Aroma: An Exploration in Tool-Assisted Refactoring

Drew Guarnera
Department of Computer Science
The University of Akron
Akron, OH USA
dtg3@zips.uakron.edu

***Abstract*—This paper proposes Aroma, an experimental tool used to assist developers in refactoring source code. Aroma takes a different approach to this task using the Fowler refactoring catalogue as iterative building blocks for each stage of the target refactoring. The motivation behind this application is to perform refactorings that mimic the behavior of a developer and to test the efficacy of using XML tools such as XSLT and Xpath combined with the srcML document format. At the conclusion of this stage of the tool's development, Aroma shows capability in performing both code level and design level transformations to source code within a single source code file and across file boundaries with input from the developer. While still in an early stage of development, a clear development path for improvement is also laid out in the form of future work for the project.**

Keywords—srcML; XML; XSLT; Xpath; refactoring; code smells

I. Introduction

Software maintenance can account for anywhere from 40% to 80% of the total software development costs during an application's evolutionary lifespan. A large portion of this cost is due to the fact that as software grows and evolves it becomes more difficult to add features and maintain the application. This problem is exacerbated by the fact that most times, the push for new features, bug fixes or other tasks supersedes the need to perform proper maintenance activities like refactoring to prepare a software system for such changes. The ultimate result is that project goes into technical debt, which leads to code decay, and an application that is exponentially more difficult and fragile to work with.

It's not surprising that refactoring gets neglected in favor of new features and fixes, as refactoring can be time consuming to perform. While refactoring strengthens the internal structure of the code, a disciplined approach must be taken to prevent causing regression of the software, which usually requires a strong testing suite behind the scenes. This can prove to be a very time consuming proposition for most developers, especially those who lack experience with the software system. With all the other responsibilities on a developer's plate at any given day, most developers or management would opt for new forward facing features or fixes in place of behind the scenes improvements.

Automated tools do exist to help mitigate the time consuming and error prone nature of refactoring. However they are not without their limitations. A survey of six refactoring tools was performed in [5] and found that on average, the tools were not up to industrial software development standards. All of the tools examined were found to not fully support the refactoring process and to make matters worse, the installation and configuration of the software alone was found to be difficult. With a general lack focus on ease of use and tutorial support combined with other issues left most tool overall rating for that study to be at or below average. With this in mind it becomes more apparent why many of the automated refactoring tools are not adopted and used by mainstream developers.

In [6] a study was performed to see how developers utilized automated refactoring tools. For this work, 1268 hours worth of interaction data was collected over three months and then interviewed nine of the twenty six developers to get more in-depth feedback on data that was collected. Their study found that simple refactorings with context-aware and lightweight invocation methods dominated the usage patterns. Developers had issues with automated naming decisions, general trust, and predictability of the automated results. This lead to developers using automated refactorings to change at most six lines in 82% of all refactorings observed and 84% of all the refactorings only made changes within a single file. Due to this lack of trust in the automated approach, tool based refactoring goes underutilized and fails to accomplish the purpose it was designed for in the first place. With these facts in mind this paper proposes Aroma, a different approach to tool based software refactoring for applications written in C++.

Aroma utilizes the srcML document format [2, 3], along with Xpath, Xslt, and Python with lxml to query and transform source code and apply transformations even across files. Unlike other tools, which strive for full or nearly full automation, Aroma focuses on assisting the developer in making refactorings. By performing automation when appropriate and requesting developer input either before or during a refactoring task, the developer is able exert more control over the final result of the refactoring.

Additionally Aroma's approach to refactoring is to perform each change as a series of refactorings instead of just having one large or final change produce the end result. Aroma utilizes the refactoring approach of Martin Fowler [4] and his catalog of refactorings to complete small refactorings and to use a series of smaller refactorings to accomplish larger changes.

This paper will introduce the tools and technologies used in the creation of Aroma (Section II), the development methodology behind Aroma (Section III), an example overview of use and results (Section IV), threats to validity (Section V) and future work and conclusion (Section (VI).

II. Tools and Technologies

To begin to understand how Aroma works, it is necessary to examine the underlying technologies.

FIGURE I. AROMA ARCHITECTURE

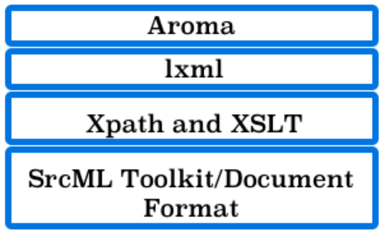


Fig. 1. Visual representation of the Aroma architecture and associated technologies

As can be seen from Fig 1, the core foundation of Aroma is based on the srcML document format and toolkit. The srcML toolkit is capable of taking source code files, either individually or as an entire project, and converting them into an XML representation to include syntactic context and then back again into source code.

The srcML toolkit is able to perform these conversions in a lossless fashion, allowing for preservation of whitespace, comments, and any other source code contents throughout the process. While this may sound time consuming, the srcML toolkit is highly optimized and these conversions can be performed in mere seconds. Recent benchmarks of the srcML toolkit show it can process the entire Linux kernel in around a minute using a modern multi-core desktop computer.

What really makes the srcML format so practical to use for Aroma is the converted XML format for the source code. Using the context provided by srcML's custom XML markup tag set allows for a multitude of lightweight tools to be applied to the document such as queries with Xpath and XML manipulations using XSLT (Extensible Stylesheet Language Transformations). Additionally, the srcML format stores the content of all the source code files in one large XML document which allows for easy access to perform modifications to any location in any file in the entire source code project.

Working with XSLT and Xpath for the uninitiated has a steep learning curve. During a majority of the initial development of Aroma, about 85% of development the time was spent trying to learn and understand XSLT. Writing transformations in XSLT is a unique experience and takes some getting used to. An XSLT stylesheet isn't like regular string manipulations in the traditional fashion with a programming or scripting language. Instead XSLT traverses the entire XML document and stops at user-defined templates where a "match" is found with a given Xpath expression. While the matching part (Xpath) isn't too hard to figure out, the actual process of keeping, removing, or moving the XML contents around can be quite troublesome.

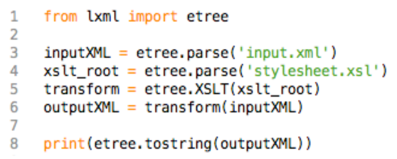
Since XSLT is a stylesheet for XML, there aren't many clean and friendly ways to run and test stylesheets. While the srcML toolkit client is capable for applying XSLT stylesheets to XML files, the output phases lacked debugging messages. This lead to, on more than one occasion, a blank or oddly transformed XML result being generated after applying a stylesheet. Without any detailed debugging or error messages it was very difficult to troubleshoot issues.

Another option would be to use a web based stylesheet tester or a web browser itself, but may of the online offerings suffered from similar issues with lack of debugging assistance and the interfaces were so cramped or poorly designed that ultimately it was impossible to incorporate them into a productive workflow. With IDEs, or other stand-alone applications, it was a similar story. The ability to simply run the stylesheets locally yielded a few options, some free but difficult to even setup and configure, and some were exclusively for XSLT development, but cost several hundred dollars.

In spite of the high learning curve and workflow issues, XSLT ended up proving to be remarkably fast and effective at manipulating the srcML XML documents. It is also a much more efficient approach than traversing an XML document manually in code and making changes that way, which was briefly attempted during the development of Aroma. Ultimately though, it was using lxml, an XML library for Python that really helped to solidify a smooth and sane workflow along with offering a complimentary feature set that integrated quite well with standard XSLT and Xpath.

Lxml has many useful features as the core template and transformation execution engine in Aroma. First it can execute raw xslt files on the local disk against XML from a string in memory or from a local file. This cleared up the work flow issue as lxml with Python helped to create a staging area for testing XSLT transformations, complete with line errors for debugging in only a few lines of python code as shown below in Fig 2.

FIGURE II. SANDBOX CODE



```python
from lxml import etree

inputXML = etree.parse('input.xml')
xslt_root = etree.parse('stylesheet.xsl')
transform = etree.XSLT(xslt_root)
outputXML = transform(inputXML)

print(etree.tostring(outputXML))
```

Fig. 2. Example Python code for sandboxing XSLT against XML

This feature alone makes lxml a must for testing and sandboxing development for the XSLT stylesheets to be used with Aroma.

Second, lxml can perform Xpath and XSLT transformations from strings in memory, which means that transformations performed by aroma can be generated dynamically. This feature comes in especially handy when trying to name parameters or classes, or when performing more operations that are more difficult with XSLT alone such as moving operations. Lxml also has the necessary functionality to create and manipulate XML structures in memory and interpret Xpath results as more than just a string result. This allows XSLT stylesheets to be run to handle bulk changes while lxml can be used directly to make more delicate changes that would be cumbersome with XSLT.

III. Development Methodology

The initial goal of Aroma was straightforward. Determine the efficacy and plausibility of using XSLT to perform Fowler refactoring transformations along with the srcML document format. After reading about the usage of automated tools for refactoring by developers [6] and on the state of automated refactoring tools [5], the goals have been broadened to a few essential principles:

A. *Minimize Setup Time*

Refactoring tools have been shown in [5] to be difficult to get setup and configured. It seems obvious that if configuration and setup is too arduous of a task then an application will most certainly not meet the needs of it users. With that in mind, Aroma strives to maintain the most basic of requirements for use:

1. Python
2. lxml
3. Aroma
4. A valid srcML document as input

Each of the preceding requirements is easy to acquire for the three major computing platforms (OSX, Windows, and Unix based systems).

B. *Small Refactorings*

The concept of performing and planning refactorings as a series of smaller mini-patterns and mini-transformations is not new [1]. For Aroma, the building blocks of each transformation start with Martin Fowler's refactoring catalog [4]. This online catalogue lists taxonomy for common refactorings as well and provides simple examples. Together this provides a clear universal terminology for each refactoring task and a basic understanding of the expected results. With small refactorings at the core of Aroma's ideology, the final results should mimic a developer's approach to the problem. This leads to the third goal.

C. *Stage Changes*

All refactorings and changes should be done in step increments and not simply applied all at once by Aroma. By performing changes in steps or stages, changes could be tracked via version control tools like Subversion or Git and in this way refactorings could be walked through or rolled back and provide a type of unlimited "undo" and "redo" feature. Developers could also merge in a complete set of changes or even just a subset, which ever better met their needs. Additionally, users should be able to apply new or the same transformations to any step in the previous transformation process in the event of a mistake by the tool or end user.

D. *Assisted vs. Automated*

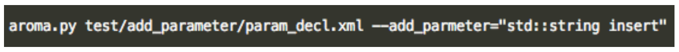
While full automation of transformations with near perfection is the proverbial holy grail of transformation tools. The reality is that a developer's input will be required for certain aspects. Instead of fully automatic transformations, Aroma focuses on making the process of refactoring as straight forward as possible so a person with little experience with a software system (like an intern) could be assigned a refactoring task by a superior and, using Aroma, be able to quickly and efficiently make changes that could be later be reviewed my a more experienced team member. This goal seems to be a far more attainable and meaningful endeavor than full automation.

With the preceding ideas at the core of Aroma's development methodology, Aroma aims at being a tool that improves on the current state of tool assisted refactoring and have real world application.

IV. Using Aroma

At this time, Aroma is in a very early experimental phase. As such, Aroma supports only a small selection of transformations from the Fowler catalog [4] with many other assisting transformations to support the core tasks.

FIGURE III. ADD PARAMETER CLI COMMAND

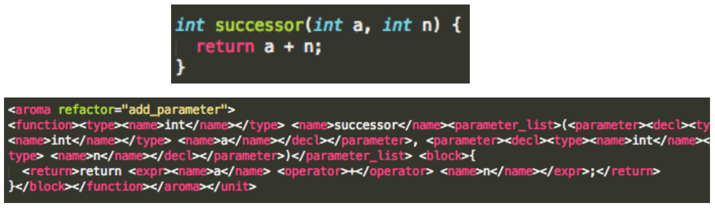


```
aroma.py test/add_parameter/param_decl.xml --add_parmeter="std::string insert"
```

Fig. 3. Example Aroma command to perform an "add parameter" refactoring

While command line only at the time of this paper, Aroma is relatively straightforward to use. Fig. 3 shows how to call an "add parameter" refactoring on a srcML input file. In the case of Fig. 3 note that the "add parameter" refactoring includes the type and name of the parameter to add as though it was typed into the source itself.

FIGURE IV. EXAMPLE FUNCTION AND SRCML



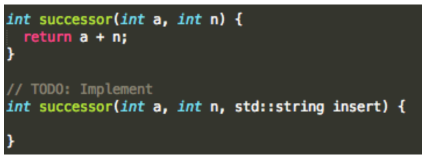
```
int successor(int a, int n) {
  return a + n;
}
```

```
<aroma refactor="add_parameter">
<function><type><name>int</name></type> <name>successor</name><parameter_list>(<parameter><decl><ty
<name>int</name></type> <name>a</name></decl></parameter>, <parameter><decl><type><name>int</name><
type> <name>n</name></decl></parameter>)</parameter_list> <block>{
  <return>return <expr><name>a</name> <operator>+</operator> <name>n</name></expr>;</return>
}</block></function></aroma></unit>
```

Fig. 4. Example function (top) and srcML version with Aroma Tags (bottom)

Looking at Fig. 4 (top), the original function is used as an example candidate for an "add parameter" refactoring. The associated srcML representation also in Fig.4 (bottom) will be used as input for the Aroma command shown in Fig. 3. While all of the srcML is standard, the `<aroma>` tag is unique for Aroma only and helps the tool to identify the region and type of refactoring to apply via the "refactor" attribute. Looking at Fig. 5, it can be seen that the refactoring was not simply tacking the new parameter at the end of the existing successor function. Instead the transformation more similarly resembles a refactoring a live developer would do by first making a duplicate of the original function and then adding the parameter to the duplicate. Also note that a convenient "TODO:" comment was added above the new function to indicate a location that requires attention from the developer, namely the implementation of the new function. While this is not a mandatory part of the refactoring, it is a nice addition to as many IDEs including Eclipse support the identification of "TODO:" in a comment as a notification similar to a warning or error to the developer. Not to mention the "TODO:" represents an addition a real developer might make during a refactoring while an automated tool might not.

FIGURE V. EXAMPLE FUNCTION REFACTORED



```
int successor(int a, int n) {
  return a + n;
}

// TODO: Implement
int successor(int a, int n, std::string insert) {

}
```

Fig. 5. Result of example function refactored by Aroma with "add parameter"

One other refactoring that Aroma is capable of performing is a "pull up field" refactoring. This transformation is unique in that it potentially requires the refactoring to be made across files. Since the srcML format is capable of storing the contents of multiple files within a single srcML document the fact that different files need to be modified to satisfy this refactoring task becomes a non-issue for Aroma.

Much like the previous example in Fig. 4, Aroma tags are added around the field (or fields) that need to be move up in an inheritance hierarchy and another around the class to receive the fields. The exception is that an additional attribute called "role" is added to the tags to indicate which part of the refactoring are the fields (given the value "source") and the class they will be moved to (given the value "destination").

Upon completion of this refactoring, the fields are moved, but transferred to the parent class with the access specifier of protected. Again, while the moved fields may be ultimately hidden by another abstraction in the parent class at a later time, refactoring the code in this manner allows the existing code to retain its same functionality.

While the refactorings mentioned above are mostly straightforward affairs, writing these transformations are still quite difficult to perform, and simply the act of moving, removing, changing or inserting text to or from one location to another is only half of the challenge. The remainder of the difficulty comes in trying to make smart decisions when performing these actions. An example of this can be seen in the duplication of the method when adding a parameter.

While the refactoring would have been perfectly valid to simply insert the parameter and check that refactoring off the list, the goal with Aroma is to make refactorings in a way that is most like what a developer would do. Attempting to minimize the amount of non-compliable code and unintended side effects are one of the primary concerns. Adding a comment to the change to draw attention to work that needs to be done much like a person editing the source would do. These decision and approaches take serious consideration and several iterations to elevate the transformations from being solely technically correct to also being thoughtfully designed in a way that a developer would find natural and useful.

V. THREATS TO VALIDITY

As stated earlier in this paper, Aroma is in a very early and experimental phase. As such, it has a limited number of refactorings it is capable of performing and has only been tested in a very limited environment. While the refactorings that are currently available for use should work properly in most cases they have yet to be proven with real world systems. Another draw back to a lack of real world testing is that some design approaches, such as the proposed notion of saving each step of a transformation, might end up scaling quite poorly when applied to real world systems.

In addition to a small number of supported refactorings that have been implemented, given time constraints only the most straightforward of refactorings could be completed in Aroma. This means that the performance of larger more complex refactorings requiring multiple intermediary refactorings based on the Fowler catalog cannot be spoken for at this time.

Compared to other refactoring tools, Aroma currently is only capable of making transformations where the locations are already tagged. Aroma on it's own is not capable of locating sections of code for refactoring. Without a GUI or other practical input method for this tool, all Aroma tags must be added manually to a srcML document prior to a transformations execution. Also because Aroma is not utilizing the new srcML Toolkit library libsrcML, Aroma requires users to utilize the srcml command line application separately from Aroma to even generate the document format. Which is not an improvement over complaints leveraged against other similar refactoring applications.

Also when compared to other refactoring tools, this more interactive or assistive approach is not the norm for this category of application. While the theory behind assistive versus automated seems to make sense given trust issues / reliability issues some developers have with fully automated tools seems reasonable to lean toward assistive, it is speculative at this time whether a tool designed in this way will be adopted at all much less become mainstream.

VI. FUTURE WORK AND CONCLUSION

Since Aroma is in a very early stage of development, the tool still has many shortcomings. However a clear development path complete with many future improvements to address these issues is proposed below with the intent of making Aroma into a tool that professional developers would find attractive to use:

A. Larger Refactoring Support

First and foremost on the list of improvements to Aroma would be to add support for additional refactorings from Fowler's catalogue. With a broader selection of refactorings to apply, Aroma's large scale refactoring capabilities via a smaller refactorings can finally be assessed.

B. *Smell Detection and Refactoring Suggestions*

Aroma has no built in capabilities to detect or suggest refactorings. This feature is necessary if for no other reason than to provide parity with other tools in its category. Additionally, this feature can also be used to quickly assist and introduce developers to using the tool and its feature set. While many tools exist for the express purpose of code smell detection or code violations, no specific tool or approach has been selected to full fill this role. Ideally, a lightweight approach also utilizing the srcML document format and similar XML tooling such as Xpath would be a natural fit for Aroma.

C. *Graphical Interface Support*

As presented in [5] GUI tools and IDE plugins were among the more widely used and preferred tools for refactoring. Ideally with a simplified graphical user interface perhaps similar to something on a mobile device, quick identification of refactoring locations and refactoring types can be selected in a few quick taps or clicks. This is also a way in which the srcML document format can be marked up with Aroma tags without the user having to delve into the raw XML text.

D. *Integration with Version Control Systems*

This proposed feature should be the one that most separates Aroma from other tools of its kind. With each stage of the transformation versioned in a separate branch of the repository, developers would be able to undo, redo, apply, and reapply transformations to any stage of the process without having to restart the entire process all over again. The changes could be viewed in a simple step-by-step way and the version control system would have a detailed and easily accessible record of all changes made to complete the refactoring. Best of all, when the refactoring is complete and approved it can simply be merged into any development branch and added to the source repository.

E. *Integration with the srcML library for Input Generation*

The addition of integrating the srcML library directly into Aroma using libsrcML would allow the XML document and input format to be a transparent aspect of the application to the user. This would support the development methodology behind Aroma by encapsulating the srcML document generation into the tool itself removing an additional requirement for setup. Also handling the XML generation internally would eliminate the need for users to use or even understand the srcML document format or toolkit allowing for their full attention on the refactoring task in font of them.

F. *Language Support*

While Aroma will always be limited in terms of language support by whatever the srcML toolkit supports. Presently Aroma only supports a single language, C++. Future versions should be expanded to support Java, C#, and C which are the remaining languages currently supported by srcML. Along with the addition of languages to widen the potential user base of Aroma, expanding the language to Java and C# will allow Aroma to be directly compared with other refactoring tools to examine how it compares. This leads to another aspect of future work.

G. *Tool Comparisons*

Once Aroma reaches a more mature development state and a more comprehensive feature set, comparisons between Aroma and other tools should be performed. Ideally a survey should be done to find the most popular tools at the time to include commercial tools like Resharper (C#) and research projects such as ClangMR (C++)[7]. Comparing these projects and others especially in different language domains could help to show not only how Aroma stacks up in terms of features, but it's performance using support for a variety of languages when compared to tools specifically designed for one development language.

In its present state, Aroma is little more than a proof of concept. However this paper has show that even in its limited capacity, Aroma's lightweight approach to refactoring using a combination of XSLT and the srcML document format shows promise, and while the refactorings are time consuming to develop, they are in fact feasible. Combined with a clear development methodology and a detailed plan for improvement in place Aroma has a road map that could lead to success. With additional support and development time, it does not seem unreasonable to believe that Aroma could become an application that developers find helpful and perhaps practical for everyday use.

References

[1] Cinneide, Mel O., and P. Nixon. "A Methodology for the Automated Introduction of Design Patterns". Software Maintenance, 1999. (ICSM '99) Proceedings. IEEE International Conference on. Web.

[2] Collard, M. L., M. J. Decker, and J. I. Maletic. "Lightweight Transformation and Fact Extraction with the srcML Toolkit". Source Code Analysis and Manipulation (SCAM), 2011 11th IEEE International Working Conference on. Web.

[3] Collard, M. L., M. J. Decker, and J. I. Maletic. "SrcML: An Infrastructure for the Exploration, Analysis, and Manipulation of Source Code: A Tool Demonstration". Software Maintenance (ICSM), 2013 29th IEEE International Conference on. Web.

[4] Fowler, M. "Refactoring.com."Web. <http://refactoring.com/catalog/>.

[5] Mealy, E., and P. Strooper. "Evaluating Software Refactoring Tool Support". Software Engineering Conference, 2006. Australian. Web.

[6] Vakilian, M., et al. "Use, Disuse, and Misuse of Automated Refactorings". Software Engineering (ICSE), 2012 34th International Conference on. Web.

[7] Wright, H. K., et al. "Large-Scale Automated Refactoring using ClangMR". Software Maintenance (ICSM), 2013 29th IEEE International Conference on. Web.