Unifying Code Refactorings of Different Languages

Introduction of CHAST and using Parser Generators as Refactoring Meta Language

Raphael Jenni
OST Eastern Switzerland University of Applied Sciences
Supervised by Prof. Dr. Farhad Mehta
FT 2022

Abstract

In this paper, the newly developed tool CHAST will be introduced, a tool that automates refactorings in a secure and reliable manner by using the concept of change isolation. This allows users to preview and confirm changes before they are applied to the actual codebase, avoiding unintended side effects and enabling users to easily roll back changes if necessary. CHAST is designed to be language-agnostic and to support a wide range of tools and functionality. It also simplifies the process of creating and sharing refactorings by providing a command-line interface and recipe format. The paper furthermore investigates the feasibility of implementing multi-language refactorings using parser generators as a refactoring meta language. It also evaluates CHAST in terms of change isolation and the effort required to create a refactoring. In future work, plans exist to continue developing CHAST and further enhancing its effectiveness in real-world scenarios. Therefore, CHAST's limits need to be explored, and potential trade-offs of using parser generators as a refactoring meta language need to be identified. Furthermore, other approaches for implementing multi-language refactorings need to be investigated. There also exist plans to add additional features to CHAST, such as installing software or running linters and formatters in a unified way.

Keywords: Refactoring, Parser Generators, Automation, AST Manipulation, Code Generation

1 Introduction

In every software project, the need for refactoring emerges sooner or later. Refactoring is understood as the process of restructuring existing computer code without changing its observable behavior. It is intended to improve the design, structure, and/or implementation of the software (or a part of it) to make it more maintainable, scalable, and efficient. Refactoring is a well-known technique in software engineering and is widely used in practice. However, much refactoring work is still done manually, which is time-consuming and error-prone. Many refactorings turn out to be repetitive and very similar to one another. Some refactorings may be automated and aided by integrated development environments (IDEs). For others, no automated support is available. Some projects aim to provide automated support for refactoring but are often limited to a specific language or a specific refactoring. If the same automation exists for a different language, it is most likely a completely different implementation, sharing nothing of its logic with the previous implementation. Trying to automate refactoring can be difficult and time-consuming. Furthermore, not all refactorings are the same. Different approaches must be taken depending on the complexity of the refactoring. Refactorings can be classified into five different categories. An overview is shown in Figure 1.

Level 1 covers very simple, text-based, refactorings and can be done with a well-crafted regex or a tool like Comby [vT]. For those, the tool does not need any or just limited explicit knowledge of the underlying language. However, although the refactoring may be simple, getting the automation right can still be very hard. The tricky part is to craft a search-and-replace expression that covers all edge cases and does not have any unwanted side effects.

At level 2 are more complex refactorings that require some knowledge of the underlying language, specifically local changes that only cover a single method or function. Such refactorings can sometimes also be done with methods described in level 1. Due to the limited scope of the refactoring, these refactorings are often easier to automate. Examples of this type of refactoring are replacing an *indexed for loop* with a *foreach loop* or replacing a *switch statement* with a *map lookup*.

At level 3, the definition of level 2 is expanded to class local changes. This means the refactoring impacts other parts of the code but only within the same class. Level 2 and level 3 refactorings together build the *syntax-based* refactoring type. By syntax-based, we mean that the refactoring knows the underlying language but does not require a semantic analysis of the code and can rely on the syntax. A general and reasonable way for this is to use a lexer and parser combination that can convert the abstract syntax tree (AST) back into code. This way, the refactoring can be applied to the AST, which then can be converted back into code. Level 3 refactorings may require a symbol table for resolving references depending on the refactoring. Some parsers include a symbol table out of the box; for others, it needs to be implemented manually.

At level 4, we move into the area of cross-cutting changes, particularly class local changes that require information from other files of the project. However, at this level, the impact on other parts of the code is limited to only needing to check

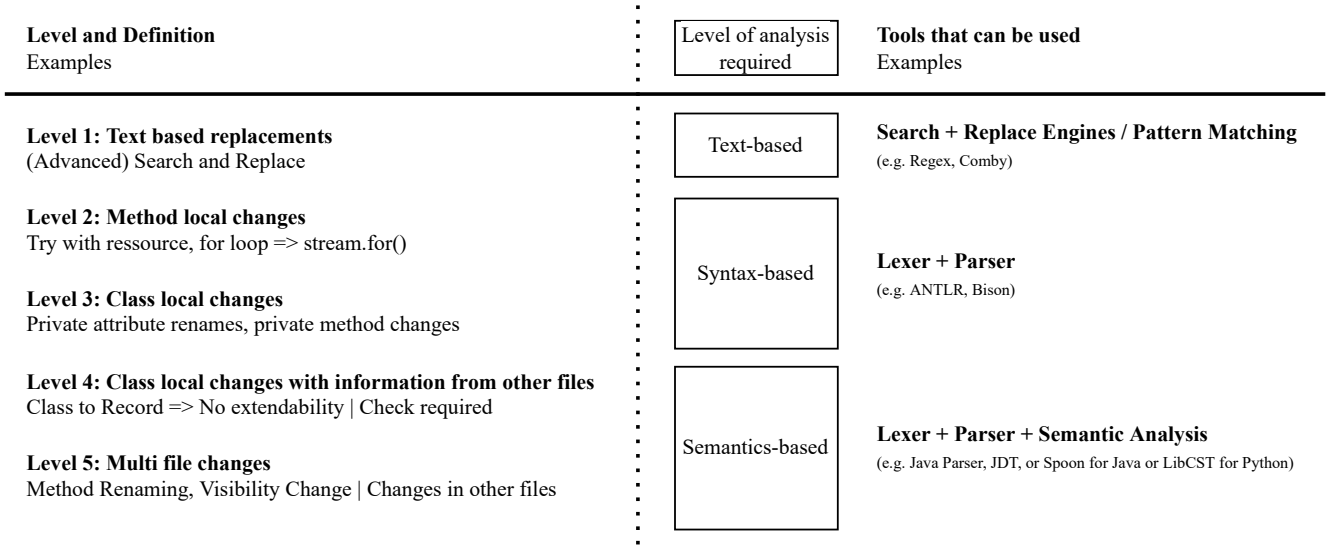

Figure 1. Types of refactorings

other files but not alter them. If, for example, a class is converted to a record and can therefore not be extended anymore, the refactoring has to check whether the class is used in other files and needs to prevent the change if there is another class extending the to-be-refactored class. This would require an analysis of the whole project and the resolution of dependencies. However, by limiting the analysis and only searching for package imports, the refactoring can still be done without a full dependency resolution.

This brings us to level 5, which is considered the most complex refactoring type. It is a cross-cutting change that results in changes in other files. An example: If a class is renamed, the refactoring has to check whether the class is used in other files and needs to rename the usages in those files. At this point, the refactoring needs to know the code's semantics and resolve dependencies. Depending on the refactoring, this might also result in changes that trigger further changes. Such refactorings need to be crafted very carefully and rigorously tested. Together with level 4, level 5 refactorings build the *semantics-based* refactoring type. Those refactorings, most of the time, have the most significant impact and yield the most considerable benefits. However, they are also the most complex and often crafted for specific tasks. For tasks that are not that common, you often are better off doing it by hand with the support of the IDE and unit tests. However, for refactoring in some new language features, i.e., features that could be used in many different projects, it is worth the effort to automate them, as other developers can use them.

Besides having different types of refactorings, there is no unified way to do refactorings. Every tool reinvents the way refactorings are done. Every command line interface (CLI) tool and every IDE has its own set of refactorings and its own way of invoking them.

In this paper, we look first at related work in the area of automated refactorings and discuss different possibilities for creating a tool that is able to specify and perform refactorings in an automated manner across many programming languages. The findings of this research led to the development and implementation of CHAST, a tool to support the creation of automated refactorings, which will be introduced. Furthermore, we present a case study that shows how CHAST can be used to create automated refactorings and how it compares against similar projects. Lastly, we propose this tool as a flexible, efficient, and secure solution to unify and standardize the creation of automated refactorings.

2 Related Work

Many approaches have been proposed to analyze code and automate refactorings. Baqais et al.[BA20] conducted a systematic literature review of papers on this topic and identified a wide range of approaches, including meta-modeling frameworks and generic refactoring languages. One well-known meta-modeling framework is Famix[BCDGa], which is used in the Moose platform [BCDGb] for software analysis. Famix can bridge the gap between programming languages and analysis tools and has been applied to the task of refactoring [DLT00, TDDN01]. Another approach found is the generic refactoring language *Re*$\mathcal{L}$ [RWZ11], which can be customized for specific programming languages and be used to execute refactorings based on a refactoring description.

A different method for automating refactorings is to convert a project into a database and use queries similar to SQL to analyze and modify the code. Kim et al.[KBDA16] proposed a tool called R3 that uses a database to store the code

and its relations, resulting in faster analysis and refactoring speeds and a smaller memory footprint. GitHub's CodeQL tool[Git] also uses a database approach but is designed for code analysis in the context of security rather than refactoring.

In addition to these general approaches, many tools focus on automating refactorings for specific programming languages. JDeodorant [MTSV16] is a tool for analyzing and refactoring Java code to improve its quality. The tool has been developing for over ten years and has even published a review of the lessons learned from the project [TCC18]. JSparrow [jSp] is another tool for Java that implements a wide range of refactorings and language upgrades. Zhang et al.[ZLS21] proposed ReSwitcher, a tool that automatically updates switch statements to the new switch expressions introduced in Java 12. Comby[vT] is a tool that can be used to search and replace code structures using a syntax that is aware of the underlying programming language, similar to a regular expression.

Despite the wide range of approaches that have been proposed, there are still many challenges and limitations in the field of automated software refactoring. Many existing approaches have limitations in the types of refactorings they can perform or are not applicable to various languages or programming paradigms. These tools can also be complex to use and may require a significant investment of time to understand and implement. Furthermore, many of these tools lack integration with other software development tools, making it challenging to implement them in real-world workflows. Additionally, many existing tools do not address refactorings' potential unintended side effects, which can lead to bugs or even security vulnerabilities. In the current study, we aim to address and eliminate these limitations.

3 Feasibility of Multi-Language Refactoring

Refactoring is a common task in software engineering. It does not matter what programming language someone uses; at some point, code needs to be refactored to either fix bugs, make the code clearer, or utilize new language possibilities. Doing a refactoring by hand once, twice, or even thrice is fine, but as soon as the task gets repetitive, it would be nice to automate it. But how can a refactoring be automated? There are several approaches to this problem, three of them discussed in this section.

3.1 Option 1: Independent Tools

Have multiple independent tools combined in a single "executable".

This is the simplest form of them all in terms of needing to create actual refactoring code. In this option, no actual knowledge of code refactoring is needed; one can simply combine existing tools. However, this is the source of its efficacy. Many tools are available on the internet, floating around and waiting to be found. The main problem is finding them and figuring out how they work. When combining multiple tools, the fact that a single tool only is available for a single language is not a problem anymore. If the language modifier framework `x` is required for the language `A`, we can use it. If for language `B` an entirely different tool is needed, we can use that. Provided that a supportive tool exists for the desired operation in the chosen programming language, it can be utilized.

The problem with this approach is that mostly the same logic needs to be created for every single language. Moreover, it needs to be updated and maintained. If a tool gets outdated and does not work with the newest language version, it gets more useless with every project that uses a more recent version. This, in turn, makes the combined tool less useful until the point where the deprecated tool needs to be dropped and the support for the language with it.

A significant advantage of this approach is that existing refactoring tools can be used to implement the functionality. Those tools often provide some features that make it easier to work with, cover the whole feature set of the language, are most of the time actively maintained, and have an active community that can provide support if needed.

3.2 Option 2: Multiple Modifiers - Single Logic

Have a tool that uses a separate language modifier per language but shares the core logic between them.

Take as a language modifier a simple parser that has the ability to change the syntax tree and converts the changed AST back into code. If such a parser is available for multiple languages, accessible through one language, the core logic can be extracted. Only a mapping between the language-specific and the language-independent parts is needed.

There exist tools that support that kind of functionality. Parser generators - ANTLR [Par] is one example. You specify a grammar (or use one provided by the community) and generate a lexer and parser for it. For ANTLR, the default output is Java, but other languages are also supported. Using ANTLR for the above-described approach works as long as the changes are local to one single file. Once the core logic is defined, only little effort is required to adapt it to support an additional language. More specifically, a new grammar file is needed and, depending on the core logic implementation, a few mappings or selector functions.

This approach's benefit is removing the need to re-implement the same core logic repeatedly. The problem is that it requires quite some initial work to design a core logic that encapsulates most of the refactoring logic and has a minimal footprint in terms of mapping to the target language. In section 4, this approach is covered in more detail.

3.3 Option 3: Domain Specific Language (DSL)

Abstract common language features like variables, methods, classes, etc., into their own domain-specific language that maps to the original code.

Creating an entire domain-specific language (DSL) is a more extreme version of option 2. In this case, a new language or at least an abstraction of a language is created. For every language that gets supported, an adapter must be implemented that maps the individual parts to the DSL. As soon as the whole language adapter is created, refactorings that work on common elements such as classes, methods/-functions, or common constructs like *if statements* can be refactored directly in this DSL.

The benefit of this option arises as soon as there is a language adapter. All new operations are automatically supported by all languages with such an adapter. This makes adding new functionality much more straightforward and faster than all the other approaches. This is especially true if the DSL is explicitly created for refactoring and has some quality-of-life features.

The obvious problem with this approach is finding the common parts of languages and creating a DSL for it. Moreover, after creating the DSL, all its adapters need to be created, tested, and updated. Adding a new language requires again implementing the full adapter. Besides that, such a tool would be very limited in extended functionality that involves some language specifics not present in other languages. Option 2 has the same problem if the goal is to create a single model. However, if there is an adjusted model per refactoring, this is not a problem anymore. The tradeoff is between the amount of work and overall code that is required and the flexibility of the tool.

3.4 Viability of Options

The conclusion of analyzing the viability of those different options is to only stick to options 1 and 2. For both options, a base refactoring framework could be created that provides the basic functionality for all refactorings and collects some common operations that can be reused. However, option 1 requires a tool to wrap the tools in a single, maintainable, and testable interface. This is where CHAST comes in, which will be discussed in section 5. Creating a dedicated DSL has already been tried several times, but no solution has become mainstream yet. Therefore going the route of options 1 and 2 seems to be better at the moment.

4 Parser Generators as Refactoring Meta Language

As discussed in subsection 3.2, it is possible to use parser generators as a refactoring meta language. This section will expand on this idea and show how it could be implemented.

First of all, it is essential to understand what parser generators are. A parser generator is a tool that generates a lexer and a parser from a grammar. This can be a grammar of an existing language or a grammar of a new language. Depending on the parser generator, different grammar formats are supported, and different output formats are generated. For example, ANTLR [Par, Par13] supports a grammar format called ANTLR4 and generates a lexer and parser in Java. However, other output languages are supported as well.

The part of the parser generator that needs to be explained for this section is how the parser is generated: Every parser parses the input code into an abstract syntax tree (AST) structure where every node represents a part of the code. Although the AST differs for every language, the general structure remains the same. Therefore, every AST can be traversed similarly, and with the power of polymorphism, the same functions can be called on every node. When only inspecting the traversing of the AST, this is the same for every language. Only the nodes themselves, i.e., the concrete code written in the respective language, are different. If we map a node's type to a general one, we can do AST manipulations by utilizing the general types instead of the language-specific ones. This allows us to extract the AST manipulation part and use it for any language that has the corresponding mapping of nodes to general types.

In Figure 2, the idea is visualized. The parser generator generates a parser for a language based on a grammar. This parser is then used to parse the input code into an AST. Consequently, the AST is traversed, and the nodes are mapped to general types. For each mapped node, the refactoring logic decides what to do with that node. If new nodes need to be created or existing nodes need to be moved or deleted, the refactoring logic can do that, and the AST is modified. After the AST is modified, the AST gets converted back into code. For this to work, the parser and its AST need to support the conversion from AST to code. In ANTLR, the AST is linked to the token stream. Each node in the AST links to the start and end token in the stream. When moving nodes around or replacing a node with another node, the token stream can be adjusted directly. After the refactoring, the token stream can be converted back into code, and all code that was not changed remains unchanged. This includes newlines, comments, or special formatting. For a better understanding of the whole process, two examples will be discussed.

4.1 Example: Rearrange Class Members

The first example is a refactoring that rearranges the members of a class. Depending on the language, this can include fields, methods, constructors, and so on. It also includes the visibility of such a member, for instance, whether a field is private or public, to name just one. The refactoring should be able to rearrange the members in a way that the user specifies. For example, the user could specify that all public fields should be at the top of the class and all private fields should be at the bottom. Here the languages Java, Kotlin, and C# are supported.

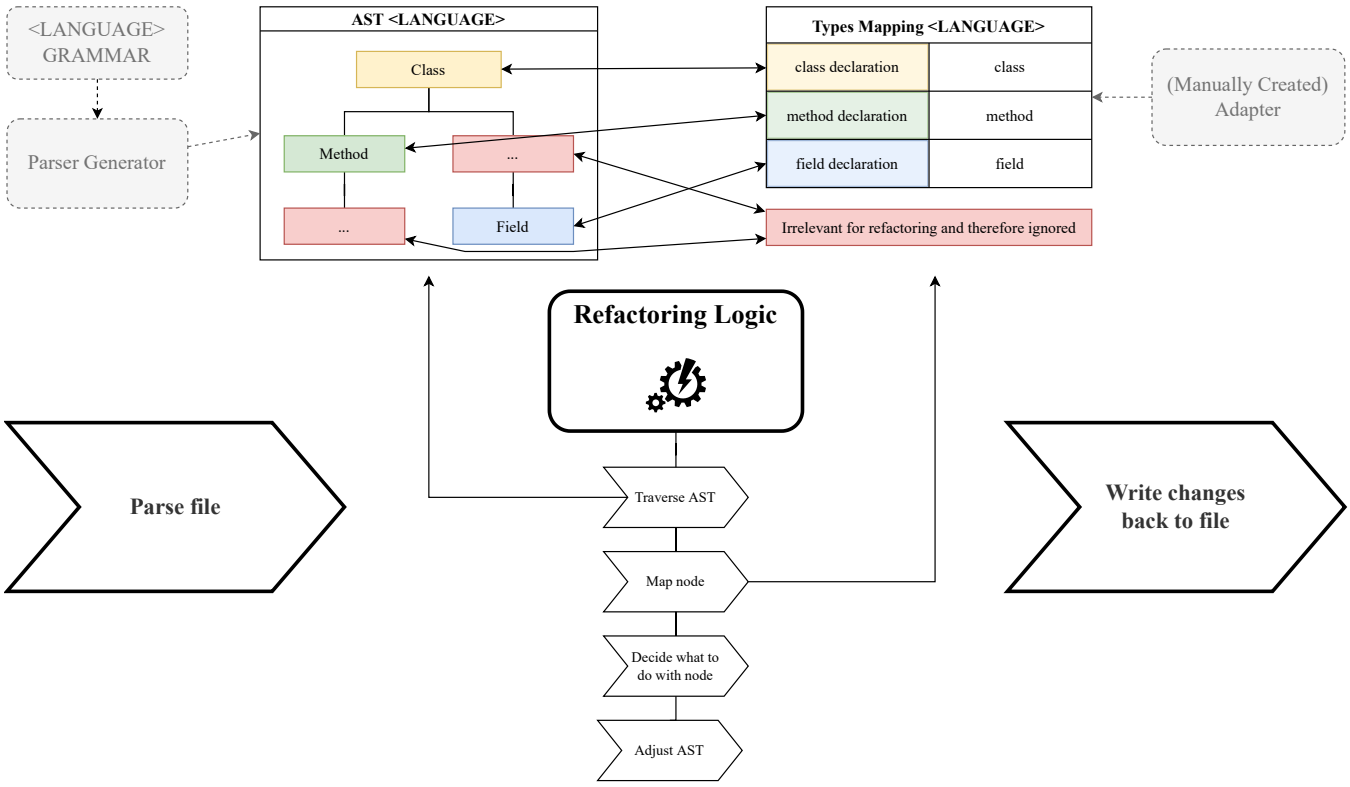

Figure 2. Using Parser Generators and Mappings for Refactorings

To start this process, we need to figure out what information and what AST types are required to rearrange the members, following these steps: First, we need to be able to detect a class. Second, we need to be able to access the members of a class. Third, we need a map from the AST type of a member to a general type and a map from the AST visibility to a general visibility. And last, we need a definition of how to handle hidden tokens like newlines, spaces, or comments.

With this information, we can start implementing the refactoring. Generally, we walk through the AST, and we rearrange the members for every class we find. The rearranging is done by first collecting all members and then sorting them. After sorting, the members are added to the class in the correct order. It is important to note that the AST traversing is not continued further down that branch after a class is found and modified. This prevents nested instances from conflicting with the outer instance, as everything is done on a single token stream. To mitigate this restriction, the reordering process is done multiple times until no more changes are made. After a refactoring, the token stream is converted back into code and then again parsed into a new AST and its corresponding token stream.

Interestingly, a lot of the logic for rearranging and walking through the AST can be reused for other refactorings. There is only a small part that is specific for a particular refactoring. All the code for this example can be found in the GitHub repository accompanying this paper.

4.2 Example: Remove Double Negation

The second example is a refactoring that removes double negations from a boolean expression. For example, the expression `!!(a == b)` can be simplified to `a == b`. This also covers cases where the double negation is separated by parentheses such as `!(!(a == b))`. In this example, the languages Java and Python will be supported. The main difference here is that Java uses a `!` for negation, and Python uses the `not` keyword. Compared to the first example, this refactoring does not move nodes but instead removes nodes and moves nested nodes to the parent node.

To start this process, we need to figure out what information and what AST types are required in order to remove double negations, following these steps: First, we need to be able to detect a boolean expression. Second, we need to be able to detect a negation. Third, we need a condition for when a negated boolean expression contains a boolean expression that is also negated. And last, we need a cancel condition not to remove negations of composite boolean expressions. A composite boolean expression is an expression that contains multiple boolean expressions, such as `a == b && c == d` or `a == b || c == d`. This is needed as removing a negation of a composite boolean expression would change the meaning

of the expression or a possible intended use of double negation. Depending on the grammar of the language, this case is already handled by the parser and the produced AST.

With those selectors and conditions in place, we can start implementing the refactoring. Similar to the first example, we walk through the AST and check whether it contains a double negation for every boolean expression we find. If it does, we replace the first negated expression with the expression of the second negated expression. We continue doing that until no more double negations are found in that AST path. The rest is the same as in the first example.

As it turns out, much of this logic can be reused for other refactorings. In fact, parts of the refactorings from the first example were used in the second example. All the code for this example can be found as well in the GitHub repository accompanying this paper.

5 CHAST: Change Stuff

Imagine the following scenario: A developer wants to perform a number of refactorings: First, they check their IDE whether it provides this functionality. If not, the search for it continues on the internet. There, with some luck, several tools can be found that do have the required functionality. Although the first link shows a project that seems to do what they are looking for, it might not be maintained anymore. While the next link is a maintained project, it is not available for the required language. The link after that leads to a project that is maintained and available for the language but has no documentation. And finally, the last link reveals a project that is maintained, available for the language, and has documentation but has a bunch of dependencies and needs to be built first. Furthermore, the available CLI is not very intuitive, has no log output, and just edits the files in place. The developer is now left with the decision of whether to use the tool or not. However, as it seems to be the best option available, it will be used. As he does not trust the tool, he will spin up a VM that is not connected to the internet and continues the evaluation from there. After a few hours of trying to get it to work, it finally works and does what it should. After some time, the tool needs to be used again, but the user has forgotten how to operate it. Therefore, the whole process starts again. However, the earlier found solution might not work anymore to build a new version. This may result in the user becoming frustrated and deciding to do the refactoring manually instead of using the tool, even for repeated refactorings. However, by doing so, the developer not only wastes time but also loses focus on the actual task at hand. Furthermore, there is a high risk of introducing bugs, as errors coming from copy-paste or search-and-replace actions are very common. As for the tool, losing users will eventually result in it being discontinued and abandoned.

Of course, this is a very simplified example and may also not always be the case, but it illustrates the problem. Besides that, if it happens, it is very annoying and leads to not looking for any tool at all anymore, even if another language would have better tools available. And this happens despite the fact that actually certain commands are present no matter what language is used, taking a linter or a formatter as an example: Every language has one in one way or the other. A single tool should be able to handle those and select the correct underlying tool based on the present project.

In conclusion, there is a need for a single tool that can handle widely used commands as well as select the correct underlying tools based on a given description. This is where CHAST comes in, addressing those mentioned issues and laying the groundwork for further feature additions in the future.

5.1 Concepts

The questions outlined in the following paragraphs have been identified and defined to create a possible solution to the above-mentioned issues. Furthermore, these tasks are associated with existing concepts and tools to discuss the proposed solution, i.e. CHAST, later. In the first version of CHAST, as presented in this paper, not all the mentioned features are implemented yet, but the groundwork is laid for future extended versions. For an up-to-date list of the implemented features, see the repository's README .

How to run an application with all its dependencies without the need to install every single one on your system? The answer here are *Docker* and *Nix* . With the help of *Docker*, tools can be run in containers already containing the required dependencies. *Nix* is a tool with the aim of supporting reproducible builds where you can define a set of dependencies and their versions, while *Nix* handles the resolution, download, and environment for you. To make use of those tools, you need to have one of the two installed. There is also the option to run NixOS in a docker container.

How to access all the tools written for CHAST that are available? Here the system is similar to every other available packaging system. There exists a repository that lets developers upload their tools and also allows users search for tools and see their documentation. This common and well-understood approach is implemented by many tools, such as Docker with DockerHub, Maven with MavenCentral, or npm with npmjs.org. Hosting private repositories for company-specific tools will be made possible as well. It is crucial that the tools need to be available in a central place, are well documented, and are easy to find. This leads us to the next point.

How to improve the documentation and quality aspects? Documentation and tests are first-class citizens of CHAST. Documentation should be provided directly in the

recipe, CHAST's definition configuration of a refactoring, itself and then automatically extracted for displaying it in the repository. Furthermore, tests provide a crucial part of the documentation as well. They clearly state what outputs based on certain inputs are expected. Tests should be treated as examples of how the tool is used. Based on documentation and tests together, the user should have a much easier time finding the relevant information and no need to dig through the source code to figure out how to run it.

How is the security of the user's system ensured? To secure the system from any unwanted side effects, the process needs to be run isolated. As demonstrated with *Docker*, this can be done by utilizing several Linux primitives internally, namely: *(user) namespaces*, *change_root* or *pivot_root*, and *union mounts*. *Namespaces* partition kernel resources such as mounting disks, network, or CPU usage to a process so that this process only sees the defined set of resources. *User namespaces* are namespaces that a non-root user can create. *Change_root* or *pivot_root* both allow us to set a new root directory (*/*) for a process. With this functionality, Docker enables us to run, for example, AlpineOS on an Ubuntu machine. Finally, *union mounts* such as *overlayfs* or *Unionfs*, enable us to create an overlay over one or multiple folders, combine them into a single one, and track the changes in a single folder. Docker uses this functionality to slice their images into pieces that only hold the changed data.

In CHAST, we can use those tools to run a command isolated from the host system and track all changes made in a persistent and non-intrusive way. We are doing this by creating an overlay over the root file system in a *user namespace* and then *change_root* into the newly overlayed root system. The process subsequently runs normally on the system as if it were the host system itself. However, changes made by the script are not directly reflected in the root file system but are written into a separate "changes" folder. The overlay gets removed when the process is done, and we are left with the changes. On these, we can now do security analysis or user confirmation before applying it to the actual file system.

For sandboxing as such, other tools on Linux are *Firejail* [Net] or bubblewrap [Con], which also use Linux primitives, but are more focused on not harming the system than on tracking changes.

How to make it easy to use and extend? CHAST builds on a configuration called recipe in two versions. A *yaml* version that is easy to read and write even with no knowledge of the tool. Even though yaml is a powerful format, it is not very powerful in terms of defining complex data structures and logic. Furthermore, due to its open nature, it is very easy to make errors and no parser can catch them all. Therefore, a second recipe is used, which is a dedicated CHAST language, which also includes some quality-of-life features and some included functionality. The yaml recipe is transpiled into the CHAST language and then executed. This allows us to have a simple and easy-to-read configuration while still having the power of a programming language.

To build extendability, CHAST recipes should be able to call other CHAST recipes inside of them. This allows us to build a library of recipes that can be used by other recipes. Each recipe can therefore be extended or just linked together to build a new recipe. This is similar to how docker images are built. Each image is built on top of another image and can be extended by adding new layers on top of it. The CHAST recipes can also be divided into separate files to make them more readable and maintainable.

With all the security in place, how is the tool kept fast? Speed is a valid concern, as nobody wants to use a slow tool. Fortunately, *overlayfs* and *unionfs* are remarkably performant, and the overhead isn't noticeable for typical tasks. Applying the changes in the end is done by moving and overwriting the files from the change folder to the original location. As moving only changes the file pointer, this operation is very fast and does not result in any heavy disk usage as long as everything takes place on the same disk. Furthermore, some of the cleanup operations are done during the execution and can be parallelized.

Why should a developer bother to add a CHAST package? For tools with an already solid CLI, CHAST is just an extension of reach and a simplification of use for the user, as there is no need to install all the dependencies. However, for developers that have created a tool and do not want to handle all the user interactions of the CLI, the website with documentation and usage instructions, and the publishing, CHAST might be the missing part in the system.

A real benefit of CHAST is that everyone can create a recipe. If there is the need to combine multiple tools into a single one, the user can easily do that and publish its recipe. Maybe someone else needs the same functionality or even extends the functionality further, it can easily be achieved. CHAST follows the from-developers-for-developers idea and depends on an active community. It is comparable to Docker and its wide variety of images provided by numerous developers to solve everyday issues. Without the community creating new images, it has to be understood that Docker would not be what it is today. Likewise, CHAST will be depending on its user community for its further growth and development.

6 Evaluation

Although many tools for task automation exist, no tool incorporates change isolation, which would lead to a much more secure and reliable refactoring process. This section presents the evaluation of the proposed approach divided into two parts. The first part evaluates the proposed approach in terms of the change isolation, and the second part evaluates the

approach in terms of the time and amount of work it takes to create a refactoring.

6.1 Change Isolation

As Docker heavily inspires CHAST's change isolation, this section will use Docker as a baseline for the evaluation. We assume that the refactoring is already packed, once in a Docker container and once in a CHAST recipe. When using Docker to do such a task, the docker container first needs to get all the necessary files from the host system. Exposing the relevant folder to the container ensures that the refactoring script has only access to the relevant files and not to the rest of the system. However, to do the refactoring not on the actual files, the container or the host system must copy the files to a temporary folder. Depending on the project size, creating a copy of the project can take a significant amount of time. Moreover, as soon as the refactoring is done, the differences between the original and the refactored files must be manually built. If parts are not needed, they need to be removed as well. Automating this removal could lead to data loss if the original files were not a copy but the actual files instead. Therefore, this step is also done manually. Lastly, the refactored files must be copied back to the original location when everything is done. However, this includes not only the modified files but the whole project, which again takes a significant amount of time. In terms of security, this variant is undoubtedly secure as the refactoring script only has access to the relevant files. Limiting the container's capabilities is also possible, for example, preventing the container from accessing the internet. However, it is not optimal for the use case of changing something in a project or on your system. This is also not the task Docker was designed for.

CHAST's approach is different in that it does not require any copying of files. Instead, the refactoring script is executed on the actual files. This means that the refactoring script has access to the whole project and host system and can change anything it wants. However, as the host system is overlayed, all changes are isolated to a separate layer. This means that the changes are not visible to the host system, and the original files are not changed as long as the user does not give his confirmation. This means that the user can inspect all changes and decide whether to keep them or not. Filtering out unwanted side effects can here also be done automatically. Filtering in this context means discarding changes that are not relevant to the refactoring.

Other approaches also automate tasks. They often also have a corresponding script to configure the task and incorporate the possibility of sharing it with others. However, such tools do not provide any isolation of the changes and act unrestricted on the host system.

One can argue that Git is also a tool that handles changes. However, Git is not designed to automate tasks. Furthermore, Git only provides a way to handle changes for checked-in files. Files not tracked by Git but changed by a refactoring script will not be detected. This would suffice for only refactorings in a single project. However, if the change affects files outside the project or even the entire system, Git will not detect those changes. This is also the case for other version control systems like SVN or Mercurial.

CHAST provides a unique approach to automating tasks and isolating the resulting changes. It tracks all changes made by a task, regardless of where the files are located on the system. This change isolation feature is similar to sandboxing tools but with the added ability to filter out unwanted changes and retain only the intended modifications. This makes CHAST a secure option for running unfamiliar scripts on your machine without requiring manual review. Overall, CHAST offers improved security and reliability for task automation.

6.2 Creating a refactoring

Creating a refactoring with CHAST involves two steps: writing the refactoring script and creating a CHAST recipe to define the script's execution and testing.

To demonstrate this process, we will use a refactoring script that converts a Java class to a Java record, introduced in Java 14. This refactoring uses the Spoon library, created by Renaud et al.[PMP+15], to analyze and refactor the Java code. We have chosen to use Spoon instead of a parser generator approach because this refactoring is specific to Java and cannot be used with other languages. Additionally, this refactoring is a level 4 refactoring (as described in Figure 1) that requires the resolution of dependencies and the creation of a symbol table. These tasks are challenging to accomplish with a parser generator approach and would require significant additional work. All code for this refactoring can be found in the CHAST repository.

The required refactoring script has two stages. In the first stage, it checks whether the class can be converted to a record by verifying that it is not abstract, has no non-final fields, and has a constructor with all fields as parameters. If the class meets these conditions, the refactoring script proceeds to the second stage. In this stage, it creates a new record with the same name and fields as the class, creates a constructor for the record with the same parameters as the class's constructor, copies all methods from the class to the record, and replaces the class with the record in the code.

The refactoring script is written in Kotlin and must be compiled into a jar file in order to be executed and convert the specified class to a record. The CHAST recipe specifies how this jar file should be executed and what inputs and outputs it needs to have. The recipe also includes a section to define any files or directories that must be included or excluded from the set of changes made by the refactoring.

Listing 1 is an example of how the CHAST recipe for the class-to-record refactoring might look: The `primaryParameter` defines the input file as the only argument to the script. The `run` task specifies that this script only applies to Java

```yaml
version: 1
type: refactoring
name: ClassToRecord
maintainer: Raphael Jenni

primaryParameter:
  id: inputFile
  type: filePath
  description: The file to be refactored.

run:
  - id: rearrange_class_members
    supportedExtensions:
      - java
    script:
      - java -jar ./class_to_record.jar $inputFile
    includeChangeLocations:
      - $inputFile
```

Listing 1. Java Class to Record CHAST Recipe

files. The `script` section defines the script to be executed with the Java binary and the input file as an argument. Lastly, the `includeChangeLocations` section defines that the refactoring script should only include the input file in the changes. This limitation is required because the refactoring creates a `./bin/` folder in the project root during its execution. As this folder is not relevant to the refactoring, it, therefore, should not be included in the changes.

The tests for this refactoring are omitted here for the simplicity of the example. Of course, the refactoring script itself can define tests with jUnit or any other testing framework. But for completeness and as an integration test of the CHAST script, tests should always be defined directly in the CHAST recipe.

Running the refactoring script with the CHAST recipe yields the output shown in Listing 2. The user can then choose to accept or reject the refactoring.

Without using CHAST to create the refactoring, the refactoring script would need to be executed manually, tested manually, and integrated into a CLI manually. This would also require separating the refactored version from the original version, creating a diff, and obtaining user confirmation, which requires a significant amount of additional work, to be repeated for each consecutive refactoring.

Using CHAST streamlines the process of creating refactorings by providing a command-line interface and recipe format for specifying and testing refactorings. This allows users to quickly add a CLI, documentation, and tests to their refactoring, improving the overall quality and reliability of the refactoring. Additionally, the change isolation feature of CHAST helps to catch errors early on and prevents unintended side effects by filtering out irrelevant changes. The initial effort required to create a refactoring remains the same, but the time spent creating the CHAST recipe is significantly reduced. As CHAST continues to grow its base set of refactoring operations, the process of creating refactorings becomes even more efficient and allows for reusing elements from existing recipes.

```
[INFO]: local.(*Runner).Run - Running pipeline PIPELINE-...
[INFO]: local.sequentialRun - Running step class_to_record-...
>>> Setting up folders, Mount UnionFs
[INFO]: namespace.nsExecution - Running in isolated environment
[DEBUG]: namespace.nsRun - Running command in isolated environment
>>> Command output
[DEBUG]: namespace.nsRun - Running command done!
>>> Cleanup, Unmount Folders
[INFO]: local.sequentialRun - Running pipeline post processing
[INFO]: report.PrintFileTree
└── samples
    └── java
        └── records
            └── [~] Person.java

[INFO]: report.PrintChanges - /samples/java/records/Person.java
-class Person {
-    private final String firstName;
-    private final String lastName;
-
-    Person(String firstName, String lastName) {
-        this.firstName = firstName;
-        this.lastName = lastName;
-    }
-
-    public String firstName() {
-        return firstName;
-    }
-
-    public String lastName() {
-        return lastName;
-    }
-
+record Person(String firstName, String lastName) {
+
=    public String name() {
=        return firstName + " " + lastName;
=    }
=
=}

Do you want to apply the refactoring? (y/N)
```

Listing 2. CHAST Java ClassToRecord Conversion Output - Adjusted

6.3 Limitations, intended improvements and further use cases

While it is obvious what can be gained by using CHAST, one limitation of CHAST's current version shows in that a new environment is created for every refactoring, which can result in some binaries not being available on the path and environment variables not being propagated. In future versions of CHAST, this issue can be addressed by supporting Docker and Nix, as well as copying the environment variables from the user session to the isolated environment. This will further improve the usability, reliability, and flexibility of CHAST, allowing it to support a broader range of tools and tasks.

Furthermore, it needs to be noted that CHAST currently supports Linux exclusively due to its reliance on Linux primitives. However, there are ways to use CHAST on other operating systems, by using it in combination with Docker or WSL. In the future, it is intended to expand the compatibility

of CHAST to other platforms, making it more accessible to a broader developer community.

Another limitation of CHAST is that it currently only supports the yaml recipe. As yaml can sometimes be verbose, it is intended to add support for the mentioned CHAST-specific DSL in the future. This addition will allow for a more concise and readable recipe format, which will improve the overall usability of CHAST.

While this paper focuses on the refactoring of source code, CHAST could, in the future, also be used for other tasks, such as automating the installation of software. This can be particularly useful for software that is not available in the operating system's package manager or for simplifying the installation process that may vary across different operating systems and flavors. In addition, CHAST can be used to unify the use of linters and formatters, which are available in most languages. By addressing these limitations and expanding the capabilities of CHAST, it has the potential to become a powerful and versatile tool not only for refactoring, as described in this paper, but for automating tasks across languages and platforms.

7 Conclusion

In conclusion, this paper presents the feasibility of multi-language refactoring and explains how it can be achieved through the use of parser generators as a refactoring meta language. By designing a core refactoring logic and using language-specific mappings, it is possible to apply the same refactoring to multiple programming languages. This approach has the potential to significantly reduce the effort required to implement and maintain refactorings for multiple languages, as well as improve the user experience by providing a unified interface for accessing these refactorings. Thus, it highly contributes to the efficiency and effectiveness of developers in their strive to continuously improving their software.

As part of this research project, the CHAST tool was developed and introduced, which aims to address the challenges of finding and using refactoring tools by providing a unified interface for accessing refactorings and other code-related tools, as well as a packaging system for distributing these tools. CHAST is designed to be language-agnostic and to support a wide range of tools and functionality. While CHAST is still in its early stages of development, it has the potential to significantly improve the accessibility and usability of refactoring tools, as well as to facilitate the development of new refactorings and code-related tools.

In future work, it will be essential to continue the development of CHAST and to evaluate its effectiveness in real-world scenarios. Further research may be necessary to explore and extend the limits and reduce potential trade-offs of using parser generators as a refactoring meta language. Overall, multi-language refactoring and the potential of tools like CHAST to facilitate this process represent significant opportunities for improving the productivity and quality of software development.

References

[BA20] Abdulrahman Ahmed Bobakr Baqais and Mohammad Alshayeb. Automatic software refactoring: a systematic literature review. *Software Quality Journal*, 28(2):459–502, 2020.

[BCDGa] Prof. Alexandre Bergel, Dr. Andrei Chis, Dr. Stéphane Ducasse, and Dr. Tudor Girba. Famix-Moose. https://pavel-krivanek.github.io/famix/.

[BCDGb] Prof. Alexandre Bergel, Dr. Andrei Chis, Dr. Stéphane Ducasse, and Dr. Tudor Girba. Moose. https://moosetechnology.org/.

[Con] Containers. containers/bubblewrap: Unprivileged sandboxing tool. https://github.com/containers/bubblewrap.

[DLT00] Stéphane Ducasse, Michele Lanza, and Sander Tichelaar. MOOSE: An Extensible Language-Independent Environment for Reengineering Object-Oriented Systems. *Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools*, pages 24–30, 2000.

[Git] GitHub. CodeQL. https://codeql.github.com/.

[jSp] jSparrow. jsparrow | an automated java refactoring tool. https://jsparrow.io/.

[KBDA16] Jongwook Kim, Don Batory, Danny Dig, and Maider Azanza. Improving refactoring speed by 10X. *Proceedings - International Conference on Software Engineering*, 14-22-May-:1145–1156, 2016.

[MTSV16] Davood Mazinanian, Nikolaos Tsantalis, Raphael Stein, and Zackary Valenta. Jdeodorant: Clone refactoring. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, page 613–616, New York, NY, USA, 2016. Association for Computing Machinery.

[Net] Netblue30. netblue30/firejail: Linux namespaces and seccomp-bpf sandbox. https://github.com/netblue30/firejail.

[Par] Terence Parr. Antlr. https://www.antlr.org/.

[Par13] Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd edition, 2013.

[PMP+15] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience*, 46:1155–1179, 2015.

[RWZ11] Thomas Ruhroth, Heike Wehrheim, and Steffen Ziegert. ReL: A generic refactoring language for specification and execution. *Proceedings - 37th EUROMICRO Conference on Software Engineering and Advanced Applications, SEAA 2011*, pages 83–90, 2011.

[TCC18] Nikolaos Tsantalis, Theodoros Chaikalis, and Alexander Chatzigeorgiou. Ten years of JDeodorant: Lessons learned from the hunt for smells. *25th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2018 - Proceedings*, 2018-March:4–14, 2018.

[TDDN01] S Tichelaar, S Ducasse, S Demeyer, and O Nierstrasz. A meta-model for language-independent refactoring. pages 154–164, 2001.

[vT] Rijnard van Tonder. Comby - structural code search and replace for every language. https://comby.dev/.

[ZLS21] Yang Zhang, Chaoshuai Li, and Shuai Shao. Reswitcher: Automatically refactoring java programs for switch expression. *Proceedings - 2021 IEEE International Symposium on Software Reliability Engineering Workshops, ISSREW 2021*, pages 399–400, 2021.