Star Diagram with Automated Refactorings for Eclipse

Alexis O'Connor, Macneil Shonle, William Griswold
Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093-0114
{*aoconnor,mshonle,wgg*}*@cs.ucsd.edu*

Abstract

One of the powerful features of the Eclipse JDT plug-in is its refactoring feature. Yet, because many refactorings are global and architectural in nature, the textual presentation of refactorings presents challenges for designers. We have designed and implemented the visual Star Diagram automated refactoring plug-in to ease such refactorings. This paper motivates and introduces star-diagram-based refactorings, demonstrates that new tools can be implemented on top of existing JDT refactorings, and reflects on the challenges that the JDT refactoring API presents in supporting novel extensions. In this last category, we found it difficult to build on the refactoring API to introduce more general refactoring semantics and visualizations of refactoring opportunities.

1. Introduction

The maintenance of a large software system is a complex task. One way to lower maintenance costs is to refactor the system into modular abstractions that both better represent the system designer's concepts and better encapsulate future changes. Refactoring is a restructuring technique to transform code without changing its behavior [4,5,6,9]. Together, many small refactorings can be applied to create a significant design change in the system. The Eclipse JDT provides a set of refactoring operations that programmers can use when writing code, as well as a refactoring API that plug-in developers can use to build new tools.

New system features or emergent concepts in the problem domain often motivate refactorings. Some changes affect a certain concern or a specific set of data structures. For example, the logging performed by an application may have slowly accreted into a major system feature supporting online visualization and steering, so its coupling to the application and its own design were never given serious consideration. Although Eclipse provides a mechanism to perform mechanical restructuring changes, it does not advise the user to the relevant refactoring opportunities. Finding the relevant refactorings to perform can be difficult when working primarily with the localized view provided by the source editor or simple search tools. Most of the source on the screen at any time is unrelated to logging, yet the computations on the logging data structures are widely dispersed in the system [7].

To address this problem in the context of procedural programs, many years ago we introduced the Star Diagram manipulable visualization [2,3]. With the emergence of the Eclipse IDE framework and the JDT's refactoring support, we revisited this work to explore two questions. One, could the star diagram, with suitable improvements, prove useful in object-oriented development? Two, what IDE features are required, and what challenges are encountered with integration into a mature IDE? This paper introduces our enhanced star diagram for OO in Eclipse (see Figure 1), partially answering the first question. The second question is explored in depth, revealing successes as well as challenges for the development of whole-program refactoring support.

2. The Star Diagram Automated Refactoring Plug-in

A star diagram is a static tree containing all computations that refer directly or indirectly to the data structure under examination, and omits computations not related to the data structure. Star diagrams illustrate both syntactic relationships and redundancies in the program. Figure 2 is an example of a star diagram that provides a representation of the program in Listing 1. A node denotes a branch of the AST. The root, at the far left (in this case, labeled "status"), represents the data structure being examined. The children of the root represent Java constructs in the program that directly reference the variable. The children of the next level represent constructs that reference the previous level's computations, and so on. A node having a stacked appearance indicates that two or more pieces of code correspond to (perhaps) the same computation, as determined by our unification algorithm. Because of the stacking of similar computations, each unique computation is represented just once. The number label on stacked nodes denotes the number of nodes in the stack. The penultimate level of children, shown as parallelograms, represent the functions that contain these computations. The last level of children denotes the classes that contain the methods [7,8].

The structure and content of a star diagram makes it possible to identify refactorings of interest when planning and performing changes to a system. Previous research has shown that restructuring through a star diagram visualization can help a developer understand and manipulate the overall structure of a program for the purposes of creating new abstractions, thus localizing future changes and lowering maintenance costs [7].

Star Diagram was originally built as a tool for analyzing procedural code. The tool included an automated refactoring feature that allowed users to manipulate the source code through the star diagram. To perform a refactoring, the user clicked on a node, then clicked on the desired refactoring button.

We addressed three shortcomings of the existing tool. The often-complex star diagram did not provide the user with knowledge about which nodes would be useful to refactor. Second, the previous star diagram tool required the user to look outside the diagram view to determine the available refactorings. Third, some refactoring relationships were not sufficiently described with the existing syntactic edges. Consequently, we designed the Star Diagram

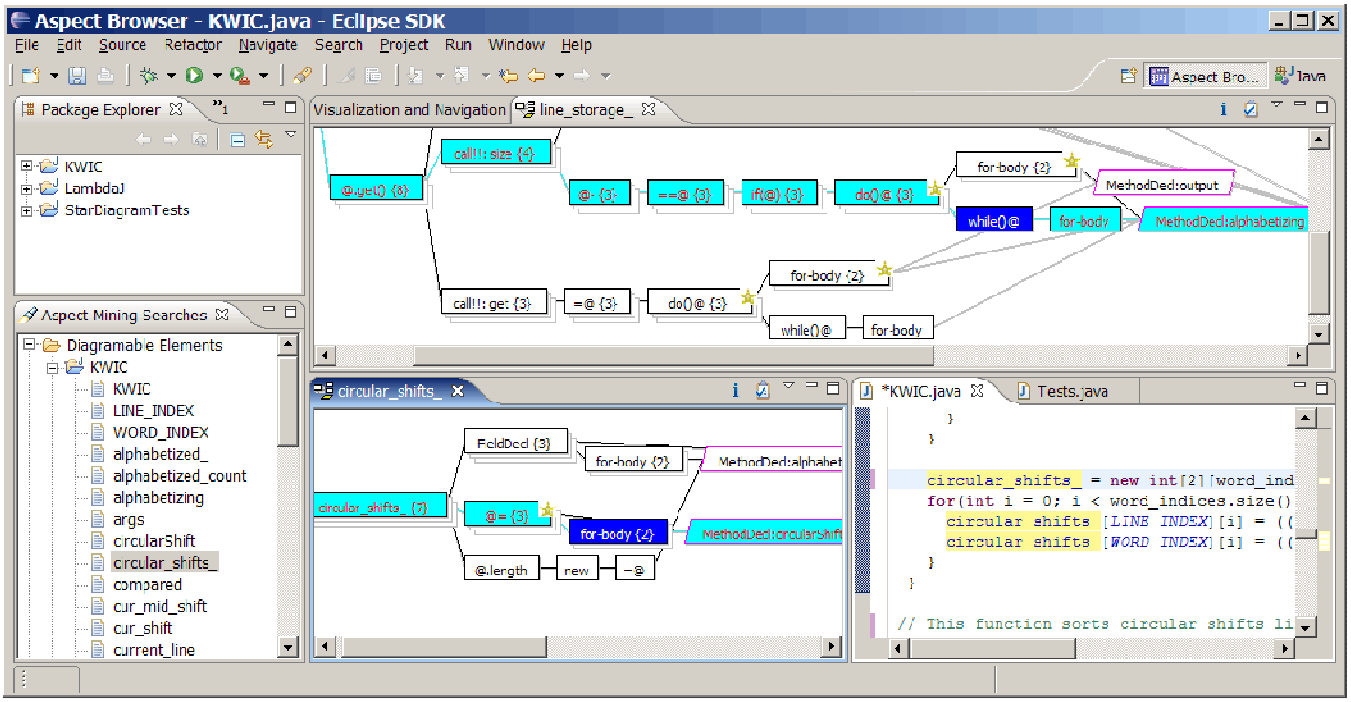

Figure 1. The SDAR Plug-in for Eclipse

Figure 2. Star Diagram with Refactoring Opportunity Hint

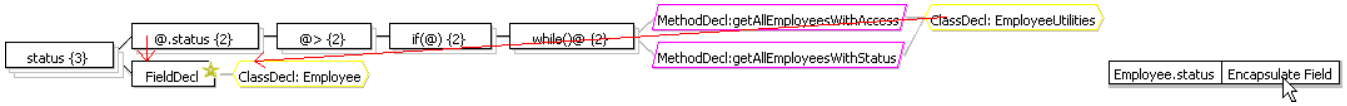

Figure 3. Star Diagram with Refactoring Opportunity Hint, Pop-up Refactoring Menu, and Dynamic Edges

Automated Refactoring Plug-in (SDAR), which is an enhanced tool for developers to see and perform both global and local refactoring opportunities using star diagrams in Eclipse.

Available refactoring opportunities in the tree may not be immediately apparent to the user. Thus, we have added an additional component to star diagrams: refactoring opportunity hints. SDAR currently creates refactoring hints for three different refactorings: Encapsulate Field, Remove Unused Parameter, and Inline Method. The hints are generated when a star diagram is created, based on the structure of the diagram and conditions we have designated. We have chosen the star symbol to indicate these refactoring hints. In Figure 2, the star symbol can be seen on the nodes labeled "while() @" and "FieldDecl"; thus, both of those nodes identify a refactoring possibility.

SDAR currently supports four different refactorings: Encapsulate Field, Extract Method, Remove Unused Parameter, and Inline Method. To communicate to the user the SDAR refactoring suggestions, we created a pop-up suggestion menu within the star diagram view. When the user mouses-over a node, the applicable refactorings for that node are shown to the right of the diagram. The context menu as a pop-up within the diagram is more convenient and apparent than viewing the options outside of the diagram. The user can then activate the refactoring by clicking on the desired refactoring in the pop-up menu, as shown in Figure 3.

In SDAR, an edge implies that there is a syntactic relationship between the nodes connected to the edge. Some refactorings can be described sufficiently with existing edges, whereas other refactorings have indirect, non-syntactic relationships. To capture this relationship, we have added dynamic edges to SDAR that are activated when the user mouses-over the refactoring in the pop-up refactoring menu. Encapsulate Field is an example of a refactoring that uses dynamic edges. SDAR suggests applying the Encapsulate Field refactoring when the root node (i.e., the data structure being examined) is used in classes other than the declaring class or its subclasses. In this case, no edge exists in the diagram that illustrates which field reference is violating the encapsulation principle. To show this relationship, SDAR generates dynamic edges when the user mouses-over the desired refactoring (shown as directed arrows in Figure 3). An edge is added from every violating reference (in this case there is only one) to the Field Declaration node. SDAR also adds an edge from the member class of every violating node to the class that declares the field so that the user can see how pervasive the violation is throughout the program.

3. Implementation

SDAR uses Eclipse's Abstract Syntax Tree components to build star diagrams and uses Eclipse's refactoring API to implement various refactorings. Our research method was to use (as much as

possible) the JDT refactoring components that were available, rather than to rewrite our own refactorings. We created a simple adapter of the analogous JDT refactoring API to invoke the Encapsulate Field refactoring initiated via SDAR. Thus, the JDT itself will infer the additional needed information to perform the refactoring. Our implementations of Remove Unused Parameter and Inline Method are accomplished similarly. This implementation approach ensures that the transformations preserve the semantics of the program. Since SDAR invokes the corresponding JDT refactoring to perform the user-selected transformation and does not change the code itself (for the refactorings listed above), the transformations ensure semantics preservation. This guarantee assumes that each JDT refactoring has been proven not to change the program's semantics.

```
public class Employee
{
   public int status;
   …
}

public class EmployeeUtilities
{
 public ArrayList getAllEmployeesWithAccess(ArrayList employees)
 {
  ArrayList empList = new ArrayList();
  int i = 0;
  while (i < employees.size()) {
    if (((Employee)employees.get(i)).status > 5) {
      empList.add(employees.get(i));
    }
    i++;
  }
  return empList;
 }

 public ArrayList getAllEmployeesWithStatus(ArrayList employees)
 {
  ArrayList empList = new ArrayList();
  int i = 0;
  while (i < employees.size()) {
    if (((Employee)employees.get(i)).status > 3) {
      empList.add(employees.get(i));
    }
    i++;
  }
  return empList;
 }
}
```

Listing 1. Employee Payroll Source Code

```
public ArrayList getAllEmployeesWithAccess(ArrayList employees)
{
  ArrayList empList = new ArrayList();
  int i = 0;
  int param0 = i;
  int param1 = employees.size();
  while (param0 < param1){
    int param2 = ((Employee)employees.get(i)).status;
    int param3 = 5;
    if (param2 > param3) {
      empList.add(employees.get(i));
    }
    i++;
  }
  return empList;
}
```

Listing 2. After Extract Local Variable Operations

```
public ArrayList getAllEmployeesWithAccess(ArrayList employees)
{
  ArrayList empList = new ArrayList();
  int i = 0;
  int param0 = i;
  int param1 = employees.size();
  int param3 = 5;
  while (param0 < param1){
    int param2 = ((Employee)employees.get(i)).status;
    if (param2 > param3){
      empList.add(employees.get(i));
    }
    i++;
  }
  return empList;
}
```

Listing 3. After Code Motion Operations

Implementation of the SDAR Extract Method refactoring was more involved. SDAR enables the Extract Method refactoring option for every node in the diagram that passes the JDT Extract Method conditions. We found that the JDT Extract Method refactoring implementation often does not recognize code as duplicate when the code corresponds to the same computation. To recognize a wider range of equivalent code as duplicates, we developed an Extract Method extension that converts the code to be extracted into a normalized form that can then be identified as equivalent by the JDT Extract Method refactoring unifying algorithm (located in the SnippetFinder class). The SDAR Extract Method extension improves the amount of code that the JDT Extract Method refactoring unifying algorithm recognizes as equivalent, but it does not necessarily locate all cases of duplicate code throughout a program.

There are two cases when the JDT Extract Method refactoring will not recognize potentially duplicate code:

(1) In otherwise-identical trees, the JDT Extract Method refactoring often does not match two nodes that are of the same scope and the same type, but are not of type *SimpleName*. For example, in Listing 1, when the literal 5 is compared against the literal 3, the comparison will not result in a match.

(2) Furthermore, the JDT Extract Method refactoring does not equate two variables that have the same type, but different scopes (including inter-class scopes). For example, if an integer *x* is considered for matching with an integer *y*, and *x* and *y* have different scopes, the two variables will not match. This is correct because there are instances when such cases result in different method generations, but there is a large class of special cases that can safely be viewed as equivalent: When the code to be extracted does not re-evaluate the expression, the expression can instead be passed as an argument to the new abstraction of the common code (i.e., a method generated by Extract Method).

Given these two issues, we decided to create an Extract Method Refactoring extension. It is essentially a refactoring that is a composite of four micro-refactorings. Three of these refactorings are JDT refactorings: Extract Local Variable, Extract Method, Inline Local Variable; and one is our own: Code Motion. (The availability of these JDT refactorings considerably mitigated the difficulty of the implementation of the Extract Method extension.) When the user chooses to extract a method from within the SDAR, each node in the stack is examined. The expressions in each node and its subnodes are converted to local variables through the JDT Extract Local Variable refactoring when extracting an expression will not change the semantics of the program (see Listing 2). Local variable extraction will not change the semantics of the program when it is performed on expressions that function as a right hand value in the Java construct (right hand side of an assignment, comparison, etc.). After the restructuring, the nodes to check as matching will be of type *SimpleName*. So, the nodes that have the same scope and member class will result in a JDT Extract Method refactoring match. This technique also partially addresses the second issue because it creates a variable declaration statement for

every applicable expression under examination, and doing so enables the expressions to attain the same scope.

For the newly created local variables to attain the same scope, we created a Code Motion Refactoring. The refactoring moves a specified node to a different location in the code. In our case, we are concerned only with the movement of variable declaration statements; specifically, those statements just created by the JDT Extract Local Variable refactoring. Our aim is to move the statements directly above the selected node to be extracted, so as to ensure that the variables have the same scope. This refactoring is able to guarantee correctness only if the expression evaluation to be moved does not need to be re-evaluated in the block of code to be extracted. In order to make this determination, we ask the user to select the expressions that only need to be evaluated once within the block of code to be extracted and thus may be moved before that block. Each expression that cannot be moved outside the block is inlined to undo the local variable extraction that was performed earlier; the rest of the expressions are moved outside the block. The result of the code motion refactoring applied to the code in Listing 2 can be viewed in Listing 3. After the Code Motion refactoring is performed, the JDT Extract Method refactoring API is called and it can now recognize a wider range of duplicate code. As explained above, only expressions that are evaluated one time in the block of code are extracted as local variables and moved outside of the block. The above series of refactorings guarantee that the semantics of the code have not been changed. Any expression that is evaluated once within a block of code will produce the same behavior as an argument passed to the block of code. Thus, assuming that the JDT refactorings have been proven to guarantee the semantics of the program, the extension has not changed the semantics.

After the method is extracted, the JDT Inline Local Variable refactoring is applied to restore the extracted local variables to their original state. This final micro-refactoring does not affect the semantics of the program since those expressions that could be safely extracted can likewise be safely inlined. If the user chooses to cancel at any point in the Extract Method Extension process, each intermediate step that has been performed is undone in reverse order.

4. Reflections on Developing for Eclipse JDT Refactoring API

The Eclipse JDT refactoring tool [1] implements many of the refactorings delineated in Fowler's book. The refactorings implemented in the JDT plug-in can be viewed as micro-refactorings, meaning that each transformation is a small change, but combined, a series of such refactorings can result in a "significant transformation" [4]. One of our research questions is to explore the IDE features required for architecture-level refactorings. Our version of Extract Method yields several insights on this question and also reflects on the analogous question of composing macro-refactorings from micro-refactorings.

Compatibility of Interfaces. All of the refactorings extend from the same super class and hence have similar APIs for creating and performing a refactoring. Such a uniform API greatly reduced the learning curve required to make use of the refactorings. Another central feature of the API is the separation of displaying a refactoring dialog and performing a refactoring. In the case of the Extract Method extension, it would have been onerous for the user to ok the extraction of every local variable and then again to inline it.

As we developed the code to perform automated refactorings, we found it necessary to use components of existing refactorings. We utilized the JDT's TextChangeCompatibility class to create the Code Motion Refactoring. The existing interface provided all that we needed. We used the static deleteTempDeclaration and createAndInsertTempDeclaration methods to move the newly created variable declaration statements from one location in the code to the other.

There were other cases where accessing the existing refactoring code was difficult. For example, there is no way that our plug-in can access the JDT's match method in the SnippetFinder's inner Matcher class. Our solution was to copy and paste the method into SDAR. This is a substandard solution because the method must be exactly the same as the one in SnippetFinder. This copying creates a potential source for bugs in our code when upgrading to future versions of the JDT plug-in.

Matching Algorithm. We found that the JDT Extract Method refactoring has some limitations. As explained earlier, we found the matching algorithm that recognizes duplicate code to be insufficient. We believe that equating two nodes of the same type and scope could be an enhancement to the JDT Extract Method refactoring by making some straightforward modifications to its matching algorithm. It would be more challenging for the JDT Extract Method refactoring to be extended to perform the second part of our Extract Method extension, which equates a class of variables that have the same type and different scopes. This extension would either require a data flow analyzer or the implementation of a method similar to ours. We would not recommend the latter of the two due to the large scale of potential duplicates that JDT Extract Method refactoring examines. This is where SDAR has an added benefit; it only analyzes the nodes in the selected stack, rather than the entire code base, making it feasible to perform all of the intermediate transformations.

Inter-Class Matching. The JDT Extract Method finds duplicates of the code to be extracted, but limits the search to the same class. This limitation made the refactoring unsuitable to be used directly for our purposes because star diagrams present a global view of the program. We speculate that the reason for this implementation decision is due to the extended computation time. If our assumption is accurate, future implementations of the tool might offer the user the option to search all classes in the project, and then select in which class the newly extracted method belongs.

Multiple Return Values. Another possible feature that the JDT Extract Method refactoring tool could have is to handle multiple return values. The current version cannot extract any code that would require greater than one return value. One approach would be to generate a class to hold the return values.

Tracking AST Node Correspondence. Creating a macro-refactoring as a composite of micro-refactorings proved to be a difficult experience. The actual creation of the refactoring in terms of subclassing the Refactoring class was straightforward. The complexity lied in tracking state from one micro-refactoring change to another. Each micro-refactoring required the changes of the previous refactoring and hence required a new compilation unit and new AST. It was necessary for us to find the corresponding nodes from the old AST to the new AST. We did not find a simple solution to do this. We investigated converting AST nodes to IJavaElements and then marking the IJavaElements as a way to save state, but there is not a one-to-one correspondence between the two: the AST is finer grained. As a solution, we tracked a node from one tree to another by calculating the location of the node of interest from the root node. When a refactoring change occurs, we update the stored location of the node. From the stored location, we can find the node in the new AST. Without the ability to use the Visitor pattern to traverse the ASTs, a much deeper understanding of the ASTs would have been required. When finding the location of a particular node of some type, the Visitor pattern allows us to view only nodes of that same type and count in which occurrence the node of interest appeared. The beauty of this is that when a refactoring change occurs, the location needs to be updated only if the changed node is of the same type as the node being tracked.

Atomic Undo. Ideally, a macro-refactoring should be able to be undone with the user clicking undo once. The current implementation provides a CompositeChange class that enables an atomic undo of a macro-change of independent changes. However, the current interfaces available in the refactoring tool do not provide a way to do this when the macro-refactoring is a sequence of dependent changes, as is our case.

5. Conclusion

As realized today, the Eclipse JDT refactorings are micro-refactorings, supporting the local repair of class designs. Refactoring, as conceptualized by a software architect, is a redesign activity, possibly spanning many classes and exposing emergent system concepts as first-class entities. Tools like the Star Diagram for extracting system-wide similarities and presenting refactoring opportunities hold considerable promise.

We have shown that star-diagram-style macro-refactorings can be built from micro-refactorings, thus supporting architecture-level refactoring. However, the micro-refactorings work better if designed for composition and extension. The main problem was not in the refactorings themselves, but in the underlying infrastructure, such as ASTs and similarity matching. We also found a need for some additional basic micro-refactorings to enable the construction of complete macro-refactorings.

6. Future Work

Our next step is to conduct an experiment that measures how useful a star diagram, with our refactoring enhancements, is in object-oriented development. Additionally, it would be interesting to explore the use of other global analysis tools within SDAR to make SDAR a more robust tool. The concept of refactoring opportunity hints and a refactoring context menu that invokes the JDT Eclipse refactorings could be extended to other visualization tools.

7. Acknowledgements

This work was supported in part by an Eclipse Innovation Award from IBM.

8. About the Authors

Alexis O'Connor is a master's student at UCSD whose research interests include software design and refactoring tools. This work represents a part of her master's thesis. Macneil Shonle is PhD student at UCSD whose research interests include aspect-oriented programming tools and languages. William Griswold is the research advisor of Alexis and Macneil.

9. References

1. Eclipse Refactoring Support http://help.eclipse.org/help31/index.jsp?topic=/org.eclipse.jdt.doc.user/concepts/concepts-9.htm

2. R. W. Bowdidge and W. G. Griswold. Automated support for encapsulating abstract data types. *In ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering,* pages 97-110, December 1994.

3. R. W. Bowdidge. *Supporting the Restructuring of Data Abstractions through Manipulation of a Program Visualization.* PhD dissertation, University of California, San Diego, Department of Computer Science & Engineering, November 1995. Technical Report CS95-457.

4. M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, Reading, MA, 1999.

5. W.G. Griswold. *Program Restructuring as an Aid to Software Maintenance.* PhD dissertation, University of Washington, Dept. of Computer Science & Engineering, August 1991. Technical Report No. 91-08-04.

6. W. G. Griswold and D. Notkin. Automated assistance for program restructuring. *ACM Transactions on Software Engineering and Methodology,* 2(3):228-269, July 1993.

7. W. G. Griswold, M. I. Chen, R.W. Bowdidge, J. D. Morgenthaler. Tool Support for Planning the Restructuring of Data Abstractions in Large Systems. In *Proceedings of the ACM SIGSOFT Conference on the Foundations of Software Engineering* (FSE-4), October 1996.

8. W. F. Korman, *Elbereth: Tool Support for Refactoring Java Programs,* M.S. Thesis, Technical Report CS98-590, Department of Computer Science and Engineering, University of California, San Diego, June 1998.

9. W. F. Opdyke. *Refactoring: A Program Restructuring Aid in Designing Object-Oriented Applications Frameworks.* PhD dissertation, University of Illinois at Urbana-Champaign, Dept. of Computer Science, 1992. Technical Report No. 1759.