Optimising the Fit of Stack Overflow Code Snippets into Existing Code

Brittany Reid
brittany.reid@adelaide.edu.au
University of Adelaide
Australia

Christoph Treude
christoph.treude@adelaide.edu.au
University of Adelaide
Australia

Markus Wagner
markus.wagner@adelaide.edu.au
University of Adelaide
Australia

ABSTRACT

Software developers often reuse code from online sources such as Stack Overflow within their projects. However, the process of searching for code snippets and integrating them within existing source code can be tedious. In order to improve efficiency and reduce time spent on code reuse, we present an automated code reuse tool for the Eclipse IDE (Integrated Developer Environment), NLP2TestableCode. NLP2TestableCode can not only search for Java code snippets using natural language tasks, but also evaluate code snippets based on a user's existing code, modify snippets to improve fit and correct errors, before presenting the user with the best snippet, all without leaving the editor. NLP2TestableCode also includes functionality to automatically generate customisable test cases and suggest argument and return types, in order to further evaluate code snippets. In evaluation, NLP2TestableCode was capable of finding compilable code snippets for 82.9% of tasks, and testable code snippets for 42.9%.

CCS CONCEPTS

• **Software and its engineering → Software libraries and repositories**.

KEYWORDS

Crowd-generated code snippets, Stack Overflow, Optimisation

ACM Reference Format:
Brittany Reid, Christoph Treude, and Markus Wagner. 2020. Optimising the Fit of Stack Overflow Code Snippets into Existing Code. In *Genetic and Evolutionary Computation Conference Companion (GECCO '20 Companion), July 8–12, 2020, Cancún, Mexico.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3377929.3398087

1 INTRODUCTION

Among software developers, reusing code snippets from the Internet is a common occurrence, with 79% of developers reporting that they copied code from the popular programming question and answer site Stack Overflow (SO) [14] for use in their own projects within the last month [1]. While the benefits of this kind of code reuse are hard to quantify, case studies have previously observed a return of investment of up to 400% [9]. With 19 million questions and 29 million answers as of March 2020 [6], Stack Overflow is one of the most popular resources for code snippets; however, due to their crowd-sourced nature, the quality of code snippets varies, with only 8.41% of answers containing compilable code [20]. This makes the process of integrating code snippets time-consuming; time that could be spent writing code is instead spent correcting compiler errors.

Existing code reuse tools like NLP2Code [4] and Blueprint [3] automate the process of searching for code snippets within the editor; however, snippets are inserted as-is and developers must still make changes in order to integrate them into existing source code. On the other hand, tools like CSNIPPEX [20] and Jigsaw [5] automate code corrections and integration, but rely on a developer to supply a code snippet. As it stands, no existing tool attempts to automate the entire code reuse process, nor assist developers with testing code from online sources.

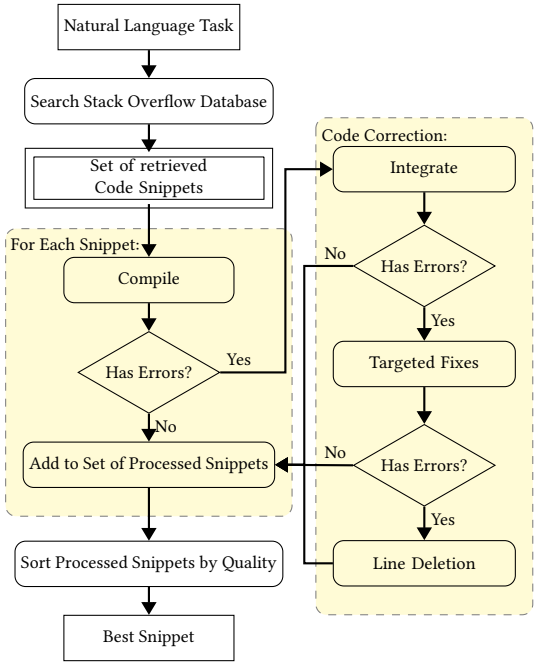

Figure 1: NLP2TestableCode's process, from task to snippet.

To address these issues, we employ a combination of methods from data-driven search-based software engineering (DSE) [12], resulting in NLP2TestableCode, a plug-in for the Eclipse IDE that (1) uses natural language tasks to search for relevant Java code

GECCO '20 Companion, July 8–12, 2020, Cancún, Mexico

ACM ISBN 978-1-4503-7127-8/20/07…$15.00
https://doi.org/10.1145/3377929.3398087

(a) Snippet

Alternatively, you can use an Ints method from the Guava library:

```
import com.google.common.primitives.Ints;
import java.util.Optional;
```

This makes for a concise way to convert a string into an int:

```
int foo = 0;
foo = Optional.ofNullable(myString)
        .map(Ints::tryParse)
        .orElse(0);
```

(b) Copy and Paste

```
public class Main{
    public static void
    main(String[] args){
        import com.google.common.primitives.Ints;
        import java.util.Optional;

        int foo = 0;
        foo = Optional.ofNullable(myString)
                .map(Ints::tryParse)
                .orElse(0);
    }
}
```

Figure 2: Example code snippet adapted from a SO post [21] and the result after being inserted into an existing file.

snippets from a database of SO threads, (2) integrates code snippets by making changes based on existing source code, (3) corrects compiler errors and (4) provides automated testing tools to find working snippets. By automating all of these individual parts of the code reuse process, NLP2TestableCode aims to improve productivity and free up developers for other work. Because context switching has been shown to have a negative effect on productivity [19], we implement NLP2TestableCode as an in-editor tool that reduces the need to switch between the IDE and web browser.

The process, from task to inserted snippet, can be seen in Figure 1. Users begin by entering a natural language task where they would like to insert a snippet. Using this task, the plug-in will find relevant SO threads to extract code snippets from. For each code snippet, a version of the user's code with this snippet inserted is constructed then compiled. Snippets that successfully compile are added to the final set of processed snippets, while non-compilable snippets undergo the code correction process. Here, NLP2TestableCode attempts to integrate, correct specific compiler errors and delete lines from snippets to reduce the number of errors. When snippets compile, or are finished being processed, they are added to the set of processed snippets, which is sorted by number of errors. The first snippet, the one with the least compiler errors, will be inserted into the user's code. From here, users can optionally test the retrieved set of snippets, being provided with input and output type suggestions and a default JUnit test case to customise. The ordering of snippets is then updated based on number of passed tests, changing the inserted snippet if necessary.

Table 1: Comparison of NLP2Code and NLP2TestableCode.

Plug-in	Snippets Retrieved	Tasks with Compilable Snippets	Tasks with Testable Snippets
NLP2Code	355	21.3%	0%
NLP2TestableCode	6,954	82.9%	42.5%

We measured NLP2TestableCode against 47 tasks and compared these results to NLP2Code. A summary of this comparison is presented in Table 1, NLP2TestableCode is capable of presenting users with compilable code snippets for 82.9% of sample tasks, while increasing the number of compilable snippets out of the total retrieved from 4.7% to 29.3% snippets using code correction approaches.

The public GitHub repository for NLP2TestableCode is available at: https://github.com/Brittany-Reid/nlp2testablecode

2 MOTIVATING EXAMPLE

Consider a typical code reuse situation where a developer would like to find an example Java code snippet illustrating how to convert a string into an integer. First, the developer would need to search for snippets; in this case the developer enters the query "How to convert string to int in Java" into their search engine. The first Stack Overflow thread returned for this query has 44 answers, each containing a code snippet. A developer cannot insert, integrate and test every snippet within a reasonable time, instead they must rely on additional information such as votes and comments, or their own programming knowledge to select suitable snippets.

```
    import com.google.common.primitives.Ints;
    import java.util.Optional;
    public class Main {
        public static void main(String[] args){
+           String myString;
+           myString = "empty";
            int foo = 0;
            foo = Optional.ofNullable(myString)
                    .map(Ints::tryParse)
                    .orElse(0);
    }
}
```

Figure 3: The snippet in Figure 2 modified to compile.

Figure 2a shows an example SO answer and the embedded snippet within. The first step to integrating this snippet is to copy and paste it into an existing file. The result of this copy and paste can be seen in Figure 2b. In this state the file will not compile; the import statements are not in the correct place and the variable myString is missing a declaration. To correctly integrate this snippet, the

developer must make a series of changes to correct these problems, including moving the import statement to the start of the file and inserting a declaration and definition for myString. The resulting compilable snippet can be seen in Figure 3.

In contrast, using NLP2TestableCode only requires a developer to enter their task within Eclipse, after which they will be presented with set of snippets modified for them and sorted by best fit. NLP2TestableCode is able to automatically move import statements, fix common syntax errors like missing semi-colons and add missing variable declarations.

3 RELATED WORK

Similar tools that help developers find online code snippet within their IDE include NLP2Code [4], Blueprint [3], Seahawk [15], Prompter [17] and Bing Developer Assistant (BDA) [25]. There has also been much research into using neural networks to map natural language to code snippets [24][23]. While some in-editor tools attempt to evaluate the quality or fit of code snippets, such as NLP2Code's use of SO vote counts to rank snippets and Prompter's ranking system that takes into account existing code [16], none focus on automating the entire code reuse process, including the integration of code snippets.

Research into automated code corrections on Stack Overflow snippets has found that the number of compilable Java snippets could be improved from 1% to 3.02% through simple code fixes, such as adding missing semi-colons [22]. Tools like Jigsaw [5] and CSNIPPEX [20] explore approaches to automated code correction and integration. Jigsaw is a plug-in for Eclipse that can take a code snippet and modify it in order to integrate it within an existing project, while CSNIPPEX takes a Stack Overflow URL and attempts to generate a compilable file from the snippet by correcting compiler errors using Eclipse's Quick Fix functionality [7]. However, neither of these tools aid the user in finding snippets.

μSCALPEL [2] is an automatic code transplant tool that uses genetic programming and testing to transplant code from one program to another. Using test cases to define and maintain functionality, small changes are made to the transplanted code, and code that does not aid in passing tests can be discarded, reducing the code to its minimal functioning form. μSCALPEL is limited in that it is designed to transplant code from one working program to another, not example code like that found on Stack Overflow.

4 APPROACH

NLP2TestableCode undergoes a multistage process to take a natural language task and insert into the user's code the snippet that best fits. These steps include retrieving relevant code snippets from a database of Stack Overflow threads, evaluating code snippets within the context of a user's existing code, modifying snippets that do not compile through the use of integration, targeted fixes and line deletion and then finally sorting the processed set of snippets and inserting the best snippet. NLP2TestableCode is implemented as a plug-in for Eclipse that functions in a single window, and users can cycle through processed snippets as they become available. This cycling replaces the currently inserted snippet with the next best, allowing a user access to more than a single chosen snippet.

An optional testing stage is included to further evaluate snippets. When evaluation and correction is complete, compilable snippets are processed to determine possible argument and return types for testing. This type information is used to suggest test input and output, then used to construct both a customisable skeleton JUnit test case and testable functions from code snippets. After testing, snippet ordering is updated based on passed tests and the inserted snippet is updated with a new best if necessary.

4.1 Using Natural Language Tasks to Find Relevant Snippets

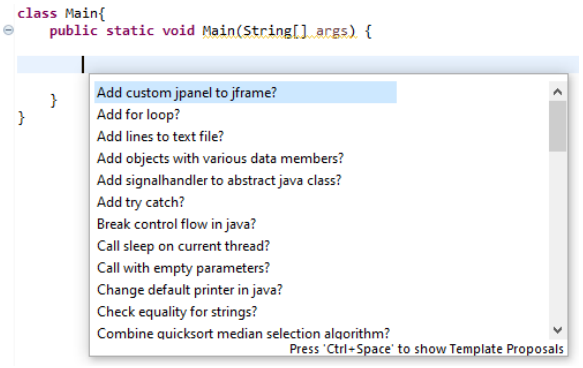

Figure 4: Task suggestions through content assist.

Searching for code snippets effectively is a crucial aspect of automating the code reuse process. Fortunately, code snippets on Stack Overflow are surrounded by natural language questions, explanations and comments. The challenge of mapping tasks to code snippets is ensuring that as many relevant snippets are retrieved as possible.

Online searches like the one used in NLP2Code [4], with a hard-coded limit of 12 snippets per task, are restricted by the time required to download each Stack Overflow thread. To address this, NLP2TestableCode uses an offline SO database, pre-filtered to only include threads tagged with Java, and is capable of retrieving hundreds of snippets in less than a second. The size of this filtered dataset is 6.84GB, containing 1.5 million questions and 2.5 million answers.

Stanford's CoreNLP [10] is used to lemmatise queries and question titles, reducing their words to common forms. Stop words like *the*, *an* and *is* are removed, using a list from NLTK [18] for Python, along with the word *java*, as threads are already filtered by language. Each word in a processed title is added to a database, associated with a list of threads with that word in the title; similarly, processed query words are used to retrieve those same sets of threads.

Users simply enter a task where they would like a snippet to be inserted, either by selecting a task from the suggestions presented through Eclipse's content assist feature, based on NLP2Code's [4] task suggestion functionality, or by ending a custom task with a question mark. With the selected task, the plug-in will begin the process of searching, evaluating and fixing snippets. Combined, these features enable users to integrate code from SO into an existing file without having to leave the editor.

4.2 Evaluating Code Quality

NLP2TestableCode presents the user with the 'best' snippet from the total set of retrieved snippets for a given task. Because the plug-in aims to reduce the amount of manual integration work, snippets should be evaluated based on how well they integrate within the user's existing code, with the snippet requiring the least work to integrate being presented to the user first. To do this, snippets and the user's existing code are combined at the point of task entry, then this code is compiled to count errors. An example of this combined code can be seen in Figure 5.

```
class Main{
    public static void main(String[] args){
+       public static void main(String[] args){
+           int result = Integer.parseInt(args[0]);
+       }
    }
}
```

Figure 5: A snippet, highlighted, inserted into existing code.

By compiling snippets within the context of a user's existing source code, NLP2TestableCode can apply a context sensitive analysis of each snippet. The logic is that snippets which integrate well within existing code should produce less errors than those that do not. By ranking snippets by compiler errors, snippets that best fit existing code can be shown to the user before others. This also means that the set and ordering of compiling snippets can change based on insertion location and surrounding code. Likewise, using compiler errors as a measure of quality enables snippets containing syntax errors or missing elements to be ranked lower than those without errors. The snippet highlighted in Figure 5 is an example of an otherwise correct snippet that contains elements that would cause it to fail to compile when inserted into an already existing main function. However, a snippet that contains only the inner statement would compile with no errors.

Because NLP2TestableCode performs many compiles, it employs the use of an in-memory compiler to reduce compile time. By using in-memory compilation, no files are written to the disk during evaluation. The plug-in makes use of the Eclipse Compiler, the same compiler used within the IDE to underline compiler errors and warnings; because of this the Eclipse Compiler is better suited to compiling incomplete or incorrect code.

4.3 Code Correction and Integration

Most code snippets on Stack Overflow do not compile when inserted as-is. Using a range of automatic code correction techniques, NLP2TestableCode is able to improve the number of compilable snippets and reduce the amount of integration work required from users. All snippets that contain compiler errors are sent through the code correction process. Changes to snippets are only accepted if they decrease the number of compiler errors.

4.3.1 Automatic Integration. Snippets on Stack Overflow are written for example purposes and thus snippets range from single statements to full classes with multiple functions. Depending on the insertion location, these elements may be unnecessary and during manual integration be removed or shifted around. NLP2TestableCode is capable of handling a subset of these instances automatically, 'snippetising' larger pieces of code.

Firstly, all snippets have any import statements extracted during initial processing. These import statements are stored separately from the rest of the snippet, to be inserted into their correct location when required. Without separating import statements, many snippets would have them inserted in incorrect locations.

Frequently SO answers wrap example code within potentially unnecessary classes and functions. Many of these class and method declarations can be removed without altering a snippet's functionality. Snippets are parsed to determine if they contain a class or function. Where snippets contain more than a single class and/or function, they are skipped; these are a more complex case the plug-in currently cannot handle. Classes that contain fields are also ignored, assuming that a snippet that includes fields is demonstrating their usage in some way, and that the class itself is part of the snippet's functionality.

NLP2TestableCode handles the simple case of inserting snippets that contain a main function into an existing main function. This is considered a simple case because the arguments of both functions will be the same. The function declaration can simply be removed along with the closing bracket. After the integration process has been performed, snippets are compiled and the changes are only kept if they reduce compiler errors. This ensures that the integration process improves the correctness of a snippet.

4.3.2 Targeted Fixes. Many snippets on SO contain syntax errors, missing imports and undeclared variables that a developer would typically need to manually correct in order to integrate a snippet. NLP2TestableCode addresses these common compiler errors with targeted fixes, and can insert missing semi-colons and other tokens, find missing import statements, add variable declarations for undefined variables and remove error causing tokens. Compiling a snippet generates a list of diagnostic objects for each error, that contain error codes, location information and error messages. Using both error codes and information within error messages, targeted fixes can be applied to a snippet. The plug-in attempts to fix each error once, and if the fix reduces the number of errors, the changes to the snippet are accepted. Because one fix can sometimes resolve multiple errors and in order to avoid skipping any errors, previously processed errors must be stored and used to recalculate the next error to process.

The plug-in is capable of looking through packages on the Eclipse project classpath to solve missing import statements. It is not uncommon for type names to be used by multiple packages, which makes determining the correct one to use a challenge. In this case, the plug-in prefers classes that belong to packages in the default Java library. This allows common packages like `java.util.List` to be preferred over less common alternatives like `com.sun.tools.javac.util.List`.

When undefined variables are found, the plug-in utilises JavaParser [8] to analyse usage and determine a type. For example, in the line of code `var = "some text";` the variable `var` can be assumed to be of type String by the value it is being assigned. This type information is then used to define and assign a default value

to the variable. If no type can be determined through usage, the plug-in brute forces common types such as `Integer`, `Character`, `String`, `Boolean`, `Double`, `Long` and `Float`.

4.3.3 Line Deletion. NLP2TestableCode's final stage of code correction is line deletion. The aim of line deletion is to reduce a snippet into its optimal form through small changes. Line deletion uses a local search algorithm, detailed in Algorithm 1. The current best, S_{best}, is initialised with the unmodified snippet. For each loop over the snippet, lines are deleted in order starting at the bottom of a snippet and each deletion is accepted if it does not increase the number of errors. A snippet is continuously looped over until no more changes can be made.

Algorithm 1: Deletion Algorithm (Local Search)

```
S_best ← Initial snippet;
done ← false;
while done == false do
    done ← true;
    line ← S_best.length;
    for int j =0 to S_best.length − 1 do
        if S_best(line).Deleted then
            line ← line − 1;
            continue
        S_current ← S_best;
        S_current.Delete(line);
        errors ← Compile(S_current);
        if errors <= S_best.Errors then
            S_best ← S_current;
            done ← false;
        line ← line − 1;
return S_best;
```

4.4 Automated Testing

NLP2TestableCode automates the testing process by integrating JUnit. After retrieving a set of snippets, a user can choose to test these snippets without leaving the current file. The plug-in provides recommendations for argument and return types, and using a given set of argument and return types can generate a default JUnit test case. This test case is inserted into the open file where a user can customise it, and with the press of a button use this test case to test the set of compiling snippets. After testing is complete, the inserted snippet updates with the new best. This testing process also requires transforming snippets into testable functions with input and output.

4.4.1 Suggesting Argument and Return Types. An important part of automating the testing process is being able to identify from a piece of code, which variables could be input and which output. By analysing compilable code snippets, the plug-in can attempt to guess appropriate input and output types. The plug-in does two things: it assumes that the last line of code in a snippet must be relevant to the functionality in some way, and analyses this line for a possible return argument, while it also looks at the variable declarations within a snippet to guess arguments. From these variables, the type information is extracted and used to provide suggestions for testing. This can be seen in the snippet in Figure 6, the last variable being assigned, `foo`, will be chosen as output, while the variable myString will be selected as input. From these variables, the types `String` and `int` will be extracted, to be used as a type suggestion for a argument and return value.

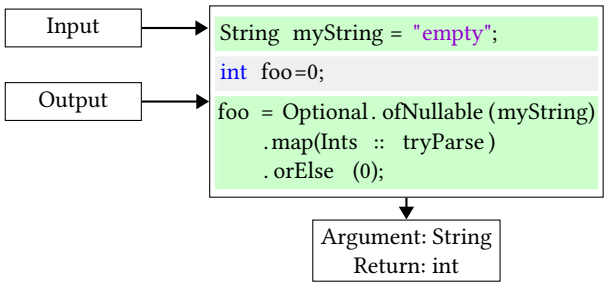

Figure 6: How a snippet is processed for input and output.

If a user chooses to test a set of retrieved snippets, the return and argument type suggestions will be generated and displayed to the user. The user can also choose to insert their own type information. This type information, as well as the number of arguments, is used to construct both a testable function from a snippet, but also a skeleton JUnit test case. The test case's default input is generated using JavaParser's default type value information. An example JUnit test case for the types `String` and `int` can be seen in Figure 7.

```
@Test
public void JUnitTest (){
    assertEquals (snippet("empty"), 0);
}
```

Figure 7: An automatically generated JUnit test case.

4.4.2 Building a Testable Function. Informed by the provided argument and return type information, the plug-in attempts to generate from each snippet a function with input and output that can be tested. This process is similar to the one used to suggest argument and return types from variables, but with added information. Instead of looking at all variables, this search only looks for variables of the given type and number. The last variable of a specified type is accepted as the return, then variables that match argument types are searched for starting at the beginning of the snippet. Where not enough variables of the given types can be found, a testable function is unable to be generated.

4.4.3 Testing. The plug-in uses JUnit to run the user's test case on the generated testable function. Both the JUnit test case and the testable function are combined into a compilable class file, and this code is run in a separate process. Running the code in a separate process allows the plug-in to effectively kill the process if it times out, for example if the code contains an infinite loop. If code runs without errors and passes the test case, the snippet is marked as passing, and ranked above all non-passing snippets.

```
public static int snippet(String myString){
    int foo = 0;
    foo = Optional.ofNullable(myString)
        .map(Ints :: tryParse)
        .orElse (0);
        return foo;
}
```

Figure 8: Snippet from Figure 6 converted into a function.

5 EVALUATION

NLP2TestableCode was evaluated against 47 sample tasks.[1] These tasks are a subset of the total 101 tasks from the original NLP2Code user study [4] for which users used NLP2Code's auto-complete feature. We chose to evaluate NLP2TestableCode against the same set of tasks because they are a representation of the types of tasks real users would find helpful.

5.1 How many code snippets can our approach retrieve?

In order to maximise the number of retrieved snippets per task, different processing techniques for keywords were compared. The initial number of retrieved snippets without the use of lemmatization and removal of stop words was 2,832. We also measured the effects of both lemmatisation using CoreNLP and stemming using the Porter stemming algorithm [11]. Stemming is the process of removing word endings, such as "-ing", "-ed" and "s", to reduce a word to a common form, or its stem. Unlike lemmatisation, stemming does not analyse word context or use dictionary look ups, meaning lemmatisation typically outperforms stemming. However, it has been noted that tools like CoreNLP can misinterpret technical language, like that used when discussing software development [13], because they are trained on more general data. For this reason, comparing both results is necessary.

Table 2: Comparison of total retrieved snippets.

Omit Stop Words?	No Processing	Stemming	Lemmatisation
No	2832	4100	5091
Yes	3464	5646	6954

Table 2 shows the effect different keyword processing techniques (no processing, stemming and lemmatisation) and the omission of stop words have on the total number of code snippets retrieved from the Stack Overflow database. The initial 2,832 snippets could be increased to 4,100 using stemming and 5,091 using lemmatisation. The use of stop words further increases the number of retrieved snippets for all processing techniques, with the highest number of snippets being 6,954 using lemmatisation and omitting stop words, with at least one code snippet for all 47 tasks. These results show that despite its limitations, lemmatisation is still more effective than stemming on programming related tasks, and based on these results we chose to implement it within NLP2TestableCode.

[1] https://github.com/Brittany-Reid/nlp2testablecode/tree/master/data/task, id47.txt

5.2 How many code snippets are compilable?

We measured the number of compilable snippets before any changes in order to provide a benchmark for the results of code corrections. Each retrieved code snippet was inserted into an empty class and main function before being compiled. Because the results of compiling snippets are dependant on a user's existing code, we chose a simple case like this to represent inserting a snippet into some existing structure while avoiding errors caused by, for example, duplicated pre-existing variables.

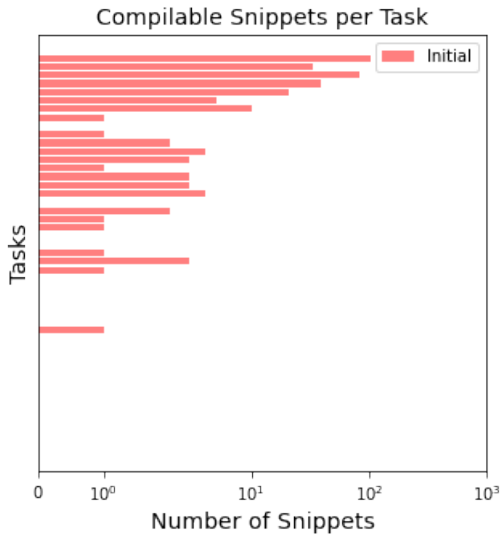

Figure 9: The initial number of compilable snippets per task.

The number of snippets that compile without changes was 327, out of 6,954 total code snippets. For all 47 tasks, 24 have at least a single compilable snippet. The per task breakdown of compilable snippets can be seen in Figure 9, with tasks sorted in descending order based on their final number of compilable snippets after correction.

5.3 What are the most common error types?

We compiled the initial set of code snippets and recorded the types of error codes generated, along with the number of occurrences per error. Each error code generated by the Eclipse compiler corresponds to a constant variable in the Eclipse `IProblem` interface that provides a short description of the error. This information was used to inform what errors should be the focus of our targeted fixes, in an effort to maximise their effect.

Figure 10 shows the 10 most common error types generated during the compilation attempt on the initial unmodified set of snippets. Many of these are parsing errors, such as missing semi-colons or incorrectly placed elements, while others include undeclared variables and types. The non-specific 'parsing error' and 'cannot be resolved' errors make up a large portion of errors and generate compiler messages indicating that these are used when a more specific

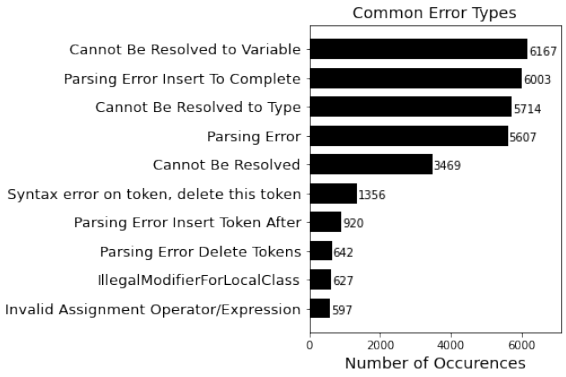

Figure 10: 10 most common compiler errors in snippets.

error cannot be found, for example, the 'cannot be resolved' error can be triggered by both variables and type names, likely when this is ambiguous.

5.4 How many code snippet can our approach make compilable?

In order to determine the effects of each fix, the number of compilable snippets were recorded after each stage of code correction. Again, code snippets were inserted into an empty main function within an empty class before being compiled.

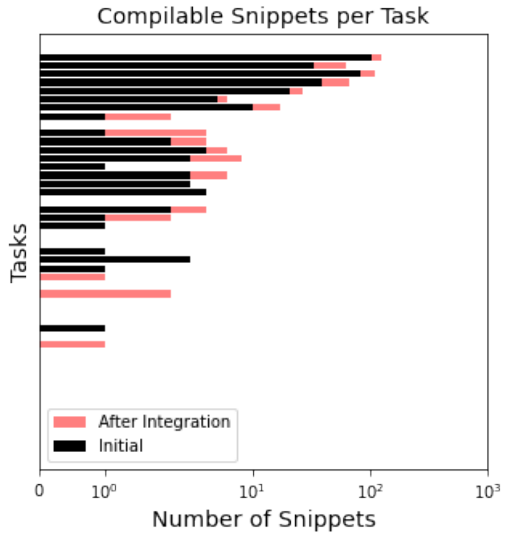

Figure 11: The per task breakdown of compilable snippets after integration in pink, compared to the initial number in black.

5.4.1 Integration. Figure 11 shows the improvement in number of compilable snippets after the integration step. The total number of compilable snippets was increased from the initial 327 to 470. This is considerable considering how limited the integration tools within NLP2TestableCode are; currently we only handle moving import statements to the top of the file and removing empty classes and duplicate main functions. These results indicate that the use of these structures surrounding example code is common enough that 'snippetising', or reducing unnecessary classes and functions down to their containing statements, can have a non-small effect on the number of compilable snippets.

5.4.2 Targeted Fixes. The use of targeted fixes alongside integration increased the number of compilable snippets from 470 to 968. The per task breakdown can be seen in Figure 12, compared to the number of compilable snippets after integration only. After these fixes, the total number of errors fell considerably, from initially 34,427 errors and 34,002 after the integration step, to 21,514 errors.

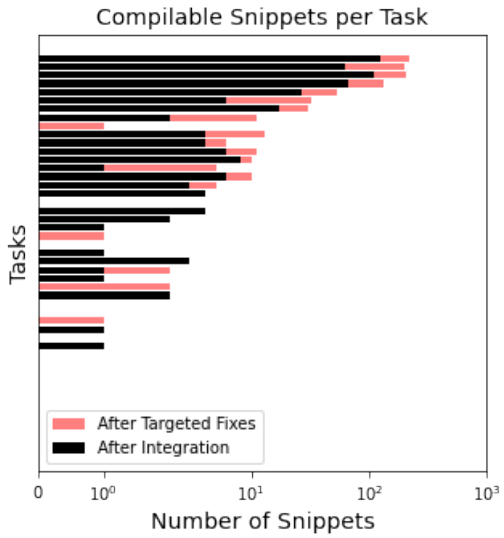

Figure 12: Compilable snippets after targeted fixes compared to after integration.

5.4.3 Line Deletion. Different line deletion configurations were tested, based on the order of line deletion, number of loops over the snippet and acceptance criteria, to determine which configuration maximised the number of compilable snippets. The order of deletion can impact results because often a line has dependencies on other parts of code; for example, deleting a variable declaration before deleting usage or assignment of this variable will generate errors. Similarly, only accepting deletions when they reduce the number of compiler errors, compared to a less-strict acceptance criteria of no increase in errors, can change results and the number of deleted lines considerably.

Figure 13 shows the results for each of eight different deletion algorithms based on three options; single or multiple loops over the snippet, strict or non-strict acceptance and descending or ascending order of line deletion. These results exclude empty snippets. In all cases, deleting lines from the bottom up results in more snippets than the same configuration with descending order deletion. In most cases the multiple loop options and non-strict options outperform their alternatives, besides from the descending order, non-strict

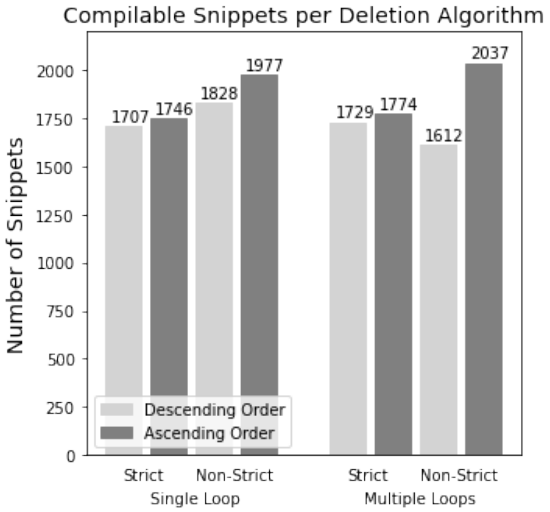

Figure 13: Compilable snippets per deletion algorithm

algorithm which results in 1,612 compilable snippets compared to 1,828 for the single loop alternative and 1,729 snippets for the strict alternative. The non-strict, descending order, multiple loop algorithm outperforms all other algorithms and, because of this, is the algorithm implemented in NLP2TestableCode.

The final number of compilable snippets after deletion is 2,037, a 522.9% increase from the initial number of compilable snippets. Figure 14 shows the final per task break down of compilable snippets compared to only integration and targeted fixes, with 39 out of 47 tasks having at least one compilable snippet.

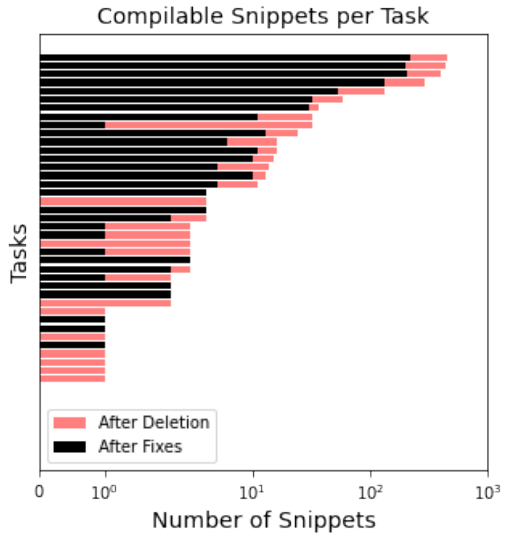

Figure 14: The per task increase in compilable snippets after deletion.

5.5 How many code snippets can our approach generate type suggestions for?

All compilable snippets were run through the type suggestion process. NLP2TestableCode was able to generate at least one type suggestion for 20 tasks, and type suggestions for 316 snippets. This means that at least 20 out of 47 tasks are testable, 51.3% of the 39 tasks with at least one compilable snippet. User supplied tests may make this number larger as the algorithm that searches for matching variables is less strict; for example, the type suggestion algorithm requires at least one argument. Examples of the types of suggestions generated for sample tasks are listed in Table 3. In these cases, the algorithm was capable of generating appropriate suggestions.

Table 3: Example of type suggestions generated for tasks.

Task	Arguments	Return
split string by whitespaces	String	String[]
convert string to integer	String	int
convert uppercase to lowercase	char	char

6 CONCLUSIONS AND FUTURE WORK

NLP2TestableCode's results are promising for the future of automated code reuse. The approaches investigated may be limited, but their effect on the number of code snippets that could be made to integrate is considerable. NLP2TestableCode shows that there is a large amount of improvement in the quality of SO snippets that can be achieved through simple fixes. In our evaluation over 47 tasks, we found that the number of snippets that compile when inserted into an existing piece of code could be increased from 327 to 2,037, and that for 51.3% of tasks, at least one type suggestion could be generated.

Based on our results, more comprehensive code correction and integration tools would likely only serve to further the number of compilable snippets. There is also room to improve the snippet search algorithm, as currently we only map task keywords to Stack Overflow question titles.

Currently the type suggestion algorithm is limited in that it must find at least one argument type and one return type – we cannot test void functions or functions without arguments. In addition to this, the type suggestion algorithm often suggests too many argument types, because it will select all variables besides the last one, which is used for the return. It could also be interesting to look at natural language information within tasks to determine testing types. Similarly, we could investigate better ways to automatically generate JUnit test cases and their automatic input and output values.

In the future, evaluating NLP2TestableCode through a user study similar to the one performed for NLP2Code, would allow us to validate if the new features are useful to developers. This could include evaluating the usefulness of aspects like the highest ranked snippets, the automatic changes made to snippets, type suggestions, automatically generated test cases and the testing process.

ACKNOWLEDGMENTS

This research is supported by an Australian Government Research Training Program (RTP) Scholarship. Christoph's and Markus' work has been supported by the Australian Research Council projects DE180100153 and DE160100850.

REFERENCES

[1] Sebastian Baltes and Stephan Diehl. 2019. Usage and attribution of Stack Overflow code snippets in GitHub projects. *Empirical Software Engineering* 24, 3 (01 Jun 2019), 1259–1295.

[2] Earl T. Barr, Mark Harman, Yue Jia, Alexandru Marginean, and Justyna Petke. 2015. Automated Software Transplantation. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*. ACM, 257–269.

[3] Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R. Klemmer. 2010. Example-centric Programming: Integrating Web Search into the Development Environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10)*. ACM, 513–522.

[4] B. A. Campbell and C. Treude. 2017. NLP2Code: Code Snippet Content Assist via Natural Language Tasks. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 628–632.

[5] Rylan Cottrell, Robert J. Walker, and Jörg Denzinger. 2008. Jigsaw: A Tool for the Small-scale Reuse of Source Code. In *Companion of the 30th International Conference on Software Engineering (ICSE Companion '08)*. ACM, 933–934.

[6] Stack Exchange. 2020. All Sites - Stack Exchange. Retrieved March 10, 2020 from https://stackexchange.com/sites?view=list#traffic

[7] Eclipse Foundation. 2020. Quick Fix and Quick Assist. Retrieved April 16, 2020 from https://help.eclipse.org/2020-03/topic/org.eclipse.jdt.doc.user/concepts/concept-quickfix-assist.htm?cp=1_2_5

[8] JavaParser. 2019. JavaParser. Retrieved March 27, 2020 from https://javaparser.org/

[9] W. C. Lim. 1994. Effects of reuse on quality, productivity, and economics. *IEEE Software* 11, 5 (Sep. 1994), 23–30.

[10] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. 2014. The Stanford CoreNLP Natural Language Processing Toolkit. In *Association for Computational Linguistics (ACL) System Demonstrations*. The Association for Computer Linguistics, 55–60.

[11] Porter Martin. 2006. Porter Stemming Algorithm. https://tartarus.org/martin/PorterStemmer/

[12] Vivek Nair, Amritanshu Agrawal, Jianfeng Chen, Wei Fu, George Mathew, Tim Menzies, Leandro Minku, Markus Wagner, and Zhe Yu. 2018. Data-Driven Search-Based Software Engineering. In *Proceedings of the 15th International Conference on Mining Software Repositories* (Gothenburg, Sweden) *(MSR '18)*. ACM, New York, NY, USA, 341–352.

[13] Fouad Nasser A Al Omran and Christoph Treude. 2017. Choosing an NLP Library for Analyzing Software Documentation: A Systematic Literature Review and a Series of Experiments. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR '17)*. IEEE, 187–197.

[14] Stack Overflow. 2020. Stack Overflow - Where Developers Learn, Share, & Build Careers. Retrieved April 8, 2020 from https://stackoverflow.com

[15] Luca Ponzanelli, Alberto Bacchelli, and Michele Lanza. 2013. Seahawk: Stack Overflow in the IDE. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE, 1295–1298.

[16] Luca Ponzanelli, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Michele Lanza. 2014. Mining StackOverflow to Turn the IDE into a Self-Confident Programming Prompter. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR 2014)*. ACM, 102–111.

[17] Luca Ponzanelli, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Michele Lanza. 2014. Prompter: A self-confident recommender system. In *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 577–580.

[18] NLTK Project. 2019. Natural Language Tool Kit. Retrieved March 27, 2020 from https://www.nltk.org/

[19] Sebastian Proksch, Veronika Bauer, and Gail C. Murphy. 2015. *How to Build a Recommendation System for Software Engineering*. Springer, 1–42.

[20] Valerio Terragni, Yepang Liu, and Shing-Chi Cheung. 2016. CSNIPPEX: Automated Synthesis of Compilable Code Snippets from Q&A Sites. In *Proceedings of the 2016 International Symposium on Software Testing and Analysis, ISSTA 2016*. ACM, 118–129.

[21] Unknown user. 2020. How do I convert a String to an int in Java? Retrieved March 27, 2020 from https://stackoverflow.com/questions/5585779/how-do-i-convert-a-string-to-an-int-in-java

[22] Di Yang, Aftab Hussain, and Cristina Videira Lopes. 2016. From Query to Usable Code: An Analysis of Stack Overflow Code Snippets. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16)*. ACM, 391–402.

[23] Ziyu Yao, Daniel S. Weld, Wei-Peng Chen, and Huan Sun. 2018. StaQC: A Systematically Mined Question-Code Dataset from Stack Overflow. In *Proceedings of the 2018 World Wide Web Conference* (Lyon, France) *(WWW '18)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 1693–1703. https://doi.org/10.1145/3178876.3186081

[24] Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to Mine Aligned Code and Natural Language Pairs from Stack Overflow. In *Proceedings of the 15th International Conference on Mining Software Repositories* (Gothenburg, Sweden) *(MSR '18)*. Association for Computing Machinery, New York, NY, USA, 476–486. https://doi.org/10.1145/3196398.3196408

[25] Hongyu Zhang, Anuj Jain, Gaurav Khandelwal, Chandrashekhar Kaushik, Scott Ge, and Wenxiang Hu. 2016. Bing Developer Assistant: Improving Developer Productivity by Recommending Sample Code. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, 956–961.