Syntactic Zoom-Out / Zoom-In Code with the Athenizer

Yossi Gil Dor Ma'ayan Niv Shalmon

Raviv Rachmiel Ori Roth

Department of Computer Science
The Technion—Israel Institute of Technology,
Technion City, Haifa 3200003, Israel

***Abstract*—Care and great effort are often taken to dress program code of libraries, just as model implementations, in its most presentable form, which includes adherence to strict coding standards, careful selection of identifiers, avoiding unnecessary constructs, etc. However, a presentable dress is not a janitor's uniform and is often inferior to the more lax working outfit.**

The spartanizer is a tool that brings Java code into a canonical, short form. Trying to say the most with the fewest words. In contrast, the athenizer is a tool that expands the code, placing it in a more maintainable form, using plenty of auxiliary variables, many potential locations for breakpoints and for change.

The tool reported on here allows developers to interactively use their joystick and its buttons for code navigation, and in particular for zooming-in into the code (athenizing) and zooming-out of it (spartanizing).

I. Introduction

Ancient Sparta was the archetype of minimalism. Spartan conduct is frugal, despising luxuries. "Laconic speech"—the art of saying little in few words—is eponym for Laconia, this region of Greece, of which Sparta was capital, and a source of great ridicule of Spartans by the hedonistic Athenians.

This work describes a tool[1] that makes it possible for programmers to (metaphorically) move from Sparta and Athens and back, i.e., go from a short, laconic version of their code (Spartan code), to its verbose version and back.

Spartan programming [1]–[3] is a coding style and abstract teaching that promotes coding with the equivalent of laconic speech—using as few words as possible to write your code, while being careful to avoid degrading into obfuscation nor to the horrors of code-golfing. Spartan code is easily recognizable, (see e.g., Fig. 1), and as our experience (both as developers and clients) indicates, is very readable after a short learning curve [1].

While reading code of others, programmers are often intrigued by questions such as: "*why did my peer define this variable?*", and "*why did he use a* **`while`** *here, when a* **`for`** *is so much more appropriate?*", etc. If the examined code is spartanized code, then the answer to these questions are often: because the standard says so.

[1]see accompanying video at http://bit.ly/athenizer

```
/** Class generated automatically by defining a generic class representing
 * a cell, then applying Eclipse to automatically generate equality, hashing
 * and other standard methods, and then fully spartanizing the resulting
 * code. */
public class Cell<T> {
  private final T inner;
  public Cell(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((Cell) ¢);
  }
  private boolean equals(Cell ¢) {
    return inner==null ? ¢.inner==null : inner.equals(¢.inner);
  }
} // class Cell
```

Figure 1: A spartanized Java class

When the same code structure can be rewritten is several different ways, the spartanizer will make a serious attempt at selecting the shortest and more canonical one.[2]

An advantage is also that this (pseudo) canonical form leads to fewer collisions [4] in using version control systems. Empirical evidence is still required to show what some believe: that the "little ink" policy of spartanization is easier on the reading eye, requires less scrolling, and reduces the time of browsing through code of others.

However, with experience in using the spartanizer [1] accumulating, so came the demand of programmers to move from the austere to the lavish, and from the spartan to the hedonistic. Mostly during debugging, but also in other occasions, there are times in which an expanded version of the code is required.

```
@Override
public int hashCode() {
  int i1;
  int i2;
  if (inner == null) {
    i2 = 0;
  } else {
    i2 = inner.hashCode();
  }
  i1 = 31 + (i2);
  return i1;
}
```

Figure 2: Function **`hashCode`** of Fig. 1 fullly expanded by the athenizer

Indeed, to check whether a hash function is homogeneous and does not include multiplication by zero, or redundant operations, one may prefer the version in Fig. 2 over the

[2]As with all refactoring tools, the spartanizer may break code in the process of transformation.

spartanized version in Fig. 1.

Contribution: Refactoring [5] was originally advertised for making it possible to improve design and other code qualities dynamically. In this work, we take refactoring to a different domain: interactive, reversible refactoring not for improving the code, but rather for inspecting it from different angles.

Our model programmer is an individual who likes to keep her code representative, neat and compact, without compromising the ability to focus into details, poke breakpoints, debug, instrument, and quickly manipulate the code in other ways. We found that athenizing the code indispensable with limitations of contemporary debuggers, in which inspection of intermediate values is often cumbersome or impossible and single stepping is still done in line-by-line mode.

II. Spartan Programming and the Spartanizer

This section is a brief overview of *Spartan programming* [3] and the spartanizer [1], which is a coding style that tries to follow the spirit of visionaries like Dijkstra [6] in focusing on *minimalism*. Towards this end, spartan programming enables its followers to minimize size metrics of their code, including

- *Code size*, measured in terms of number of characters, lines, and language tokens;
- *Vertical complexity*, specifically, the number of lines in each individual 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.

The spartanizer is an Eclipse refactoring plugin composed of *tippers*—nano refactorings trying to minimize these metrics as much as possible while keeping the semantics of the code unchanged. The plugin can be applied in batch, to spartanize large projects and interactively as part of the development process.

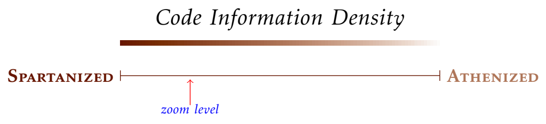
III. Zooming In and Out of Code

The athenizer is the dual of the spartanizer: The spartanizer shrinks the code without changing its functionality; the athenizer expands the code without changing its functionality. Why have the two? For the same reason we have casual and formal items in our wardrobe, and for the same reason that Eclipse offers an outline and a type hierarchy views of the code.

Think of these two extremes of modes of the code:

- *Working Mode*: describes the code during writing and debugging. During these stages of development we would like to "zoom-in" on specific parts of the code and see them in as much details as possible. In this mode, we can follow the flow of the program in a finer granularity. For example, this makes finding the specific point at which a bug occurs easier.
- *Polished Mode*: describes the code before sharing it with other developers and before releasing it. At this mode we would like the code to be as *canonical* as possible. In this mode, the programmers has a broader, bird's eye view of the code.

Accordingly, the zoom-in/zoom-out operations into code can be visualized like so;



The left hand side of the diagram denotes fully spartanized code, with high information density. At the right, we see athenized code. Dynamic zooming allows one to change the information density, moving between the two ends at small steps.

IV. The Athenizer as Minimizer of Metrics

While the spartanizer minimizing the above mentioned spartan programming metrics, the athenizer tries to minimize the effect of each individual statement, breaking each statement into many smaller statements which together preform the same operation as the original.

The statement `int i=1;` can be broken into two statements `int i; i=1;` each of which is essential, i.e., eliminating any one of them would either change the semantics of the program, or prevents it from compiling. The athenizer thus never expands the statement `int i;` into the two statements:

```
int i;
;
```

By asking that the impact of a statement be as small as possible (but without being zero), we allow the programmer to examine each little step individually, concentrate on the smallest steps, but spend time on steps that do nothing.

Other metrics we use are

- Total length of the code (by number of lines)
- Number of control structures
- Number of variables

By increasing these metrics we provide an ability to *"Zoom-In"* into the code and look at it in its most extended way, including separate `if` statements, separate `catch` clauses, temporary variables in complicated computations and many other.

In order to demonstrate some of the refactorings done by the Athenizer, Fig. 2 presents the original code from Fig. 1 fully athenized.

We see that the athenized code is greatly expanded compared to the original code: the ternary expression was expanded to `if` statements, and the complicated arithmetic computation of the *hash code* was split into temporary variables. This example demonstrates the differences between *spartanized* code to *athenized* code: While the *spartanized* version minimized the total length of the code, the *athenized* version minimized the operation each one of the statements does.

Minimizing the operation of each code statement may be useful during the development and debugging. Consider e.g., the four dense lines in Fig. 3.

The semantics of these are clear to most readers, in the sense that the algorithm is clear to people familiar with the domain: examining a ternary expression, choose either its **elze** or **then** branch (one of which must be **boolean**), and then convert it to a simpler boolean expression.

In contrast, the zoomed-in version of the code in Fig. 4, allows one to side step the algorithm, debug it and modify it: In particular, we can see that every sub-expression is assigned to a temporary variable and that ternary expressions are expanded into **if** statements.

V. User Interface

A. Graphical User Interface

The athenizer comes is part of a larger Eclipse plugin available through the Eclipse marketplace [3]. The functionalities of the athenizer are accessible through the Spartanizer menu, in the Athenizer sub menu (Fig. 5). The Athenizer offers the following operations:

- Current Selection Expand once withing the current selection of test
- Active Window Fully expand the active window
- Toggle Athenizer Toggle the athenize mode
- Zoom In Once Activate one tipper on the active window (from the Spartanizer)
- Zoom Out Once Expand once in the active window

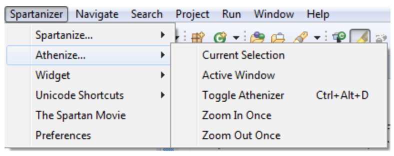


Figure 5: The main menu of the Spartanizer plugin

While the athenize mode it activated, the code can be expanded and contract the code freely by holding the control button and scrolling the mouse wheel. This allows you to control exactly how much the code expands through an easy and intuitive interface. You can tell whether the athenizer is activated by the left most icon in the Spartanizer tool-bar.

B. Programming With Joystick

Besides of zooming in and out with the mouse wheel, we tried to follow a futuristic vision of multi dimensional code viewing. By this we mean that the programmer can zoom into the code both horizontally (athenizing) and vertically (expanding short names). Yet a third dimension, font size, is also possible, but is out of this scope.

Using the joystick as a tool for coding is a revolutionary step, follows the vision of others [7]: "*The developer could stay focused on the code when he used marking menus for refactoring, as opposed to linear menus, which were distracting*".

We speculate that refactoring using gestures of a joystick will be more convenient for programmers. In other words, it can be inferred that using the joystick will be an intuitive way for refactoring, or, in our case, spartanizing and athenizing (both described as refactoring methods(see Sect. II)).

We noticed that the use of the joystick is intuitive and straightforward according to the user interface of the project. In addition to zooming in and out with the joystick, we utilized the left and right gestures in order to expand and spartanize variable names. Thus, names are generated for variables according to their type (using binding, as a function of the desired length (expanded or spartanized)).

VI. Design and Implementation

The athenizer relies on the modular design of the *spartanizer*, which makes it easy to encapsulate code transformations and apply these at will.

Code analysis is by JDT [8] (Eclipse Java development Tools) and the support of the Java IDE to Eclipse platform. The main data type offered the JDT is the **AST-Node**, i.e., a node in the abstract syntax tree.

The JDT represents a Java [9] source file as an AST (Abstract Syntax Tree) that includes all syntactical elements (including JavaDOC comments, but not other comments). Nodes in this tree come in many different types that represent the many different Java syntactical categories, including node types such as **TypeDeclaration**, **MethodDeclaration**, **Statement**, **CompilationUnit**, etc. All these types are organized in an inheritance hierarchy comprising circa 100 different types. The JDT provides mechanisms for traversing the tree and modifying it. These mechanisms are augmented by *binding information*: With the help of the IDE, the client can follow a reference from a definition of an entity, such as a local variable

```
final Thingy thing = new Thing())
```

to the typing information assumed for this entity, such as a reference to the definition

```
class Thingy {...}
```

The *AST* tool is a Java model which provides an API for many others, fully modeling the syntax of Java and providing us a full analysis of the code. An API for traversing different ASTNodes is provided as well and we found it useful for the implementation of some advanced refactoring techniques. We also use tools from eclipse RCP (rich client platform) [4] in order to design our graphical user interface. *Bloaters*

[3]https://marketplace.eclipse.org/content/spartan-refactoring-0

[4]https://wiki.eclipse.org/Rich_Client_Platform

```
final boolean~$ = !iz.booleanLiteral(then);
final Expression other =~$ ? then : elze;
final boolean literal = az.booleanLiteral($ ? elze : then).booleanValue();
return subject.pair(literal !=~$ ? main : make.notOf(main), other).to(literal ? CONDITIONAL_OR : CONDITIONAL_AND);
```

Figure 3: Four lines of heavily spartanized code

```
final boolean~$;
boolean b1;
b1 = iz.booleanLiteral(then)
    ;
$ = !b1;
final Expression other;
if ($) {
  other = then;
} else {
  other = elze;
}
final boolean literal;
BooleanLiteral l1;
ASTNode n1;
if ($) {
  n1 = elze;
} else {
  n1 = then;
}
```

```
l1 = az.booleanLiteral(n1);
literal = l1.booleanValue();
InfixExpression x1;
Pair p1;
Expression x2;
if (literal !=~$) {
  x2 = main;
} else {
  x2 = make.notOf(main);
}
p1 = subject.pair(x2, other);
Operator o1;
if (literal) {
  o1 = CONDITIONAL_OR;
} else {
  o1 = CONDITIONAL_AND;
}
x1 = p1.to(o1);
return x1;
```

Figure 4: Athenized version of the code in Fig. 3

are refactoring rules which operate over **ASTNode**s. Each bloater implement a nano-scale[5] rafactoring—in the code while preserving the code semantics.

Bloaters follow the *typestate* [10] semantics. Before any change is done over the actual **ASTNode** of the code, prerequisites are checked and only if they are satisfied, the modification of the *AST* occurs. Checking prerequisites before any operation of the bloater makes assures that the applied transformation on the **ASTNode**s do not lead to unpredictable, potentially detrimental, or even dangerous, side effects.

In order to keep the refactored code as safe as possible, any bloater must satisfy a list of prerequisites before operating over an **ASTNode**.

VII. Conclusion

The athenizer is a new part of the *spartan sefactoring*[6] Eclipse plugin that features reverse refactoring techniques to those the original spartanizer offers. *Spartan programming* is a methodology that strives to obtain an extreme minimalism of code element. Athenization does the opposite—try to minimize the impact of each individual statement.

With the athenizer, the plugin now offers programmers the ability to "Zoom-In" into their code and "Zoom-Out" of it. These operations can be done with the mouse wheel or a joystick controller. We explained how these operations correspond to a working mode and polished mode of the code, and how the zoom operations moves the code along the scale between the two modes.

As described on Sect. V-B, turning a joystick right and left while using the plugin will increase or decrease the length of identifiers in the code. Currently, we demonstrate a basic heuristic for giving names to identifiers. The issue of giving names to identifiers is also reflected while using bloaters which create new variables. We identify a potential for future research over the issue of giving meaningful names to identifiers in a code, possibly by machine learning techniques.

The plugin is actively maintained. Latest release is 2.12.1 (June 2017). The plugin is available at the eclipse marketplace[7], and open source on GitHub [8]

Acknowledgments: We pay tribute to Dr. Adi Omari who wisely and patiently lead the production team. Tomer Dragucki and Yuval Simon were active in the implementation of the athenizer. See also the full list of contributors to the spartanizer project in [1] and in [9].

This research was supported by the Israel science foundation (ISF grant No. 1803/13*).

References

[1] Y. Gil and M. Orrù, "The spartanizer: Massive automatic refactoring," in *Soft.Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th Int. Work. on.* IEEE, 2017, pp. 477–481.

[2] J. Atwood, "Spartan prog." https://blog.codinghorror.com/spartan-programming/.

[3] J. Y. Gil, *Reflections on Spartan Prog.and the No-Debugger Principle.* Berlin, Heidelberg: Springer Verlag, Oct. 2011, pp. 5–8. [Online]. Available: http://dx.DOI.org/10.1007/978--3--642--19583--9_4

[4] M. L. Guimarães and A. R. Silva, "Improving early detection of soft. merge conflicts," in *Proc. 34th Int. Work. on Soft.Eng.*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 342–352. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337264

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

[6] 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

[7] E. Murphy-Hill, M. Ayazifar, and A. P. Black, "Restructuring Soft.with Gestures," in *2011 IEEE Symp. Visual Lang. and Human-Centric Computing (VL/HCC).* IEEE, 2011, pp. 165–172. [Online]. Available: http://people.engr.ncsu.edu/ermurph3/papers/vlhcc11.pdf

[8] E. Gamma and K. Beck, *Contributing to Eclipse: Principles, Patterns, and Plugins.* Addison-Wesley, 2003. [Online]. Available: http://www.amazon.com/Contributing-Eclipse-Principles-Patterns-Plugins/dp/0321205758

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

[5]small changes to the code aimed to increase the metrics listed above while preserving the code semantic

[6]https://github.com/SpartanRefactoring/Main/

[7]http://marketplace.eclipse.org/content/spartan-refactoring-0

[8]https://github.com/SpartanRefactoring/Main/releases

[9]http://bit.ly/spartan_contributors

[10] R. E. Strom and S. Yemini, “Typestate: A prog. lang.concept for enhancing soft.reliability,” *IEEE Trans.Soft.Eng.*, vol. 12, no. 1, pp. 157–171, Jan. 1986. [Online]. Available: http://dx.DOI.org/10.1109/TSE.1986.6312929

Appendix

This is the typescript of the plug-in screen cast[10]:

“Hello everyone, and welcome to the Athenizer plug-in. With the Athenizer you can zoom in and out of your code, so when you want to concentrate on some hard part in your code you can zoom in the code and understand it better.

Here is an example. This example is be from a real GitHub project we imported. Take a look at this Java *program.*

We have a bug inside this project, so when we run the tests, the tests fail as you can see now when we go to the line which failed the test. We see it is really hard to understand the line because there is a long if and we dont know exactly where the bug is. We would like to expand this code. Maybe expand the ternary expression. When I say “ternary” I mean the funny “question-mark” column” operator, *which is similar to an* `if` *statement, except that it is an* operator, *not a* statement.

So let us expand the code using our joystick. As you can see we expand the code, making it clearer and easier to understand. We expand ternary expressions, extract variables, expand computations to temporary variables and increase dramatically the total size of the code while keeping the original logic of it.

As you can see the code is longer now and it will be easier to detect the exact error, so lets run again the tests.

Now we can see that we get an exception on a much more concrete line of the code. We get a null pointer exception on `ownerType` *variable and now, when we see more details about it, we can fix it.*

Let’s run again the tests and here we see that all tests pass. Hooray!

All that is left for now is to squeeze the code back again, before we push it back to GitHub. So let’s do it.

Now lets see some other features we have in the Athenizer plug-in.

We can expand and squeeze the whole code in the active window with a click of a button or, as we see here, using a button on the joystick.

The athenizer plug-in is available for free download on the eclipse marketplace and as an open source project on GitHub.

Thats it for today, thank you for watching wish you a joy(stick)ful coding.

[10]http://bit.ly/athenizer