Report from Dagstuhl Seminar 14211

# The Future of Refactoring

**Edited by**

**Danny Dig[1], William G. Griswold[2], Emerson Murphy-Hill[3], and Max Schäfer[4]**

1 **Oregon State University – Corvallis, US,** digd@eecs.oregonstate.edu
2 **University of California – San Diego, US,** wgg@cs.ucsd.edu
3 **North Carolina State University, US,** emerson@csc.ncsu.edu
4 **Semmle Ltd., Oxford, UK,** max@semmle.com

**Abstract**

This report documents the program and the outcomes of Dagstuhl Seminar 14211 on "The Future of Refactoring." Over the past decade, refactoring has become firmly established as an essential part of industrial software development. At the same time, academic interest in refactoring has grown at a fast pace, resulting in a large body of literature on many different aspects of refactoring. The aim of this seminar was to provide a forum for refactoring researchers and practitioners to discuss what has been achieved, get to know each others' work, and plan future collaboration. This report presents abstracts of the participants' talks and summaries of breakout sessions, and introduces some joint projects that were started as a result of the seminar.

**Seminar** May 18–23, 2014 – http://www.dagstuhl.de/14211
**1998 ACM Subject Classification** D.2.7 Restructuring, reverse engineering, and reengineering
**Keywords and phrases** Refactoring
**Digital Object Identifier** 10.4230/DagRep.4.5.40

## 1 Executive Summary

*Danny Dig*
*William G. Griswold*
*Emerson Murphy-Hill*
*and Max Schäfer*

**License** Creative Commons BY 3.0 Unported license
© Danny Dig, William G. Griswold, Emerson Murphy-Hill, and Max Schäfer

The Dagstuhl seminar on "The Future of Refactoring" brought together 41 researchers and practitioners from academia and industry working on different aspects of refactoring. Participants had the opportunity to introduce their own work both in short plenary talks and more detailed presentations during breakout sessions, with daily keynote talks by eminent refactoring researchers providing historical background. Given the rapid growth of the field over the past decade, special emphasis was put on providing opportunities for researchers with similar interests to meet and survey the state of the art, identify open problems and research opportunities, and jointly chart the future of refactoring research.

We believe the seminar achieved its goal of providing a forum for in-depth discussion of recent research in the area, and of fostering collaboration. In particular, it kickstarted several collaborative projects, among them a book on refactoring tools, a special journal issue on refactoring and a survey article on refactoring research over the last decade.

Except where otherwise noted, content of this report is licensed
under a Creative Commons BY 3.0 Unported license

The Future of Refactoring, *Dagstuhl Reports*, Vol. 4, Issue 5, pp. 40–67
Editors: Danny Dig, William G. Griswold, Emerson Murphy-Hill, and Max Schäfer

Dagstuhl Reports
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

## Research Context

Modern software is rarely written from scratch. Instead, it usually incorporates code from previous systems, and is itself reincarnated in other programs. Modern software is also not static. Instead, it constantly changes as bugs are fixed and features added, and usually these changes are performed by more than one programmer, and not necessarily by the original authors of the code.

Refactoring is a technique for supporting this highly dynamic software life cycle. At its most basic, refactoring is the process of improving the internal structure of an existing piece of code without altering its external behavior. It can be used for cleaning up legacy code, for program understanding, and as a preparation for bug fixes or for adding new features. While any behavior-preserving change to a program can be considered a refactoring, many particularly useful and frequently recurring refactoring operations have been identified and catalogued. Over the past decade, popular development environments have started providing automated support for performing common refactorings, making the process of refactoring less tedious and error-prone.

Based on the accumulated experience with refactorings both in practical applications and in research, this seminar aimed to identify open problems and challenges and to foster collaboration between researchers and between academia and industry to address these issues and actively shape the future of refactoring.

## Seminar Format

Given the large number of participants, the standard conference format with one in-depth talk per participant would have been impractical. Instead, we decided to split up the schedule: during the first three days, the mornings were allocated to plenary sessions. Each day began with a keynote by a distinguished speaker with decades of experience with refactoring, in which they presented their perspective on refactoring. The rest of the morning was allocated to "lightning talks" where each participant was given a 7-minute presentation slot for providing a quick, high-level overview of their work without getting bogged down in detail, followed by a few minutes for questions. While this format was not easy for the speakers, everyone rose to the challenge, and reactions from both presenters and audience were broadly positive.

Monday afternoon was given over to four parallel breakout sessions organized along thematic lines: novel domains for refactoring, user experience in refactoring, refactoring tools and meta-tools, and refactoring in education. While participants appreciated the opportunity for more in-depth presentations and discussion, this format had the unfortunate but inevitable drawback that several talks were held in parallel, and not everyone was able to attend all the talks they were interested in.

Tuesday afternoon had an industry panel, followed by another round of breakout sessions. Discussion and exchange continued in an informal setting during Wednesday afternoon's excursion to Mettlach.

On Thursday morning, we had another keynote followed by a final round of breakout sessions. While the focus of the breakout sessions on Monday and Tuesday had been on surveying recent work and getting an overview of the state of the art, Thursday's sessions were aimed at gathering together the threads, and identifying common themes, open problems and research opportunities.

The outcome of these group discussions were then briefly presented in a plenary on Thursday afternoon, and opportunities for collaborative projects were identified. Specifically,

the following projects were discussed and planned in group discussions on Thursday afternoon:
- a book on refactoring tools;
- a special issue of IEEE Software on refactoring;
- a survey paper on refactoring research in the last decade;
- an informal working group on the place of refactoring in the Computer Science curriculum.

Friday morning saw a final plenary discussion, summarizing the project discussions of Thursday afternoon and ending with a retrospective session on which aspects of the seminar are worth keeping for the future, what needs to change, and what still puzzles us.

We hired George Platts, a professional artist, to facilitate games he designed and tangential thinking activities to help the participants develop a sense of scientific community. During each of the five days of the Seminar, George ran 30-minute games sessions at the beginning of the day which doubled as times for announcements to be given and daily reports to be delivered. In the early afternoon, we had a 30-minute game session to energize participants for the afternoon's workshops. For the rest of the time in his 'studio', he has been playing music, showing short films, facilitating drawing and painting activities, composing sound composition for all participants to perform.

## 2 Table of Contents

**Executive Summary**
*Danny Dig, William G. Griswold, Emerson Murphy-Hill, and Max Schäfer* . . . . 40

**Perspective Talks**

The Birth of Refactoring – A Personal Perspective
*William G. Griswold* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46

Concerns in Refactoring
*Bill Opdyke* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46

Two Decades of Refactoring Tools
*Don Roberts* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46

Refactoring using Type Constraints
*Frank Tip* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47

**Lightning Talks**

Teaching Refactoring
*Andrew P. Black* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47

Retrofitting Parallelism through Refactoring
*Danny Dig* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47

Refactoring for Usability of Web Applications
*Alejandra Garrido* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48

Automated Behavioral Testing of Refactoring Engines
*Rohit Gheyi* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49

Refactoring Refactoring History
*Shinpei Hayashi* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49

Refactoring Spreadsheets
*Felienne Hermans* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50

Awareness of Refactoring Tools
*Emerson Murphy-Hill* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50

Agile Software Assessment
*Oscar M. Nierstrasz* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50

Wrangler – Writing Refactorings Made Easy
*Huiqing Li* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51

Proof Improving Refactoring
*Francesco Logozzo* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51

Usage Contracts
*Kim Mens* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52

Domain-Specific Model Refactoring
*Tom Mens* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52

Detection and Correction of Anti-Patterns
*Naouel Moha* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53

Can we Mine and Reapply Refactoring Strategies?
*Francisco Javier Perez Garcia* . . . . . 53

Refactoring with Synthesis
*Veselin Raychev* . . . . . 53

Identifying Overly Strong Conditions in Refactoring Implementations
*Gustavo Soares* . . . . . 54

The History of C++ Refactoring (for Eclipse CDT)
*Peter Sommerlad* . . . . . 54

A Brief History of Eclipse-based Refactorings by HSR
*Peter Sommerlad* . . . . . 55

Extract+Move=Bug
*Volker Stolz* . . . . . 55

Why Should I Trust Your Refactoring Tool?
*Simon J. Thompson* . . . . . 56

To the Cloud and Back: Automated Inter-Address Space Component Migration to Support Software Evolution
*Eli Tilevich* . . . . . 56

Automated Decomposition of Software Modules
*Mohsen Vakilian* . . . . . 57

Complexity of Maintenance – Refactoring for the Reproducible Evaluation of Design Choices
*Jurgen Vinju* . . . . . 57

**Demonstrations**

IDEs are Ecosystems
*Andrew P. Black* . . . . . 58

Tools for Retrofitting Parallelism
*Danny Dig* . . . . . 58

Tools for Refactoring of Web Applications
*Alejandra Garrido* . . . . . 59

WitchDoctor: IDE Support for Real-Time Auto-Completion of Refactorings
*William G. Griswold* . . . . . 59

REdiffs: Refactoring-aware Difference Viewer for Java
*Shinpei Hayashi* . . . . . 60

Refactoring via Pretty-Printing
*Jongwook Kim* . . . . . 60

Interactive Quick Fix
*Emerson Murphy-Hill* . . . . . 61

Cloud Refactoring
*Eli Tilevich* . . . . . 61

A Universal Type Qualifier Inference System
*Mohsen Vakilian* . . . . . 61

Rascal for Experimenting with New Intermediate Formats for Source Code Analysis
*Jurgen Vinju* . . . . . . . . . . 62

How Can We Do Better than Search and Replace?
*Jan Wloka* . . . . . . . . . . 62

**Working Groups**

User Experience Breakout: Dimensions of Refactoring
*Emerson Murphy-Hill* . . . . . . . . . . 63

User Experience Breakout: The Future of Refactoring
*Emerson Murphy-Hill* . . . . . . . . . . 63

Plenary Discussion on Refactoring in Education, Corpora and Benchmarks
*Simon J. Thompson* . . . . . . . . . . 63

Novel Applications of Refactoring Breakout
*Bill Opdyke* . . . . . . . . . . 63

Refactoring Tools and Meta-Tools
*Max Schaefer* . . . . . . . . . . 65

**Industry Roundtable** . . . . . . . . . . 66

**Participants** . . . . . . . . . . 67

# 3 Perspective Talks

## 3.1 The Birth of Refactoring – A Personal Perspective

*William G. Griswold (University of California – San Diego, US)*

**License** Creative Commons BY 3.0 Unported license
© William G. Griswold
**Joint work of** Griswold, William G.; Notkin, David; Bowdidge, Robert W.

Software Refactoring was invented in the late 1980's at two institutions – the University of Illinois by Bill Opdyke and Ralph Johnson, and the University of Washington by myself and David Notkin. In this talk I revisit the surprising events at the birth of refactoring – what we called meaning-preserving restructuring – at the University of Washington. I'll talk about how the ideas came about, and the research agenda and results that emerged. In the course of the presentation, I'll highlight several lessons for researchers seeking high impact in their work.

## 3.2 Concerns in Refactoring

*Bill Opdyke (JP Morgan Chase – Chicago, US)*

**License** Creative Commons BY 3.0 Unported license
© Bill Opdyke

What are the four key reasons why software developers might be reluctant to refactor their code even if they think refactoring is, at least in the abstract, a good idea? How might one effectively address those concerns? These four concerns, and the means for addressing them, have applicability far beyond refactoring. In this talk, I discussed these lessons learned in a refactoring context and how they subsequently helped me as an architect and in other roles.

## 3.3 Two Decades of Refactoring Tools

*Don Roberts (University of Evansville, US)*

**License** Creative Commons BY 3.0 Unported license
© Don Roberts
**Joint work of** Brant, John; Roberts, Don

We released the first industrial refactoring tool for Smalltalk 20 years ago this month. In this talk, we will present the history of the Refactoring Browser along with the other tools that we have developed to solve rewriting problems. The tools have been used to replace a database layer in a commercial application. We have also developed a process that, along with our rewriting tool, allows us to migrate existing systems between languages while not sacrificing development time. We will also present what we've learned about how end-users interact with refactoring tools.

## 3.4 Refactoring using Type Constraints

*Frank Tip (University of Waterloo, CA)*

**License** Creative Commons BY 3.0 Unported license
© Frank Tip
**Joint work of** Tip, Frank; Fuhrer, Robert; Kieżun, Adam; Ernst, Michael; Balaban, Ittai; De Sutter, Bjorn

Type constraints express subtype relationships between the types of program expressions, for example, those relationships that are required for type correctness. Type constraints were originally proposed as a convenient framework for solving type checking and type inference problems. This work shows how type constraints can be used as the basis for practical refactoring tools. In our approach, a set of type constraints is derived from a type-correct program P. The main insight behind our work is the fact that P constitutes just one solution to this constraint system, and that alternative solutions may exist that correspond to refactored versions of P. We show how a number of refactorings for manipulating types and class hierarchies can be expressed naturally using type constraints. Several refactorings in the standard distribution of Eclipse are based on our work.

# 4 Lightning Talks

## 4.1 Teaching Refactoring

*Andrew P. Black (Portland State University, US)*

**License** Creative Commons BY 3.0 Unported license
© Andrew P. Black
**Joint work of** Black, Andrew P.; Noble, James; Bruce, Kim B.
**Main reference** A. P. Black, K. B. Bruce, Michael Homer, J. Noble, "Grace: the absence of (inessential) difficulty," in Proc. of the 2012 ACM Int'l Symp. on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!'12), pp. 85–98, ACM, 2012.
**URL** http://dx.doi.org/10.1145/2384592.2384601

I'm engaged in designing Grace, a new programming language for teaching object-oriented programming. We hope that Grace will be used for teaching object-oriented concepts, testing, debugging, design, and refactoring. Our motivation is to have a language with low 'accidental' complexity, so that students can focus on the essential complexity of the task.

I have questions, not answers. How should one introduce refactoring to novices? If we teach 'red—green—refactor', what are the important refactorings? What about the refactorings that are no-ops in our language, such as abstract instance variable? Should refactoring be taught early, or late? I would like to discuss these questions with the group.

## 4.2 Retrofitting Parallelism through Refactoring

*Danny Dig (Oregon State University, US)*

**License** Creative Commons BY 3.0 Unported license
© Danny Dig
**Main reference** D. Dig, "A Refactoring Approach to Parallelism," IEEE Software 28(1):12–22, 2011.
**URL** http://dx.doi.org/10.1109/MS.2011.1

In the multicore era, programmers have to work harder to introduce parallelism for performance or to enable new applications and services not possible before. In this talk I present

our ever-growing toolset of interactive refactorings for adding parallelism into sequential programs. This toolset is grounded on empirical studies that shed light into the practice of using, misusing, underusing, or abusing parallel libraries. Our refactoring toolset supports refactorings from three domains: adding thread-safety, improving throughput, and scalability. Empirical evaluation shows that our toolset is useful: (i) it dramatically reduces the burden of analyzing and changing code, (ii) it is fast so it can be used interactively, (iii) it correctly applies transformations that open-source developers applied incompletely, and (iv) users prefer the improved quality of the changed code. I muse on lessons that can be learned as we move onto automated refactoring for mobile apps.

## 4.3 Refactoring for Usability of Web Applications

*Alejandra Garrido (University of La Plata, AR)*

**License** Creative Commons BY 3.0 Unported license
© Alejandra Garrido
**Joint work of** Garrido, Alejandra; Rossi, Gustavo

Refactoring represents an essential activity in today's software lifecycle and a powerful technique against software decay. Software decay, however, is not only about code becoming legacy, but it is also about systems becoming less usable compared to competitor solutions. We propose refactoring to progressively and systematically improve the external quality of an existing web application, like usability and accessibility. The transformations can be applied at the model level (the navigation, presentation or process model) or at the implementation level. We created a framework where refactorings can also be applied at the client-side, as DOM changes, which allows for personalization. We are now working on the automatic detection of bad usability smells from user interaction logs.

### References

1 Alejandra Garrido, Gustavo Rossi, Damiano Distante. *Refactoring For Usability In Web Applications.* IEEE Software 28(3):60–67. 2011.
2 Alejandra Garrido; Sergio Firmenich; Gustavo Rossi; Julian Grigera; Nuria Medina Medina; Ivana Harari. *Personalized Web Accessibility using Client-Side Refactoring.* IEEE Internet Computing 17(4):58–66. 2013.
3 Alejandra Garrido; Gustavo Rossi; Nuria Medina Medina; Julian Grigera; Sergio Firmenich. *Improving Accessibility of Web Interfaces: Refactoring to the Rescue.* Universal Access in the Information Society 13(4). Springer. 2014.
4 Julian Grigera, Alejandra Garrido and Jose Matias Rivero. *A Tool for Detecting Bad Usability Smells in an Automatic Way.* Int. Conf. On Web Engineering (ICWE 2014). Demo and poster track. Toulouse, France. July, 2014.

## 4.4 Automated Behavioral Testing of Refactoring Engines

*Rohit Gheyi (Universidade Federal – Campina Grande, BR)*

**License** Creative Commons BY 3.0 Unported license
© Rohit Gheyi
**Joint work of** Soares, Gustavo; Gheyi, Rohit; Massoni, Tiago
**Main reference** G. Soares, R. Gheyi, T. Massoni, "Automated behavioral testing of refactoring engines," IEEE Transactions on Software Engineering, 39(2):147–162, 2013.
**URL** http://dx.doi.org/10.1109/TSE.2012.19

Proving refactoring sound with respect to a formal semantics is considered a challenge. In practice, developers write test cases to check their refactoring implementations. However, it is difficult and time consuming to have a good test suite since it requires complex inputs (programs) and an oracle to check whether it is possible to apply the transformation. In this talk, I discuss the challenges of automated testing of refactoring engines. Moreover, I present our current technique that detected more than 200 bugs related to compilation errors, behavioral changes and overly strong conditions in the best refactoring engines (Eclipse, NetBeans and JRRT) [1].

### References

1. G. Soares, R. Gheyi, T. Massoni, Automated behavioral testing of refactoring engines, IEEE Transactions on Software Engineering 39 (2) (2013) 147–162.
2. G. Soares, R. Gheyi, D. Serey, T. Massoni, Making program refactoring safer, IEEE Software 27 (2010) 52–57.
3. G. Soares, M. Mongiovi, R. Gheyi, Identifying overly strong conditions in refactoring implementations, in: Proceedings of the 27th IEEE International Conference on Software Maintenance, ICSM '11, Washington, USA, 2011, pp. 173–182.
4. G. Soares, R. Gheyi, E. Murphy-Hill, B. Johnson, Comparing approaches to analyze refactoring activity on software repositories, Journal of Systems and Software 86 (4) (2013) 1006–1022.
5. M. Mongiovi, R. Gheyi, G. Soares, L. Teixeira, and P. Borba, Making refactoring safer through impact analysis ,Science of Computer Programming, 2014, to appear.

## 4.5 Refactoring Refactoring History

*Shinpei Hayashi (Tokyo Institute of Technology, JP)*

**License** Creative Commons BY 3.0 Unported license
© Shinpei Hayashi
**Joint work of** Hayashi, Shinpei; Omori, Takayuki; Zenmyo, Teruyoshi; Maruyama, Katsuhisa; Saeki, Motoshi
**Main reference** S. Hayashi, T. Omori, T. Zenmyo, K. Maruyama, M. Saeki, "Refactoring Edit History of Source Code," in Proc. of the 28th IEEE Int'l Conf. on Software Maintenance (ICSM'12), pp. 617–620, IEEE, 2012.
**URL** http://dx.doi.org/10.1109/ICSM.2012.6405336

In this talk, we present a concept for refactoring an edit history of source code in a refactoring process and a technique for its automation. The aim of our history refactoring is to improve the clarity and usefulness of the history without changing its overall effect. We have defined primitive history refactorings including their preconditions and procedures, and large refactorings composed of these primitives. Our tool enables developers to pursue some useful applications using history refactorings such as task level commit from an entangled edit history in an floss refactoring process, and support for reviewing the difference obtained from tangled edits.

## 4.6 Refactoring Spreadsheets

*Felienne Hermans (TU Delft, NL)*

License Creative Commons BY 3.0 Unported license
© Felienne Hermans
Joint work of Hermans, Felienne; Dig, Danny
URL http://www.felienne.com/BumbleBee

Spreadsheets are code! They are just as complex, used for similar purposes and suffer from similar problems like a long life span and lack of documentation. Therefore, we can apply methods from software engineering to spreadsheets to address those problems.

For refactoring, we propose a tool called BumbleBee that can refactor spreadsheet formulas, which the user can define themselves in a little language.

## 4.7 Awareness of Refactoring Tools

*Emerson Murphy-Hill (North Carolina State University, US)*

License Creative Commons BY 3.0 Unported license
© Emerson Murphy-Hill
Main reference E. R. Murphy-Hill, "Continuous Social Screencasting to Facilitate Software Tool Discovery," in Proc. of the 34th Int'l Conf. on Software Engineering (ICSE'12), pp. 1317–1320, IEEE, 2012; pre-print available from author's webpage.
URL http://dx.doi.org/10.1109/ICSE.2012.6227090
URL http://people.engr.ncsu.edu/ermurph3/papers/icseNIER12.pdf

One of the main challenges developers face when using refactoring tools is not even knowing that the refactoring tools are there. This lack of awareness is a problem because programmers, without tools, are otherwise refactoring manually, which is both slow and error-prone. In this talk, I discuss the causes of lack of awareness among programmers, existing solutions, and some open questions.

## 4.8 Agile Software Assessment

*Oscar M. Nierstrasz (Universität Bern, CH)*

License Creative Commons BY 3.0 Unported license
© Oscar M. Nierstrasz

Modern IDEs are largely code-centric, and do not support developers well in understanding the software systems they need to develop, maintain and refactor. We believe that developers need a flexible environment of "meta-tools" that can be easily adapted to the project at hand, to query, browse, debug and monitor software systems and the ecosystems they belong to.

## 4.9 Wrangler – Writing Refactorings Made Easy

*Huiqing Li (University of Kent, GB)*

**License** Creative Commons BY 3.0 Unported license
© Huiqing Li
**Joint work of** Li, Huiqing; Thompson, Simon
**Main reference** S. J. Thompson, "Refactoring tools for functional languages," Journal of Functional Programming, 23(3):293–350, 2013.
**URL** http://dx.doi.org/10.1017/S0956796813000117
**URL** https://github.com/RefactoringTools/wrangler

This talk and demo shows a framework built into Wrangler – a refactoring and code inspection tool for Erlang programs – that allows users to define for themselves refactorings that suit their needs. With this framework, elementary refactorings are defined using a template- and rule-based program transformation and analysis API; composite refactorings are scripted using a high-level domain-specific language(DSL). User-defined refactorings, both elementary and composite, are fully integrated into Wrangler and so can be previewed, applied interactively and 'undone'.

## 4.10 Proof Improving Refactoring

*Francesco Logozzo (Microsoft Research – Redmond, US)*

**License** Creative Commons BY 3.0 Unported license
© Francesco Logozzo
**Main reference** P. Cousot, R. Cousot, F. Logozzo, M. Barnett, "An abstract interpretation framework for refactoring with application to extract methods with contracts," in Proc. of the 27th Annual ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'12), pp. 213-232, ACM, 2012.
**URL** http://dx.doi.org/10.1145/2384616.2384633
**URL** http://research.microsoft.com/apps/pubs/default.aspx?id=170382

Traditional refactoring modifies the program source while preserving the concrete semantics of the program. Proof-improving refactoring, on the other hand, aims at preserving or improving the proof of correctness of a program, i.e., its abstract semantics. In the talk I presented three examples of proof-improving refactoring. The first one is useful to make code-bases ready for automatic program verification. Starting from an un-annotated code base, we automatically insert CodeContracts (preconditions, postconditions, and object invariants). The inferred contracts are sound, in that no good execution is removed, only bad ones. The injected contracts enable a modular correctness proof of the program The second example are automated code repairs. Starting from the alarms of a sound static analyzer, we propose a set of program transformations to fix bug in the programs and/or to let its correctness proof succeed. Finally, the last example is an abstract interpretation framework for refactoring. We instantiate it to a new refactoring: extract method with contracts. In addition to extracting the method, we endow it con preconditions and postconditions which satisfy some constraints, namely to be a valid, general, and complete. The extract method with contracts guarantee that the proof of correctness of the program proceeds even when the refactoring is applied.

## 4.11 Usage Contracts

*Kim Mens (UCL, Belgium)*

**License** Creative Commons BY 3.0 Unported license
© Kim Mens
**Joint work of** K. Mens, A. Lozano and A. Kellens

Developers often encode design knowledge through structural regularities such as API usage protocols, coding idioms and naming conventions. As these regularities express how the source code should be structured, they provide vital information for developers (re)using that code. Adherence to such regularities tends to deteriorate over time when they are not documented and checked explicitly. Our uContracts tool and approach allows to codify and verify such regularities as 'usage contracts'. The contracts are expressed in an internal domain-specific language that is close to the host programming language, the tool is tightly integrated with the development environment and provides immediate feedback during development when contracts get breached, but the tool is not coercive and allows the developer to decide if, when and how to correct the broken contracts (the tool just highlights the errors and warnings in the integrated development environment). In spirit, the approach is very akin to unit testing, except that we do not test behaviour, but rather verify program structure. The tool, of which some screenshots can be found below, was prototyped in the Pharo dialect of the Smalltalk programming language.

## 4.12 Domain-Specific Model Refactoring

*Tom Mens (University of Mons, Belgium)*

**License** Creative Commons BY 3.0 Unported license
© Tom Mens

Model-driven software engineering is becoming an established discipline. In this presentation we present the challenge of providing generic support for domain-specific model refactoring. While refactoring tools and technology are well established for programming languages, it is much less the case for (software) modeling languages. For domain-specific modeling languages (DSMLs), there is even no or very little refactoring support. In this presentation, we provide a case study in which we are developing a domain-specific modelling language (DSML) for developing executable models of applications that use gestural interactions (hand movements) to control virtual objects in a 3D environment. We explain the need for refactoring such models, and the need for dealing with different notions of "behaviour preservation". We illustrate how one can provide generic support for domain-specific models, using the AtomPM transformation tool (https://www.youtube.com/watch?v=iBbdpmpwn6M, http://syriani.cs.ua.edu/atompm/atompm.htm), that combines graph transformation technology with the use of a concrete visual model syntax. Preservation of desirable model properties can be verified using the most appropriate formalism (e.g. model checkers for verifying temporal properties; OCL checkers for verifying structural properties; or any other tool that may be more appropriate for expressing and verifying the property of interest).

## 4.13 Detection and Correction of Anti-Patterns

*Naouel Moha*

**License** Creative Commons BY 3.0 Unported license
© Naouel Moha

Anti-patterns are design problems that come from "poor" recurring design choices. They may hinder development and maintenance of systems by making them hard for software engineers to change and evolve. A semi-automatic detection and correction are thus key factors to ease the maintenance and evolution stages. Several techniques and tools have been proposed in the literature both for the detection and correction of anti-patterns in object-oriented systems. However, works in service-based systems are still in their infancy despite their importance. In this seminar, I presented a novel and innovative approach supported by a framework for detecting antipatterns in service-based systems. For the correction, we are still investigating some techniques for correcting service-based antipatterns.

## 4.14 Can we Mine and Reapply Refactoring Strategies?

*Francisco Javier Perez Garcia*

**License** Creative Commons BY 3.0 Unported license
© Francisco Javier Perez Garcia

I believe reuse is the single most beneficial strategy in software engineering and it can be fostered by harnessing today's wide available data and extensive collaborative software development environment. In this context, I want to propose a challenge. Can we mine and reuse successful complex refactoring strategies? In the past I have developed a technique to compute refactoring plans – complex refactoring sequences – from refactoring strategies for correcting bad smells, using automated planning. The future challenge I present involves studying: how to analyse software projects' history to identify refactoring patterns that were successful in the past for removing bad smells; and how to collect and represent these strategies so they can be automatically re-applied in other projects.

## 4.15 Refactoring with Synthesis

*Veselin Raychev (ETH Zürich, CH)*

**License** Creative Commons BY 3.0 Unported license
© Veselin Raychev
**Joint work of** Raychev, Veselin; Schäfer, Max; Sridharan, Manu; Vechev, Martin
**Main reference** V. Raychev, M. Schäfer, M. Sridharan, M. Vechev, "Refactoring with synthesis," in Proc. of the 28th Annual ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'13), pp. 339–354, ACM, 2013.
**URL** http://dx.doi.org/10.1145/2509136.2509544

Modern IDEs provide a fixed set of supported refactorings listed in a menu, which limits the possible use cases and additionally leads to poor discoverability of the available refactoring tools. In this talk, I show a new approach "Refactoring with Synthesis", where the user demonstrates an edit on a piece of code and then a refactoring engine synthesizes a sequence of existing refactorings that perform the task demonstrated by the user task on the entire project.

I present an Eclipse plug-in that operates as "Refactoring without Names": 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.

I show how our approach extends the capabilities of current refactorings: with only minimal user input, the synthesizer was able to quickly discover complex refactoring sequences for several challenging realistic examples. Then, I discuss the concept of local refactorings that we introduce, and how it helps synthesize sequences in an extensible and scalable way.

## 4.16 Identifying Overly Strong Conditions in Refactoring Implementations

*Gustavo Soares (Universidade Federal – Campina Grande, BR)*

**License** Creative Commons BY 3.0 Unported license
© Gustavo Soares
**Joint work of** Soares, Gustavo; Mongiovi, Melina; Gheyi, Rohit
**Main reference** G. Soares, M. Mongiovi, R. Gheyi, "Identifying overly strong conditions in refactoring implementations,"' in Proc. of the 27th Int'l Conf. on Software Maintenance (ICSM'11), pp. 173–182, IEEE, 2011.
**URL** http://dx.doi.org/10.1109/ICSM.2011.6080784

Each refactoring implementation must check a number of conditions to guarantee behavior preservation. However, specifying and checking them are difficult. Sometimes refactoring tool developers may define overly strong conditions that prevent useful behavior-preserving transformations to be performed. We propose an approach for identifying overly strong conditions in refactoring implementations. We automatically generate a number of programs as test inputs for refactoring implementations. Then, we apply the same refactoring to each test input using two different implementations, and compare both results. We use Safe Refactor to evaluate whether a transformation preserves behavior. We evaluated our approach in 10 kinds of refactorings for Java implemented by three tools: Eclipse and Netbeans, and the JastAdd Refactoring Tool (JRRT). In a sample of 42,774 transformations, we identified 17 and 7 kinds of overly strong conditions in Eclipse and JRRT, respectively.

## 4.17 The History of C++ Refactoring (for Eclipse CDT)

*Peter Sommerlad (Hochschule fur Technik – Rapperswil, Switzerland)*

In this talk I present the history of refactoring support within Eclipse-based IDEs. While C++ was the first language addressed by Bill Opdyke that coined the term Refactoring, it was a long way to get working Refactoring support within an IDE. IFS Instiute for Software contributed over almost a decade now to Eclipse and provided infrastructure and plug-ins for better refactoring and code transformation support of C++ code.

The presentation gives a historical overview and shows some of the challenges that need to be addressed when building a refactoring plug-in for C++ in Eclipse CDT. For example, testing refactorings can be tough when formatting details make test cases indadvertedly fail, or when interaction with a wizard makes using a refactoring unbearable. For the

latter, instead of a "Change Function Signature" refactoring with usability problems, the author invented "Toggle Function Definition" quick-refactoring that eases the manual burden of changing a function signature in C++. Another example is interactive guidance for modernizing C++ code to conform to new standard versions or ridding it from bad practices like macros implemented through the refactoring engine.

The talk concludes with an overview of the lessons learned over the many years, such as "automate refactoring tests".

## 4.18 A Brief History of Eclipse-based Refactorings by HSR

*Peter Sommerlad (Hochschule fur Technik – Rapperswil, Switzerland)*

This Lightning Talk gives an overview of the many attempts to create Refactoring plug-ins for Eclipse-based IDEs by IFS Institute for Software students for many different languages, some succeeded and some failed.

The failures happened because of technology, student quality but also for political reasons. The only languages for which we can sustain supporting the refactoring tooling today are C++ and Scala. Our first attempt at Ruby refactoring succeeded technology wise, but failed in the end for political reasons, as well as PHP refactoring which was overtaken by Zend Studio. Parts of our Python refactorings still seem to live within PyDev. With our Groovy Refactoring we were among the first to provide cross-language rename refactoring, but due to lack of financing and personnel we abandoned supporting it. Javascript Refactoring failed for all of the above reasons and because to make it useful it must support the conventions of the JS framework du jour.

One of the lessons learned that even with very good student project results, it still requires work to productize a new refactoring plug-in. When that happens and it gets integrated into "the official IDE" like it happened with the Scala-IDE, then results will be used.

## 4.19 Extract+Move=Bug

*Volker Stolz (University of Oslo, NO)*

**License** Creative Commons BY 3.0 Unported license
© Volker Stolz

Extracting a chunk of code and moving it to a more suitable class to reduce coupling may change the behaviour of the code. A possible solution (apart from reasoning on the code) is to add assertions specific to the extract&move refactoring which track the information necessary to decide (at runtime) whether the behaviour has changed with respect to the old code. We use a search-based, heuristic approach to identify candidates for the refactoring, and evaluate it on the Eclipse JDT UI-project. Existing unit tests provide the necessary coverage.

## 4.20 Why Should I Trust Your Refactoring Tool?

*Simon J. Thompson (University of Kent, GB)*

**License** Creative Commons BY 3.0 Unported license
© Simon J. Thompson
**Joint work of** Thompson, Simon J.; Li, Huiqing; Sultana, Nikolai.

A common question for refactoring tool builders is one of trust. While there are many social, organisational and psychological aspects to this, there are two technical aspects too.

The first is of strength of the assurance: do we test, or do we try to prove correctness in some sense? Secondly, do we aim to verify the results if a single refactoring, or the refactoring itself: that is, verifying it for all possible invocations.

Work has been done by us and others on this, and I survey that and conclude with two suggestions: full verifying a refactoring tool for a formally-verified language, CakeML; and using SMT solving automatically to verify the results of refactorings – initially for Haskell.

## 4.21 To the Cloud and Back: Automated Inter-Address Space Component Migration to Support Software Evolution

*Eli Tilevich (Virginia Polytechnic Institute – Blacksburg, US)*

**License** Creative Commons BY 3.0 Unported license
© Eli Tilevich
**Joint work of** Kwon, Young-Woo; Tilevich, Eli
**Main reference** Y.-W. Kwon, E. Tilevich, "Cloud refactoring: automated transitioning to cloud-based services," Automated Software Engineering, 21(3):345–372, 2014.
**URL** http://dx.doi.org/10.1007/s10515-013-0136-9

The modern computing landscape is increasingly mobile and distributed, characterized by rapidly evolving hardware platforms and network technologies. As a particular example, mobile software designed yesterday will have to run on mobile hardware to be designed tomorrow. Adapting modern software applications for changing execution environments, hardware setups, and user requirements often requires moving software components across address spaces. To facilitate these non-trivial program transformations, this lightning talk introduces two refactoring techniques: Cloud Refactoring and Component Insourcing. Cloud Refactoring renders a portion of an application's functionality remotely accessible as a Web service, including migrating to the cloud the functionality to be accessed as remote cloud-based services, re-targeting the client code accordingly, and handling the faults raised while invoking the services. Component Insourcing moves a remotely accessed component into its client's address space, replacing accesses through a middleware interface with those through local method calls. This talk highlights how these refactoring techniques can facilitate the process of evolving modern software and outlines some of their implementation challenges.

## 4.22 Automated Decomposition of Software Modules

*Mohsen Vakilian (University of Illinois – Urbana, US)*

**License** Creative Commons BY 3.0 Unported license
© Mohsen Vakilian
**Joint work of** Vakilian, Mohsen; Sauciuc, Raluca; Morgenthaler, J. David; Mirrokni, Vahab
**Main reference** M. Vakilian, R. Sauciuc, J. D. Morgenthaler, V. Mirrokni, "Automated Decomposition of Build Targets," Technical Report, 2014.
**URL** http://hdl.handle.net/2142/47551

Large software is often organized as a set of interdependent modules. As the software evolves, the cost of managing the dependencies between the modules tends to grow. A common dependency problem is underutilized modules. An underutilized module is one whose dependents need only a small part of it. Underutilized modules increase the cost of building, testing, and deploying software. Thus, programmers often manually decompose modules. However, decomposing underutilized targets manually is tedious. We propose a greedy algorithm that proposes effective module decompositions by analyzing both intra-module and inter-module dependencies. We implemented the algorithm and evaluated it at Google. The results show that the algorithm is efficient and the decompositions that it proposes significantly reduce the cost of testing.

## 4.23 Complexity of Maintenance – Refactoring for the Reproducible Evaluation of Design Choices

*Jurgen Vinju (CWI – Amsterdam, NL)*

**License** Creative Commons BY 3.0 Unported license
© Jurgen Vinju
**Joint work of** Hills, Mark; Klint, Paul; Vinju, Jurgen
**Main reference** M. Hills, P. Klint, J. J. Vinju, "A case of visitor versus interpreter pattern," in Proc. of the 49th Int'l Conf. on Objects, Models, Components and Patterns (TOOLS'11), LNCS, Vol. 6705, pp. 228–243, Springer, 2011; pre-print available from author's webpage.
**URL** http://dx.doi.org/10.1007/978-3-642-21952-8_17
**URL** http://homepages.cwi.nl/~jurgenv/papers/TOOLS2011.pdf

This lightning talk had two messages. The first is the existence of Rascal, a meta programming language designed to cover the requirements for both source code analysis and transformation. Refactoring requires them both. Rascal emphasizes programming over specification, is based on powerful pattern matching and substitution primitives and relational calculus.

The second message was that refactoring tools can also be used to research trade- offs in design choices. We report on the creation of an ad-hoc refactoring from the Visitor design pattern to the Interpreter design pattern [1]. Using this refactoring we could create two versions of a complex system which differ only in this single design choice: isolating it from all other factors on code quality. We then experimented by executing maintenance scenarios on both systems and measuring the complexity of analyzing and transforming the source code manually. The manual tasks were recorded as "meta-programs" as well. We found out that Visitor is better, surprisingly, even in cases where in theory Interpreter should be better.

### References

1 Mark Hills, Paul Klint, and Jurgen J. Vinju. *A case of visitor versus interpreter pattern*. Proceedings of the 49th International Conference on Objects, Models, Components and Patterns, TOOLS, 2011.

# 5 Demonstrations

## 5.1 IDEs are Ecosystems

*Andrew P. Black (Portland State University, US)*

**License** Creative Commons BY 3.0 Unported license
© Andrew P. Black
**Joint work of** Vainsencher, Daniel; Black, Andrew P.
**Main reference** D. Vainsencher, A. P. Black, "A pattern language for extensible program representation," Transactions on Pattern Languages of Programming 1, LNCS, Vol. 5770, pp. 1–47, Springer, 2009.
**URL** http://dx.doi.org/10.1007/978-3-642-10832-7_1

For many years, implementors of multiple view programming environments have sought a single code model that would form a suitable basis for all of the program analyses and tools that might be applied to the code. They have been unsuccessful. The consequences are a tendency to build monolithic, single- purpose tools, each of which implements its own specialized analyses and optimized representation. This restricts the availability of the analyses, and also limits the reusability of the representation by other tools. Unintegrated tools also produce inconsistent views, which reduce the value of multiple views.

This talk is an advertisement for a paper that describes a set of architectural patterns that allow a single, minimal representation of program code to be extended as required to support new tools and program analyses, while still maintaining a simple and uniform interface to program properties. The patterns address efficiency, correctness and the integration of multiple analyses and tools in a modular fashion.

## 5.2 Tools for Retrofitting Parallelism

*Danny Dig (Oregon State University, US)*

**License** Creative Commons BY 3.0 Unported license
© Danny Dig
**Main reference** D. Dig, "A Refactoring Approach to Parallelism," IEEE Software 28(1):12–22, 2011.
**URL** http://dx.doi.org/10.1109/MS.2011.1

In the multicore era, programmers have to work harder to introduce parallelism for performance or to enable new applications and services not possible before. In this talk I present our ever-growing toolset of interactive refactorings for adding parallelism into sequential programs. This toolset is grounded on empirical studies that shed light into the practice of using, misusing, underusing, or abusing parallel libraries. Our refactoring toolset supports refactorings from three domains: adding thread-safety, improving throughput, and scalability. Empirical evaluation shows that our toolset is useful: (i) it dramatically reduces the burden of analyzing and changing code, (ii) it is fast so it can be used interactively, (iii) it correctly applies transformations that open-source developers applied incompletely, and (iv) users prefer the improved quality of the changed code. I muse on lessons that can be learned as we move onto automated refactoring for mobile apps.

## 5.3 Tools for Refactoring of Web Applications

*Alejandra Garrido (University of La Plata, AR)*

**License** Creative Commons BY 3.0 Unported license
© Alejandra Garrido
**Joint work of** Garrido, Alejandra; Firmenich, Sergio; Grigera, Julián; Rossi, Gustavo

Refactoring can be applied to improve external quality attributes of web applications, and thus provides the ideal context to incite developers to experiment new interface metaphors, and keep them or discard them after usage testing or client feedback. We have extended a tool for web application modeling to support refactoring in a model-driven approach. We have also developed a framework that allows for refactoring on the client-side. This makes it possible to have different views of the same application, customized for and by users, depending on their experience, preferences, or accessibility issues. We are currently developing a tool to automatically detect bad usability smells from user interaction logs.

### References

1 Alejandra Garrido, Gustavo Rossi, Damiano Distante. *Refactoring For Usability In Web Applications*. IEEE Software 28 (3): 60–67. 2011.
2 Alejandra Garrido; Sergio Firmenich; Gustavo Rossi; Julian Grigera; Nuria Medina Medina; Ivana Harari. *Personalized Web Accessibility using Client-Side Refactoring*. IEEE Internet Computing 17 (4): 58–66. 2013.
3 Julian Grigera, A. Garrido and J.M. Rivero. *A Tool for Detecting Bad Usability Smells in an Automatic Way*. Int. Conf. On Web Engineering (ICWE 2014). Demo and poster track. Toulouse, France. July, 2014.

## 5.4 WitchDoctor: IDE Support for Real-Time Auto-Completion of Refactorings

*William G. Griswold (University of California – San Diego, US)*

**License** Creative Commons BY 3.0 Unported license
© William G. Griswold
**Joint work of** Foster, Stephen R; Lerner, Sorin; Griswold, William G.;
**Main reference** S. R. Foster, W. G. Griswold, S. Lerner, "WitchDoctor: IDE support for real-time auto-completion of refactorings," in Proc. of the 34th Int'l Conf. on Software Engineering (ICSE'12), pp. 222–232, IEEE, 2012.
**URL** http://dx.doi.org/10.1109/ICSE.2012.6227191

Integrated Development Environments (IDEs) have come to perform a wide variety of tasks on behalf of the programmer, refactoring being a classic example. These operations have undeniable benefits, yet their large (and growing) number poses a cognitive scalability problem. Our main contribution is WitchDoctor – a system that can detect, on the fly, when a programmer is hand-coding a refactoring. The system can then complete the refactoring in the background and propose it to the user long before the user can complete it. This implies a number of technical challenges. The algorithm must be 1) highly efficient, 2) handle unparseable programs, 3) tolerate the variety of ways programmers may perform a given refactoring, 4) use the IDE's proven and familiar refactoring engine to perform the refactoring, even though the the refactoring has already begun, and 5) support the wide range of refactorings present in modern IDEs. Our techniques for overcoming these challenges are the technical contributions of this paper. We evaluate WitchDoctor's design and implementation by simulating over 5,000 refactoring operations across three open-source

projects. The simulated user is faster and more efficient than an average human user, yet WitchDoctor can detect more than 90% of refactoring operations as they are being performed – and can complete over a third of refactorings before the simulated user does. All the while, WitchDoctor remains robust in the face of non-parseable programs and unpredictable refactoring scenarios. We also show that WitchDoctor is efficient enough to perform computation on a keystroke-by-keystroke basis, adding an average overhead of only 15 milliseconds per keystroke.

## 5.5 REdiffs: Refactoring-aware Difference Viewer for Java

*Shinpei Hayashi (Tokyo Institute of Technology, JP)*

**License** Creative Commons BY 3.0 Unported license
© Shinpei Hayashi
**Joint work of** Hayashi, Shinpei; Thangthumachit, Sirinut; Saeki, Motoshi
**Main reference** S. Hayashi, S. Thangthumachit, M. Saeki, "REdiffs: Refactoring-Aware Difference Viewer for Java," in Proc. of the 20th Working Conf. on Reverse Engineering (WCRE'13), pp. 487–488, IEEE, 2013.
**URL** http://dx.doi.org/10.1109/WCRE.2013.6671331

Comparing and understanding differences between old and new versions of source code are necessary in various software development situations. However, if changes are tangled with refactorings in a single revision, then the resulting source code differences are more complicated. We propose an interactive difference viewer which enables us to separate refactoring effects from source code differences for improving the understandability of the differences.

## 5.6 Refactoring via Pretty-Printing

*Jongwook Kim (University of Texas – Austin, US)*

**License** Creative Commons BY 3.0 Unported license
© Jongwook Kim

We demonstrate a new refactoring engine called Relativistic Reflective Refactoring that uses a projection or pretty-printer technology based on Simonyi's Intentional Programming. Using main-memory databases to encode containment and inheritance relationships among program elements (like classes, methods, fields, and interfaces), we can encode the changes made by refactorings within the database itself, and not modifying existing program ASTs. By displaying the contents of the database through ASTs, we can emulate many different and classical refactorings and design patterns without using "program transformation" technologies.

## 5.7 Interactive Quick Fix

*Emerson Murphy-Hill (North Carolina State University, US)*

**License** Creative Commons BY 3.0 Unported license
© Emerson Murphy-Hill
**Joint work of** Song, Yoonki; Barik, Titus; Johnson, Brittany; Murphy-Hill, Emerson
**URL** https://www.youtube.com/watch?v=y4BhIF0mMZg

Quick fixes are a great way to fix problems when the number of possible solutions are easily enumerable. However, when this is not the case, they fail to adequately support the programmer. In this demo, I talk about our approach called Interactive Quick Fix, which allows a developer to benefit from the structured help of tools yet still explore the full design space of the solution.

## 5.8 Cloud Refactoring

*Eli Tilevich (Virginia Polytechnic Institute – Blacksburg, US)*

**License** Creative Commons BY 3.0 Unported license
© Eli Tilevich
**Joint work of** Kwon, Young-Woo; Tilevich, Eli
**Main reference** Y.-W. Kwon, E. Tilevich, "Cloud refactoring: automated transitioning to cloud-based services," Automated Software Engineering, 21(3):345–372, 2013.
**URL** http://dx.doi.org/10.1007/s10515-013-0136-9

We demonstrate a set of Cloud Refactoring techniques, which we have implemented as automated, IDE- assisted program transformations that render a portion of an application's functionality accessible remotely as a Web service. In particular, we show how a programmer can extract services, add fault tolerance functionality, and adapt client code to invoke cloud services via refactoring transformations integrated with a modern IDE. The running example refactors a bioinformatics application to use a remote sequence alignment service.

## 5.9 A Universal Type Qualifier Inference System

*Mohsen Vakilian (University of Illinois – Urbana, US)*

**License** Creative Commons BY 3.0 Unported license
© Mohsen Vakilian
**Joint work of** Vakilian, Mohsen; Phaosawasdi, Amarin; Johnson, Ralph E.

Type qualifiers augment an existing type system to check more properties, such as safety against null dereferences and SQL injections. To get the benefits of type qualifiers, programmers have to add type qualifiers to the source code. Realizing the burden of manually adding type qualifiers to existing code, researchers have proposed inference systems for each type qualifier system. Each of these inference systems operates in the batch mode, gives little control to the programmer, and is limited to a single type qualifier system. A combination of two concepts, compositional refactoring and speculative analysis, enabled us to develop the first universal type qualifier inference system called Cascade. Cascade is an interactive system that achieves universality by repeatedly invoking the checker for a given type qualifier system and proposing a composition of changes to fix the errors reported by the checker.

## 5.10 Rascal for Experimenting with New Intermediate Formats for Source Code Analysis

*Jurgen Vinju (CWI – Amsterdam, NL)*

**License** Creative Commons BY 3.0 Unported license
© Jurgen Vinju
**Joint work of** Klint, Paul; Van der Storm, Tijs; Vinju, Jurgen
**Main reference** P. Klint, T. van der Storm, J. J. Vinju, "Rascal: A Domain Specific Language for Source Code Analysis and Manipulation," in Proc. of the 9th IEEE Int'l Working Conf. on Source Code Analysis and Manipulation (SCAM'09), pp. 168–177, IEEE, 2009; pre-print available from author's webpage.
**URL** http://dx.doi.org/10.1109/SCAM.2009.28
**URL** http://homepages.cwi.nl/~jurgenv/papers/SCAM-2009.pdf

In this live coding demonstration we demonstrate the power of Rascal as a language to introduce new intermediate representations, extracting these from source code, then analyzing them. Models of source code in Rascal are all represented as immutable data: terms in many-sorted algebras, parse trees over context-free grammars, sets, relations, maps, etc.

As an example we translated Java to the Object Flow Language [Tonella] and then extracted an over-approximated object flow graph from this as a binary relation. Then we visualize this graph by exporting a graphviz dot graph.

We claim that experimenting with new representations and new source code extractors and new analysis requires hardly any boilerplate using the Rascal language, which makes it more fun and more effective to explore new ideas in refactoring.

## 5.11 How Can We Do Better than Search and Replace?

*Jan Wloka (IBM Research GmbH – Zürich, CH)*

**License** Creative Commons BY 3.0 Unported license
© Jan Wloka

When using a refactoring tool to automate incremental design improvements in mixed-language programs you can get the impression that current tools are very limited in how refactoring targets can be selected and when a refactoring can be applied. The tool expects a single program element as target, e.g. a method declaration, before it tries to find and change all referencing elements in the program. The resolution of declaration-reference bindings is difficult and it is often impossible for a program analysis to determine whether a certain change preserves program behavior.

Future refactoring tools can overcome these limitations by enabling their users to provide (non- determinable) declaration-reference bindings. Developers would use the tool to change declaring program elements and their bindings separately. The developer would specify a search term and a substitution template in a unified pattern language, and the tool would search, preview and then consistently refactor all matching bindings in the different programming language files. The refactoring tool would know how to match and refactor the individual elements of each programming language with the unified search and substitution template provided by the developer.

The use of syntax trees for each supported language would allow for context-dependent matches and substitutions and introduce fewer programming errors than textual search-and-replace. Even if possibly not behavior preserving, refactorings automated by such a tool would enable developers to perform vast and complex changes in a consistent way, and automated tests would catch unintended behavioral changes.

Until such a refactoring tool is available, developers will continue to change mixed-language programs with search-and-replace in their favorite editor.

# 6 Working Groups

## 6.1 User Experience Breakout: Dimensions of Refactoring

*Emerson Murphy-Hill (North Carolina State University, US)*

**License** Creative Commons BY 3.0 Unported license
© Emerson Murphy-Hill
**URL** https://docs.google.com/document/d/1GmEtQWz4xUnBdpDucfJV0fnUMaIVmWxIzPEWfnNJuIw/edit

The user experience breakout group (about 10 people, including Emerson) brainstormed "dimensions of refactoring". Later, Oscar and Jurgen pulled Emerson aside to augment those dimensions with some Oscar came up with based on the industry panel. The results are in the linked Google Doc. Figure 1 shows the dimensions visualized by Emerson, Friedrich, Jurgen, and Oscar.

## 6.2 User Experience Breakout: The Future of Refactoring

*Emerson Murphy-Hill (North Carolina State University, US)*

**License** Creative Commons BY 3.0 Unported license
© Emerson Murphy-Hill
**URL** https://docs.google.com/document/d/1xHqizIEjVZaYxsTfTFVv1IEjyPkt9slcds-KYVkZPH8/edit

The user experience refactoring group met to discuss the future of refactoring. We discussed current problems, solutions, and future opportunities. The results are in the linked Google Doc.

## 6.3 Plenary Discussion on Refactoring in Education, Corpora and Benchmarks

*Thompson, Simon J.*

**License** Creative Commons BY 3.0 Unported license
© Simon J. Thompson
**URL** https://docs.google.com/document/d/1q3E1n6tJbX7W-V0jjtgObpoR8P37JL2ja8rj4yC2cFQ/edit?usp=sharing

In a plenary discussion, the seminar talked about the various roles of refactoring in higher education, and the roles of corpora and benchmarks in refactoring research.

## 6.4 Novel Applications of Refactoring Breakout

*Bill Opdyke (JP Morgan Chase – Chicago, US)*

**License** Creative Commons BY 3.0 Unported license
© Bill Opdyke
**URL** https://drive.google.com/folderview?id=0B7DV-T4_2mpKcXNYb0VJR3NqN28&usp=sharing

During the discussion breakout session, the "Novel Applications of Refactoring" participants listed several opportunity areas (problems/ challenges and open issues):

- How to introduce refactoring to children.
- Teaching that you don't need to have it right the first time.

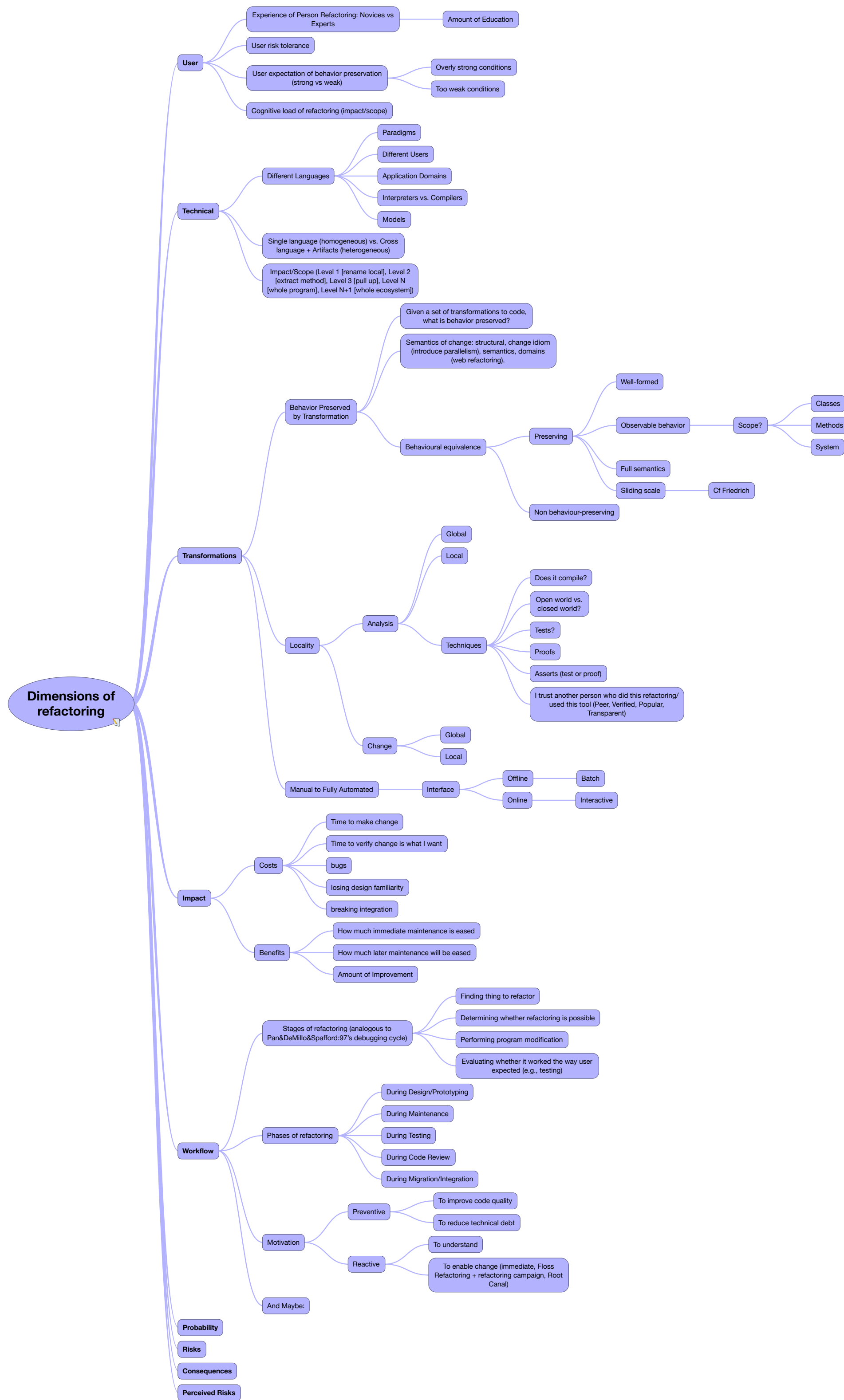


**Figure 1** Dimensions of Refactoring.

- Refactoring (in the mobile area), tailored to environments with special constraints (e.g., security, screen space, battery power, efficiency, latency, data usage, network connectivity, preferences, heterogeneity).
- Use of resources more efficiently (green computing).
- Refactoring to distribute resources on the cloud.
- Refactoring of big data.
- Expressing refactorings as goals to non-expert developers or users.
- Globalization, internationalization (cultural awareness, use of color and fonts, other factors).
- How to transfer insights of basic refactoring research to new domains.

The breakout group noted several existing solutions:

- Self-adaptation and self-healing to handle dynamic user of resources.
- Software product lines for internationalization.
- Refactoring for accessibility.

The breakout group also noted several works in progress:

- Refactoring for improving user responsiveness on mobile devices (extracting long running – blocking u/p computation from the UI event to asynchronous task).
- Annotation refactoring (before and after examples).
- Record and replay of web macros.
- Finding bad usability smells (as part of solving limitations of systems with respect to internationalization).

## 6.5 Refactoring Tools and Meta-Tools

*Max Schäfer (Semmle Ltd., Oxford, UK)*

**License** Creative Commons BY 3.0 Unported license
© Max Schaefer

The group discussed open problems in refactoring tools, existing solutions and work in progress, and ideas for approaching the open problems. It was agreed that the main problem facing authors of refactoring tools is the great complexity of real-world languages and code bases. Moreover, languages continue to evolve and thus become more complex. At the same time, commercial tool vendors seem to have little interest in improving existing refactoring tools. On the research side, many participants found that there was too little exchange and collaboration between, thus leading people to solve the same problems over and over again. This shows the importance of broad-based seminars such as this one.

## 7 Industry Roundtable

We held one industry panel, moderated by Bill Griswold, and attended by Robert Bowdidge, Loius Wasserman, Don Roberts, John Brant, Ira Baxter, and Bill Opdyke. Panelists were asked: "Tell us one surprising fact about industry refactoring that you'd like academics to know."

Here are some issues that came out during the session:

### Refactoring Process

- Why? (what's the trigger?)
- When? (on-the fly Agile, post facto re-architecting, etc.)
- Scope? (small scale, system-wide, etc.)
- How? (tools and techniques, meaning-preserving or not)
- Who? (everybody, specialists, etc.)

### Critical Analysis

- Surprising fact(s)
- Barriers to adoption
- Skill set required to perform refactoring (and how common that skill set is)
- How important to the on-going success of the software
- How related or compared to re-development and other maintenance

## Participants

- Don Batory
University of Texas – Austin, US
- Ira D. Baxter
Semantic Designs – Austin, US
- Andrew P. Black
Portland State University, US
- Robert Bowdidge
Google Inc. –
Mountain View, US
- John Brant
The Refactory Inc. – Urbana, US
- Caius Brindescu
Oregon State University, US
- Marcio Cornelio
Federal University of
Pernambuco – Recife, BR
- Stephan Diehl
Universität Trier, DE
- Danny Dig
Oregon State University, US
- Ran Ettinger
Ben Gurion University –
Beer Sheva, IL
- Alejandra Garrido
University of La Plata, AR
- Rohit Gheyi
Universidade Federal – Campina
Grande, BR
- William G. Griswold
University of California –
San Diego, US
- Shinpei Hayashi
Tokyo Institute of Technology, JP
- Felienne Hermans
TU Delft, NL
- Jongwook Kim
University of Texas – Austin, US
- Huiqing Li
University of Kent, GB
- Francesco Logozzo
Microsoft Res. – Redmond, US
- Kim Mens
University of Louvain, BE
- Tom Mens
University of Mons, BE
- Naouel Moha
Univ. of Quebec – Montreal, CA
- Emerson Murphy-Hill
North Carolina State Univ., US
- Oscar M. Nierstrasz
Universität Bern, CH
- Bill Opdyke
JP Morgan Chase – Chicago, US
- Chris Parnin
Georgia Inst. of Technology, US
- Javier Perez
University of Antwerp, BE
- Veselin Raychev
ETH Zürich, CH
- Don Roberts
University of Evansville, US
- Max Schaefer
Semmle Ltd. – Oxford, GB
- Gustavo Soares
Universidade Federal – Campina
Grande, BR
- Peter Sommerlad
Hochschule für Technik –
Rapperswil, CH
- Friedrich Steimann
Fernuniversität in Hagen, DE
- Kathryn T. Stolee
Iowa State Univ. – Ames, US
- Volker Stolz
University of Oslo, NO
- Simon J. Thompson
University of Kent, GB
- Eli Tilevich
Virginia Polytechnic Institute –
Blacksburg, US
- Frank Tip
University of Waterloo, CA
- Mohsen Vakilian
Univ. of Illinois – Urbana, US
- Jurgen Vinju
CWI – Amsterdam, NL
- Louis Wasserman
Google Inc. –
Mountain View, US
- Jan Wloka
IBM Res. GmbH – Zürich, CH