Acta Cybernetica **23** (2017) 715–735.

Automating the Refactoring Process

Gábor Szőke[a]

Abstract

To decrease software maintenance cost, software development companies use static source code analysis techniques. Static analysis tools are capable of finding potential bugs, anti-patterns, coding rule violations, and they can also enforce coding style standards. Although there are several available static analyzers to choose from, they only support issue detection. The elimination of the issues is still performed manually by developers.

Here, we propose a process that supports the automatic elimination of coding issues in Java. We introduce a tool that uses a third-party static analyzer as input and enables developers to automatically fix the detected issues for them. Our tool uses a special technique, called reverse AST-search, to locate source code elements in a syntax tree, just based on location information. Our tool was evaluated and tested in a two-year project with six software development companies where thousands of code smells were identified and fixed in five systems that have altogether over five million lines of code.

Keywords: refactoring, code smells, reverse ast-search, spatial index

1 Introduction

Refactoring is a common practice for improving software maintainability. By definition, refactoring is "*a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior*" [6]. The original declaration introduces the understandability and changeability aspect of refactoring. However, other research shows that the idea can also be applied to other purposes [18], such as improving performance, security, and reliability. Developers use it regularly to transform code into a more maintainable form. About 70–80% of all structural changes in the code are due to refactorings [32, 2]. Industrial case studies [14] revealed that the refactoring definition in practice is not confined to a rigorous definition of semantics-preserving code transformations and that developers perceive that refactoring involves substantial cost and risks.

Many IDEs (integrated development environments) offer refactoring features to provide assistance in these regular tasks. Eclipse, for instance, has a separate *Refactor menu* where such operations are available, e.g., developers can rename a source

[a]University of Szeged, E-mail: `gabor.szoke@inf.u-szeged.hu`

DOI: 10.14232/actacyb.23.2.2017.16

code element (e.g., a variable) and the IDE will correct all its references. Similarly, it is possible to extract local variables, methods, classes, or move elements, among several other operations for refactoring purposes.

Tools which support automatic refactorings often assume that programmers already know how to refactor, and they have knowledge about the catalog of refactorings [6], but this is typically not a reasonable assumption. As Pinto et al. found in their study where they examined questions of refactoring tools on Stack Overflow, programmers are usually unable to identify refactoring opportunities, because of lack of knowledge in refactoring, or do not understand of the legacy code. They also claim that "*refactoring recommendations is one of the features that most of Stack Overflow users desire (13% of them)*" [24].

In order, to offer refactoring recommendations to developers, some studies [5] use static code analyzers to report problematic code segments. Static analysis tools are capable of finding potential bugs, anti-patterns, and coding rule violations. Fontana et al. compare the capabilities of refactoring tools to remove code smells, and they identify only one tool (*JDeodorant*) which can support code smell detection and then to suggest which refactoring to apply to remove the detected smells.

In this study, we introduce a process which suggests refactoring opportunities for coding issues and it is able to automatically refactor them. Our solution uses the PMD static analyzer to find Java coding rule violations. We created several refactoring transformations that support fixing these coding issues.

To perform refactoring operations, we created a mapping between the textual output of the static analyzer and the structural representation of the source code. This task required us to create an algorithm that takes textual source code position (i.e. start and end line) and type information (i.e. `for` loops) of the problematic code segment and executes a search on the syntax tree to locate the related source code element in the tree. To make reverse searching possible, we use a *spatial database*. The database is created by transforming the source code into a *geometric space*. Line numbers and column positions from the AST are used to define areas. These areas are used in *R-trees*, where area-based searching is possible. To evaluate our approach we created a tool that applies the proposed process. The evaluation showed that our approach is adaptable in a real-life scenario, and it can provide viable results. Our tool was used in a project where it assisted in more than 6,000 automated refactorings covering systems ranging from 200 to 2,500 kLOC.

2 Overview

This study was part of an R&D project supported by the EU and the Hungarian Government. The goal of the 2-year project was to develop a software refactoring framework, methodology and software tools to support the "continuous reengineering" methodology, hence provide support to identify problematic code parts in systems and to refactor them so as to enhance maintainability. During the project, we developed a semi-automatic refactoring framework [29] and tested it on the source code of the industrial partners, having an *in vivo* environment and live feedback on

the tools. This is why partners not only participated in this project to help develop the refactoring tools, but they also tested and used the toolset on the source code of their own products. This provided a good chance for them to refactor their code and improve its maintainability.

During the development of the framework, we found several interesting problems and solutions to these problems which we think worth sharing.

Before going into details, we should overview the general refactoring process. As Fowler says, refactoring is *"a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior"* [6]. Based on this definition if we model the code as a graph – which every compiler does – a refactoring is a (behavior preserving) transformation on a graph. More specifically, it can be viewed as a transformation on the *abstract syntax tree (AST).* Executing such transformation requires three components. They are:

- An AST as a representation of the source code.
- A transformation algorithm.
- Starting points (AST nodes) called the *„origin"* where the transformation algorithm begins.

First, we parse the source code with a parser which builds an AST. Second, we create a transformation algorithm that will make modifications on the AST (i.e. pull up a method from one class to its ancestor). Next, we pick a node on the AST as the origin where the transformation algorithm will start working (i.e. selecting the method to pull up). After the transformation is performed, we get a modified (refactored) AST and the refactoring is complete.

Because our goal was to create automated refactorings, we extended the process with a few additional steps. These steps allow us to interact with developers and to make the transformations automatically. Our process works as follows:

Process 1.

1. *We create an AST representation of the source code.*
2. *We use static analysis on the source code to find problematic code parts, i.e. coding issues, rule violations, metric warnings. We list these issues as suggestions to help the user in finding candidates for refactoring.*
3. *The user selects one of the issues as the target of the refactoring.*
4. *Based on the type of the issue chosen by the user, a refactoring algorithm is selected that is capable of fixing the given type of issue.*
5. *Based on the selected issue a proper origin node is chosen from the AST.*
6. *The algorithm makes the transformation and modifies the AST, while keeping track of what modifications it made. A modified AST is created.*
7. *We generate source code from the modified AST.*
8. *The newly refactored source code is shown to the user where he or she can test the code and decide whether to accept or reject the refactoring.*

Next, we will present these steps in greater detail and discuss our findings.

3 Process Details

3.1 Building AST

To build an AST from the source code we use the SourceMeter [7] tool. SourceMeter uses OpenJDK [21] as a backend to parse the code and build an *abstract semantic graph (ASG)*. ASG is an extended version of the AST with cross-edges and much more [4]. This extra information was crucial in the automation process. Hence it allowed us to create flawless transformations which otherwise wouldn't be possible. *Thus Step 1 of Process 1 is covered.*

3.2 Finding Refactoring Suggestions

Choosing which part of the source code to refactor is quite hard. To improve the maintainability of the code, one can either start optimizing for metric values or try to get rid of anti-patterns by introducing design patterns to the code. Any of them might be a good solution. However, we will choose coding issues as the main target of our automated refactorings because our preliminary study [27] suggested that this is what developers want the most. To identify coding issues we choose the well known PMD static source code analyzer. It is a widely used and accepted tool among developers, especially for checking Java rule violations. Because all of the participating project members had a Java code base, it was an optimal choice to integrate it into the framework.

The output of PMD worked well in identifying coding issues and even in presenting some of these to developers as refactoring suggestions. Let us examine the sample in Listing 1. In this simple example, PMD finds 9 rule violations (with default settings). It finds issues like missing package declaration, missing comments, short variable names, magic numbers, missing braces and the list goes on. Even in this simple sample of code, there are many issues that can be fixed with computer assistance.

To extract the issues we use the XML output of PMD. This file contains a list of violations for each file with name, description, priority, and position information. An example violation is shown in Listing 2. This is one taken from the list of issues we got as output after running PMD on Listing 1. It clearly states that we should use curly braces in the `if` statement in Line 3.

After presenting these kinds of issues to developers, they are usually able to select one for refactoring. *Thus Step 2 and 3 of Process 1 are covered.*

3.3 Selecting the *Right* Transformation for the Job

After the user selected an issue, the next step is to find the right transformation. Many researchers are working on solutions to automate this process using machine learning techniques [11, 19, 22]. However, we used a much simpler approach. We created several general refactoring transformations, i.e. for moving, adding, deleting, and swapping source code elements. We created a mapping between PMD

Listing 1: Example code

```
1 public class Example {
2     public static int limiter(int x) {
3         if (x > 10)
4             return 10;
5         return x;
6     }
7 }
```

Listing 2: PMD's XML report

```
<file name="Example.java">
    ...
    <violation beginline="3" endline="4" begincolumn="3" endcolumn="
        13" rule="IfStmtsMustUseBraces" ruleset="Braces" class="
        Example" method="limiter" priority="3">
    Avoid using if statements without curly braces
    </violation>
    ...
</file>
```

violations and the transformations. For example, to fix the curly braces issue, we mapped it to an insertion transformation where a *block* element will be injected below the `if` statement (See the illustration in Figure 1). Each mapping defines different parameters based on the type of issue. The transformation in the former example requires an `if` statement as a parameter. *Thus Step 4 of Process 1 is covered.*

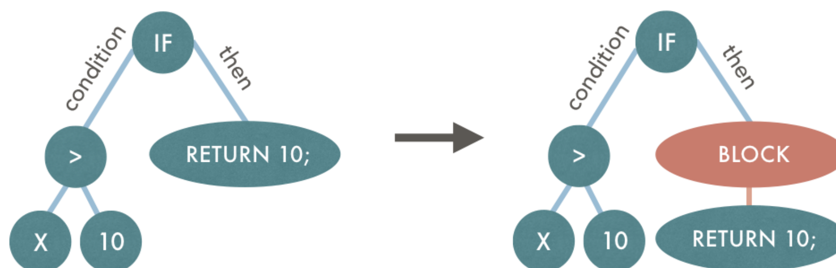

Figure 1: Simplified illustration of a refactoring on the AST of Listing 1.

Listing 3: PMD highlight

```
1 public class Example {
2     public static int limiter(int x) {
3         if (x > 10)
4             return 10;
5         return x;
6     }
7 }
```

Listing 4: SourceMeter highlight

```
1 public class Example {
2     public static int limiter(int x) {
3         if (x > 10)
4             return 10;
5         return x;
6     }
7 }
```

3.4 Selecting a Proper Origin

After the user has selected an issue to fix and we chose the right transformation, the next step in the process is to choose an origin point on the AST, which we can give as a parameter to the transformation algorithm. In other words, one has to perform a search on the AST to find an element that matches both the description provided by the PMD report and the type of the parameter the transformation algorithm requires.

The report provides only a few key point for a violation, i.e. beginning line and column, ending line and column, class, and method. Also, the source file is available in the `file` tag. If we look at the example in Listing 2 and the results on the example in Listing 1, we get the problematic code part *highlighted*. The highlighted part in Listing 3 shows the particular `if` statement that needs braces. Although this highlighted segment is a good visual aid for the developer to find where the violation is, it is problematic for the computer to find nodes in the AST based on little more than position information.

A way to address the problem is to store position information for each source code element in the AST during the parsing process. Fortunately, SourceMeter does this already. Now that we know the positions on the AST, we can attempt to match the violation location information to the ones we have on the AST. It may come as no surprise that a simple equality match did not work. PMD and SourceMeter have different parsers and therefore it is more unlikely they will have the same position information for each and every source code element. Listing 4

shows the position SourceMeter has identified for the `if` statement in question. Looking at both highlighted cases reveals that a simple approximation will not suffice to get a match. To handle the problem, we took a different direction which we call *reverse AST-search*.

Reverse AST-Search

This idea was born when we decided to take a different direction and start looking at the source code elements, not as just data or nodes in a tree. In the text editor, they look like as little areas or patches. Since they all have begin and end lines and columns, they can be viewed as coordinates on a map. This led us to the idea of transforming the source code into a *geometric space*.

We took line numbers and column positions from the AST and used them to define areas. These areas form rectangles where the corner points are the begin and end positions of the source elements. The rectangular areas are then used to build a *spatial database*, where area-based queries are possible.

Spatial Databases and R-trees A spatial database [25] is a database that is optimized to store and query data that represents objects defined in a geometric space. Common database systems use indexes to quickly look up values and the way that most databases index data is not optimal for spatial queries. Instead, spatial databases use a *spatial index* to speed up database operations. To create spatial index data, we decided to use *R-trees* [10].

An R-tree is a data structure where the key idea is to group together information based on spatial data and index these groups by using their minimum bounding rectangles[1]. Next, these groups are bound together at the next level of the tree by their minimum bounding rectangles, and so on. This way, a query cannot intersect any of the objects contained because all the objects within a bounding rectangle occur together. The input of a search is a rectangle called a *query box*. Every rectangle in a node (starting from the root node) is checked to see whether it overlaps with the search rectangle or not. If it does, the same thing happens with its corresponding child nodes. The search goes on recursively until all matching nodes get visited. Meanwhile, when a leaf node is found and it overlaps with the query box it is added to the result set.

R-tree applications cover a wide spectrum, ranging from spatial and temporal to image and video databases. In industry, it is used where multi-dimensional data needs to be indexed. For example, a common application is in digital maps where R-trees are used to link geographical coordinates to POIs [15].

Building Spatial Index for the AST To create a spatial database for the source code, we used the *si* method in Algorithm 1. The method requires C, an AST element with position information. This might be a root node or a class node, say. When the algorithm commences, it creates an R-tree for storing the spatial

[1]The “R" in R-tree is for rectangle.

index *(Alg. 1 Line 1)* and begins to traverse the descendants of the C element *(Alg. 1 Line 2)*. Note that every kind of traversal is acceptable since the position of the elements do not depend on each other.

Algorithm 1 Building Spatial Index for the AST.

Funct $si(C)$

Require: C is an AST element

1: Let $\mathcal{I}$ be a new R-Tree
2: **for all** $c \in descendants(C)$ **do**
3: Let $P_{start_{line}}, P_{start_{col}}, P_{end_{line}}, P_{end_{col}}$ be the position coordinates of c
4: **if** $P_{start_{line}} = P_{end_{line}}$ **then**
5: Add rectangle $\{P_{start_{line}}, P_{start_{col}}, P_{end_{line}}, P_{end_{col}}\}$ to $\mathcal{I}(c)$
6: **else**
7: Add rectangle $\{P_{start_{line}}, P_{start_{col}}, P_{start_{line}}, \infty\}$ to $\mathcal{I}(c)$
8: Add rectangle $\{P_{end_{line}}, 0, P_{end_{line}}, P_{end_{col}}\}$ to $\mathcal{I}(c)$
9: **if** $P_{end_{line}} - P_{start_{line}} > 1$ **then**
10: Add rectangle $\{P_{start_{line}} + 1, 0, P_{end_{line}} - 1, \infty\}$ to $\mathcal{I}(c)$
11: **end if**
12: **end if**
13: **end for**
14: **return** $\mathcal{I}$

For each source code element c, the algorithm takes into account their position P; namely, start line, start column, end line, end column *(Alg. 1 Line 3)*. Next, using these positions rectangles are created. To be precise, it creates one, two or three rectangles, depending on the length of the current source element. If the element position is confined within a single line, one rectangle is created *(Alg. 1 Line 5)*, which is a line on the 2D plane. In the case of multiple lines, we have a multiline element, which is an element that starts at a line in a column, and it ends in another line in a column. All positions between these two positions are part of the element. For example, in Listing 4 the `if` statement is a two-line element and the highlight indicates the positions belonging to the statement. To handle this, we create two rectangles. The first line has no end column *(Alg. 1 Line 7)*, and the second begins at zero *(Alg. 1 Line 8)*. If there more lines between them, we create a rectangle that covers all the space between the two lines *(Alg. 1 Line 10)*.

Each time a rectangle is created it is added to the R-Tree with a binding to the AST element. This way when the spatial query function starts running, we will get AST elements instead of rectangles. Once all the descendant source code elements get visited, we can return the resulting R-tree, and the spatial index is ready.

When we tested the algorithm in the example in Listing 1, we got the search space shown in Listing 5. Note that every node that has multiple lines are separated into more rectangles, such as the `if` statement.

Listing 5: Search space for the example in Listing 1 with the rectangles and the type of their referred source code element

```
Rect (1, 1), (1, INF) = Class
Rect (2, 0), (6, INF) = Class
Rect (7, 0), (7, 2) = Class
Rect (2, 2), (2, INF) = Method
Rect (3, 0), (5, INF) = Method
Rect (6, 0), (6, 3) = Method
Rect (2, 16), (2, 19) = PrimitiveTypeExpression
Rect (2, 28), (2, 33) = Parameter
Rect (2, 28), (2, 31) = PrimitiveTypeExpression
Rect (2, 35), (2, INF) = Block
Rect (3, 0), (5, INF) = Block
Rect (6, 0), (6, 3) = Block
Rect (3, 3), (3, INF) = If
Rect (4, 0), (4, 14) = If
Rect (3, 6), (3, 14) = ParenthesizedExpression
Rect (3, 7), (3, 13) = InfixExpression
Rect (3, 7), (3, 8) = Identifier
Rect (3, 11), (3, 13) = IntegerLiteral
Rect (4, 4), (4, 14) = Return
Rect (4, 11), (4, 13) = IntegerLiteral
Rect (5, 3), (5, 12) = Return
Rect (5, 10), (5, 11) = Identifier
```

The Search Algorithm Once we have built our spatial index, we can use it to locate a node in the AST based on position information. We created a method that uses inputs such as the ASG from SourceMeter, the parameter type of the transformation, and the violation position from the PMD report in order to search the geometric space. The **R**everse **AST**-search **A**lgorithm (*rasta* for short) is listed in Algorithm 2 below.

Algorithm 2 The reverse AST-search algorithm.

Funct rasta(C, P, t)
Require: C is an AST element
Require: P is a position
Require: t is a type
1: Set the candidate list $\mathcal{R} := \{\}$
2: Compute $si(C)$
3: Let S be the set of all AST nodes whose have intersecting rectangles with P
4: **for all** $s \in \mathcal{S}$ **do**
5: **if** $type(s)$ is t **then**
6: Store s in $\mathcal{R}$
7: **end if**
8: **end for**
9: **return** $\mathcal{R}$

The purpose of the algorithm is to find the source code element that is highlighted in PMD's report. The function begins by creating a list of the source code element candidates. Next, it builds a spatial index with the C AST parameter *(Alg. 2 Line 2)*. The newly constructed index is used in the next step to query the candidates. As mentioned earlier, the spatial database requires a rectangle, called the query box as a search parameter. The query box in our case is the "highlight" from the PMD's output. We use this box to ask the R-tree which previously added rectangles intersect with the parameter. The R-tree returns with a set of AST nodes whose rectangles satisfied the query *(Alg. 2 Line 3)*. As an example, Listing 5 shows which source code elements (highlighted lines) remain after the query has been performed with the `Rect(3,3)(4,13)` box as the parameter.

Even with a small sample like Listing 5, the resulting set of the query can be quite big. To narrow the result set we use the third parameter, namely the type of the input parameter of the refactoring transformation, to filter the results *(Alg. 2 Line 5)*. The filtering is achieved by going through the result set and by inserting only those source code elements onto the candidate list whose type matches (actually, whose type is compatible with) the input type. After the filtering, the candidate list is returned as the result of the function.

To continue with the example in Listing 5, we have to filter the result set with the input parameter type of the refactoring transformation. In Section 3.3 we identified this source code element type as an `if` statement. Even with a quick glance, we can see that there are two rectangles where their type is an `if` statement. Since both

of the two rectangles refer to the same `if` element, the algorithm terminates. We have found an origin where the refactoring transformation can begin its operation.

Heuristics As we saw previously, the best case scenario is when the algorithm ends up having only a single element in the list of candidates. This happens when the result set has only one source code element with the type of parameter. In this case, it is evident which element was highlighted as the source of the violation. Nevertheless, there are times when the candidate list has multiple source code elements of the same type. In such cases, we have to select the proper element as the origin, otherwise the refactoring will be executed wrongly.

To remove ambiguity we decided to use a "distance" measure to find the best candidate. We defined the distance as the number of characters between the position of the source code element and the highlight. More specifically, on one hand, it is a metric of the number of characters between the start position (line and column) of the source code element and the start position (line and column) of the highlight. On the other hand, it is the number of characters between the end position (line and column) of the element and the end position (line and column) of the highlight. The distance is just the sum of the two. To calculate this value we use the original source file where the exact number of characters could be measured.

This method allows us to select the proper source code element from the list of candidates in almost every case. Still, there is a certain mathematical probability that the calculated distances will be equal for all candidates. However, because the chance of this event is astronomically small (it never even happened once in our exhaustive testing period, see Section 4), we chose to notify the user with a message that the refactoring failed because of ambiguity.

Alternative Method The algorithm above uses two-dimensional information to back-propagate code smells to AST elements and handles code as lines and columns. However, source code can be viewed as a linear sequence of characters as well. Here, a simple one-dimensional data structure and interval operations could replace the fairly complicated two-dimensional approach. Despite this mechanism being seemingly simpler, it would require different input data. Both the given inputs – the AST and PMD – work with two-dimensional data. This would mean that we would either require the source file as input or the AST and PMD must provide one-dimensional data.

- The latter requirement would be needed from both tools to replace position information with char-sequence index or store it as additional data. This new data would cause an increase both in the processing time and storage space for the tools for information that probably no one else will ever use. Still, if char-sequence index data is available as input, the one-dimensional approach is preferable.

- The former requirement would introduce an additional parameter to the algorithm; namely, the original source file. In the case where source code is given

it is possible to handle the text as one-dimensional data and map the source code elements to char-sequence indexes. However, this approach has several drawbacks. First, every search would need to read the source file, which is an i/o intensive task. This is required to calculate the end of line positions. Next, the mapping of two dimensional data to character sequence indexes would have to consider whitespace. For example, when reading a tab character from the file, the algorithm has to know it is 2, 4, 8 (or other) characters long. Both parsers could have mixed tab size settings, which would make the mapping difficult. This would also affect the two-dimensional approach, but since a line just contains only a few tabs it easier to match the source code elements in a line than in the entire file.

The reverse AST-search algorithm works only with an AST and PMD's report as input. These tools and inputs are treated as third-party from the algorithm's point-of-view. Since these inputs have two-dimensional data and source code is not available, the one-dimensional mechanism would not suffice. Nevertheless, from the point-of-view of the refactoring process, the source file is given; but we still choose the two-dimensional approach because it provides more accurate matches in real-life scenarios.

Summary The reverse AST-search algorithm enabled us to select the proper origin, which is a source code element in the AST that is the input of the refactoring transformation later on. *Thus Step 5 of Process 1 is covered.*

3.5 Executing the Refactoring

Now that we have covered each preceding step, we have all the components and we are ready to perform the refactoring. As mentioned earlier in Section 3.3, the refactoring algorithms are defined as smaller, multiple generic transformations. The type of the PMD violation determines which transformation(s) will be executed. There are some complex cases where a simple transformation will not suffice and fixing will require multiple operations.

In order to fix the *missing curly braces* (the issue in Listing 1), the transformation inserts a *block* statement into the *then* clause of the `if` statement and rewires every former member of the *then* clause to make a member of the *block* statement. See Figure 1 as an illustration of a process like this.

Once the refactoring transformation completes its operations, a new, modified, issue-free AST is created. *Thus Step 6 of Process 1 is covered.*

3.6 Generating Source Code and Creating Diffs

In the previous step, we completed the refactoring at the model level. Even though the refactoring process came to an end, there is one more thing to do. Because our main goal is to assist developers, the next task is to translate the AST back to source code where they can readily interpret the changes.

Listing 6: Refactored code

```
1 public class Example {
2     public static int limiter(int x) {
3         if (x > 10) {
4             return 10;
5         }
6         return x;
7     }
8 }
```

3.6.1 Generating Source Code

The source code generation is realized by systematically going through the AST and writing code to a text file according to the underlining source code element. For example, if we start at a file, if there is a package declaration we write the package keyword following the name of the package and a semicolon to close the statement. This is followed by import statements and so on. The generation goes on until every source code element is visited and the code is fully reconstructed from the AST.

In the case of the example in Listing 1, after the refactoring transformation we get the code shown in Listing 6. As expected, both curly braces appeared and therefore the `return` statement got a *block* around it. In consequence, the PMD rule violation got fixed, and our code maintainability improved. *Thus Step 7 of Process 1 is covered.*

3.6.2 Keeping Track of Modifications

Previously, we showed how to fully reconstruct the code from the AST. However, generating the whole code base is unnecessary. It is sufficient to recreate only those code segments where the changes occurred. There are, however, multiple ways to reduce the amount of generated code. An easy solution would be to create only those files which were affected by the refactoring. Our only concern, in this case, was that our code generation cannot reproduce exactly (100%) equivalent source code. This happens because though SourceMeter stores source code element positions and even comments, it does not store data concerning whitespaces and indentation. Despite this, we created the source code generator in such a way that it “pretty prints” the code, but what is considered “pretty” is subjective. For example, in Listing 6 the beginning bracket is positioned after the method declaration, but someone may prefer it to be on the next line. On one hand, it is possible to make this configurable. On the other hand, there are other remaining issues, such as whether there are two spaces between the *public* and the *static* keywords or whether they should be written in separate lines.

To reduce the former anomalies, we sought to minimize generation even within

files themselves. Our approach keeps track of which nodes and at what level they are modified when the refactoring operation is running. We mark those nodes that get, for instance, inserted, deleted, and swapped. Furthermore, we mark those nodes which we visited during the operation but did not modify them. For example, in Listing 6 we put a *block* statement into the *then* clause of the `if` statement and rewired the former content (the `return`) of the *then* clause as a statement of the *block* node. We marked this *block* statement as inserted, and the `return` as unmodified. The latter was required because marking a node is a recursive operation which will mark the entire subtree of that node as well. By marking the `return` statement as unmodified will leave this subtree unaffected and cause efficiency benefits.

Keeping track of modifications allowed the generator to only modify those places where it was necessary. Only the new or modified source code elements get generated, and every other part of the source code gets copied from the original source file. In the example, this works in the following way. The generator starts traversing the refactored AST from root to bottom, in a preorder strategy. While it finds unmodified nodes, it just copies the source code from the original file into the refactored file. This is based on the position information stored in the AST. This goes on until it finds a modified node in this case, an inserted *block* statement. Next, it generates the *block* statement. More precisely, it generates only the starting bracket, because there are still unvisited descendants of the *block* node in the AST. When the traversal goes to the next child, it finds an unmodified node again. It does so the same way as before, it copies the code from the original source code, only this time, it will insert the copied code with an increased indent, because the generator keeps track of the fact that we are now in a *block* statement. After every descendant has been visited and copied, we arrive back at the leave-visit for the *block* statement. The generator inserts the closing bracket into the right place, and a visit continues. Since every other node is unmodified, everything else is copied, and the generation is complete.

Generating only the required code parts created nearly the same code as the original, with most indentation and whitespaces in the right place. This was an important request from developers because interviews showed that they did not want to bother with fixing the indentation [30].

3.6.3 Creating Diffs

As soon as the generation process ended, it become possible to present the refactored code to the developers. However, reviewing entire unannotated files is not a welcomed idea by developers. Since this is the output of an automated process, users like to check what changes the automation process will apply.

To meet the need of the users, we will only show their a patch (unified diff file) as output. This patch file contains the differences between the refactored source file and the original one. This enables the developer to review what changes the automation process made on the code. Besides this, it allows the user to make a decision at the end of the process of whether to accept or reject the suggested

Listing 7: Output diff file

```
--- Example.java (original)
+++ Example.java (refactored)
@@ -1,7 +1,8 @@
 public class Example {
     public static int limiter(int x) {
-        if (x > 10)
+        if (x > 10) {
             return 10;
+        }
         return x;
     }
 }
```

refactoring. If it is the former, the user can apply the diff on the original source code, and this will transform it into the refactored code. *Thus Step 8 of Process 1 is covered.*

For example, Listing 7 shows the diff file for the refactoring of the example in Listing 1. Note that this introduces which lines are marked for deletion (starting with a "-") and which ones are marked as added (starting with a "+").

4 Evaluation

Previously, in Section 2 we mentioned that we were able to test our approach on a number of systems with the assistance of software engineering companies in an EU project. These companies were founded only in the last two decades and some of their projects were initiated before the millennium. Their projects consisted of about 5 million lines of code altogether, written mostly in Java, and covered different IT areas like ERPs (enterprise resource planning and business process management), ICMS (integrated collection management systems) and online PDF Generation. Participating companies received an extra budget to improve their source code by refactoring. First, they did the task manually, and later, after we had developed the automated tool, they used the tool to commit thousands of automatic refactorings.

At the end of the project, our tool was tested exhaustively. The companies participating in the project performed around 6,000 refactorings altogether which fixed over 11,000 coding issues. Interviews with the developers revealed that they found it to be easy to work with the tool in a lots of situations and they intend to keep using it in the future. An in-depth analysis of the results was discussed in a previous study [30] and a summary of our observations is available as an experience report [28].

5 Discussion

5.1 Performance

Our refactoring tool was implemented in Java. One of the requirements for our tool was for it to be quick-acting from a user perspective. This required that we to optimize each step for speed. The most important optimization we preformed was with the Reverse AST-search algorithm.

Building the spatial database for the whole system was an unnecessary overhead. To reduce the search space, we built the spatial index based on the issue the user had selected. The rule violation has information about which source file it is in. Only these file elements were added to the R-tree and it greatly reduced the search space. As a comparison, building the entire search space on a PC[2] for the log4j[3] project took 221 ms and used 46 MB of memory, while building only one file took 53 ms and memory used was less than a kilobyte.

Further optimizations helped to speed up the process as well. For example, the the filtering step moved a few steps ahead in the Reverse AST-search algorithm's execution order. Filtering was applied while building the search space. Only those AST nodes were added to the R-tree where the type of node matched the type of refactoring transformation parameter. This way, executing a single search operation takes less than a millisecond.

The above improvements besides some other tweaks made our tool quick and this appealed to developers. We did not make detailed measurements of the tool's performance in our studies, but in general the tool performed well.

5.2 Threats to Validity

We have identified a few validity threats that might affect the the internal and external validity of our results. Here, we discuss the validity of our findings.

Usage of Java

We have only considered Java as the target of our actions. Some of the other languages may require different approach. Nevertheless, our process is readily adaptable to most text-based programming languages.

Application of a Third-Party Tool

We provided support to the developers in identifying coding issues with a third-party static analyzer, namely PMD. Naturally, this was a great help in identifying problematic code fragments, but it might have introduced many unnecessary steps during the refactoring process. There is a risk here that by using other analyzers or by using our own, we might skip the AST-search part. For example, if we were to develop our own issue finder tool, we could directly report the AST node where the

[2]Intel i7 3.40 GHz with 8 GB ram
[3]`http://logging.apache.org/log4j/1.2`

problem is located. However, our process makes our refactoring tool independent of a single third-party static analyzer. The way, it is constructed makes it capable to switch to another analyzer with only a slight modification.

6 Related Work

Since Opdyke introduced the term *refactoring* in his PhD dissertation [20] and Fowler published a catalog of refactoring 'bad smells' [6], many researchers have studied this technique to improve the maintainability of software systems. Just a few years later, Wake [31] published a workbook on the identification of 'smells', and suggested ways of recognizing the most important ones and some possible ways to fix them by applying the appropriate refactoring techniques. Five years after the appearance of Fowler's book, Mens et al. [16] published a survey with over 100 related papers in the area of software refactoring. But the popularity of the topic continues to this day.

Automation techniques can support the regular task of refactoring and they are intensively studied by researchers. Ge et al. implemented the BeneFactor tool which detects developers' manual refactoring and reminds them that automation is available; and then it performs the refactoring automatically [9, 8]. Vakilian et al. proposed a compositional paradigm for refactoring (automate individual steps and letting programmers manually compose the steps into a complex change) and implemented a tool available that support it. Henkel et al. implemented a framework which captures and replays refactoring actions [12]. Jensen et al. used genetic programming for automated refactoring and the introduction of design patterns [13]. Also, there are many approaches to support specific refactoring techniques, like extract method [3, 26], refactoring to design patterns [1] and performing clone refactoring [33].

In a recent study, Fontana et al. examined refactoring tools designed to remove code smells [5]. They evaluated the following tools: Eclipse,[4] IntelliJ IDEA,[5] JDeodorant,[6] and RefactorIT.[7] In the case of JDeodorant, they said that this "*is the only software currently available [that is] able to provide code smell detection and then to suggest which refactoring to apply to remove the detected smells.*" To evaluate the other refactoring tools, they relied on the code smell identification of iPlasma[8] and inCode[9]. In an earlier study, Pérez et al. also identified smell detection and automatic corrections as an open challenge for the community, and proposed an automated bad smell correction technique based on the generation of refactoring plans [23]. Code-Imp is an automated refactoring platform designed for the Java language by Moghadam at el. [17]. It can apply a range of refactorings,

[4]`https://www.eclipse.org/`
[5]`http://www.jetbrains.com/idea/`
[6]`http://www.jdeodorant.com/`
[7]`http://sourceforge.net/projects/refactorit/`
[8]`http://loose.upt.ro/reengineering/research/iplasma`
[9]`http://www.intooitus.com/products/incode`

it supports several search types, and implements over 25 software quality metrics that can be combined in a variety of whys to form a fitness function.

There are many IDEs available with automatic refactoring capabilities and they support typical code restructurings (e.g. renaming variables, classes) and some common refactorings from the Fowler catalog. For instance, IntelliJ IDEA was one of the first IDEs to implement these techniques and it is able to support many languages (e.g. PHP, JavaScript, Python), not just Java which it was originally designed for. Eclipse and NetBeans[10] also implement similar algorithms. However, these IDEs have only limited support for the automatic refactoring of programming flaws. Eclipse calls these *Quick Fixes*. For example, it can organise imports, remove dead code and optimize string concatenations. NetBeans calls them *Java Hints*. These are able to fix things such as missing switch cases, suspicious equals calls or a dead branch. NetBeans also reports Java code metrics flaws, like *Class has too many methods* or *Method with multiple return points*. Moreover, it is possible to create custom, user-defined Java Hints.

The IDEs mentioned above use an incremental build system to bound text to certain source code elements. In contrast, our approach uses a reverse AST-search method to locate AST nodes. What is more, the above tools are IDEs with refactoring capabilities, while our tool is a server-side refactoring framework and its IDE plugins are simply interfaces that communicate with the server.

7 Final Remarks

In this article, we presented an automated process for refactoring coding issues. We used the output of a third-party static analyzer to find refactoring suggestions. Then we created an algorithm that is capable of locating a source code element in an AST based on textual position information. The algorithm transforms the source code into a searchable geometric space by building a spatial database. After, we conducted an exhaustive evaluation which confirmed that our approach can be adapted to a real-life scenario, and it definitely provides viable results.

In the future, we plan to use a reverse AST-search to create a mapping between multiple, different structured ASTs. Also, we would like to investigate how effectively our approach can be used to determinate what source code element the user is examining, just by looking at their text selection.

Acknowledgment

This research work was supported by the EU supported Hungarian national grant GOP-1.2.1-11-2011-0002. Here, I would also like to thank my supervisor, Rudolf Ferenc, for his support in this research work.

[10] `https://www.netbeans.org/`

References

[1] Christopoulou, Aikaterini, Giakoumakis, E. A., Zafeiris, Vassilis E., and Soukara, Vasiliki. Automated refactoring to the strategy design pattern. *Information and Software Technology*, 54(11):1202–1214, November 2012.

[2] Dig, Danny and Johnson, Ralph. The Role of Refactorings in API Evolution. In *Proc. of the 21st IEEE Int. Conference on Software Maintenance (ICSM2005)*, pages 389–398. IEEE Comp. Soc., 2005.

[3] Feldthaus, Asger and Møller, Anders. Semi-automatic rename refactoring for javascript. *SIGPLAN Not.*, 48(10):323–338, October 2013.

[4] Ferenc, Rudolf, Beszédes, Árpád, Tarkiainen, Mikko, and Gyimóthy, Tibor. Columbus – Reverse Engineering Tool and Schema for C++. In *Proceedings of the 18th International Conference on Software Maintenance (ICSM'02)*, pages 172–181. IEEE Computer Society, 2002.

[5] Fontana, Francesca Arcelli, Mangiacavalli, Marco, Pochiero, Domenico, and Zanoni, Marco. On experimenting refactoring tools to remove code smells. In *Scientific Workshop Proceedings of the XP2015*, XP '15 workshops, pages 7:1–7:8, New York, NY, USA, 2015. ACM.

[6] Fowler, Martin. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Longman Publishing Co., Inc., 1999.

[7] FrontEndART Ltd. SourceMeter website: https://www.sourcemeter.com.

[8] Ge, Xi, DuBose, Quinton L., and Murphy-Hill, Emerson. Reconciling manual and automatic refactoring. In *Proc. of the 34th Int. Conference on Software Engineering (ICSE2012)*, pages 211–221. IEEE Press, 2012.

[9] Ge, Xi and Murphy-Hill, Emerson. Benefactor: A flexible refactoring tool for eclipse. In *Proc. of the ACM Int. Conference Companion on Object Oriented Programming Systems Languages and Applications Companion (OOPSLA2011)*, pages 19–20. ACM, 2011.

[10] Guttman, Antonin. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD '84, pages 47–57, New York, NY, USA, 1984. ACM.

[11] Harman, Mark and Tratt, Laurence. Pareto optimal search based refactoring at the design level. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, GECCO '07, pages 1106–1113, New York, NY, USA, 2007. ACM.

[12] Henkel, Johannes and Diwan, Amer. Catchup!: Capturing and replaying refactorings to support api evolution. In *Proc. of the 27th Int. Conference on Software Engineering (ICSE2005)*, pages 274–283. ACM, 2005.

[13] Jensen, Adam C. and Cheng, Betty H.C. On the use of genetic programming for automated refactoring and the introduction of design patterns. In *Proc. of the 12th Annual Conference on Genetic and Evolutionary Computation (GECCO2010)*, pages 1341–1348. ACM, 2010.

[14] Kim, Miryung, Zimmermann, T., and Nagappan, N. An empirical study of refactoring challenges and benefits at Microsoft. *IEEE Transactions on Software Engineering*, 40(7):633–649, July 2014.

[15] Manolopoulos, Y., Nanopoulos, A., Papadopoulos, A.N., and Theodoridis, Y. *R-Trees: Theory and Applications*. Advanced Information and Knowledge Processing. Springer London, 2010.

[16] Mens, Tom and Tourwé, Tom. A Survey of Software Refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004.

[17] Moghadam, Iman Hemati and Ó Cinnéide, Mel. Code-imp: A tool for automated search-based refactoring. In *Proceedings of the 4th Workshop on Refactoring Tools*, WRT '11, pages 41–44, New York, NY, USA, 2011. ACM.

[18] Mylopoulos, John, Chung, Lawrence, and Nixon, Brian. Representing and using nonfunctional requirements: A process-oriented approach. *IEEE Transactions on Software Engineering*, 18(6):483–497, 1992.

[19] O'Keeffe, Mark and Cinnéide, Mel Ó. Search-based refactoring for software maintenance. *Journal of Systems and Software*, 81(4):502 – 516, 2008. Selected papers from the 10th Conference on Software Maintenance and Reengineering (CSMR 2006).

[20] Opdyke, William F. *Refactoring object-oriented frameworks*. PhD thesis, University of Illinois, 1992.

[21] Oracle Corporation. OpenJDK website: http://openjdk.java.net/.

[22] Ouni, A., Kessentini, M., Sahraoui, H., and Hamdi, M. S. Search-based refactoring: Towards semantics preservation. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 347–356, Sept 2012.

[23] Pérez, Javier and Crespo, Yania. Perspectives on automated correction of bad smells. In *Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (Evol) Workshops*, IWPSE-Evol '09, pages 99–108, New York, NY, USA, 2009. ACM.

[24] Pinto, Gustavo H. and Kamei, Fernando. What Programmers Say About Refactoring Tools?: An Empirical Investigation of Stack Overflow. In *Proc. of the 6th Workshop on Refactoring Tools*, pages 33–36. ACM, 2013.

[25] Rigaux, Philippe, Scholl, Michel, and Voisard, Agnes. *Spatial databases: with application to GIS*. Morgan Kaufmann, 2001.

[26] Silva, Danilo, Terra, Ricardo, and Valente, Marco Tulio. Recommending automated extract method refactorings. In *Proc. of the 22nd Int. Conference on Program Comprehension (ICPC2014)*, pages 146–156. ACM, 2014.

[27] Szőke, Gábor, Nagy, Csaba, Ferenc, Rudolf, and Gyimóthy, Tibor. A Case Study of Refactoring Large-Scale Industrial Systems to Efficiently Improve Source Code Quality. In *Proc. of ICCSA 2014*, pages 524–540. Springer, 2014.

[28] Szőke, Gábor, Nagy, Csaba, Ferenc, Rudolf, and Gyimóthy, Tibor. Designing and Developing Automated Refactoring Transformations: An Experience Report. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 693–697, March 2016.

[29] Szőke, Gábor, Nagy, Csaba, Fülöp, Lajos Jenő, Ferenc, Rudolf, and Gyimóthy, Tibor. FaultBuster: An Automatic Code Smell Refactoring Toolset. In *Proc. of SCAM 2015*, pages 253–258. IEEE, 2015.

[30] Szőke, Gábor, Nagy, Csaba, Hegedűs, Péter, Ferenc, Rudolf, and Gyimóthy, Tibor. Do Automatic Refactorings Improve Maintainability? An Industrial Case Study. In *Proc. of ICSME 2015*, pages 429–438. IEEE, 2015.

[31] Wake, William C. *Refactoring Workbook*. Addison-Wesley Longman Publishing Co., Inc., 1 edition, 2003.

[32] Xing, Zhenchang and Stroulia, Eleni. Refactoring Practice: How It is and How It Should Be Supported - An Eclipse Case Study. In *Proc. of the 22nd IEEE Int. Conference on Software Maintenance (ICSM2006)*, pages 458–468. IEEE Comp. Soc., 2006.

[33] Yoshida, Norihiro, Choi, Eunjong, and Inoue, Katsuro. Active support for clone refactoring: A perspective. In *Proc. of the 2013 ACM Workshop on Workshop on Refactoring Tools (WRT2013)*, pages 13–16. ACM, 2013.