Will it Blend? Composable Source Code Analysis

Khashayar Etemadi
KTH Royal Institute of Technology
Stockholm, Sweden
khaes@kth.se

Matthías Páll Gissurarson
Chalmers University of Technology
Gothenburg, Sweden
pallm@chalmers.se

Idriss Riouak
Lund University
Lund, Sweden
idriss.riouak@cs.lth.se

Momina Rizwan
Lund University
Lund, Sweden
momina.rizwan@cs.lth.se

Mohammed Reza Saleh
Umeå University
Umeå, Sweden
msaleh@cs.umu.se

Deepika Tiwari
KTH Royal Institute of Technology
Stockholm, Sweden
deepikat@kth.se

ABSTRACT

In the beginning, there were command line tools. Each was beautifully designed and served a single function, such as `wc` and `cat`. A single editor was born, `ed`, to serve as the standard editor. However, mankind was convinced of its greatness, and wanted more. They started small and wrote `vi`, a visual interface for `ed`. An editor with a simple interface and humble features, which greatly improved productivity. However, in its success, mankind grew vain and wrote ever more complex editors, with grand interfaces and rich graphics, with feature upon feature built into the editor itself. But they could not agree. Some wanted to use a LISP dialect, others shouted for JavaScript. And thus, like the fall of the tower of Babel, the editor wars began, with each editor writing features integrated into its very being, completely assimilated into its very core. Mankind grew to like some of these features, but a problem emerged: how can we share these features between editors so that we may all benefit, and avoid another bloodshed like the editor wars? In this paper, we describe an interface and units for that interface which abstract common GUI editor operations into simple, one-feature-per-unit command line programs, and peace in our time.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; **Documentation**; • **Computing methodologies → Natural language generation**; **Neural networks**.

1 INTRODUCTION

State-of-the-art software technology tools analyse the software and transform it into meaningful output to improve developer productivity. Researchers in academia are putting a lot f effort to develop program analysis tools which analyze the code for finding bugs that fit a bug pattern [8] and tools for program analysis like Soot [20], Opal [9] or IntraJ [14]. But these tools are not being widely used in industries because they are either too complex to use Modern software tools are generally written as monolithic heaps of code to operate as part of a framework within Integrated Development Environments (IDEs) or other specialized tools. Modern IDEs provide you capabilities like code completion, suggested refactoring and automatic deployment systems. Such a complex software tool is difficult to use and integrate within an automated industrial process where the software needs to be distributed organisation-wide, and where developers are continuously testing and prototyping new features because testing might need adding and removing new tools and the modern tools are not very modular. Having to deal with complex software tools places a strain on developer's productivity because it requires a lot of time and engineering efforts to set them up in the CI/CD integration cycle. One way to solve this problem is to increase software flexibility and re-usability by introducing a framework that allows software components to be plugged in and out, easily exchanged and adapted to other purposes.

One of the best feature of a standard UNIX shell is the ability to use pipes to compose modular software programs. The main principle of UNIX is to compose minimal software modules/units should do one task and do it well [11]. To the best of our knowledge, this principle has never been applied to software tools in a systematic and meaningful way[2]. Although, piping mechanism allows us to compose software tools by feeding the output of one tool to the input of the other tool, the design of the interface between different composable units is crucial. To be able to mix and match different components into a full-fledged software, the components are required to fit perfectly in the chain. This is only possible if we choose such an interface between different components which is generalisable (means being used by many shell tools already), and easily extendable. To allow re-usability of components, each component should perform one single task which cannot be decomposed into smaller units.

Contribution:

- Applying UNIX philosophy to software tools and verifying its practical significance by implementing real-life use cases using this architecture.
- Design decision: Choosing an interface which is widely applicable and language implementation independent.
- Experimental evaluation of our prototype to examine and validate some research questions about
  - modules being interchangeable,
  - the scalability of our approach

2 ARCHITECTURE OVERVIEW

This section will describe our approach to decomposing software components into the units implemented in the pipeline. We describe the communication protocols (i.e., interactions) and the interface between the different units. Moreover, we will discuss the limitations and advantages of all our design decisions.

2.1 Unit isolation

As mentioned in section 1, developers rely heavily on IDEs to perform many tasks that are actually composites of multiple atomic tasks. We use this fact as motivation to decompose these tasks into smaller, isolated units that "do one thing, and do it well" [11]. For example, with a few clicks, a developer may be able to replace all occurrences of a literal within a project with another, all within her IDE. The same outcome may be achieved by chaining together independent units that are invoked on the developer's shell, outside of the IDE. As such, a single unit may be responsible for parsing the source files in the project, and building an Abstract Syntax Tree (AST) representation for it. This output may then be piped into another unit that filters a subset of nodes in this AST corresponding to all literals. Finally, another unit performs the actual replacement of the input literal with another. Moreover, the scalability of this design also allows standard UNIX programs, such as `grep`, `cat`, or `wc`, to be chained together with these implemented units for finer-grained control of the output and to achieve more potential use cases.

We implement units that perform *parsing* of source files into ASTs, as well as *filtering* and *replacement* of a set of nodes within the AST. We design them so that they can interface with each other on the shell, and can be chained together to achieve several use cases relevant to developers. We illustrate this by implementing a unit that interacts with other shell utilities, such as `git`, in order to generate documentation for methods defined within source files. The following subsection presents the details of the implementations of these units.

2.2 Unit implementation

To demonstrate that we can easily plug in or swap different modules in the pipeline for different purposes, we implemented different instances of the same functionalities (each with their own strengths and limitations). For examples, we might need a faster parser (e.g., ExtendJ) that can parallelize the AST evaluation for the use cases where the performance is important or we might need a parser (e.g., Soot) that can keep track in the AST of the comments in the source file. The example shows that there is a need to have different tools that perform the same functionality but have their own strengths and weaknesses for different purposes/use cases. So, to improve a developer's productivity, there should be a simple way to just use the most suitable parser for the every job. This convenience can only be achieved if the units are modular and simple rather than cumbersome and complicated.

The units implemented in the pipeline are summarized as follows.

mp-parser: This unit transforms an input source file into an AST, using `Json4Spoon`[1]. It primarily outputs the AST as nested JavaScript Object Notation (JSON), with information on the children of each node. It also includes a built-in converter to transform this output into a Comma-Separated Values (CSV) format. It is available on GitHub at https://github.com/Tritlo/JavaToJSON.

k-parser: This unit parses a source file into an AST, using the `Spoon` library [12]. It produces a CSV file with information on all the nodes of the AST. This parser is available on Github at https://github.com/khaes-kth/Simple-Parser.

[1]https://github.com/SpoonLabs/gumtree-spoon-ast-diff

NAME	...	VISIBILITY
`CtClassImpl`	...	`public`
`CtMethodImpl`	...	`private`
`CtInvocationImpl`	...	`null`
...		
`CtMethodImpl`	...	`public`
...		

Table 1: CSV with parsed AST nodes used as input to `d-filter`

NAME	...	VISIBILITY
`CtMethodImpl`	...	`private`

Table 2: Filtered CSV output from `d-filter`

im-parser: This unit parses a source file and construct the respective AST using the Java Extensible compiler `ExtendJ`. The unit produces a CSV file containing the source location (i.e., `line-start`, `line-end`, `column-start`, `column-end`,`relative and absolute path`) and a set of composable properties. These properties are composed assembling different `ExtendJ` APIs. This approach allows the unit's user to generate ad-hoc information, e.g., a variable's unique identifier and enables advanced analyses. The limitation of the *im-parser* are the following:

- *ExtendJ* supports all java versions up to Java 8,
- It generates only CSV, therefore, the output is limited by the CSV representation,
- The comments are discarded by *ExtendJ*, therefore, all the comments information are lost.

The strength of the *im-parser* are as follow:

- Reference Attributed Grammars (RAGs) are used to compute the properties. RAGs enable on-demand evaluation with the possibility to memoize already computed properties, allowing the parallel computation of attributes.
- It is easy to fetch information from a parent node in the AST thanks to the expressivity of RAGs (i.e., inherited attributes and synthesized attributes).

The *im-parser* is available on GitHub at https://github.com/IdrissRio/JAVA2AST.

d-filter: This unit accepts a list of nodes in an AST, in CSV format, and extracts a subset of the nodes that correspond to methods or classes. The filtering is done based on the input criterion for visibility, i.e., public or private. For example, it can produce the CSV presented in Table 2, which is a subset of the CSV in Table 1, corresponding to private methods. `d-filter` is available on GitHub at https://github.com/Deee92/ast-filter.

m-filter: This unit accepts a list of nodes in an AST, in CSV format, and extracts a subset of the nodes that correspond to syntactic constructs. This unit filters the input list based on other input criterion for syntactic constructs such as `loops`, `if`, `assertions`,

ASTNode	...	Type	Value
VariableAccess	...	int	BAR
IntegerLiteral	...	int	2
VariableDeclarator	...	int	BAR

Table 3: CSV input to `im-filter`

ASTNode	...	Type	Value
VariableAccess	...	int	BAR
VariableDeclarator	...	int	BAR

Table 4: CSV generated by `im-filter`

`flow breaks`, `switch`, `synchronized` and `try` constructs. For instance, it can produce the CSV or tabular format presented listing of syntactic constructs a. `m-filter` is available on Github at https://github.com/salehsedghpour/SyntacticConstructLocator.

im-filter: This filter reads from standard input a CSV generate by a previous unit. It filters the rows according to a specific criteria specified by the user. For example:

By running the *im-filter* on the Table 3 with the criteria specified as the following command line flag `-filterby=type{int},value{bar}` gives an output as in Table 4.

k-replacer: This unit accepts a literal node in an AST, in CSV format, and replaces them with a new literal with a given value. This replacement is done by manipulating the source code at the location specified by the given node. For example, it can replace a string like “old-str” with a new string like “new-str”. `k-replacer` is available on Github at https://github.com/khaes-kth/kh-replacer.

doctor: Short for *DOCumentation generaTOR*, this unit accepts a list of public methods, and produces Javadoc-like documentation for them. More specifically, it populates a template with the following information:

- method description, which in the current preliminary implementation, is extracted from the method name. In future, this may be replaced with more sophisticated analyses or even with natural language processing techniques.
- author information, which is obtained using `git blame` on the lines in the source file that correspond to the method.
- parameters,
- return type, and
- thrown exceptions, which are obtained with through static analysis of the method with `Spoon` [12].

Figure 3 is an example of the documentation generated by `doctor` for the method `isPrime(int)` in the `commons-math` project. The output is redirected to the shell by default. Alternatively, modified versions of the source files with the added documentation can also be produced. `doctor` is available on GitHub at https://github.com/Deee92/doctor.

2.3 Interface design

Whereas some units introduced in subsection 2.1 produce outputs in the JSON format, such as `mp-parser`, others produce CSV outputs, as in the case of `k-parser`, or `d-filter`. Both the JSON and CSV interfaces have their merits and drawbacks. For example, JSON outputs can be nested and describe the structure of the AST in more detail, using properties and child nodes. However, this also means that these outputs may not be very command-line friendly. On the other hand, CSV formats may not be as nested or descriptive, but are typically very easy to work with on the command line. As illustrated in Figure Figure 1, we have tried to utilize the advantages of each format and mitigate these differences by allowing conversion from the JSON to CSV output for readability. Note that conversion in the opposite direction is currently not supported, but is proposed as an extension.

2.4 Supported use cases

We now present a list of the use cases achievable with the current implementation of our pipeline.

- Semantic replacing or renaming of any name (e.g., method or variable) across multiple compilation units.
- Adding documentation to method declarations, which can be achieved by chaining together the parser, filter, and the documentation generator (`doctor`). Therefore, this invocation can be illustrated as
  `parser | d-filter | doctor`
  where `parser` may be substituted with any parser that produces an appropriate output that is compatible with `d-filter`. In subsection 3.2, we also illustrate this use case as part of our evaluation.
- Finding syntactic constructs in the source based on input criteria (e.g., `for-loops` longer than *n* lines).
- Constant propagation and constant folding, exploiting compiler’s analyses.
- Method extraction.
- Quick Fixes (e.g., replacing reference with structural equality).

3 METHODOLOGY

3.1 Research Questions

We now present the research questions we aim to answer with our evaluation.

- RQ1: *Can the individual units be replaced in the pipeline with their alternative implementations?*
  The same goal in programming may be achieved through diverse implementations [6]. For example, corresponding to an input, a unit may achieve two equivalent outputs following two different algorithms or through the use of two distinct libraries. This implies that alternative implementations of a unit may be used interchangeably by developers to produce the same result. We elaborate on the protocol used to evaluate this property in our pipeline, in subsection 3.2.
- RQ2: *Can decomposing large programs help increase their performance?*
  Decomposing large programs into small units enables us to fine-tune the interactions between these units and create new efficient compositions of them. In this research question, we evaluate whether this gives us an opportunity to increase the performance of our programs.

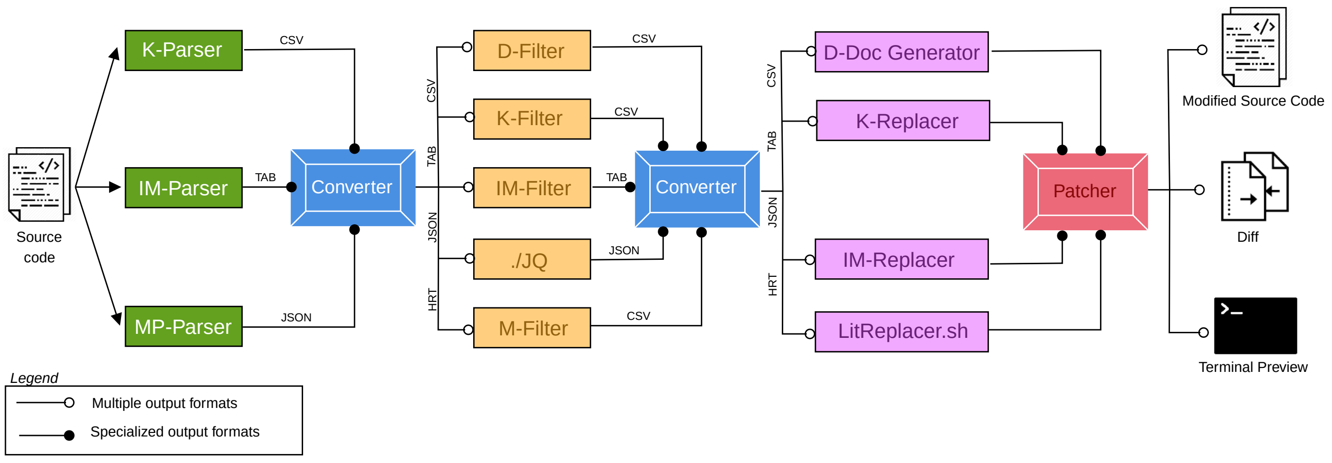

Figure 1: The units and their relations

- RQ3: *What is the appropriate interface in terms of ease-of-use for developers and user readability?*
  Choosing the appropriate interface is crucial for achieving unit developer ease-of-use and user readability, when many different parties are involved in writing units and reading the output. We illustrate the merits and drawbacks of different interfaces by answering this question by answering the following questions:
  - RQ3.1 *What interface designs simplify integration with other established tools outside of our own framework?*
  - RQ3.2 *What interface designs do end-users find easy to use?*

3.2 Protocol for RQ1

In order to demonstrate the interchangeability of alternative implementations of a unit, we replace one implementation with another and compare the outputs obtained for a specific use case. Moreover, to ensure that this evaluation is performed on a real-world source file, we use one source file each from two widely used, open-source projects of the Apache Software Foundation, called `commons-math`[2] and `commons-lang`[3]. Table 5 presents some details about these projects, such as the number of stars on GitHub (#STARS), the VERSION used for the evaluation, the fully qualified name of the class source file (SOURCE_FQN), as well as the number of lines of code (LOC) in the file.

Figure 2 illustrates the protocol for this evaluation. For each source file, we use `mp-parser` and `k-parser` to parse it into an AST. Next, for each output AST, we obtain a set of public methods with the `d-filter`. We then use the documentation generation unit, `doctor`, to generate documentation for both sets of outputs. Finally, we verify the equivalence of the two outputs of `doctor`.

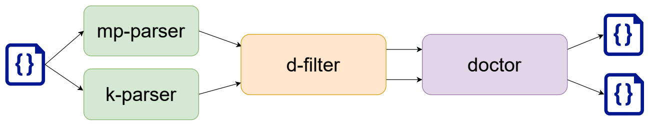

Figure 2: Evaluation protocol for RQ1

This allows us to draw conclusions about the equivalence of alternative implementations of the parsers. We present the results of this evaluation in subsection 4.1.

Note that, for the current pipeline, this evaluation can only be performed for alternative implementations of the parsers. Recall from subsection 2.1 that the two parsers have diverse implementations and equivalent outputs. However, the same is not true for the 5 filters in the pipeline. As highlighted in subsection 2.2, the filters produce a disjoint set of outputs. For example `m-filter` can produce a set of syntactic constructs such as loops, while `d-filter` outputs a set of methods or classes of the specified visibility. Therefore, for the evaluation of RQ1, we analyze the interchangeability of the parsers only. However, we propose that an extension to the pipeline with the addition of more equivalent filters will facilitate the demonstration of their interchangeability as well.

3.3 Protocol for RQ2

To assess the performance improvement that our proposed method provides, we conduct the following experiment. First, we develop a simple Java program that creates a hard-coded array of 100 strings from "1" to "100" and then prints elements of this array. Next, we compare the time required for replacing these literals of this program using two different versions of a "String Modifier" program.

The first version of the string modifier is a composition of three basic units: a parser (here k-parser), a filter (an ast-filter, namely k-filter), and a literal replacer (here k-replacer). The parser gets a Java program and outputs its AST nodes in a CSV file. The filter gets the CSV output of a parser and outputs a CSV that only contains

[2]https://github.com/apache/commons-math/tree/MATH_3_6_1
[3]https://github.com/apache/commons-lang/tree/rel/commons-lang-3.12.0

PROJECT	#STARS	VERSION	SOURCE_FQN	LOC
`commons-math`	425	3.6.1	`org.apache.commons.math3.primes.Primes`	56
`commons-lang`	2.2k	3.12.0	`org.apache.commons.lang3.ArraySorter`	41

Table 5: Source files used for the evaluation of RQ1

the literal nodes with a given value. Finally, the literal replacer gets a filtered AST node and replaces it in the source code with a given new literal. These three units work in isolation and can be executed in separated processes. The output of each unit can be used by other units or even other programs, since they can be executed separately. We call this version the I-StringModifier.

The second version of the string modifier contains the exact same units as the first one, while its units are not isolated. This means to run the string modifier we have to execute all three units in a single process. Consequently, the interactions between units only happen inside the program and they cannot be reused by other units or programs. We call this version the W-StringModifier.

As mentioned above, the goal of our experiment is to replace all 100 strings with new ones. For this purpose, we target replacing every string of a number n with a string of number n+100. For example, "1" becomes "101". Doing this with I-StringModifier requires running the parser once to get the AST and running the filter and literal replacer units one hundred times to find and replace each string. On the other hand, when we use W-StringModifier, we have to run the whole pipeline (i.e. parser, filter, and literal replacer) one hundred times. Note that in this experiment I-StringModifier represents a software project that follows our proposed approach, as it decomposes a large program into small units and allows us to reuse the output of small units. On the other hand, W-StringModifier represents a software project that does not take advantage of decomposition of large units.

We conduct the experiment using each version of the string modifier with two different configurations: with one process dedicated to running the experiment, and with four processes dedicated to the task. Comparing results for these two different configurations can help us better understand the impact of parallelization of the task using each version of the string modifier.

After running the experiments on the same machine, we compare the time needed to finish the replacing. A faster finish shows a more scalable method. We run the experiment on a machine with an Intel Core i5-6260U CPU running at 1.80GHz and 16GB of DDR4 2133MHz RAM.

3.4 Protocol for RQ3

What is the appropriate interface in terms of ease-of-use for developers and user readability?

As part of our work, we implemented interfaces following two different approaches: one based on a tabular format (encoded as CSV) and another based on a nested format (encoded as JSON).

To compare the two, we look at two measures:

- RQ3.1 *What interface designs simplify integration with other established tools outside of our own framework?*

  An important metric for the interface is the ease-of-use for unit developers, as approximated by implementation complexity. High implementation complexity would make the tool hard to work with for unit developers, which are an important part of the tool ecosystem. To assess the implementation complexity, we compare the lines-of-code-as-implemented for the two comparable units, namely the Parser, the Literal Replacement, and the visibility filter.

- RQ3.2 *What interface designs do end-users find easy to use?*

  An important part of designing the interface is the "readability" of the output, i.e. how easy it is for users to parse without machine assistance. As "readability" is not very well defined, we use the amount of data returned after running the parsers on different programs as a proxy metric for readability. Too much data on the command line can overwhelm even the most dedicated users, and thus compromise readability.

4 RESULTS

This section presents the results from our evaluation of the three research questions.

4.1 Results for RQ1

Per the protocol presented in subsection 3.2, we verify that the output of the documentation generation unit, `doctor`, is equivalent for a source file parsed with either `mp-parser` or `k-parser`.

Figure 3 presents the output of `doctor` for a method in `commons-math` called `isPrime`, which accepts an integer parameter and returns true if it is prime. The Javadoc-like documentation generated by `doctor` includes information on the author, the parameter, and the return type for the method. This output is the same, regardless of the parser used to parse the source class, `Primes`. Similarly, Figure 4 presents the output for the method `sort` in the `commons-lang` project, which returns a sorted version of an input array of float values. As in the previous case, this documentation is also identical for both parsers used on the source class, `ArraySort`.

This evaluation demonstrates that both the `d-filter` and the `doctor` are independent of the implementation of the parser, and that their output is solely dependent on the input they receive from the preceding unit in the pipeline. Individual units with diverse implementations in the pipeline, such as the parser, may therefore be used interchangeably, provided their outputs are equivalent. This is because the unit that consumes the output of such units is

```
/**
 *
 * is prime
 * @author  Luc Maisonobe 2013-03-10 21:05:20
 * @param  n
 * @return boolean
 */
public boolean isPrime(int n) {
  ...
}
```

Figure 3: Output of doctor for `isPrime(int)` in `commons-math`

```
/**
 *
 * sort
 * @author  Gary Gregory 2020-12-21 16:43:30
 * @param  array
 * @return float[]
 */
public float[] sort(final float[] array) {
   ...
}
```

Figure 4: Output of doctor for `sort(float[])` in `commons-lang`

	One Process	Four Processes
I-StringModifier	172 (sec)	136 (sec)
W-StringModifier	399 (sec)	302 (sec)

Table 6: Time spent on each execution of the RQ2 experiment.

oblivious to the exact protocol, algorithm, or library used in their implementation.

Answer to RQ1: Can the individual units be replaced in the pipeline with their alternative implementations?
Provided that a unit has diverse implementations but equivalent outputs, either implementation may be used to achieve the same output for a use case.

4.2 Results for RQ2

Table 6 and fig. 5 present the results of our experiment for answering RQ2. As it is shown, I-StringModifier is significantly faster than W-StringModifier with both configurations (one process or four processes). More specifically, when one process is used, I-StringModifier is 57%$((399 - 172)/399)$ faster than W-StringModifier and when four processes are used, it is 55%$((302 - 136)/302)$ faster.

Note that I-StringModifier runs the parser only once and the filter and literal replacer one hundred times. On the other hand, W-StringModifier runs all three units one hundred times, as it is not decomposed into small units like I-StringModifier and its whole pipeline should be run to get the results. Therefore, we can conclude that almost the whole 136 seconds is spent on running the filter

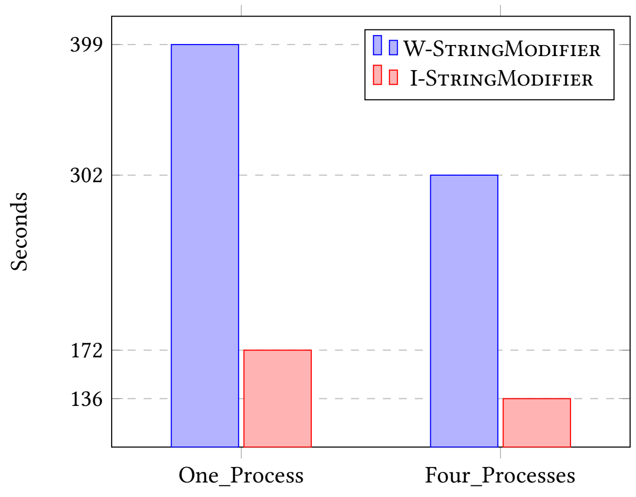

Figure 5: Time spent using a replacer with each configuration.

and literal replacer a hundred times, and 263 seconds (399-136) is spent on running the parser. This means I-StringModifier saves a significant time by skipping the parsing step in 99 executions. Also, it is worth mentioning that the filter is a script that just selects the lines of the parser's output based on a given condition. Therefore, we know the time it takes to complete the task is not significant compared to the parser and literal replacer.

The main reason that I-StringModifier is faster than W-StringModifier is that when the parser produces the AST, I-StringModifier uses it for all replacement, while W-StringModifier has to regenerate the AST every single time. This happens because decomposing the replacer program in I-StringModifier enables us to reuse the result of a single unit in many executions, while this option is not available in W-StringModifier. Also, with a closer look we realize the reduction in the time is almost the same as when one process is used and when four processes are used (57% and 55%). The reason behind this narrow gap is that since I-StringModifier generates the AST once, that task is not performed multiple times and parallelization does not reduce the time spent for it. On the other hand, W-StringModifier runs both parsing and literal replacing tasks multiple times which gives it a chance to reduce the time spent for both parsing and literal replacing with parallelization.

Answer to RQ2: Can decomposing large programs help increase their performance?
According to our evaluation, our proposed method for decomposing large programs enables developers to take advantage of the results of a small part of the program multiple times with running it only once. This can cut the resources required for a task as much as 57%.

4.3 Results for RQ3

To evaluate the two interfaces, we looked at the lines of code for each unit, and ran the parsers on their own source as well as a small `Hello.java` file, as seen in fig. 6.

Units	JSON	CSV
Parser	130	744
Literal Replacer	24	93
Visibility Filter	16	256

Table 7: Lines of Code Comparison between units

```
public class HelloWorld {
  public static void main(String [] args){
    System.out.println("hello, world");
    System.out.println("ok, world");
  }
}
```

Figure 6: Hello.java used for evaluation

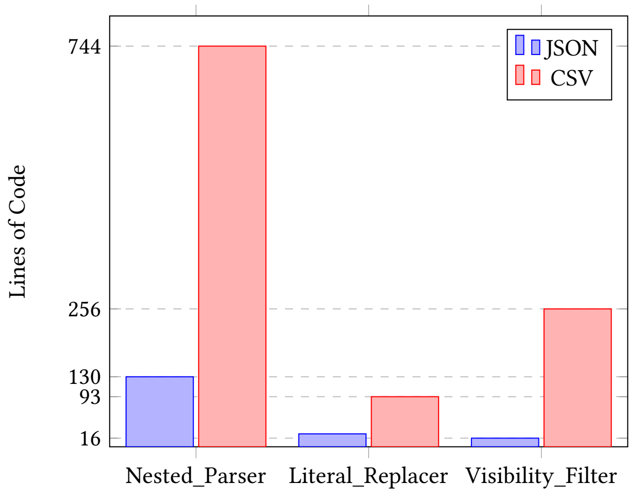

Figure 7: Lines of Code for each implementation

4.3.1 RQ3.1 Outside tool integration.

What interface designs simplify integration with other established tools outside of our own framework?

The results of our evaluation can be seen in table 7 and fig. 7. As the data suggests, unit implementation is simpler when using the JSON interface. We believe this is mainly due to the nested structure of JSON being closer to the nested structure of the packages and source code ASTs, as well as the availability of jq and its query language making querying the JSON structure easier than for the CSV case [7].

Answer to RQ3.1: What interface designs simplify integration with other established tools outside of our own framework?

Using unit implementation complexity as a proxy measure for simplicity of integration with other established tools, the nested JSON-like format does better than the CSV-like tabular format.

	LOC	JSON (Lines)	CSV (Lines)
`Hello.java`	6	243	57
`NestedParser`	130	4080	919
`TabularParser`	744	26536	6581

Table 8: Relevant output comparison.

4.3.2 RQ3.2 Ease-of-use.

RQ3.2 *What interface designs do end-users find easy to use?*

The lines of output per size of code base being evaluated (with JSON processed by jq) is shown in table 8 and fig. 8. As seen from the data, the JSON output is a sizeable 243 lines of output for only 6 lines of code, and growing to a massive 26536 lines for a only 744 lines of code. It is evident that this is quite a lot of output for a human to parse without machine assistance.

Answer to RQ3.2: What interface designs do end-users find easy to use?
A good answer to this question could have been obtained by running a user study, which we unfortunately did not have the resources for. However, using the amount of lines in the output as a proxy for readability, and assuming that readability is an important consideration for interface designs in terms of end-users ease-of-use, using a CSV-like tabular format is better than a nested JSON-like format.

Answer to RQ3: What is the appropriate interface in terms of ease-of- use and readability?
Both approaches have their merits: JSON more closely represents the nested structure of the source files, and is thus easier to query and manipulate in a programmatic way, resulting in a low implementation complexity and ease-of-use for unit developers. The CSV format is however much more succinct, and has less extraneous data and is more easily readable for a human without specialized tools.

5 RELATED WORK

Modern software technology gives us tools that can systematically analyse and transform software and thereby dramatically improve developer productivity. We divide these earlier studies into works concerning finding bugs or errors, refactoring, integration to IDEs, program analysis, extended tools aimed at end-users and large-scale program analysis.

5.1 Finding bugs or errors

One of the key outcome of static analysis could be finding bugs or errors. Knizhnik and Artho [10], proposed Jlint, which checks Java source code and find bugs, inconsistencies and synchronization problems by analyzing data flow and building the lock graph. Ayewah et al. [4] analyzed the use of FindBugs in production; Findbugs is an open source static-analysis tool for Java. In another study Ayewah et al. [3], discussed the lesson learned from fixing the findbugs warnings. Aftandilian et al. [1] report how they used three different tools on Google Java code base, from which, one of them analyzed the source code to check for errors at compilation process and automates repair of those errors. To summarize, most of efforts

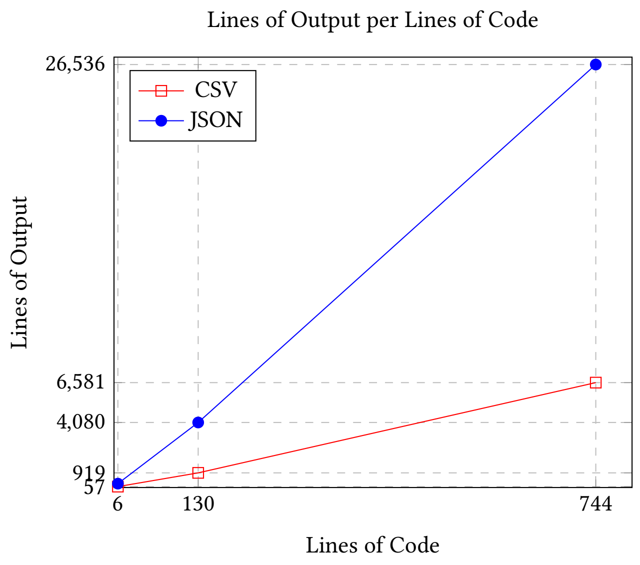

Figure 8: Lines of parser output per number of lines in the input between the two approaches.

to find bugs or errors in programs, analyze commonly found bugs in the source code and create a repository of rules to capture various kinds of bugs.

5.2 Refactoring

Refactorings are behaviour-preserving program transformations, typically for improving the structure of existing code. Mathieu et al. [21] present JunGL; a domain-specific language for refactoring, which manipulates a graph representation of the program and let the developers script their desired refactoring. The authors in [13], break refactorings into smaller steps that need not preserve behaviour individually. Schäfer et al. [16] present the idea to implement the refactoring by embedding the source program into an extended language on which the refactoring operations are easier to perform, and then translating the refactored program back into the original language.

5.3 Integration to IDEs

Most of software developers are using Integrated Development Environments (IDE) for their daily tasks. Authors in [19] presented Monto architecture which deals with integrating new functionality into these environments. Siemund and Tovesson [18] also used the Language Server Protocol to make java developers able to have high level support such as code completion, hover tool tips, jump-to-definition and find references as an extension of ExtendJ.

5.4 Program analysis

Program analysis play an important role, as they can help software engineers spot potential run-time errors (e.g. null pointer de-reference, array out of bounds indexing) already at compile time. Vallée-Rai et al. [20] presented Soot, a framework for optimizing Java bytecode, which lead to achieve higher performance. Schubert et al. [17] also presented PhASAR, which is a LLVM-based static analysis framework for C/C++ codes. Bravenboer and Smaragdakis [5] also presented DOOP framework to specify pointer analysis algorithms declaratively, using Datalog.

5.5 Extended tools

Various libraries has been implemented for the sake of program analysis. For instance, Pawlak et al. [12] presented Spoon which is a library for the analysis and transformation of Java source code.

5.6 Large scale program Analysis

Many tools have been proposed to analyze the programs, but when it comes to large scale programs, they couldn't be a solution at all. For instance the authors in [3, 4], discussed the usage of FindBugs in the Google codebase scale. According to [15], most of engineers at Google work in an extremely large codebase, where they perform more than 800k builds, run 100M test cases, produce 2PB of build outputs, and send 30k changelist snapshots (patch diffs) for review. Sadowsky et al. [15] also presented Tricorder, a program analysis platform aimed at building a data-driven ecosystem around program analysis in large-scale codebase.

6 CONCLUSION AND FUTURE WORK

Conclusion. In this project, we applied the UNIX philosophy to implement different tools for composable source code analysis. We achieve this through the implementation of several units that can be invoked independently. The outputs of these units can be chained together in a pipeline in order to accomplish typical use cases that may be relevant for software developers on the job. Some use cases we evaluated include AST node filtering, document generation, and literal replacement. We compared two interfaces with respect to amount of output and implementation complexity.

Future Work. However, with the lessons learned during this project, we find that it may be beneficial to scale up the pipeline, and analyze the impact of new units in real-world scenarios. We therefore propose to build on the results presented herein by implementing other units. For example, one extension could be to add a unit that integrates with CI/CD pipelines to perform some pre-deployment actions. Yet another one could delete a specified set of elements from the AST. There can also be units that analyze code complexity, suggest variable or method names, or sort methods based on input criteria to enable the end-users to improve the quality of their codes. Another proposed extension is a unit that instruments source files for dynamic analysis with different configurable goals. The possibilities are endless! In terms of the evaluation, an important direction is to compare the outputs in terms of bytes produced rather than lines, and use a metric to better compare "command line" friendliness, especially in terms of tools pre-installed on systems. There is also a need for further evaluation on the ease-of-development of units with respect to libraries and pre-existing tools beyond what is covered in this report, and add more exotic interfaces than CSV and JSON into the mix.

REFERENCES

[1] Edward Aftandilian, Raluca Sauciuc, Siddharth Priya, and Sundaresan Krishnan. 2012. Building Useful Program Analysis Tools Using an Extensible Java Compiler.

In *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*. 14–23. https://doi.org/10.1109/SCAM.2012.28

[2] Dimitar Asenov, Peter Müller, and Lukas Vogel. 2016. The IDE as a scriptable information system. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 444–449.

[3] Nathaniel Ayewah and William Pugh. 2010. The Google FindBugs Fixit *(ISSTA '10)*. Association for Computing Machinery, New York, NY, USA, 241–252. https://doi.org/10.1145/1831708.1831738

[4] Nathaniel Ayewah, William Pugh, David Hovemeyer, J. David Morgenthaler, and John Penix. 2008. Using Static Analysis to Find Bugs. *IEEE Software* 25, 5 (2008), 22–29. https://doi.org/10.1109/MS.2008.130

[5] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-to Analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications* (Orlando, Florida, USA) *(OOPSLA '09)*. Association for Computing Machinery, New York, NY, USA, 243–262. https://doi.org/10.1145/1640089.1640108

[6] Liming Chen and Algirdas Avizienis. 1978. N-version programming: A fault-tolerance approach to reliability of software operation. In *Proc. 8th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-8)*, Vol. 1. 3–9.

[7] Stephen Dolan. 2021. jq 1.6 Manual.

[8] Alexandru Dura, Christoph Reichenbach, and Emma Söderberg. 2021. JavaDL: automatically incrementalizing Java bug pattern detection. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–31.

[9] Dominik Helm, Florian Kübler, Michael Reif, Michael Eichberg, and Mira Mezini. 2020. Modular collaborative program analysis in OPAL. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 184–196.

[10] Konstantin Knizhnik and Cyrille Artho. [n.d.]. Jlint - Find bugs in java programs. http://jlint.sourceforge.net/

[11] M. D. McIlroy, E. N. Pinson, and B. A. Tague. 1978. UNIX time-sharing system: Foreword. *The Bell System Technical Journal* 57, 6 (1978), 1899–1904. https://doi.org/10.1002/j.1538-7305.1978.tb02135.x

[12] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. 2015. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience* 46 (2015), 1155–1179. https://doi.org/10.1002/spe.2346

[13] Christoph Reichenbach, Devin Coughlin, and Amer Diwan. 2009. Program Metamorphosis. In *ECOOP 2009 – Object-Oriented Programming*, Sophia Drossopoulou (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 394–418.

[14] Idriss Riouak, Christoph Reichenbach, Görel Hedin, and Niklas Fors. 2021. A Precise Framework for Source-Level Control-Flow Analysis. In *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 1–11. https://doi.org/10.1109/SCAM52516.2021.00009

[15] Caitlin Sadowski, Jeffrey Van Gogh, Ciera Jaspan, Emma Soderberg, and Collin Winter. 2015. Tricorder: Building a Program Analysis Ecosystem. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 598–608. https://doi.org/10.1109/ICSE.2015.76

[16] Max Schäfer, Mathieu Verbaere, Torbjörn Ekman, and Oege Moor. 2009. Stepping Stones over the Refactoring Rubicon. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming* (Italy) *(Genoa)*. Springer-Verlag, Berlin, Heidelberg, 369–393. https://doi.org/10.1007/978-3-642-03013-0_17

[17] Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. 2019. PhASAR: An Inter-procedural Static Analysis Framework for C/C++. In *Tools and Algorithms for the Construction and Analysis of Systems*, Tomáš Vojnar and Lijun Zhang (Eds.). Springer International Publishing, Cham, 393–410.

[18] Fredrik Siemund and Daniel Tovesson. [n.d.]. Language Server Protocol for ExtendJ.

[19] Anthony M. Sloane, Matthew Roberts, Scott Buckley, and Shaun Muscat. 2014. Monto: a disintegrated development environment. In *Software Language Engineering (Lecture Notes in Computer Science)*, B Combemale, DJ Pearce, O Barais, and JJ Vinju (Eds.). Springer, Springer Nature, United States, 211–220. https://doi.org/10.1007/978-3-319-11245-9_12

[20] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research* (Mississauga, Ontario, Canada) *(CASCON '99)*. IBM Press, 13.

[21] Mathieu Verbaere, Ran Ettinger, and Oege de Moor. 2006. JunGL: A Scripting Language for Refactoring. In *Proceedings of the 28th International Conference on Software Engineering* (Shanghai, China) *(ICSE '06)*. Association for Computing Machinery, New York, NY, USA, 172–181. https://doi.org/10.1145/1134285.1134311