ein Refactoring wird daher bei jedem Wechsel der Rechnerarchitektur notwendig. Aus dem angepassten Code muss ein allgemeiner Code entstehen ohne maschinenspezifische loop-unrollings, loop-tilings oder dergleichen. Diese allgemein lauffähigen Programme sind jedoch laufzeitkritisch und müssen daher erneut auf die neue Maschine angepasst werden. Siehe dazu die mittleren Balken in Grafik 4. [Wang et al., 2015] zeigt, dass es mithilfe von Auto-Tuning-Techniken möglich ist das Refactoring des alten Codes durchzuführen ohne anschließend Veränderungen an diesem vornehmen zu müssen und dennoch die Laufzeit auf ein Minimum zu beschränken (siehe Abb. 4).

Hierzu werden dem Code pragma-Anweisungen hinzugefügt, die einem anmerkungsbasierten Codegenerator wie HMPPCG dienen, bestimme Optimierungen wie zum Beispiel das Loop-Unrolling zu übernehmen (aus [CAPS, 2008]):

```
#pragma hmpccg unroll i:4, contiguous
for(i = 0 ; i < n ; i++ ) {
  v1[i] = alpha * v2[i] + v1[i];
}
```

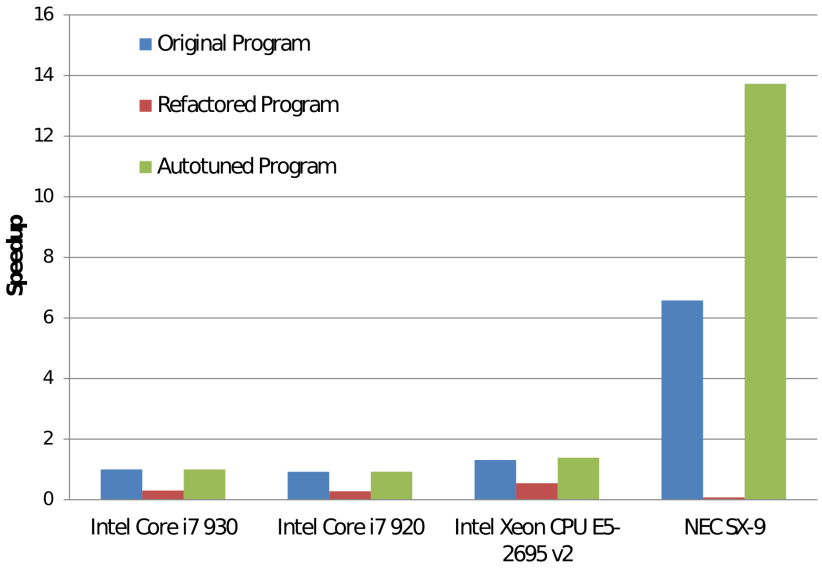

Abbildung 4: Vergleich zwischen manuell optimiertem Code, umstrukturiertem Code und auto-tuned Code. Abbildung nach [Hartono et al., 2009]

Der Codegenerator generiert daraus:

```
for (i_1 = 0, __hmppcg_end = (n / 4) -1; i_1 <= __hmppcg_end; i_1 += 1)
{
v1[4 * i_1] = (alpha * (v2[4 * i_1])) + (v1[4 * i_1]);
v1[(4 * i_1) + 1] = (alpha * (v2[(4 * i_1) + 1])) + (v1[(4 * i_1) + 1]);
v1[(4 * i_1) + 2] = (alpha * (v2[(4 * i_1) + 2])) + (v1[(4 * i_1) + 2]);
v1[(4 * i_1) + 3] = (alpha * (v2[(4 * i_1) + 3])) + (v1[(4 * i_1) + 3]);
}
```

Diese Anmerkungen bewahren die Übersichtlichkeit und können von Auto-Tuning-Tools wie Orio benutzt werden, um die optimalen Optionen zu finden. Dabei werden die pragma-Anweisungen vom Tuning-Tool um mögliche Optionen erweitert (im oberen Beispiel der unroll-Faktor 4 und die contiguous-Option). Jede Optimierung hat dabei einen gültigen Wertebereich, sodass im oberen Beispiel auch die Faktoren 1 und 2 und die Optionen split und changestep verwendet werden können. Das Auto-Tuning-Tool setzt dabei alle gültigen Optionen ein, generiert den optimierten Code und führt eine Leistungsanalyse durch. Auf diese Weise wird empirisch der schnellste Code generiert, der anschließend mit den üblichen Kompilern ausführbar gemacht werden kann. Die allgemeine Version bleibt bei diesem Verfahren immer erhalten, sodass Änderungen am Code oder an der Architektur leicht zu bewerkstelligen sind, da die Optimierung vom Auto-Tuning-Tool übernommen wird. Um bei kleinen Code-Änderungen nicht alle Optimierungsmöglichkeiten erneut durchzugehen, können auch andere Suchalgorithmen benutzt werden, die bereits über

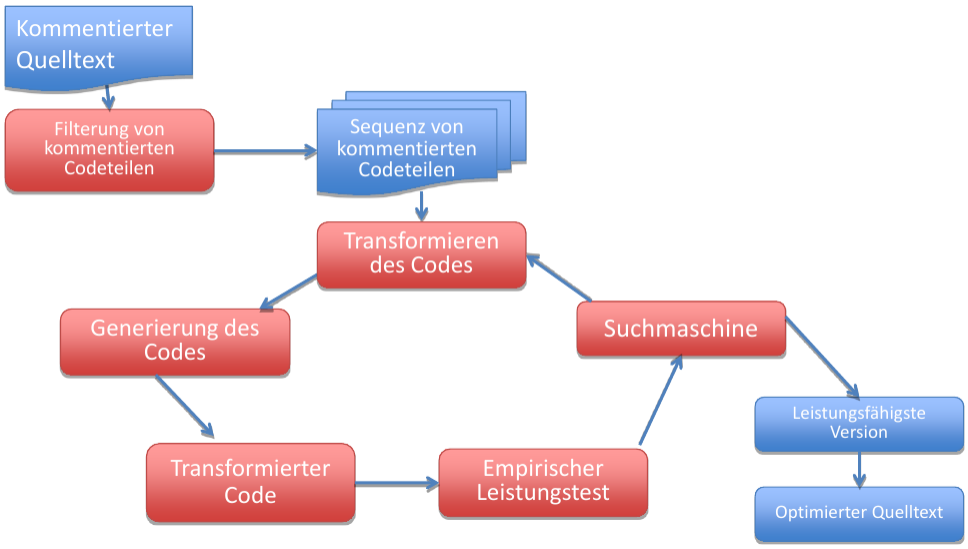

Abbildung 5: Ablaufdiagramm des Auto-Tuning-Prozesses: Der mit pragma-Anweisungen kommentierte Quelltext wird nach eben diesen und den dazugehörigen Abschnitten gefiltert. Für jeden dieser einzelnen Abschnitte wird in einem iterativen Prozess der bestmögliche Optimierungsparameter gesucht. Nach dem Tuning aller Parameter wird ein optimierter Quelltext ausgegeben, der mit handelsüblichen Kompilern lauffähig gemacht werden kann. Abbildung nach [Hartono et al., 2009]

das Vorwissen der letzten Leistungsanalyse besitzen und so gezielter suchen können. Das genaue Verfahren ist in Abbildung 5 abgebildet.

7 Zusammenfassung

Zusammenfassend lässt sich sagen, dass sich mit den zur Verfügung stehenden Hilfsmitteln ein Refactoring in den meisten Fällen lohnt und selbst bei laufzeitkritischen Anwendungen für das Hochleistungsrechnen keine Kompromisse gemacht werden müssen. Wie komplex dieser Refactoringprozess ist, hängt dabei zum einen von den Codeeigenschaften ab, zum anderen aber auch von den verwendeten Hilfswerkzeugen. Ausgehend von einer weiteren Entwicklung der Hilfswerkzeuge wird in zukünftigen Jahren die Arbeit des Refactorings weiter vereinfacht werden und die Kinderkrankheiten die noch in einigen Werkzeugen stecken ausgemerzt sein. Abschließend sei daran erinnert, dass das Refactoring auch eine große Chance ist, um Programme über die eigenen Grenzen hinweg zu verbreiten und somit Refactoring in jedem Falle in Erwägung gezogen werden sollte.

Literatur

[Barnes and Jones, 2011] Barnes, N. and Jones, D. (2011). Clear Climate Code: Rewriting Legacy Science Software for Clarity. *IEEE Software*, 28(6):36–42.

[CAPS, 2008] CAPS (2008). *HMPP Codelet Generator Directives*.

[Chidamber and Kemerer, 1994] Chidamber, S. R. and Kemerer, C. F. (1994). A metrics suite for object oriented design. *Software Engineering, IEEE Transactions on*, 20(6):476–493.

[Hartono et al., 2009] Hartono, A., Norris, B., and Sadayappan, P. (2009). Annotation-based empirical performance tuning using orio. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–11. IEEE.

[Kallen et al., 2014] Kallen, M., Holmgren, S., and Hvannberg, E. (2014). Impact of Code Refactoring Using Object-Oriented Methodology on a Scientific Computing Application. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 125–134.

[Kimura et al., 2012] Kimura, S., Higo, Y., Igaki, H., and Kusumoto, S. (2012). Move code refactoring with dynamic analysis. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 575–578.

[Mens and Tourwé, 2004] Mens, T. and Tourwé, T. (2004). A survey of software refactoring. *Software Engineering, IEEE Transactions on*, 30(2):126–139.

[Neto et al., 2015] Neto, B. F. d. S., Ribeiro, M., Torres da Silva, V., Braga, C., Pereira de Lucena, C. J., and Costa, E. d. B. (2015). Autorefactoring: A platform to build refactoring agents. *Expert Systems with Application*, 42:1652–1664.

[Sokol et al., 2013] Sokol, F. Z., Aniche, M. F., and Gerosa, M. A. (2013). Does the Act of Refactoring Really Make Code Simpler? A Preliminary Study. In *4th Brazilian Workshop on Agile Methods*.

[Wang et al., 2015] Wang, C., Hirasawa, S., and Takizawa, H. (2015). Combining code refactoring and auto-tuning to improve performance portability of high-performance computing applications. *Computation Tools*.