Aktuelle Themen der Softwaretechnologie
M-ATSWT-S

Prof. Dr. Christoph Reichenbach
Software Engineering & Programming Languages

WS 2013/2014

1 Struktur des Seminars

1.1 Sprache

Dieses Seminar wurde ursprünglich sowohl auf Deutsch als auch auf Englisch angeboten. Da unter den angemeldeten Teilnehmern diejenigen, die Englisch verstehen, in der Minderheit sind, wird das Seminar auf Deutsch durchgeführt. Ausarbeitungen und Feedback (s.u.) dürfen aber auf Englisch abgegeben werden.

1.2 Erfolgreiche Teilnahme

Um an dem Seminar erfolgreich teilnehmen zu können, müssen Sie:

- Eine 4,0 oder besser sowohl auf Ihren Vortrag als auch auf Ihre Ausarbeitung (individuell) erreichen
- An so vielen Seminarvorträgen wie möglich teilnehmen
- Zu mindestens $\frac{2}{3}$ der Vorträge Feedback senden.

Ihre Note wird wie folgt berechnet:

- Ausarbeitung: $\frac{2}{3}$
- Vortrag: $\frac{1}{3}$

1.3 Vortragsblöcke

Seminarvorträge sind in Blöcken alloziert, die alle zwei Wochen stattfinden. Im ersten Block wird:

1. die Struktur des Seminars vorgestellt,
2. allgemeines Hintergrundmaterial zum Seminar vorgestellt,
3. ein kurzer Überblick über die angegebenen Themen gegeben, und
4. ein Beispielvortrag gehalten.

Sie haben bis zum Ende der ersten (Arbeits)woche, um Ihre gewünschten Themen auszusuchen, und diese dann in einer prioritisierten Liste (in absteigender Reihenfolge) an den Veranstalter zu senden. Ein fairer Algorithmus wird Ihnen dann ein Thema, möglichst aus Ihrer Liste, zuweisen; für beliebte Themen kann dies jedoch nicht garantiert werden, hier entscheidet der Pseudozufallszahlengenerator.

1.4 Zeitliche Abfolge im Semester

Die zugewiesenen Themen werden in der gleichen Reihenfolge präsentiert, in der sie auch in diesem Dokument aufgeführt sind, wobei die Vortragszeitpunkte so spät wie möglich gesetzt werden. Die Vortagstage werden dabei zusammen mit den Vorträgen zugewiesen. Vortragenden können untereinander Tage/Uhrzeiten innerhalb des gleichen Themengebietes tauschen, aber zwischen Themengebieten nur mit Zustimmung des Veranstalters. In jedem Fall muß der Veranstalter von einem solchen Tausch rechtzeitig zur Vortragstauschdeadline in Kenntnis gesetzt werden.

Fall Sie einen Vortrag nicht halten oder Ihre Ausarbeitung nicht rechtzeitig einreichen, können Sie das Seminar nicht bestehen, es sei denn, Sie können triftige Gründe nachweisen (nach Ermessen des Veranstalters), die es Ihnen unerwartet nicht gestattet haben, Ihre Seminarbeiträge rechtzeitig fertigzustellen.

2 Deadlines

Themenwahl:	19. Oktober 2013
Vortragstauschdeadline:	Eine Woche vor dem frühesten betroffenen Vortrag
Deadline für Ausarbeitungsentwurf:	Eine Woche vor dem Vortrag
Deadline für Ausarbeitung:	10 Tage nach dem Vortrag
Deadline zum Einsenden der Folien:	Zwei Tage vor dem Vortrag
Deadline zum Einsenden des Feedbacks:	Drei Tage nach dem betroffenen Vortrag

In diesem Zusammenhang wird ein Tag als ein 24h-Block inerpretiert. Wenn der Vortrag also auf 14:00 an einem Dienstag gelegt ist, müssen die Folien spätestens um 14:00 am vorherigen Sonntag eingereicht werden.

3 Ausarbeitung

Ihre Ausarbeitung muß dem Veranstalter spätestens zehn Tage nach dem Vortrag zugesendet werden; einen vollständigen Entwurf müssen Sie bereits eine Woche zuvor einsenden. Die Ausarbeitung muß folgenden Anforderungen genügen:

- Die Ausarbeitung muß das gewählte Thema erklären und seine Bedeutung im Kontext des Themengebietes (Abschnitt 7) erläutern.
- Die Ausarbeitung muß Quellen für jegliche Aussagen, die Sie treffen, zitieren, sofern Sie diese Aussagen nicht durch eigne Experimente belegt haben oder sie in typischen B.Sc.-Curricula abgedeckt werden und somit als Allgemeinwissen gelten können.

Falls die Ausarbeitung diese Anforderungen erfüllt, wird es nach den Richtlinien aus Abschnitt 5 bewertet.

Die Gesamtgröße der Ausarbeitung sollte bei ca. 15 Seiten liegen.

Entwurf Der Entwurf Ihrer Ausarbeitung sollte die oben angegebenen Anforderungen im Wesentlichen bereits erfüllen. Insbesondere sollte der Entwurf

- Strukturell vollständig sein (d.h., die schlußendlichen Inhalte der Ausarbeitung sollten klar ersichtlich sein, auch, wenn sie eventuell noch nicht in vollem Detail vorliegen)
- Klar ersichtlich machen, welche Komponenten (z.B. fehlende Beweise oder experimentelle Ergebnisse) bis zum Vortrag bzw. zur endgültigen Abgabe noch ausstehen.

4 Vortrag

Die Foliensätze müssen zwei Tagen vor dem Vortrag eingesendet werden, damit wir sicherstellen können, daß sie korrekt dargestellt werden können. Daher bitten wir Sie, die Folien im `pdf`- oder postscript-Format einzusenden. Sie können auch Ihren eigenen Laptop zur Präsentation verwenden; eventuell zum Konfigurieren des Systems notwendige Zeit wird Ihnen aber von der Präsentationszeit abgezogen. Dennoch ist es empfehlenswert, ein `pdf`-Backup zu arrangieren.

Beachten Sie den Abschnitt über Feedback (Abschnitt 6) für Hinweise dazu, welche Fragen Ihr Vortrag beantworten sollte. Wenn Sie sich insbesondere auf die Beantwortung von Fragen 1–6 ausrichten, sollten Sie die typischen Inhalte von Konferenzvorträgen abdecken. Falls Sie von diesen wesentlich abweichen sollten, ist es empfehlenswert, den Veranstalter im Vorfeld (mit genug Vorlauf, um die Folien zu überarbeiten) zu kontaktieren.

Zusätzlich zur Vorbereitung des Vortrags sollten Sie sich auch auf wahrscheinliche Fragen vorbereiten. Da Sie nicht Autor des ursprünglichen Papieres sind, wird von Ihnen nicht erwartet, daß Sie alle Fragen beantworten können, aber Sie sollten zeigen können, daß Sie ein gutes Verständnis des zugrundeliegenden Materials gewonnen haben.

Zielen Sie auf eine Länge von 30-45 Minuten (15-23 Folien), plus 10 Minuten für Fragen. Zum Vergleich: Typische Konferenzvorträge dauern 15-25 Minuten, plus 5 Minuten für Fragen.

Nach Ihrem Vortrag wird Ihnen evtl. Feedback der anderen Studenten und des Veranstalters gesendet. Dieses Feedback kann inhaltlich oder strukturell sein; es ist für Sie eine Gelegenheit, zu verstehen, wie gut Ihr Vortrag beim Publikum angekommen ist.

4.1 Vortragsvorbereitung

Üben Sie Ihren Vortrag. Es ist nicht unüblich, daß ein Vortrag nach den ersten 2–3 Durchläufen wesentliche Veränderungen erlebt, und bei dem ersten Vortrag vor einem Probepublikum noch weiter umstrukturiert wird. Wenn möglich, üben Sie daher auch vor Freunden.

Es wird nicht erwartet, daß die Qualität Ihres Vortrages auf dem Stand eines Konferenzvortrages ist (oft 10 oder mehr Iterationen), aber er sollte die wesentlichen Fragen (Abschnitt 6) beantworten und die Bewertungskriterien (Abschnitt 5) im Auge behalten.

5 Benotung

Ihre Note wird hauptsächlich durch fünf Kriterien bestimmt:

1. Wie *verständlich* ist Ihre Ausarbeitung/ihr Vortrag?
2. Wie *genau* ist Ihre Darstellung des Sachverhaltes?
3. Wie *vollständig* behandeln Sie die Fragestellungen aus Abschnitt 6 (außer Frage 9)?
4. Wie *breit* decken Sie Hintergrund und den state-of-the-art der für diese Arbeit relevanten Aspekte des Feldes ab?
5. Bieten Sie Einsichten, die über das in dem zugrundeliegenden Papier angegebene Material und verwandte existierende Arbeiten hinausgehen?

Sie können eine Note von 2,0 erreichen, wenn Sie Kriterien 1–4 gut abdecken. Für eine bessere Note sollten Sie zusätzlich eigene Analysen, Experimente, Beweise o.ä. gemäß Kriterium 5 einbringen.

Die Kriterien werden in Vortrag und Ausarbeitung unterschiedlich gewichtet:

- Im *Vortrag* wird primär nach Kriterien 1–3 bewertet; es wird erwartet, daß Kriterium 4 angemessen angerissen aber nicht im gleichen Detail wie in der Ausarbeitung behandelt wird. Arbeiten zu Kriterium 5 sollten in angemessenem Umfang beschrieben werden, müssen aber nicht als solche

ausgezeichnet werden. Wenn Sie z.B. einen neuen Beweis entwickelt haben, ist es ausreichend, auf die Ausarbeitung zu verweisen.

- Die *Ausarbeitung* wird zunächst danach bewertet, ob Kriterien 1 und 2 hinreichend für eine Note von 3,0 oder besser erfüllt sind. Falls ja, werden Kriterien 1–4 ungefähr gleich gewichtet. Ansonsten wird das Gewicht von Kriterien 1–2 erhöht.

Anders ausgedrückt: *Verständlichkeit* und *Genauigkeit* sind die wichtigsten Faktoren.

6 Feedback

Nach jedem Vortrag müssen Sie innerhalb von 3 Tagen in einer Mail an den Veranstalter Feedback senden. Dieses Feedback kann in Deutsch oder Englisch gesendet werden; es kann anonymisiert an den Vortragenden weitergeleitet werden, wenn Sie sich nicht explizit dagegen aussprechen.

Dieses Feedback muß folgende Fragen beantworten:

1. Welches grundlegende Problem wird in dem Vortrag behandelt?
2. Warum ist dieses Problem wichtig?
3. Warum haben frühere Arbeiten das Problem nicht lösen können?
4. Welche neuen Ideen und Beiträge sind ersichtlich?
5. Wie wurden die vorgetragenen Ideen und Beiträge ausgewertet?
6. Welche Resultate sind aus der Auswertung ersichtlich?
7. Was ist der stärkste Punkt der vorgestellten Arbeit?
8. Was ist der schwächste Punkt der vorgestellten Arbeit?
9. Wie würden Sie den vorgestellten Ansaz verbessern?

Versuchen Sie, die Fragen nach Ihrer eigenen Ansicht und Interpretierung der Arbeit zu beantworten; Ihre Antwort darf Aussagen aus dem Vortrag durchaus widersprechen. Falls Sie während des Vortrages feststellen, daß eine dieser Fragen nicht beantwortet wurde, fragen Sie den Vortragenden.

Ihr Feedback wird nicht bewertet, aber Sie müssen zu mindestens $\frac{2}{3}$ der Vorträge Feedback einreichen, um die Veranstaltung zu bestehen.

7 Themengebiete

In diesem Seminar werden drei Themengebiete angeboten:

- Automatisches Refactoring, Abschnitt 7.1
- Generative und Aspektorientierte Programmierung, Abschnitt 7.2
- Parallele Programmiersprachen, Abschnitt 7.3

Jedes Themengebiet hat mehrere Themen, die in eine der folgenden Kategorien fallen:

- *Thema mit einem Papier*: Die meisten Themen fallen in diese Kategorie. Hier erhalten Sie eine Referenz auf ein Forschungspapier, das Sie direkt als Ausgangsbasis für Ihren Vortrag nehmen sollen.

- Breites Thema (**B**): Hier wird Ihnen ein breiteres Thema gegeben, oft nur mit Buchverweisen oder mehreren Forschungspapieren. Sie müssen teilweise selbst relevante Literatur identifizieren.

  Da Sie in dieser Kategorie implizit eigene Beiträge bringen müssen, ist das notwendige (!) Kriterium zur Überschreitung der Note 2,0 automatisch erfüllt.

Breite Themen können auch von zwei Studierenden gemeinsam in einer Ausarbeitung und einem Vortrag bearbeitet werden. Die Bewertungskriterien werden entsprechend höher angesetzt.

7.1 Automatisches Refactoring

Refactoring ist die Transformation der Struktur von Programmen, ohne deren (beobachtbares) Verhalten zu ändern. Automatisches Refactoring ist eine maschinengestützte Form des Refactoring, wie z.B. im SmallTalk Refactoring Browser, den eingebauten Refactorings in Eclipse, oder dem HaRe Refactoringsystem für Haskell.

Automatisches Refactoring folgt traditionell folgendem Schema:

1. Zunächst erfolgt *Benutzerauswahl* des Refactorings und zu verändernden Programmstücks
2. Danach überprüft das System eine Refactoring-spezifische *Vorbedingung* über die Auswahl des Benutzers. Die Idee dabei ist, daß die Vorbedingung überprüft, ob eine Programmtransformation das Programmverhalten erhalten wird.
3. Falls die Vorbedingung erfolg signalisiert, *transformiert* das Refactoringsystem das Programm entsprechend der Benutzerauswahl.

Literatur (Bibliothek oder Bürobestand)

- "Refactoring: Improving the Design of Existing Code" (Martin Fowler et al.; 1999, Addison-Wesley)
- "Refactoring to Patterns" (Joshua Kerievsky, 2004; Addison-Wesley)
- "Compilers: Principles, Techniques, and Tools" (Aho, Lam, Sethi, Ullman; 2nd. Ed., 2006, Addison-Wesley)

7.1.1 A comparative study of manual and automated refactorings

Stas Negara, Nicholas Chen, Mohsen Vakilian, Ralph E. Johnson, und Danny Dig, "A comparative study of manual and automated refactorings"[1], PLDI 2012 (Peking)

Despite the enormous success that manual and automated refactoring has enjoyed during the last decade, we know little about the practice of refactoring. Understanding the refactoring practice is important for developers, refactoring tool builders, and researchers. Many previous approaches to study refactorings are based on comparing code snapshots, which is imprecise, incomplete, and does not allow answering research questions that involve time or compare manual and automated refactoring.

We present the first extended empirical study that considers both manual and automated refactoring. This study is enabled by our algorithm, which infers refactorings from continuous changes. We implemented and applied this algorithm to the code evolution data collected from 23 developers working in their natural environment for 1,520 hours. Using a corpus of 5,371 refactorings, we reveal several new facts about manual and automated refactorings. For example, more than half of the refactorings were performed manually. The popularity of automated and manual refactorings differs. More than one third of the refactorings performed by developers are clustered in time. On average, 30% of the performed refactorings do not reach the Version Control System.

[1] `dig.cs.illinois.edu/papers/ECOOP13-NegaraETAL-Inferring.pdf`

7.1.2 Use, Disuse, and Misuse of Automated Refactorings

Mohsen Vakilian, Nicholas Chen, Stas Negara, Balaji Ambresh Rajkumar, Brian, P. Bailey, und Ralph E. Johnson, "Use, Disuse, and Misuse of Automated Refactorings"[2], ICSE 2012 (Zurich)

Though refactoring tools have been available for more than a decade, research has shown that programmers underutilize such tools. However, little is known about why programmers do not take advantage of these tools. We have conducted a field study on programmers in their natural settings working on their code. As a result, we collected a set of interaction data from about 1268 hours of programming using our minimally intrusive data collectors. Our quantitative data show that programmers prefer lightweight methods of invoking refactorings, usually perform small changes using the refactoring tool, proceed with an automated refactoring even when it may change the behavior of the program, and rarely preview the automated refactorings. We also interviewed nine of our participants to provide deeper insight about the patterns that we observed in the behavioral data. We found that programmers use predictable automated refactorings even if they have rare bugs or change the behavior of the program. This paper reports some of the factors that affect the use of automated refactorings such as invocation method, awareness, naming, trust, and predictability and the major mismatches between programmers' expectations and automated refactorings. The results of this work contribute to producing more effective tools for refactoring complex software.

[2] `http://dl.acm.org/citation.cfm?id=2337251` (Aktuelle ACM-Papiere sollten aus dem Uni-Netzwerk heraus kostenlos lesbar sein))

7.1.3 Refactoring with Synthesis

Veselin Raychev, Max Schäfer, Manu Sridharan, Martin Vechev, "Refactoring with Synthesis"[3], OOPSLA 2013 (Indianapolis)

Refactoring has become an integral part of modern software development, with wide support in popular integrated development environments (IDEs). Modern IDEs provide a fixed set of supported refactorings, listed in a refactoring menu. But with IDEs supporting more and more refactorings, it is becoming increasingly difficult for programmers to discover and memorize all their names and meanings. Also, since the set of refactorings is hard-coded, if a programmer wants to achieve a slightly different code transformation, she has to either apply a (possibly non-obvious) sequence of several built-in refactorings, or just perform the transformation by hand.

We propose a novel approach to refactoring, based on synthesis from examples, which addresses these limitations. With our system, the programmer need not worry how to invoke individual refactorings or the order in which to apply them. Instead, a transformation is achieved via three simple steps: the programmer first indicates the start of a code refactoring phase; then she performs some of the desired code changes manually; and finally, she asks the tool to complete the refactoring.

Our system completes the refactoring by first extracting the difference between the starting program and the modified version, and then synthesizing a sequence of refactorings that achieves (at least) the desired changes. To enable scalable synthesis, we introduce local refactorings, which allow for first discovering a refactoring sequence on small program fragments and then extrapolating it to a full refactoring sequence.

We implemented our approach as an Eclipse plug-in, with an architecture that is easily extendable with new refactorings. The experimental results are encouraging: with only minimal user input, the synthesizer was able to quickly discover complex refactoring sequences for several challenging realistic examples.

[3] `http://researcher.ibm.com/researcher/files/us-msridhar/OOPSLA13Refactoring.pdf`

7.1.4 Program Metamorphosis

Christoph Reichenbach, Devin Coughlin, und Amer Diwan, "Program Metamorphosis"[4], ECOOP 2008 (Genova)

Modern development environments support refactoring by providing atomically behaviour-preserving transformations. While useful, these transformations are limited in three ways: (i) atomicity forces transformations to be complex and opaque, (ii) the behaviour preservation requirement disallows deliberate behaviour evolution, and (iii) atomicity limits code reuse opportunities for refactoring implementers. We present 'program metamorphosis', a novel approach for program evolution and refactoring that addresses the above limitations by breaking refactorings into smaller steps that need not preserve behaviour individually. Instead, we ensure that sequences of transformations preserve be- haviour together, and simultaneously permit selective behavioural change. To evaluate program metamorphosis, we have implemented a prototype plugin for Eclipse. Our analysis and experiments show that (1) our plugin provides correctness guarantees on par with those of Eclipse's own refactorings, (2) both our plugin and our approach address the aforementioned limitations, and (3) our approach fully subsumes traditional refactoring.

[4] `http://creichen.net/papers/pm.pdf`

7.1.5 "Crossing the Gap from Imperative to Functional Programming through Refactoring"

Alex Gyori, Lyle Franklin, Danny Dig, and Jan Lahoda, "Crossing the Gap from Imperative to Functional Programming through Refactoring"[5], ESEC/FSE 2013 (St. Petersburg)

Java 8 introduces two functional features: lambda expressions and functional operations like map or filter that apply a lambda expression over the elements of a Collection. Refactoring existing code to use these new features enables explicit but unobtrusive parallelism and makes the code more succinct. However, refactoring is tedious: it requires changing many lines of code. It is also error-prone: the programmer must reason about the control-, data-flow, and side-effects. Fortunately, refactorings can be automated.

We designed and implemented LambdaFicator, a tool which automates two refactorings. The first refactoring converts anonymous inner classes to lambda expressions. The second refactoring converts for loops that iterate over Collections to functional operations that use lambda expressions. Using 9 open-source projects, we have applied these two refactorings 1263 and 1709 times, respectively. The results show that LambdaFicator is useful: (i) it is widely applicable, (ii) it reduces the code bloat, (iii) it increases programmer productivity, and (iv) it is accurate.

[5] `http://refactoring.info/tools/LambdaFicator/` und `http://dig.cs.illinois.edu/papers/lambdaRefactoring.pdf` für das Papier

7.2 Generative und Aspektorientierte Programmierung

Das vielleicht mächtigste Konzept in Mathematik und Softwareenwicklung ist das Prinzip der *Abstraktion*. Die meisten Programmiersprachen bieten daher zahlreiche Abstraktionsmechanismen: Funktionen, Module, Signaturen, Funktoren, Überzetzungszeitparameter, Parametrischer Polymorphismus, Subtyppolymorphismus, Generics, Varianz generischer Parameter, und viele andere.

Programmiersprachendesigner tendieren allerdings dazu, diese Konzepte gezielt so zu entwerfen, daß sie möglichst viele Probleme gleichzeitig lösen, meist also insbesondere modular übersetzbar sind. Dies ist in viele Fällen bequem, da es Softwareentwicklern hilft, effektiv zu arbeiten (also insbesondere modular).

Es gibt aber viele Fälle, in denen die existierenden Mechanismen nicht ausreichen:

- Softwareentwickler müssen oft Abwägungen treffen, die nicht denen der Sprachentwickler entsprechen. Zum Beispiel könnte der Sprachentwickler sich für kleine Codegrößen bei der Wiederverwertung entschieden haben (wie man es im Subtyppolymorphismus oder bei Boxing oft sieht), während der Enwickler lieber die zusätzliche Performanz von spezialisiertem Code hätte— oder umgekehrt.

- Einige nützliche Abstraktionsformen passen nicht mit traditionellen Softwarebibliothekskonzepten zusammen, insbesondere solche, die aus einer Applikationsdomäne mit einer eigenen Sprache stammen. Beispiele hierfür sind reguläre Ausdrücke und kontextfreie Sprachen, die in eignenen Formalismen kompakt notiert werden können.

- Einige Abstraktionsformen durchbrechen die Abstraktionsschienen der Sprachen. Beispielsweise ist es in einigen Situationen nötig, bestimmte Operationen nachzuverfolgen, wenn diese sicherheits— oder performanzkritisch sind. Die meisten Programmiersprachen bieten aber keine einfache Möglichkeit, um Konzepte der Form ‘jedes Mal, wenn jemand diese Methode dieser Klasse ausführt, soll diese Operation aufgerufen werden’ umzusetzen.

Generative und Aspektorientierte Programmierung sind Techniken, um Programmierern zusätzliche Ausdruckskraft zu geben. Dies kann die obigen und andere Einschränkungen umgehen, indem entweder neuer Programmcode erzeugt oder existierender Programmcode abgefangen und transformiert wird, bevor er an den Übersetzer weitergereicht wird.

Literatur (größtenteils Bibliothek oder Bürobestand)

- “Generative Programming: Methods, Tools, and Applications” (Czarnecki, Eisenecker; 2000, Addison-Wesley)

- “Aspektorientierte Programmierung mit AspectJ 5: Einsteigen in AspectJ und AOP” (Oliver Böhm; 2005, dpunkt.Verlag)

- “Compilers: Principles, Techniques, and Tools” (Aho, Lam, Sethi, Ullman; 2nd. Ed., 2006, Addison-Wesley)

- “Software Product Lines: Practices and Patterns” (Clements, Northrop; 3rd Ed., 2001, Addision-Wesley)

7.2.1 Aspektorientierte Programmierung und AspectJ (B)

Dieses Thema ist eingeschränkt auch ohne Englischkenntnisse verfügbar; es handelt sich um ein *breites Thema*.

Erforschen Sie anhand der verfügbaren Literatur, von geeigneten Forschungspublikationen und eigenen Experimenten das Konzept der *aspektorientierten Programmierung*:

- Was ist Aspektorientierte Programmierung?
- Was ist AspectJ?
- Wie kann man AspectJ verwenden?
- Wie funktioniert AspectJ "unter der Haube"?

Spezifische Literatur:

- "Aspektorientierte Programmierung mit AspectJ 5: Einsteigen in AspectJ und AOP" (Oliver Böhm; 2005, dpunkt.Verlag)
- Kiczales, Gregor; John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin, "Aspect-Oriented Programming" (1997). Proceedings of the European Conference on Object-Oriented Programming, vol.1241. pp. 220–242[6].
- Kiczales, Hilsdale, Hugunin et al., "An overview of AspectJ"[7], ECOOP 2001

[6] `www.cs.ubc.ca/~gregor/papers/kiczales-ECOOP1997-AOP.pdf`

[7] `dl.acm.org/citation.cfm?id=680006` (Aktuelle ACM-Papiere sollten aus dem Uni-Netzwerk heraus kostenlos lesbar sein)

7.2.2 Execution Levels for Aspect-Oriented Programming

Eric Tanter, "Execution Levels for Aspect-Oriented Programming"[8], AOSD 2010 (Rennes and Saint-Malo)

In aspect-oriented programming languages, advice evaluation is usually considered as part of the base program evaluation. This is also the case for certain pointcuts, such as if pointcuts in AspectJ, or simply all pointcuts in higher-order aspect languages like AspectScheme. While viewing aspects as part of base level computation clearly distinguishes AOP from reflection, it also comes at a price: because aspects observe base level computation, evaluating pointcuts and advice at the base level can trigger infinite regression. To avoid these pitfalls, aspect languages propose ad-hoc mechanisms, which increase the complexity for programmers while being insufficient in many cases. After shedding light on the many facets of the issue, this paper proposes to clarify the situation by introducing levels of execution in the programming language, thereby allowing aspects to observe and run at specific, possibly different, levels. We adopt a defensive default that avoids infinite regression in all cases, and give advanced programmers the means to override this default using level shifting operators. We formalize the semantics of our proposal, and provide an implementation. This work recognizes that different aspects differ in their intended nature, and shows that structuring execution contexts helps tame the power of aspects and metaprogramming.

[8] `dl.acm.org/citation.cfm?id=1739236` (Aktuelle ACM-Papiere sollten aus dem Uni-Netzwerk heraus kostenlos lesbar sein)

7.2.3 Meta-Object Protocols (B)

Metaobjektprotokolle sind Mechanismen, mit denen die Erzeugung, Verbindung, und Verwendung von Objekten in Programmiersprachen manipuliert werden können.

Ihre Aufgabe in diesem Thema ist es, Metaobjektprotokolle mindestens zweier Programmiersprachen (z.B. Groovy und Common LISP) bezüglich Ihrer Ausdruckskraft und weiteren angemessenen Kriterien zu untersuchen.

Spezifische Literatur:

- "The Art of the Metaobject Protocol", (Kiczales, des Rivieres, Bobrow), Bürobestand
- Shigeru Chiba, "A Metaobject Protocol for C++"[9], OOPSLA 1995 (Austin, TX)

[9] `www.cis.uab.edu/courses/cs493/spring2004/chiba95metaobject.pdf`

7.2.4 Template Haskell

Tim Sheard, Simon Peyton Jones, "Template Meta-programming for Haskell"[10], SIGPLAN Notices, vol. 37, Nr. 12, December 2002

We propose a new extension to the purely functional programming language Haskell that supports compile-time meta-programming. The purpose of the system is to support the algorithmic construction of programs at compile-time.The ability to generate code at compile time allows the programmer to implement such features as polytypic programs, macro-like expansion, user directed optimization (such as inlining), and the generation of supporting data structures and functions from existing data structures and functions.Our design is being implemented in the Glasgow Haskell Compiler, ghc.

Weitere Literatur:

- Seefried, Chakravarty, Keller, "Optimising Embedded DSLs using Template Haskell"[11] (GPCE 2004)
- Norell, Jansson, "Prototyping Generic Programming using Template Haskell"[12] (MPC 2004)

[10] `research.microsoft.com/~simonpj/Papers/meta-haskell/meta-haskell.pdf`
[11] `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.11.2883`
[12] `www.cse.chalmers.se/~ulfn/papers/genericth.pdf`

7.2.5 Language Workbenches (B)

Most new ideas in software developments are really new variations on old ideas. This article describes one of these, the growing idea of a class of tools that I call Language Workbenches - examples of which include Intentional Software, JetBrains's Meta Programming System, and Microsoft's Software Factories. These tools take an old style of development - which I call language oriented programming and use IDE tooling in a bid to make language oriented programming a viable approach. Although I'm not enough of a prognosticator to say whether they will succeed in their ambition, I do think that these tools are some of the most interesting things on the horizon of software development. Interesting enough to write this essay to try to explain, at least in outline, how they work and the main issues around their future usefulness.

— Martin Fowler [13]

Language workbenches („Arbeitswerkbänke") sind ein neuer Trend in der Entwicklung von *erweiterbaren Programmiersprachen*, die das Integrieren neuer syntaktischer und semantischer Konstrukte mit der Erweiterung von interaktiven Entwicklungsumgebungen (IDEs) wie Eclipse[14] verbinden.

Ihre Aufgabe hier ist es spezifisch, mindestens zwei aktuelle *language workbenches* wie `xtext` und `Spoofax` nach geeigneten Kriterien zu untersuchen und zu vergleichen, und deren wesentlichste Publikationen zu untersuchen. Suchen Sie auch gezielt nach bereits existierenden Vergleichskriterien.

Literatur

- Obengenanntes Essay von Martin Fowler
- Lennart C. L. Kats, Eelco Visser. "The Spoofax Language Workbench. Rules for Declarative Specification of Languages and IDEs."[15] In Martin Rinard, editor, Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2009, Reno, NV, USA. 2010
- Markus Völter, "oAW xText: A framework for textual DSLs"[16], Eclipse Summit Europe 2006

[13] `http://martinfowler.com/articles/languageWorkbench.html`

[14] `http://eclipse.org`

[15] `dl.acm.org/ft_gateway.cfm?id=1869497` (Aktuelle ACM-Papiere sollten aus dem Uni-Netzwerk heraus kostenlos lesbar sein)

[16] `http://www.eclipse.org/Xtext/` und `http://eclipsecon.org/summiteurope2006/presentations/ESE2006-EclipseModelingSymposium12_xTextFramework.pdf`

7.3 Parallele Programmiersprachen

Parallele Programmierung ist schon fast seit Beginn der Computerentwicklung Teil der Softwareentwicklungslandschaft. Allerdings wurden parallele Systeme erst in den frühen 2000er-Jahren Teil des Mainstream: zu diesem Zeitpunkt begann die Beschleunigung der Taktfrequenz nachzulassen; die von Moore's Regel weiterhin verfügbaren zusätzlichen Transistoren wurden von den Chipherstellern stattdessen in zusätzliche parallele Rechenkerne investiert.

Entsprechend finden sich in der parallelen Softwareentwicklung sowohl traditionelle Ideen, die auf verteilten Systemen aufbauen, als auch Konzepte, die gezielt auf lokal-paralleler Berechnung mit zwischen den Rechenkernen geteiltem Speicher aufsetzten, sei es durch GPUs, mehrere Prozessorkerne, oder Vektorprozessoren. Letztere Konzepte kommen urpsrünglich aus der Hochperformanzberechnung, bahnen sich nun aber langsam ihren Weg in die Alltagsprogrammierung.

Literatur (Bibliothek oder Bürobestand)

- "Computer Architecture: A Quantitative Approach" (John LeRoy Hennessy, David Patterson; 2006, Morgan Kaufmann)
- "Patterns for Parallel Programming" (Mattson, Sanders, Massingill; 2004, Addison-Wesley)
- Jeffrey Dean and Sanjay Ghemawat,"MapReduce: Simplified Data Processing on Large Clusters"[17], OSDI 2004

[17] `www.usenix.org/event/osdi04/tech/full_papers/dean/dean.pdf`

7.3.1 Seq no more: Better Strategies for Parallel Haskell

Simon Marlow, Patrick Maier, Hans-Wolfgang Loidl, Mustafa K Aswad, Phil Trinder "Seq no more: Better Strategies for Parallel Haskell"[18], Proceedings of the third ACM Haskell symposium on Haskell, Seiten 91–102

We present a complete redesign of Evaluation Strategies, a key abstraction for specifying pure, deterministic parallelism in Haskell. Our new formulation preserves the compositionality and modularity benefits of the original, while providing significant new benefits. First, we introduce an evaluation-order monad to provide clearer, more generic, and more efficient specification of parallel evaluation. Secondly, the new formulation resolves a subtle space management issue with the original strategies, allowing parallelism (sparks) to be preserved while reclaiming heap associated with superfluous parallelism. Related to this, the new formulation provides far better support for speculative parallelism as the garbage collector now prunes unneeded speculation. Finally, the new formulation provides improved compositionality: we can directly express parallelism embedded within lazy data structures, producing more compositional strategies, and our basic strategies are parametric in the coordination combinator, facilitating a richer set of parallelism combinators. We give measurements over a range of benchmarks demonstrating that the runtime overheads of the new formulation relative to the original are low, and the new strategies even yield slightly better speedups on average than the original strategies.

Empfolenes Vorwissen:

- Haskell
- Monaden

[18] `community.haskell.org/~simonmar/papers/strategies.pdf`

7.3.2 Relaxed Separation Logic: A Program Logic for C11 Concurrency

Viktor Vafeiadis, Chinmay Narayan, "Relaxed Separation Logic: A Program Logic for C11 Concurrency"[19], OOPSLA 2013 (Indianapolis)

We introduce relaxed separation logic (RSL), the first program logic for reasoning about concurrent programs running under the C11 relaxed memory model. From a user's per- spective, RSL is an extension of concurrent separation logic (CSL) with proof rules for the various kinds of C11 atomic accesses. As in CSL, individual threads are allowed to access non-atomically only the memory that they own, thus pre- venting data races. Ownership can, however, be transferred via certain atomic accesses. For SC-atomic accesses, we per- mit arbitrary ownership transfer; for acquire/release atomic accesses, we allow ownership transfer only in one direction; whereas for relaxed atomic accesses, we rule out ownership transfer completely. We illustrate RSL with a few simple ex- amples and prove its soundness directly over the axiomatic C11 weak memory model.

Empfohlenes Vorwissen:

- Lambdakalkül (insbes. calculus of constructions)
- Formale Beweissysteme / Beweisassistenten (speziell Coq)

[19] `www.mpi-sws.org/~viktor/papers/oopsla2013-rsl.pdf`

7.3.3 Concurrent ML (B)

John H. Reppy, "CML: A Higher-order Concurrent Language"[20], Cornell

Concurrent ML (CML) is a high-level, high-performance language for concurrent programming. It is cm extension of Standard ML (SML), and is implemented on top of Standard ML of New Jersey (SML/NJ). CML is a practical language and isbeing used to build real systems. It demonstrates that we need not sacrifice high-level notation in order to have good performance. Although most research in the area of concurrent language design has been motivated by the desire to improve performance by exploiting multiprocessors, we believe that concurrency is a useful programming paradigm for certain application domains. For example, interactive systems often have a naturally concurrent structure. Another example is distributed systems: most systems for distributed programming provide multi-threading at the node level. Sequential programs in these application domains often must use complex and artificial control structures to schedule and interleave activities (e.g., event-loops in graphics libraries). They are, in effect, simulating concurrency. These application domains need a high-level concurrent languagethat provides both effi- cient sequential execution and efficient concurrent execution: CML satisfies this need.

Dies ist ein etwas älteres Papier, dessen Konzepte allerdings weiterhin aktuell sind. Zur Bearbeitung des Themas sollten entweder neuere Entwicklungen oder konkurrierende Ansätze vergleichend berücksichtigt werden.

[20] `dl.acm.org/ft_gateway.cfm?id=113470&type=pdf`

7.3.4 PLINQ, PQL, Java8 lambdas (B)

Es existieren zur Zeit zumindest drei konkurrierende Ansätze zur Intergrierung von „bequemem“ Parallelismus in Java-artige Sprachen. Vergleichen Sie die folgenden drei Ansätze nach geeigneten Kriterien:

- Parallele Ausführung in Java 8 mit Lambdas (Project Lambda[21])
- Parallel LINQ (PLINQ, für Microsoft's .NET-System)
- PQL (Reichenbach, Smaragdakis, Immerman, “PQL: A Purely-Declarative Java Extension for Parallel Programming”[22], ECOOP 2012)

[21] `openjdk.java.net/projects/lambda`
[22] `http://creichen.net/pql/ecoop-2012.pdf`

7.3.5 Partitionierende parallele Sprachen (B)

Die parallelen Programmiersprachen X10, Chapel, und Fortress unterstützten ein Konzept, das oft als *locale* bezeichnet wird: eine Abstraktion, die es erlaubt, separate an einer Berechnung beteiligte Hardwaresysteme anzusprechen und diesen Teile einer Berechnung zuzuweisen.

Ihre Aufgabenstellung in diesem Thema ist, dieses Konzept und andere Sprachkonstrukte zur expliziten und impliziten Partitionierung von Berechnungen in den drei genannten Sprachen mit geeigneten Kriterien zu analysieren und zu vergleichen. Sie können sich dabei auf die Sprachspezifikationen (soweit vorhanden) und auf wissenschaftliche Publikationen berufen. Untersuchen Sie an Beispielen, für welche Arten von parallelen Probleme die expliziten Partitionierungskonzepte geeignet sind.