ACCU 2016

19 – 23 April,
Bristol, UK

Refactoring: 25 Years On

Chris Simons

@chrislsimons
chris.simons@uwe.ac.uk
www.cems.uwe.ac.uk/~clsimons/

Interactive workshop

Part 1

What is contemporary refactoring?

Part 2

What tool support exists, and what is needed?

A sense of journey, so first, the fossil record and a little archaeology…

FOCUS: REFACTORING

The Birth of Refactoring

A Retrospective on the Nature of High-Impact Software Engineering Research

William G. Griswold, University of California, San Diego

William F. Opdyke, JPMorgan Chase

// This article reflects on how the idea of refactoring arose and was developed in two PhD dissertations. The analysis provides useful insights for both researchers and practitioners seeking high impact in their work. //

Griswold, W.G. and Opdyke, W.F., 2015. The Birth of Refactoring: A Retrospective on the Nature of High-Impact Software Engineering Research. *IEEE Software*, *32*(6), pp. 30-38.

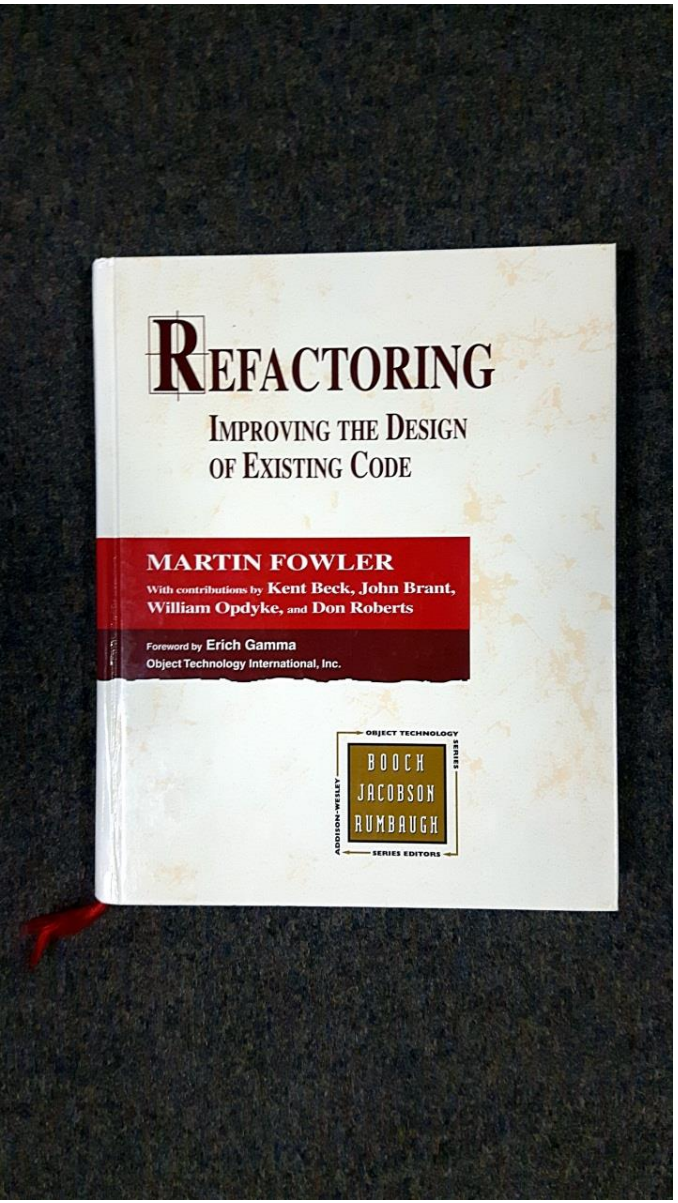
ABOUT THE AUTHORS

WILLIAM G. GRISWOLD is a professor of computer science and engineering at the University of California, San Diego. His research interests include software engineering, ubiquitous computing, and educational technology. Griswold was a pioneer of software refactoring. Later, he built ActiveCampus, an early mobile location-aware system. His CitiSense project is investigating technologies for low-cost ubiquitous real-time air quality sensing. Griswold received a PhD in computer science from the University of Washington. He's a member of the IEEE Computer Society and ACM and a former chair of ACM SIGSOFT. Contact him at wgg@cs.ucsd.edu.

WILLIAM F. OPDYKE works at JPMorgan Chase, where he focuses on agile and related technology training. His research interests include refactoring applied to agile development and legacy system evolution, and organizational and process improvements to support software innovation. Opdyke received a PhD in computer science from the University of Illinois at Urbana-Champaign; his doctoral research led to the foundational thesis in object-oriented refactoring. Contact him at opdyke@acm.org.

Software refactoring is the systematic practice of improving application code's structure without altering its behavior. An attendee of the 199(ented I Practica where O per to u perhaps ing was perience his code ing abou process ing rese followed software plicitly a years lat central part of software development practice. It's a core element of agile methodologies, and most professional IDEs include refactoring tools.

where Opdyke presented the first paper to use the term "refactoring,"[1] perhaps summed it up best: Refactoring was something that he as an experienced developer naturally did to his code, without consciously thinking about it; this research made that process explicit. Our early refactor-



"Refactoring is the process of changing a software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs."

(page xvi)

"Improving the design after it has been written."

(1999)

List of Refactorings

Smell	Common Refactorings
Alternative Classes with Different Interfaces, p. 85	*Rename Method (273), Move Method (142)*
Comments, p. 87	*Extract Method (110), Introduce Assertion (267)*
Data Class, p. 86	*Move Method (142), Encapsulate Field (206), Encapsulate Collection (208)*
Data Clumps, p. 81	*Extract Class (149), Introduce Parameter Object (295), Preserve Whole Object (288)*
Divergent Change, p. 79	*Extract Class (149)*
Duplicated Code, p. 76	*Extract Method (110), Extract Class (149), Pull Up Method (322), Form Template Method (345)*
Feature Envy, p. 80	*Move Method (142), Move Field (146), Extract Method (110)*
Inappropriate Intimacy, p. 85	*Move Method (142), Move Field (146), Change Bidirectional Association to Unidirectional (200), Replace Inheritance with Delegation (352), Hide Delegate (157)*
Incomplete Library Class, p. 86	*Introduce Foreign Method (162), Introduce Local Extension (164)*
Large Class, p. 78	*Extract Class (149), Extract Subclass (330), Extract Interface (341), Replace Data Value with Object (175)*
Lazy Class, p. 83	*Inline Class (154), Collapse Hierarchy (344)*
Long Method, p. 76	*Extract Method (110), Replace Temp with Query (120), Replace Method with Method Object (135), Decompose Conditional (238)*

Smell	Common Refactorings
Long Parameter List, p. 78	*Replace Parameter with Method (292), Introduce Parameter Object (295), Preserve Whole Object (288)*
Message Chains, p. 84	*Hide Delegate (157)*
Middle Man, p. 85	*Remove Middle Man (160), Inline Method (117), Replace Delegation with Inheritance (355)*
Parallel Inheritance Hierarchies, p. 83	*Move Method (142), Move Field (146)*
Primitive Obsession, p. 81	*Replace Data Value with Object (175), Extract Class (149), Introduce Parameter Object (295), Replace Array with Object (186), Replace Type Code with Class (218), Replace Type Code with Subclasses (223), Replace Type Code with State/ Strategy (227)*
Refused Bequest, p. 87	*Replace Inheritance with Delegation (352)*
Shotgun Surgery, p. 80	*Move Method (142), Move Field (146), Inline Class (154)*
Speculative Generality, p. 83	*Collapse Hierarchy (344), Inline Class (154), Remove Parameter (277), Rename Method (273)*
Switch Statements, p. 82	*Replace Conditional with Polymorphism (255), Replace Type Code with Subclasses (223), Replace Type Code with State/Strategy (227), Replace Parameter with Explicit Methods (285), Introduce Null Object (260)*
Temporary Field, p. 84	*Extract Class (149), Introduce Null Object (260)*

REFACTORING

Refactoring is basically a straightforward technique. However, researchers have introduced additional concepts that paint a more complex picture of refactoring in development workflows.

Martin Fowler defined refactoring as making behavior-preserving design improvements right after adding and testing new functionality.[1] But developers can also refactor code opportunistically—recurrent "upkeep refactoring" has been called "floss refactoring," "litter-pickup refactoring," or "comprehension refactoring."[2–4] Adding new functionality might trigger "preparatory refactoring" that makes the changes easier.[4]

Leppanen, M., Makinen, S., Lahtinen, S., Sievi-Korte, O., Tuovinen, A.P. and Mannisto, T., 2015. Refactoring - A Shot in the Dark? *IEEE Software*, *32*(6), pp. 62-70.

Without floss refactoring, developers might need to apply "root canal refactoring" or "planned refactoring"—that is, change large parts of the code, which requires specific backlog items.[3,4] Fowler considered a recurring need for planned refactoring as a bad smell.[4] He suggested that architectural changes could be done as "long-term refactoring," in which developers gradually and knowingly migrate the system to a new architecture by applying opportunistic changes. Michael Stal recommended recurrent refactoring of the architectural design directly.[5] Although everyday work can easily include upkeep refactoring, large-scale or architectural refactoring is difficult to justify to stakeholders, especially customers, who might not understand the nature of software.[2,4,5]

Leppanen, M., Makinen, S., Lahtinen, S., Sievi-Korte, O., Tuovinen, A.P. and Mannisto, T., 2015. Refactoring - A Shot in the Dark? IEEE *Software*, *32*(6), pp. 62-70.

The views on refactoring have been mixed. Some people see refactoring as beneficial, even a success factor, whereas others still strongly advocate the "if it ain't broke, don't fix it" mentality.[6] Developers feel that refactorings' benefits are difficult to measure and sell to management.[7] Developers don't care to make quality measurements; they want to tackle the concrete problems in code, rather than trying to improve internal code quality metrics.[8] There are conflicting results if refactoring actually improves code metrics.[9]

Leppanen, M., Makinen, S., Lahtinen, S., Sievi-Korte, O., Tuovinen, A.P. and Mannisto, T., 2015. Refactoring - A Shot in the Dark? IEEE *Software*, *32*(6), pp. 62-70.

Leppanen, M., Makinen, S., Lahtinen, S., Sievi-Korte, O., Tuovinen, A.P. and Mannisto, T., 2015. Refactoring - A Shot in the Dark? IEEE *Software*, *32*(6), pp. 62-70.

Benefits	Risks
• Easier future development • Understandability • Reuse • Improving quality attributes such as performance • Boosting morale and motivation	• Breaking something • Causing externally visible changes • Worsening the code quality • Wasting time and effort

FIGURE 1. Refactoring's benefits and risks, according to the interviewees.

BREAK OUT DISCUSSIONS, 15-20 MINUTES

25 years on… part 1

Question 1

What is the intent of contemporary refactoring?

For example, if the original intent of refactoring was focussed on architectural and design-level restructuring, is it now the case that refactoring relates to more fine-grained (code) changes? Have ‘extract xxx’, ‘push up xxx’ and ‘pull down xxx’ patterns been superseded with a more fine-grained duplication avoidance?

Question 2

What is the philosophy of contemporary refactoring?

For example, if the original philosophy of refactoring was that cleaning code must preserve behaviour, is this strictly necessary at a fine-grained code level? Might minor changes in program behaviour be tolerated for the sake of clean code, improved elegance and comprehension in design, code and test?

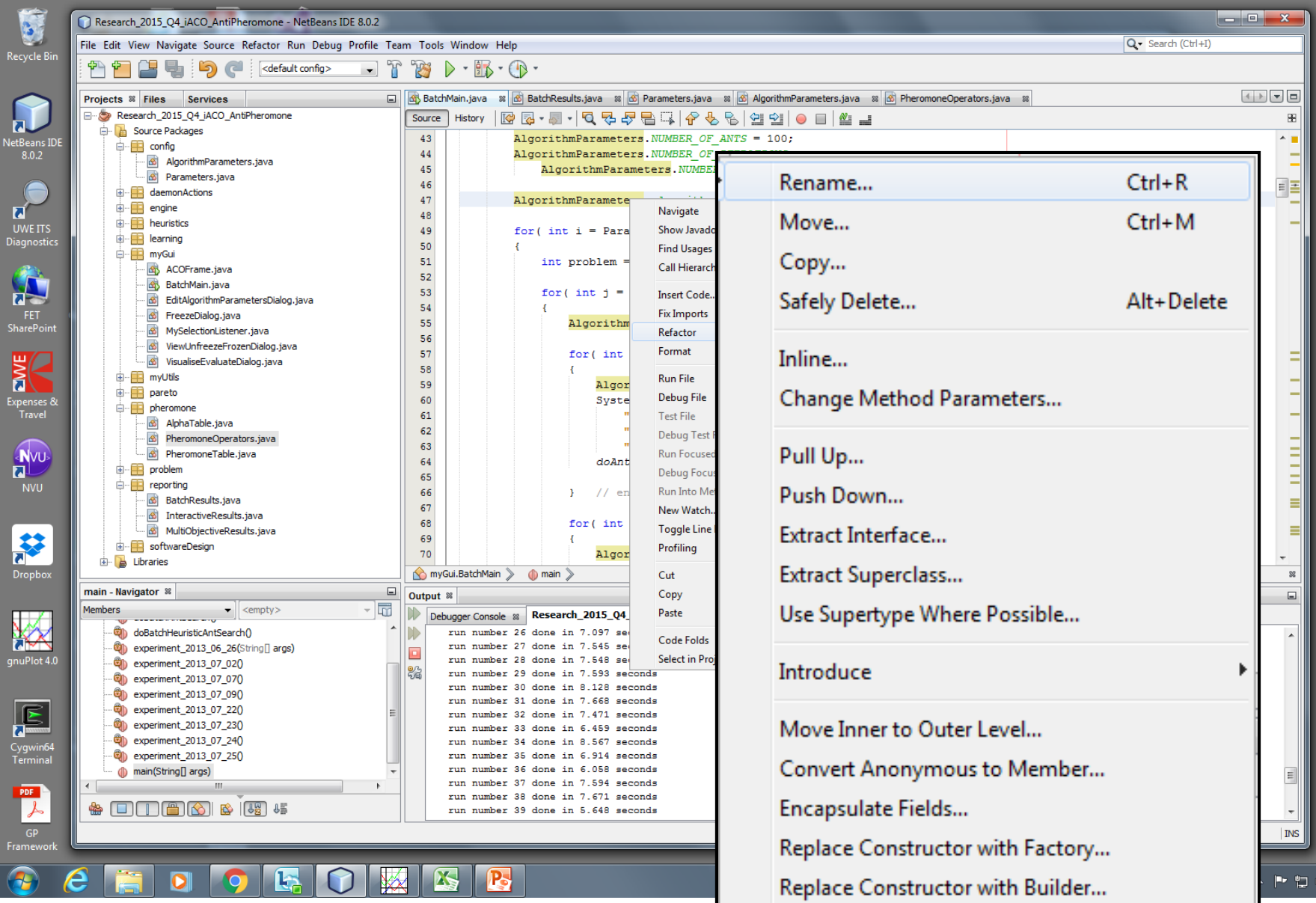
Part 2
Tool support for refactoring

Manual

(Reactive Tool)

Automated

(Proactive tool)



Netbeans for Java

Visual C++ Team Blog

C++ Core Guidelines Checkers available for VS 2015 Update 1

Andrew Pardoe [MSFT] 3 Dec 2015 11:57 AM 30

RATE THIS

[This post was written by Andrew Pardoe and Neil MacIntosh]

Back in September at CppCon 2015 Neil announced that we would be shipping new code analysis tools for C++ that would enforce some of the rules in the C++ Core Guidelines. (A video of the talk is available here: https://www.youtube.com/watch?v=rKlHvAw1z50 and slides are available on the ISOCpp GitHub repo.)

Earlier this week we made the first set of those code analysis tools freely available as a NuGet package that can be installed by users of Visual Studio 2015 Update 1. The package currently contains checkers for the Bounds and Type profiles. Tooling for the Lifetime profile demonstrated in Herb Sutter's plenary talk (video at https://www.youtube.com/watch?v=hEx5DNLWGgA) will be made available in a future release of the code analysis tools.

The package is named "Microsoft.CppCoreCheck", and a direct link to the package is here: http://www.nuget.org/packages/Microsoft.CppCoreCheck.

http://blogs.msdn.com/b/vcblog/archive/2015/12/03/c-core-guidelines-checkers-available-for-vs-2015-update-1.aspx

The Myth about Robustness

This myth is that refactoring tools are robust.

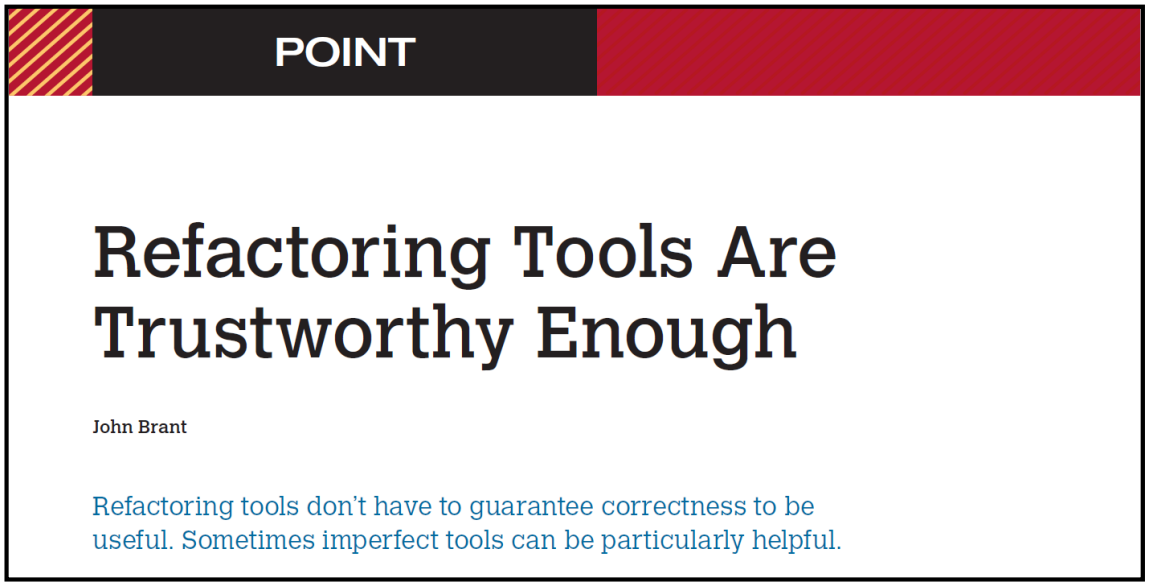
Robustness is a desirable property of software development tools—refactoring tools included. Developers won't use tools that seem unreliable. So, the widespread use of refactoring tools speaks to their

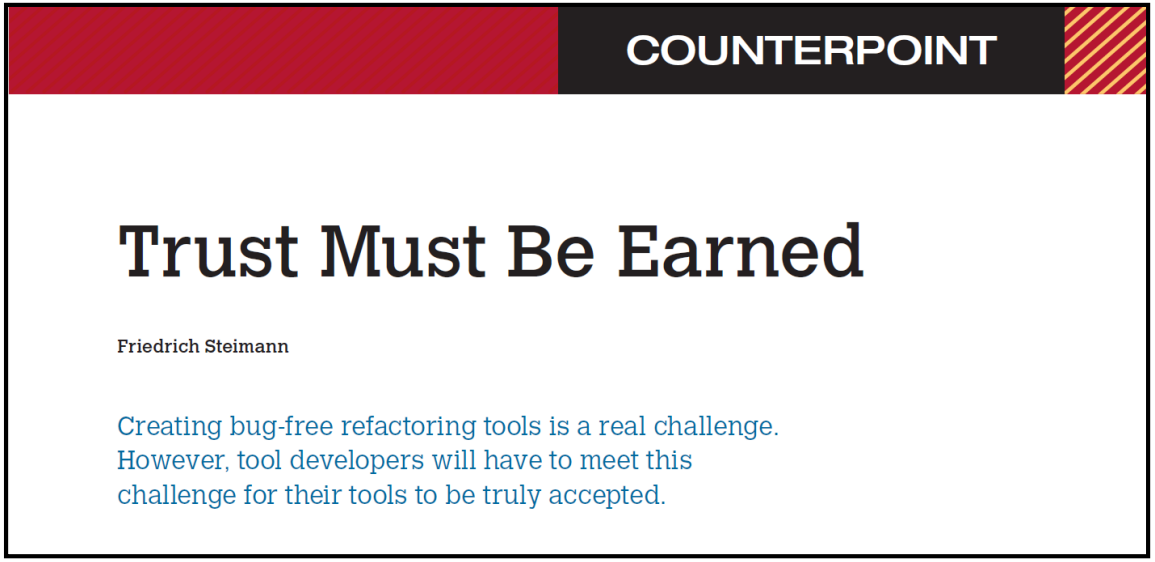
> Refactoring tools aren't error-free. They work just well enough to be useful, and they break in relatively unimportant ways.

mon features. On average, approxi- mately 2 percent of the refactoring

find bugs in refactored code. This simple implementation is neces-

Hafiz, M. and Overbey, J., 2015.
Refactoring Myths.
IEEE Software, *32*(6), pp.39-43.





IEEE Software, *32*(6), pp. 80-83.

BREAK OUT DISCUSSIONS, ANOTHER 15-20 MINUTES

25 years on… part 2

Question 1

What is the state of current tool support for refactoring?

For example, how useable is contemporary tool support? How robust? Are refactoring tools error free? Might they even introduce errors in design and code? After refactoring, is a simple syntax check sufficient?

Question 2

What tool support is needed?

For example, could automation speed up refactoring? How proactive could refactoring tools be? Would proactive tools be trusted? Might they be dynamic and adaptive? Should they be prominent in development IDEs, possibly as recommendation engines, or might they work offline from a command line?

RELATED WORK IN REFACTORING SURVEYS

Tom Mens and Tom Tourwe reviewed refactoring research in terms of the refactoring activities supported, the techniques and formalisms for supporting these activities, and refactoring's effect on the software process.[1] The literature also includes surveys and evaluations of refactoring tools (for example, the technical report by Jocelyn Simmonds and Tom Mens[2]), but they don't focus on challenges to adopting refactoring in an industrial context.

At Microsoft, Miryung Kim and her colleagues surveyed and interviewed 328 developers and analyzed version history data to identify refactoring benefits and challenges.[3] Respondents cited six key risk factors: regression bugs, code churns, merge conflicts, time taken from other tasks, difficulty performing code reviews after refactoring, and overengineering.

Emerson Murphy-Hill and Andrew Black surveyed 112 Agile Open Northwest conference attendees.[4] They found that professional programmers underused refactoring tools and that better, more usable, refactoring tools were needed. Our survey (see the main article) echoes these findings; we're trying to discover the specific problems architects and their teams face while using refactoring tools.

Finally, Aiko Yamashita and Leon Moonen surveyed 85 software professionals on code smells and related tooling.[5] They reported a finding similar to ours: refactoring tools should provide better support for refactoring suggestions.

References

1. T. Mens and T. Tourwe, "A Survey of Software Refactoring," *IEEE Trans. Software Eng.*, vol. 30, no. 2, 2004, pp. 126–139.
2. J. Simmonds and T. Mens, *A Comparison of Software Refactoring Tools*, tech. report vub-prog-tr-02-15, Programming Technology Lab, Vrije Univ. Brussel, 2002.
3. M. Kim, T. Zimmermann, and N. Nagappan, "An Empirical Study of Refactoring Challenges and Benefits at Microsoft," *IEEE Trans. Software Eng.*, vol. 40, no. 7, 2014, pp. 633–649.
4. E. Murphy-Hill and A.P. Black, "Refactoring Tools: Fitness for Purpose," *IEEE Software*, vol. 25, no. 5, 2008, pp. 38–44.
5. A. Yamashita and L. Moonen, "Do Developers Care about Code Smells? An Exploratory Survey," *Proc. 20th Working Conf. Reverse Eng.* (WCRE 13), 2013, pp. 242–251.

Sharma, T., Suryanarayana, G. and Samarthyam, G., 2015.
Challenges to and Solutions for Refactoring Adoption: An Industrial Perspective.
IEEE Software, *32*(6), pp.44-51.

Chapter
Search-Based Software Engineering
Volume 9275 of the series Lecture Notes in Computer Science pp 47-61

Date: 28 July 2015

Search-Based Refactoring: Metrics Are Not Enough

Chris Simons ✉ , Jeremy Singer, David R. White

Abstract

Search-based Software Engineering (SBSE) techniques have been applied extensively to refactor software, often based on metrics that describe the object-oriented structure of an application. Recent work shows that in some cases applying popular SBSE tools to open-source software improved version of the software as assessed by some subjective criteria. Through a survey of professionals, we investigate the relationship between popular SBSE refactoring metrics and the subjective opinions of software engineers. We find little or no correlation between the two. Through qualitative analysis, we find that a simple factors that are not amenable to measurement via metrics. We recommend that future SBSE refactoring research should incorporate information about the dynamic behaviour of software, and conclude that a human-in-the-loop approach may be the only way to refactor software in a manner helpful to an engineer.

By the way, I’m very interested in how AI can learn and search for refactoring suggestions…

The Journal of Systems and Software 113 (2016) 101–113

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

What recommendation systems for software engineering recommend: A systematic literature review

Marko Gasparic*, Andrea Janes

Free University of Bolzano, Italy

ARTICLE INFO

Article history:
Received 19 November 2014
Revised 14 November 2015
Accepted 17 November 2015
Available online 26 November 2015

Keywords:

ABSTRACT

A recommendation system for software engineering (RSSE) is a software application that provides information items estimated to be valuable for a software engineering task in a given context. Present the results of a systematic literature review to reveal the typical functionality offered by existing RSSEs, research gaps, and possible research directions. We evaluated 46 papers studying the benefits, the data requirements, the information and recommendation types, and the effort requirements of RSSE systems. We include papers describing tools that support source code related development published between 2003 and 2013. The results show that RSSEs typically visualize source code artifacts. They aim to improve system quality, make the development process more efficient and less expensive, lower developer's cognitive load, and help developers to make better decisions. They mainly support reuse actions and debugging, implementation, and maintenance phases. The majority of the systems are reactive. Unexploited opportunities lie in the development of recommender systems outside the source code domain. Furthermore, current RSSE systems use very limited context information and rely on simple models. Context-adapted and proactive behavior could improve the acceptance of RSSE systems in practice.

Elegant Object-Oriented Software Design via Interactive, Evolutionary Computation

Christopher L. Simons and Ian C. Parmee

***Abstract*—Design is fundamental to software development but can be demanding to perform. Thus, to assist the software designer, evolutionary computing is being increasingly applied using machine-based, quantitative fitness functions to evolve software designs. However, in nature, elegance and symmetry play a crucial role in the reproductive fitness of various organisms. In addition, subjective evaluation has also been exploited in interactive evolutionary computation (IEC). Therefore, to investigate the role of elegance and symmetry in software design, four novel elegance measures are proposed which are based on the evenness of distribution of design elements. In controlled experiments in a dynamic IEC environment, designers are presented with visualizations of object-oriented software designs, which they rank according to a subjective assessment of elegance. For three out of the four elegance measures proposed, it is found that a significant correlation exists between elegance values and reward elicited. These three elegance measures assess the evenness of distribution of 1) attributes and methods among classes; 2) external couples between classes; and 3) the ratio of attributes to methods. It is concluded that symmetrical elegance is in some way significant in software design, and that this can be exploited in dynamic, multiobjective IEC to produce elegant software designs.**

the results of evolutionary search supported by interactive software agents in which a population of object-oriented software design individuals is evolved with preference-based designer interaction.

The search techniques that are reported previously rely solely on quantitative computational measures of fitness to direct search and exploration. However, just as evolutionary computing draws inspiration from evolutionary processes in nature, is it also possible to draw from nature to specifically address the "quality" or "appearance" of an individual? Certainly, the influence of symmetry of appearance in the reproductive fitness of an organism has been noted by evolutionary biologists. For example, Schilthuizen [6] explains that "the significance of symmetry was only made clear with the discovery that stress and disease make it harder for an individual to develop a perfectly symmetric body. Small differences on either side of an imaginary mid-plane therefore betray genetic quality, and potential mates use this to gauge each other's desirability. Put simply, symmetry is sexy." Drawing from evolutionary biology, it seems likely

ACCU 2016

19 – 23 April,
Bristol, UK

Thank you!

Chris Simons

@chrislsimons
chris.simons@uwe.ac.uk
www.cems.uwe.ac.uk/~clsimons/