The Spartanizer: Massive Automatic Refactoring

Yossi Gil
Computer Science Dept. The Technion—I.I.T
Taub Building, Haifa 3200003, Israel
yogi@cs.technion.ac.il

Matteo Orrù
Computer Science Dept. The Technion—I.I.T
Taub Building, Haifa 3200003, Israel
matteo.orru@cs.technion.ac.il

***Abstract*—The *Spartanizer* is an eclipse plugin featuring over one hundred and fifty refactoring techniques, all aimed at reducing various size complexity of the code, without changing its design, i.e., inheritance relations, modular structure, etc. Typical use case of the Spartanizer is in an *automatic mode*: refactoring operations are successively selected and applied by the tool, until the code is reshaped in *spartan style* (a frugal coding style minimizing the use of characters, variables, tokens, etc.). The Spartanizer demonstrates the potential of *automatic* refactoring: tens of thousands of transformations are applied in matter of seconds, chains of dependent applications of transformations with tens of operations in them, significant impact on code size, and extent reaching almost every line of code, even of professional libraries.**

I. Introduction

Refactoring is the practice of altering the structure of the code without changing its observable behavior, in the sake of improving design, and in particular, extensibility and modularity [1] [2].

The *Spartanizer*[1], an interactive Eclipse plugin, which currently offers over 150 refactorings techniques whose application is geared towards achieving the end of *Spartan programming*. Spartan programming [3], [4] is a methodology and coding style which, in the spirit of visionaries like Dijkstra [5], focuses on *minimalism*. Spartan code makes frugal use of code elements: lines, characters, arguments, nesting, use of conditional and iterative control, etc. Spartan code is the software equivalent of laconic speech: *saying the most in fewest words*.

We stress that minimalism is not the same as obfuscation, even though part of the obfuscating process is minimizing code length. Obfuscated code takes pride in using the programming language in unexpected ways. In contrast, spartan code tries to follow familiar programming idioms and establish uniformity whenever possible.

Minimalism is also very different from code-golfing[2]. Spartan code is code that follows the same algorithmic structure of any other code, except that it tries to use fewer words and less complex verbose grammatical structures.

The Spartanizer produces *spartanization tips*, which are suggestions to make *nano*-refactorings: small changes of the code to help it say the same, but using fewer characters, tokens, etc. Tips include, e.g., removing redundant curly brackets, offers to use terse, yet sensible variable naming scheme, replacing a conditional `if...else` statement with the *ternary* `· ? · : ·` operator, replace re-head a `while` loop with a `for (·;·; ·)`, inline a local variable used only once, and many more nano-refactorings of other varieties. All these varieties occur at the method level. The Spartanizer has also a programmer-activated extract method component, which is not discussed here.

Programmers can follow the tips manually, witnessing how their code changes, in almost infinitesimally small steps, into a more compact form. Alternatively, the programmer can *spartanize*[3] the code automatically: The Spartanizer can run in automatic mode, by repeatedly following tips, in its own preferred order, until no more tips can be followed. At a typical automatic run over non-spartanized code, 20–40 tips are executed per class, touching around 80% of lines of code. In batch spartanization, even of large projects, is quite efficient. Spartanizing a version of the **guava** project applying a total of 14,816 tips in 16 rounds on 526 files took 06:40 minutes on a contemporary laptop. Despite the large number of transformations, which, like all refactorings, strike a gentle balance between conservativeness and optimism, failure rate is small, and recovery from over-optimistic transformation is easy. Our experience is that in many projects the automatic spartanization never breaks the code, nor tests. Moreover, when code breaks, it tends to do so in a few selected files. Fixing these is done by manual spartanization, to identify and then disable spartanization tips that break code. For example, automatic spartanization of **junit** broke code at 14 out of 422 files. Manually correcting this errors took less than five minutes.

The Spartanizer works with a toolbox equipped with tippers. A tipper is a reification of transformation rule, specifically, a (nano) refactoring technique, or more specifically, a generator of spartanization tips. This modular structure makes it possible to easily add tippers: which are just classes that implement the `Tipper` protocol. The first version of the plugin (circa 2013) had six tippers. Their number increased to over 150 as of print.

Each of the spartanization tips produced by the tippers improves at least one of the code size metrics, without (too) negatively affecting any of the others. Occasionally, there are cases that raise a dilemma of trade off between metrics. Such dilemmas were solved with attempt to follow idioms

[1]https://github.com/SpartanRefactoring/Spartanizer
[2]http://codegolf.stackexchange.com/

[3]The term "laconize" is also used at occasion, mostly in older versions of the tool.

978-1-5090-5501-2/17/$31.00 © 2017 IEEE

and patterns as much as possible. For example, the phrase `while(true)` is equivalent to `for(;;)`, except that it features one fewer token at the "cost" of two extra characters. Instead of abstaining, the automatic spartanization process replaces occurrences of `while(true)` by `for(;;)`, just because (in our experience) `for(;;)` is more "idiomatic". Similarly, tips make code gravitate towards linear chains of conditionals structures that resemble a generalized `switch` statement, annotations and modifiers are sorted, etc.

Outline: The remainder of this article is organized as follows. Sect. II discusses the spartan programming principles, showing some examples of spartan code. Sect. III describes the architecture and operation. Sect. IV presents some empirical results. Related work is in Sect. V. Sect. VI concludes.

II. Spartan Programming

Good principles of the software engineering dictate that code should to be organizing in small cohesive modules depending on few globals, avoids heavily nested and complex control structures, with a sparing use of parameters, methods and classes. The spartan programming *methodology* aims at these by asking the programmer to *simultaneous* minimize (but not at all costs) the following metrics:

- *Code size*, measured in terms of number of characters, lines, language tokens;
- *Vertical complexity*, specifically, the number of lines in software modules;
- *Horizontal complexity*, i.e., the number of parameters in classes or methods, the level of nesting of control commands, such as **`while`**, **`switch`**, **`if`**, etc.
- *Control complexity*, namely the total number of control commands.
- *Variability*, which implies a minimization of the variables ability to change, not to mention visibility, lifetime, and scope, just as their number.
- *Exploitation of environment*, in terms of number of distinct identifiers (of fields, methods, etc.) used in any module.
- *Life and size of stacked context*, which is the most intricate of the spartan minimization ends.

For example, in the few cases of conflicting metrics, common sense is called for the rescue. For example, adding the `final` modifier to a variable in the sake of reducing variability, increases the number of tokens by one, and the number of (non-white) characters by five. This issue is a no-brainer, since if a swiping decision has to be made, then it makes more sense to pay this little price to make the code more robust. And, the sake of uniformity dictates

There are differences between the methodology and the tool, the Spartanizer, that help programmers implement it. Other tools include those offered by the IDEs, code stylers, etc. The Spartanizer is not smart enough to correctly add `final` modifier to variables. This however can be done by the IDE. In contrast, the Spartanizer can remove `final`s in cases they are syntactic baggage, contributing nothing to semantics, as e.g., when they attach to arguments to abstract functions.

Indeed, much of the spartanization work can be automated, and the Spartanizer is a powerful, but not omni-potent tool for doing so. The Spartanizer can even resolve simple tradeoff dilemmas, favoring uniformity and idiomatic, common form of expression when possible. However, tools are not omnipotent, and so is not the Spartanizer. Hard tradeoff decisions come at the design level, which the Spartanizer leaves to the human operator.

Tippers and the tips they produce come in several varieties. For example, *Nominal* tippers are tippers that rename local variables and parameters. *Modular* tippers (not employed here) involve method extraction. *Structural* tippers restructure the code using changing loop headers, moving `break`s and `continue`s. *Syntactic Baggage* tippers remove redundant modifiers such as `abstract` and `public` from declarations within an `interface`, extraneous nesting, etc. A full list of the tippers and refactoring techniques is available at the tools' web page.

Repeated application of tippers converts

```
if (f() == true) {
  return false;;
} else {
  return true;;
}
```

into

```
return !f();
```

Steps include the removal of redundant curly brackets and semicolons, conversion of `if` into a *ternary* expression (`·?·:·`), and simplification of the obtained ternary expressions with boolean literals.

Spartanization is applied to the maximum extent[4], in other words till there are no other spartanization opportunities, by exhausting the tippers in the toolbox, reaching a (local) minimum. Otherwise, the Spartanizer proceeds by selecting one such tipper and continuing with the code obtained after applying this tipper. The code which underwent automatic spartanization is often significantly different from the original. As an example, we report the case of the generic class `C0`, defined by the developer to include only instance fields `inner` of the generic type parameter (Fig. 1). Its laconic version is reported in Fig. 2.

After the laconization all conditionals have been replaced with the ternary operator and the short-circuit operators, "`||`" and "`&&`". There have been also abbreviation of `obj` to `o` and then the latter has been renamed. The spurious `super()` call has been eliminated, and the same happened to the two variables used in the `hashCode()` method, that now has just a single expression.

The resulting code is shorter, not as deeply nested as before and, presumably, easier to explain and understand. The Spartanizer allows developers to perform this process automatically and semi-automatically or iteratively.

[4]"*As far as this can reach.*" K. Agesilaus the great (*about the extent of Sparta's border*)

```java
public class C0<T> {
  private T inner;
  public C0(T inner) {
    super();
    this.inner = inner;
  }
  public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + ((inner == null) ? 0 : inner.
    hashCode());
    return result;
  }
  public boolean equals(Object obj) {
    if (this == obj)
      return true;
    if (obj == null)
      return false;
    if (getClass() != obj.getClass())
      return false;
    C0 other = (C0) obj;
    if (inner == null) {
      if (other.inner != null)
        return false;
    } else if (!inner.equals(other.inner))
      return false;
    return true;
  }
}
```

Fig. 1. A generic class representing a cell, along with its Eclipse automatically generated methods (28 lines, 94 words, and 649 characters).

```java
public class C1<T> {
  private final T inner;
  public C1(T inner) {
    this.inner = inner;
  }
  public int hashCode() {
    return 31 + ((inner == null) ? 0 : inner.hashCode());
  }
  public boolean equals(Object ¢) {
    return ¢ == this || //
      ¢ != null && getClass() == ¢.getClass() && //
      equals((C1) ¢);
  }
  private boolean equals(C1 ¢) {
    return inner == null ? ¢.inner == null : inner.equals(¢.
    inner);
  }
}
```

Fig. 2. A spartanized version of the JAVA class in Fig. 1 automatically generated methods (16 lines, 70 words and 424 characters).

III. THE SPARTANIZER

A. How It Works

The Spartanizer is at the core of the Spartan Refactoring[5] Eclipse[6] plugin available through the Eclipse Marketplace [7]. Code laconization, also known as spartanization, can be performed using the menu **`Laconize ...`** under the Refactoring menu of the GUI. It is possible to ***(i)*** spartanize the code in the active window, ***(ii)*** apply all the tippers to the entire project, ***(iii)*** refresh the tippers on all the projects in the workspace and ***(iv)*** activate/deactivate tippers on all the projects.

[5]Latest release is 2.7.9, December 2016
[6]http://eclipse.org
[7]https://marketplace.eclipse.org/content/spartan-refactoring-0

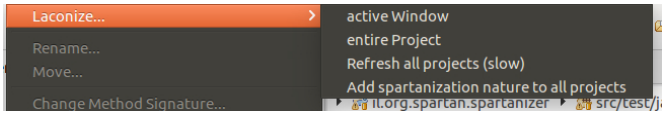

Fig. 3. The main menu

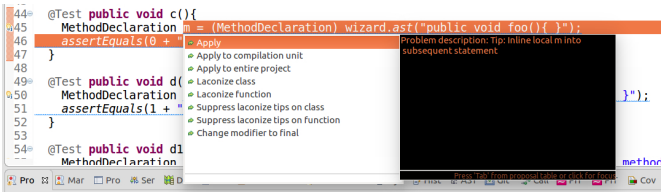

Fig. 4. The QuickFix menu

Each tipper can also be applied to specific snippets of code (*tip* in the Spartan Refactoring jargon). The spartanization opportunities are highlighted by a Quick Fix (see Fig. 4). For each Quick Fix (which corresponds to a Marker) by clicking on the icon on the left, the user can choose among several actions: ***(i)*** apply the tipper to the highlighted code (Apply), ***(ii)*** spartanize to different levels of granularity (method, class, file or project), ***(iii)*** disable spartanization on specific elements (methods or classes) and ***(iv)*** altering the variability of some elements, namely adding the **`final`** modifier to parameters, fields and variables.

All the spartanization tips are available in the "Problems" view under the **`Infos`** category with their description and resource location (file name and path). There is also the opportunity to run a global spartanization using the **`Laconize`** button on the toolbar. It is also possible to choose which tippers group to apply by selecting it in the preferences menu.

It is acknowledged that refactoring tools are often underused [6]–[8]. Murphy-Hill and Black established 5 principles that might favor the usability of these tools and the Spartanizer is compliant with some of them (for example, the user can navigate the code while using the tool) [9] [10].

B. Design and Architecture

One of the basic elements of Spartan Refactoring is the **`Toolbox`**, a class that contains all the available **`Tippers`**. Each tipper can be applied to only one determined type of **`ASTNode`** (i.e., **`ForStatement`**, **`InfixExpression`**, etc.) based on some *prerequisites*. Multiple tippers can be applied one tipper at a time, in sequence, depending on a defined policy. Presently, the first compliant tipper found on the Toolbox is applied to a given **`ASTNode`**. Any Tipper extends

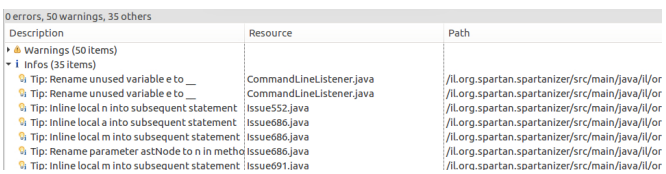

Fig. 5. The problems view with information about spartanizing opportunities

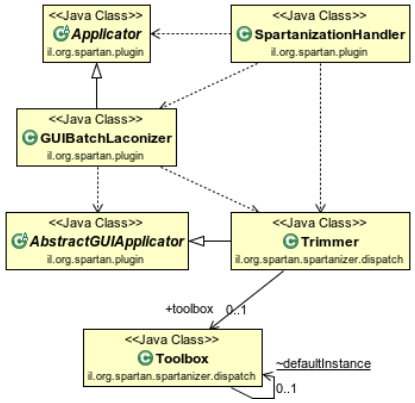

Fig. 6. A simplified UML diagram of Trimmer, Toolbox, Handlers and Applicators

a *tipping* class (i.e. **`CarefulTipper`**, **`EagerTipper`**, etc.) which implements a *tipping strategy*.

Spartanization tips are generated at Eclipse startup and after any changes[8] by an **`IncrementalProjectBuilder`**. Anytime the builder runs it loads (or reloads, in case of changes) all the tippers in the **`Toolbox`**, then parses the code looking for suggestions. Each time it occurs, the markers are first removed (see Fig. 4), optionally clearing the previous suggestions (i.e., in case of changes) before adding the suggestions again. These suggestions are then highlighted (more specifically, underlined) in the editor as it is shown in Fig. 4. The suggestions appear also in the problems view, as reported in Fig. 5.

The Spartanizer can be seen as an Abstract Rewriting System (ARS) or, more specifically, a term rewriting system. In fact each tipper is associated to a rewriting rule that has to be applied to an Abstract Syntax Tree (AST) using an **`ASTRewriter`**. The transformations are performed by a **`Trimmer`**. The trimmer scans the toolbox looking for an appropriate tipper and when it finds it, it applies the corresponding rewriting rule.

The automatic spartanization of whole systems is performed by the **`entire project`** command in the main top menu or the **`Apply to entire project`** command in the Quick Fix menu (but, in this case, just for the selected tipper). The process is handled by a **`GUIBatchLaconizer`** which is an **`Applicator`**, meaning that it applies the tippers to a specified **`Selection`** (i.e., file, method or an entire project). The **`SpartanizationHandler`** properly dispatch the selection, namely the group of files or element of code which need to be automatically spartanized.

IV. Empirical Results

We applied the Spartanizer to a corpus of eight large, professionally developed software systems, of different application domains, as reported in Table I. Table I reports the minimization effect of the process in terms of LOC and NOT (number of tokens). Extra verbosity removed amounts to ten, and sometimes more, of the code. The second column of Table II is the percentage of spartanized files. Not all the files were touched by the Spartanizer, but coverage is extensive, ranging from 70% to 87% (**`Gora`**). The last column in Table I reports the time required for the process (in ms). The fastest was commons-logging (13 seconds), and the slowest was commons-math (10 minutes) which is required for the application of about 50 thousands tips.

It is worth to note that the Spartanizer has been applied in a previous study of the authors [4]. We analyzed the Gal-Lalouche corpus [11], a dataset of 26 open source Java [12] software systems taken from *GitHub's Trending repositories*[9] and the *GitHub Java Corpus* list [13], finding that spartanized code is more "natural" than the original, using a variant of Hindle's et al. [14] naturalness metric.

Table I
Analyzed systems including number of classes, number of tippers applied, and effect the LOC and NOT (number of tokens) size metrics.

Systems	*n.classes*	*n.tippers*	*LOC*	*NOT*	*Time (ms)*
junit4	464	4,118	−9.4%	−3.0%	29,156
commons-logging	54	1,060	−6.5%	−3.6%	13,253
commons-bcel	386	8,905	−3.5%	−0.8%	90,432
commons-lang	374	15,578	−6.6%	−2.9%	262,555
commons-math	1,174	50,008	−10.1%	−4.4%	600,457
mina	316	4,139	−4.0%	−1.3%	37,126
gora	271	8,685	−10.3%	−4.3%	80,290

Table II
Statistics of the automatic spartanization

		Tippers Applied Per File			
Project	*File Touched (%)*	*Mean*	*Q1*	*Median*	*Q3*
junit4	71	31.4	3	12.3	24
commons-logging	75	63.4	6.3	30.4	65.3
commons-bcel	86	162.9	1	10	16
commons-lang	88	276.5	4	41.4	65.3
commons-math	78	898.8	5	28.5	40
mina	83	70.5	2	11.5	17
gora	87	281	5	19.5	26

V. Related Work

There are many tools available for automatic refactoring. Due to space constraint we limit our presentation to those that are, in our opinion, the most meaningful. The most popular IDEs (i.e. Eclipse, Intellij[10], etc.) include refactoring engines to perform refactoring in a semi-automatic way. The number of refactoring are usually taken from the Fowler's catalog.

Conderser [11] and jCOSMO [12] are academic project, code smells detectors. RefactorIT is a commercial tool that allows to detect and fix code smells, even though they are less sophisticated than those we can find in the Fowler's catalog [15].

[8] If the automatic build option is enabled

[9] https://github.com/trending?l=javasince=monthly

[10] https://www.jetbrains.com/idea/

[11] http://condenser.sourceforge.net/

[12] http://projects.cwi.nl/renovate/javaQA/intro.html

BeneFactor [16] is a tool that allows developer to intervene after refactoring begins in order to complete a refactoring change safely. Code-Imp [17] combines the refactoring automation with the computation of a series of metrics that can be used as fitness function. Similarly, the Spartanizer tries to minimize the metrics reported in Section Sect. II.

KinEdit [18] goal is to improve the manual refactoring work. RefactoringNG [19] applies refactoring based on patterns by exploiting two AST one for the original code (that matches the pattern) and the other for the substitute code and apply a rewriting rule. RefaFlex analyzes reflective calls that can alter program's behavior, in order to prevent the developer to refactor them [20]. ReLooper automatically refactors an array into a ParallelArray [21].

Autorefactor[13] is able to apply a number of refactorings automatically. This tool is close to Spartanizer in the intent, but provides a less extended set of 27 refactorings to apply. To the best of the authors knowledge, presently the Spartan Refactoring plugin offers the widest range of refactorings among the available tools.

VI. Conclusion

This paper introduces the Spartanizer, an Eclipse plugin in that helps programmer apply principles of spartan programming and to automatically apply over one hundred and fifty refactorings to an entire software system in a matter of seconds. We briefly presented the basic principles of the spartan programming methodology including some examples of the spartanized code.

We explained how the Spartanizer works, its design and reported some evidence of its efficiency on a dataset of seven popular software systems. We also pointed out that the Spartanizer was used in a previous study [4][14], where we reported that spartanized code seems to be more natural, in the sense that it is not only shorter, it is more idiomatic and easier to compress.

The plugin is actively maintained. A release of version 2.79 was made on December 2016. The next release, planned for February 2017 shall include a mouse roll button that would allow zoom into the code by applying tippers, and, when rolled at the other direction, zooming out, by increasing code verbosity, adding intermediate variables, breaking computation to smaller steps, etc. The tool is available as installable plugin at the Eclipse marketplace, and open source on GitHub.

Acknowledgment

The Spartanizer begun as student project conducted by Artium Nihamkin, a second major version was by Boris van Sosin and a third one by Ofir Elmakias and Tomer Zeltzer. Next came Daniel Mittelman who was succeeded by Ori Roth. Major contributions to the project were made by the team of Alexander Kopzon and Dan Greenstein, and, Dor Maayan and Niv Shalmon. Ori Marcovitch contribution was indispensable. The full list of contributors to the software, spanning over 70 names is available at the GitHub project page. This research was supported by THE ISRAEL SCIENCE FOUNDATION (grant No. 1803/13 *)

[13] http://autorefactor.org/

[14] based on an older, less powerful version of the tool. Other changes include the *laconization*, which has been since removed from the GUI.

References

[1] M. Fowler, *Refactoring: improving the design of existing code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[2] W. Opdyke and R. Johnson, "Refactoring: An aid in designing application frameworks and evolving object-oriented systems," in *Proc. 1990 Symp. Object-Oriented Programming Emphasizing Practical Applications*, ser. SOOPPA 90. ACM, 1990.

[3] J. Gil, "Reflections on spartan prog. and the no-debugger principle," in *Hardware and Soft.: Verification and Testing - 6th Int. Haifa Verification Conf. , HVC 2010, Haifa, Israel, October 4-7, 2010. Revised Selected Papers*, ser. Lecture Notes in Computer Science, S. Barner, I. G. Harris, D. Kroening, and O. Raz, Eds., vol. 6504. Springer, 2010, pp. 5–8. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-19583-9_4

[4] Y. Gil and M. Orrú, "Code spartanization," in *Proc. of SAC'17, the 32nd ACM Symposium on Applied Computing*, Marrakesh, Morocco, April 3–7 2017.

[5] E. W. Dijkstra, "Letters to the editor: Go to statement considered harmful," *Comm. ACM*, vol. 11, no. 3, pp. 147–148, Mar. 1968. [Online]. Available: http://doi.acm.org/10.1145/362929.362947

[6] E. Murphy-hill and A. P. Black, "Why dont people use refactoring tools," in *In Proceedings of the 1 st Workshop on Refactoring Tools. ECOOP 07. TU Berlin, ISSN 14369915*, 2007.

[7] G. C. Murphy, M. Kersten, and L. Findlater, "How are java software developers using the eclipse ide?" *IEEE Softw.*, vol. 23, no. 4, pp. 76–83, jul 2006.

[8] M. V. Mäntylä and C. Lassenius, "Drivers for software refactoring decisions," in *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, ser. ISESE '06. New York, NY, USA: ACM, 2006, pp. 297–306.

[9] E. Murphy-Hill and A. Black, "Refactoring tools: Fitness for purpose," *IEEE Software*, vol. 25, no. 5, 2008.

[10] E. Murphy-Hill, "Improving usability of refactoring tools," in *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA*, vol. 2006, 2006.

[11] J. Y. Gil and G. Lalouche, "When do soft. complexity metrics mean nothing? —when examined out of context," *Journal of Object Technology*, vol. 15, no. 1, pp. 2:1–25, 2016.

[12] K. Arnold and J. Gosling, *The Java Programming Language*, ser. The Java Series. Reading, MA: Addison-Wesley, 1996.

[13] M. Allamanis and C. Sutton, "Mining source code repositories at massive scale using lang. modeling," in *Proc. of 10th Working Conf. on Mining Soft. Repositories (MSR'13)*, San Francisco, California, USA, May18-19 2013, pp. 207–216.

[14] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of soft." in *Proc. of the 34th Int. Conf. on Soft. Eng.*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 837–847.

[15] E. Mealy and P. Strooper, "Evaluating software refactoring tool support," in *Proceedings of the Australian Software Engineering Conference, ASWEC*, vol. 2006, 2006.

[16] X. Ge and E. Murphy-Hill, "BeneFactor: A flexible refactoring tool for eclipse," in *SPLASH'11 Compilation - Proceedings of OOPSLA'11, Onward! 2011, GPCE'11, DLS'11, and SPLASH'11 Companion*, 2011.

[17] I. Moghadam and M. Cinnéide, "Code-Imp: A tool for automated search-based refactoring," in *Proceedings - International Conference on Software Engineering*, 2011.

[18] J. Terrell, "KinEdit: A tool to help developers refactor manually," in *SPLASH Companion 2015 - Companion Proceedings of the 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*, 2015.

[19] Z. Troníček, "RefactoringNG: A flexible Java refactoring tool," in *Proceedings of the ACM Symposium on Applied Computing*, 2012.

[20] A. Thies and E. Bodden, "RefaFlex: Safer refactorings for reflective Java programs," in *2012 International Symposium on Software Testing and Analysis, ISSTA 2012 - Proceedings*, 2012.

[21] D. Dig, M. Tarce, C. Radoi, M. Minea, and R. Johnson, "ReLooper: Refactoring for loop parallelism in Java," in *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA*, 2009.