Analyzing and Supporting Adaptation of Online Code Examples

***Abstract*—Developers often resort to online Q&A forums such as Stack Overflow (SO) for filling their programming needs. Although code examples on those forums are good starting points, they are often incomplete and inadequate for the local context of the developers; adaptation of those examples is necessary to integrate them to production code. As a consequence, the process of adapting online examples is done over and over again, by multiple developers independently. Our work extensively studies these adaptations and variations, serving as the basis for a tool that helps integrate these online code examples in a target context in an interactive manner.**

We perform a large-scale empirical study about nature and extent of adaptations and variations of SO snippets. We construct a comprehensive dataset linking SO posts to GitHub counterparts based on clone detection, time stamp analysis, and explicit URL references. We then qualitatively inspect 400 SO examples and their GitHub counterparts and develop a taxonomy of 24 adaptation types. Using this taxonomy, we build an automated adaptation analysis technique on top of GumTree to classify the entire dataset into these categories. We build a Chrome extension called EXAMPLESTACK that automatically lifts an adaptation-aware template from each SO example and its GitHub counterparts to identify hot spots where most changes happen. A user study with twelve programmers shows that seeing the commonality and variations in similar GitHub counterparts increases their confidence about how to reuse the given SO example, and helps grasp a more comprehensive view about how to reuse the example differently and avoid common pitfalls.

***Index Terms*—online code examples, code adaptation**

I. INTRODUCTION

Nowadays, a common way of quickly accomplishing specific programming tasks is to search, copy, paste, and adapt code examples in online Q&A forums such as Stack Overflow [1]–[3]. A case study at Google shows that developers issue an average of twelve code search queries per weekday [4]. As of July 2018, Stack Overflow has accumulated 26M answers to 16M programming questions. Copying code from Stack Overflow is common [5] and adapting code to fit a target program is recognized as a top barrier when reusing code from Stack Overflow [6]. Stack Overflow examples are created for illustration purposes, which can serve as a good starting point. However, these examples may be insufficient to be ported to a production environment, as previous studies find that Stack Overflow examples may suffer from API usage violations [7], insecure coding practices [8], unchecked obsolete usage [9], and incomplete code fragments [10]. Therefore, developers may have to manually customize online examples when importing them into their own projects.

Our goal is to investigate the common adaptation types and frequencies of online code examples, such as those found in Stack Overflow, which are used by a large number of software developers around the world. To study how they are adopted and adapted in real projects, we contrast them against similar code snippets in GitHub projects. The insights gained from this study could inform the design of tools for helping developers adapt code snippets they find in Q&A sites. In this paper, we describe one such tool we developed, EXAMPLESTACK, which works as a Chrome extension.

In broad strokes, the design and main results of our study are as follows. We link SO examples to GitHub counterparts using multiple complementary filters. First, we quality-control GitHub data by removing forked projects and selecting projects with at least five stars. Second, we perform clone detection [11] between 312K SO posts and 51K non-forked GitHub projects to ensure that SO examples are similar to GitHub counterparts. Third, we perform timestamp analysis to ensure that GitHub counterparts are created later than the SO examples. Fourth, we look for explicit URL references from GitHub counterparts to SO examples by matching the post ID. As the result, we construct a comprehensive dataset of *variations* and *adaptations*.

When we use all four filters above, we find only 629 SO examples with GitHub counterparts. Recent studies find that very few developers explicitly attribute to the original SO post when reusing code from Stack Overflow [5, 6, 12]. Therefore, we use this resulting set of 629 SO examples as an *under-approximation* of SO code reuse and call it as an *adaptations* dataset. If we apply only the first three filters above, we find 14,124 SO examples with GitHub counterparts that represent potential code reuse from SO to GitHub. While this set does not necessarily imply any causality or intentional code reuse, it still demonstrates the kinds of common variations between SO examples and their GitHub counterparts, which developers might want to consider during code reuse. Therefore, we consider this second dataset as an *over-approximation* of SO code reuse, and call it simply a *variations* dataset.

We randomly select 200 groups from each of the *adaptations* and *variations* datasets respectively and manually inspect SO examples and their GitHub counterparts to develop an adaptation taxonomy. This taxonomy consists of 6 high-level categories and 24 specialized types. We then develop an automated adaptation analysis technique built on top of GumTree [13] to categorize syntactic program differences into different adaptation types. The precision and recall of this technique are 98% and 96% respectively. This technique allows us to quantify the extent of common adaptations and variations in each dataset. The analysis shows that both the adaptations and

variations between SO examples and their GitHub counterparts are prevalent and non-trivial. It also highlights several adaptation types that are frequently performed yet not automated by existing code integration techniques [14]–[16], e.g., type conversion, handling potential exceptions, adding `if` checks.

Building on this adaptation analysis technique, we develop a Chrome extension called EXAMPLESTACK and conduct a user study with twelve developers to understand how EXAMPLESTACK may guide developers in adapting and customizing online code examples. For a given SO example, EXAMPLESTACK shows a list of similar code snippets in GitHub and presents an adaptation-aware template lifted from those snippets by identifying common, unchanged code, as well as the hot spots where most changes happen. In the user study, participants using EXAMPLESTACK annotate 55% more locations to modify and are able to provide a more comprehensive view of how to use APIs differently or how to avoid common pitfalls. In the post survey, participants find it useful to see the commonality and variations between the SO example and GitHub counterparts, saying that it helps them to easily reach consensus on how to reuse the code example. Participants also feel more confident of reusing the SO example after seeing how similar code is used in GitHub counterparts, which one participant describes as "*asynchronous pair programming*."

In summary, this work makes the following contributions:

- It makes a publicly available comprehensive dataset of *adaptations* and *variations* between SO and GitHub. The adaptation dataset includes 629 groups of GitHub counterparts with explicit references to SO posts, and the variation dataset includes 14,124 groups. These datasets are created with care using multiple complementary methods for quality control—clone detection, time stamp analysis, and explicit references. A large number of samples are manually inspected by the authors.
- It puts forward an adaptation taxonomy of online code examples and an automated technique for classifying adaptations. This taxonomy is sufficiently different from other change type taxonomies from refactoring [17] and software evolution [18, 19], and it captures the particular kinds of adaptations done over online code examples.
- It provides browser-based tool support, called EXAMPLESTACK that displays the commonality and variations between a SO example and GitHub counterparts along with their adaptation types and frequencies. Participants find that seeing GitHub counterparts increases their confidence on how to reuse the SO example and helps them understand different corner cases of reusing code.

The rest of the paper is organized as follows. Section II describes the data collection pipeline and compares the characteristics of the two datasets. Section III describes the adaptation taxonomy development and an automated adaptation categorization technique. Section IV describes the quantitative analysis of adaptations and variations. Section V explains the design and implementation of EXAMPLESTACK. Section VI describes a user study that evaluates the usefulness of EXAMPLESTACK. Section VII discusses several adaptation automation opportunities highlighted by our findings and threats to validity. Section VIII contrasts our work with related work, and Section IX concludes the paper.

II. LINKING STACK OVERFLOW TO GITHUB

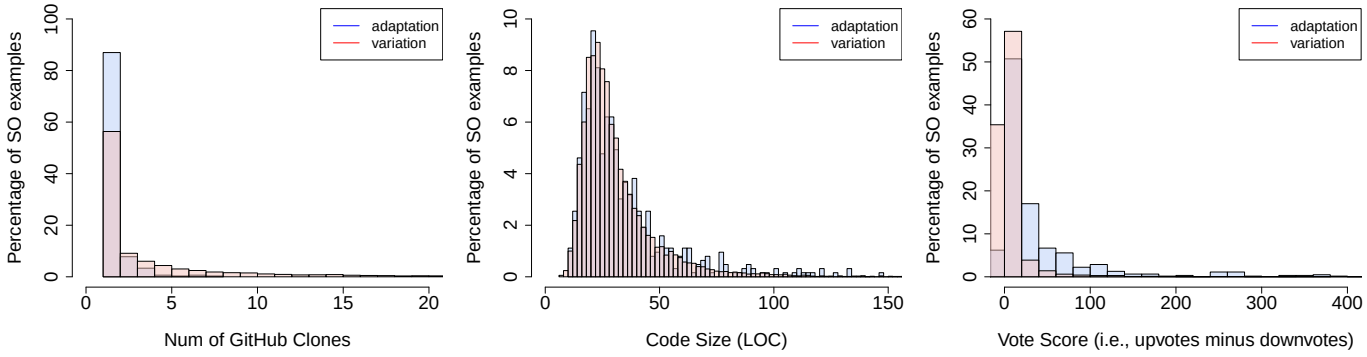
This section describes the data collection pipeline. Due to the large portion of unttributed SO examples in GitHub [5, 6, 12], it is challenging to construct a complete set of reused code from SO to GitHub. To overcome this limitation, we apply four quality-control filters to *underapproximate* and *overapproximate* code examples reused from SO to GitHub, resulting in two complementary datasets.

GitHub project selection and deduplication. Since GitHub has many toy projects that do not adequately reflect software engineering practices [20], we only consider GitHub projects that have at least five stars. To account for internal duplication in GitHub [21], we choose non-fork projects only and further remove duplicated GitHub files since such the entire file duplication may unnecessarily skew our analysis. As a result, we download 50,826 non-forked Java repositories with at least five stars from GitTorrent [22]. After deduplication, 5,825,727 distinct Java files remain.

Detecting GitHub candidates for SO snippets. From the SO dump taken in October 2016 [23], we extract 312,219 answer posts that have `java` or `android` tags and also contain code snippets in the `<code>` markdown. We consider code snippets in answer posts only, since snippets in question posts are rarely used as examples. Then we use a token-based clone detector, SourcererCC (SCC) [11] to find similar code between 5.8M distinct Java files and 312K SO posts. We choose SCC because it has high precision and recall and also scales to a large code corpus. Since SO code snippets are often free-standing statements [24, 25], we wrap them as Java methods and tokenize them using a customized Java parser [26] to exclude comments, Java keywords, and other special characters. Prior work finds that larger SO snippets have more meaningful clones in GitHub [27]. Based on this insight, we choose to study SO examples with no less than 50 tokens. We set the similarity threshold to 70% since it yields the best precision and recall on multiple clone benchmarks [11]. We run SCC on a server machine with 116 cores and 256G RAM. It takes 24 hours to complete, resulting in 21,207 SO methods that have one or mode similar code fragments (i.e., clones) in GitHub.

Timestamp analysis. If the GitHub clone of a SO example is created before the SO post, we consider it unlikely to be reused from SO and remove it from our dataset. To identify the creation date of a GitHub clone, we write a script to retrieve the Git commit history of the file and match the clone snippet against each file revision. We use the timestamp of the earliest matched file revision as the creation time of a GitHub clone. As a result, 7,083 SO examples (33%) are excluded since all their GitHub clones are committed before the SO posts.

Scanning explicitly attributed SO examples. Despite the large portion of unattributed SO examples, it is certainly possible to



(a) Examples with different number of clones (b) Examples with different code length (c) Examples with different vote scores

Fig. 1: Comparison between SO examples in the adaptation dataset and the variation dataset.

scan GitHub clones for explicit references such as SO links in code comments to confirm whether a clone is copied from SO. Note that a SO link may point to a question post instead of an answer post. In such cases, we check whether the post id of the corresponding SO example matches any of the answer posts to the question. As a result, we find 629 explicitly referenced SO examples.

Overapproximating and underapproximating reused code. We use the set of 629 explicitly attributed SO examples as an *underapproximation* of reused code from SO to GitHub, which we call an *adaptation* dataset. We consider the remaining 14,124 SO examples after timestamp analysis as an *overapproximation* of potentially reused code, which we call a *variation* dataset. Figure 1 compares the characteristics of these two datasets of SO examples in terms of the number of GitHub clones, code length, and vote score (i.e., upvotes minus downvotes). Since developers do not often attribute SO code examples, explicitly referenced examples have a median of one GitHub clone only, while SO examples have a median of two clones in the variation dataset. Both sets of SO examples have similar length, 26 vs. 25 lines of code in median. However, SO examples from the adaptation dataset have significantly more upvotes than the variation dataset: 16 vs. 1 in median. In the following sections, we inspect, analyze, and quantify the adaptations and variations evidence by both datasets.

III. Adaptation Type Analysis

A. Manual Inspection

To get insights into adaptations and variations of SO examples, we randomly sample 200 SO examples and corresponding GitHub counterparts from each of the two datasets. Then we inspect the differences between each SO example and its GitHub counterpart using a program differencing tool, GumTree [13]. Below, while variations evidenced by the second dataset do not necessarily show intentional adaptation or reuse, we use the term, adaptation type for presentation purposes throughout this paper.

The first and the last authors jointly labelled these SO examples with the descriptions of adaptations and grouped the edits with similar descriptions to identify common adaptation types. The two authors then discussed with the other authors to refine the adaptation types. Finally, we grouped fine-grained adaptation types to high-level categories based on underlying intent. Table I describes 24 adaptation types in 6 categories.

Code Hardening. This category includes four adaptation types that strengthen SO examples in a target project. *Insert a conditional* adds an `if` statement that checks for corner cases or protects code from invalid input data such as `null` or an out-of-bound index. *Insert a final modifier* enforces that a variable can only be initialized once and the assigned value or reference cannot be changed, which is generally recommended for clear design and better performance due to static inlining. *Handle a new exception* improves the reliability of a code example by handling any missing exceptions, since exception handling logic is often omitted in examples in SO [7]. *Clean up unmanaged resources* helps release unneeded resources such as file streams and web sockets to avoid resource leaks [28].

Resolve Compilation Errors. SO examples are often incomplete with undefined variables and method calls, which requires resolving compilation errors [25, 29]. *Declare an undeclared variable* inserts a statement to declare an unknown variable. *Specify a target of method invocation* resolves an undefined method call by specifying the receiver object that the method is invoked on. In a SO example of getting CPU usage [30], one comment complains that the example does not compile due to an unknown method call, `getOperatingSystemMXBean`. Another suggests to preface the method call with an instance, `ManagementFactory`, which is evidenced by its GitHub counterpart [31]. Sometimes, statements that use undefined variables and method calls are simply deleted.

Exception Handling. This category represents adaptations on the exception handling logic of an example, including changes in `catch/finally` blocks and thrown exceptions declared in the method header. One common change is to customize the actions in a `catch` block, such as printing a short error message instead of the entire stack trace. Sometimes exceptions are handled locally using `try-catch` blocks rather than throwing exceptions in the method header. For example, while the SO example [32] throws a generic

TABLE I: Common adaptation types, categorization, and implementation.

Category	Adaptation Type	Rule
Code Hardening	Add a conditional	Insert(t_1, t_2, i) $\wedge$ NodeType(t_1, IfStatement)
	Insert a final modifier	Insert(t_1, t_2, i) $\wedge$ NodeType(t_1, Modifier) $\wedge$ NodeValue(t_1, final)
	Handle a new exception type	Exception(e, GH) $\wedge$ $\neg$Exception(e, SO)
	Clean up unmanaged resources (e.g. close a stream)	(LocalCall(m, GH) $\vee$ InstanceCall(m, GH)) $\wedge$ $\neg$LocalCall(m, SO) $\wedge$ $\neg$InstanceCall(m, SO) $\wedge$ isCleanMethod(m)
Resolve Compilation Errors	Declare an undeclared variable	Insert(t_1, t_2, i) $\wedge$ NodeType(t_1, VariableDeclaration) $\wedge$ NodeValue(t_1, v) $\wedge$ Use(v, SO) $\wedge$ $\neg$Def(v, SO)
	Specify a target of method invocation	InstanceCall(m, GH) $\wedge$ LocalCall(m, SO)
	Remove undeclared variables or local method calls	(Use(v, SO) $\wedge$ $\neg$Def(v, SO) $\wedge$ $\neg$Use(v, GH)) $\vee$ (LocalCall(m, SO) $\wedge$ $\neg$LocalCall(m, GH) $\wedge$ $\neg$InstanceCall(m, GH))
Exception Handling	Insert/delete a try-catch block	(Insert(t_1, t_2, i) $\vee$ Delete(t_1)) $\wedge$ NodeType(t_1, TryStatement)
	Insert/delete a thrown exception in a method header	Changed(t_1) $\wedge$ NodeType(t_1, Type) $\wedge$ Parent(t_2, t_1) $\wedge$ NodeType(t_2, MethodDeclaration) $\wedge$ NodeValue(t_1, t) $\wedge$ isExceptionType(t)
	Update the exception type	Update(t_1, t_2) $\wedge$ NodeType(t_1, SimpleType) $\wedge$ NodeType(t_2, SimpleType) $\wedge$ NodeValue(t_1, v_1) $\wedge$ isExceptionType(v_1) $\wedge$ NodeValue(t_2, v_2) $\wedge$ isExceptionType(v_2)
	Change statements in a catch block	Changed(t_1) $\wedge$ Ancestor(t_2, t_1) $\wedge$ NodeType(t_2, CatchClause)
	Change statements in a finally block	Changed(t_1) $\wedge$ Ancestor(t_2, t_1) $\wedge$ NodeType(t_2, FinallyBlock)
Logic Customization	Change a method call	Changed(t_1) $\wedge$ Ancestor(t_2, t_1) $\wedge$ NodeType(t_2, MethodInvocation)
	Update a constant value	Update(t_1, t_2) $\wedge$ NodeType(t_1, Literal) $\wedge$ NodeType(t_2, Literal)
	Change a conditional expression	Changed(t_1) $\wedge$ Ancestor(t_2, t_1) $\wedge$ (NodeType(t_2, IfCondition) $\vee$ NodeType(t_2, LoopCondition) $\vee$ NodeType(t_2, SwitchCase))
	Change the type of a variable	Update(t_1, t_2) $\wedge$ NodeType(t_1, Type) $\wedge$ NodeType(t_2, Type)
Refactoring	Rename a variable/field/method	Update(t_1, t_2) $\wedge$ NodeType(t_1, Name)
	Replace hardcoded constant values with variables	Delete(t_1) $\wedge$ NodeType(t_1, Literal) $\wedge$Insert(t_1, t_2, i) $\wedge$ NodeType(t_1, Name) $\wedge$ Match(t_1, t_2)
	Inline a field	Delete(t_1) $\wedge$ NodeType(t_1, Name) $\wedge$Insert(t_1, t_2, i) $\wedge$ NodeType(t_1, Literal) $\wedge$ Match(t_1, t_2)
Miscellaneous	Change access modifiers	Changed(t_1) $\wedge$ NodeType(t_1, Modifier) $\wedge$ NodeValue(t_1, v) $\wedge$ v $\in$ {private, public, protected, static}
	Change a log/print statement	Changed(t_1) $\wedge$ NodeType(t_1, MethodInvocation) $\wedge$ NodeValue(t_1, m) $\wedge$ isLogMethod(m)
	Style reformatting (i.e., inserting/deleting curly braces)	Changed(t_1) $\wedge$ NodeType(t_1, Block) $\wedge$ Parent(t_2, t_1) $\wedge$ $\neg$Changed(t_2) $\wedge$ Child(t_3, t_1) $\wedge$ $\neg$Changed(t_3)
	Change Java annotations	Changed(t_1) $\wedge$ NodeType(t_1, Annotation)
	Change code comments	Changed(t_1) $\wedge$ NodeType(t_1, Comment)

GumTree Edit Operation	Syntactic Predicate	Semantic Predicate
Insert(t_1, t_2, i) inserts a new tree node t_1 as the i-th child of t_2 in the AST of the GitHub snippet.	**NodeType**(t_1, X) checks if the node type of t_1 is X.	**Exception**(e, P) checks if e is an exception caught in a `catch` clause or thrown in a method header in program P.
	NodeValue(t_1, v) checks if the corresponding source code of node t_1 is v.	**LocalCall**(m, P) checks if m is a local method call in program P.
Delete(t) removes the tree node t from the AST of the SO example.	**Match**(t_1, t_2) checks if t_1 and t_2 are matched based on surrounding nodes regardless of node types.	**InstanceCall**(m, P) checks if m is an instance call in program P.
	Parent(t_1, t_2) checks if t_1 is the parent of t_2 in the AST.	**Def**(v, P) checks if variable v is defined in program P.
Update(t_1, t_2) updates the tree node t_1 in the SO example with t_2 in the corresponding GitHub snippet.	**Ancestor**(t_1, t_2) checks if t_1 is the ancestor of t_2 in the AST.	**Use**(v, P) checks if variable v is used in program P.
	Child(t_1, t_2) checks if t_1 is the child of t_2.	**IsExceptionType**(X) checks if X contains "Exception".
Move(t_1, t_2, i) moves an existing node t_1 in the AST of the SO example as the i-th child of t_2 in the corresponding GitHub snippet.	**Changed**(t_1) is a shorthand for **Insert**(t_1, t_2, i) $\vee$ **Delete**(t_1) $\vee$ **Update**(t_1, t_2) $\vee$ **Move**(t_1, t_2), which checks any edit operation on t_1	**IsLogMethod**(X) checks if X is one of the predefined log methods, e.g., log, println, error, etc.
		IsCleanMethod(X) checks if X is one of the predefined resource clean-up methods, e.g., close, recycle, dispose, etc.

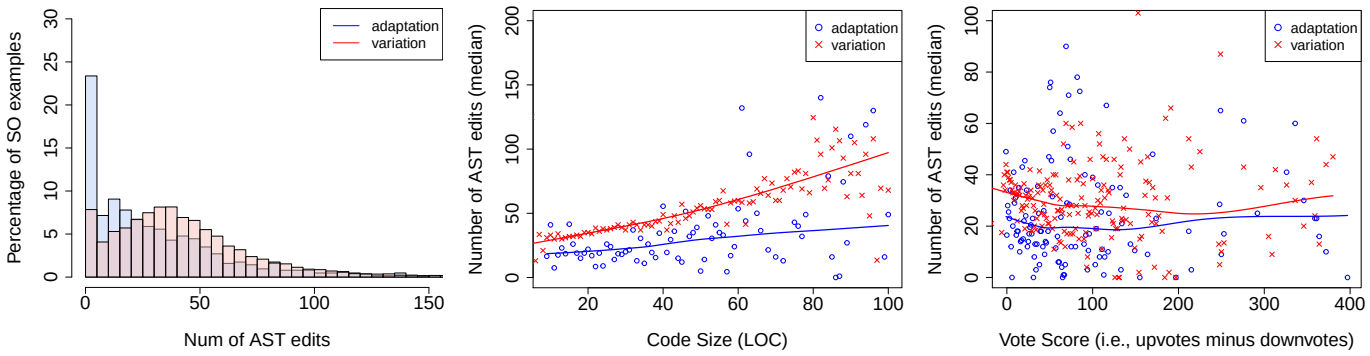
`Exception` in the `addLibraryPath` method, the counterpart GitHub method [33] enumerates all possible exceptions such as `SecurityException` and `IllegalArgumentException` in a `try-catch` block. By contrast, propagating the exceptions to upstream by adding `throws` in the method header is another way to handle the exceptions.

Logic Customization. Customizing the functionality of a code example to fit a target project is a common and broad category. We categorize logic changes to four basic types. *Change a method call* includes any edits in a method call, e.g., adding or removing a method call, changing its arguments or receiver, etc. *Update a constant value* changes a constant value such as the thread sleep time to another value. *Change a conditional expression* includes any edits on the condition expression of an `if` statement, a `loop`, or a `switch case`.

Update a type name replaces the variable type or the method return type with another type. For example, `String` and `StringBuffer` appear in multiple SO examples, and a faster type, `StringBuilder`, is used instead in their GitHub counterparts. Such type replacement often involves extra changes such as updating methods calls to fit the replaced type or adding method calls to convert one type to another. For example, instead of `InetAddress` in the SO example [34], its GitHub counterpart uses the return type `String` and subsequently converts the IP address object to its string format using a new `Formatter` API. Such type conversion can be automated via type analysis.

Refactoring. 31% of inspected GitHub counterparts use a method or variable name different from the SO example. Instead of `slider` in a SO example [35], `timeSlider` is used in one GitHub counterpart [36] and `volumnSlider` is used in another counterpart [37]. Because SO examples often use hardcoded constant values for illustration purposes, GitHub counterparts may use variables instead of hardcoded constants. However, sometimes, a GitHub counterpart such as [38] does the opposite by inlining the values of two constant fields, `BUFFER_SIZE` and `KB`, since these fields do not appear along with the copied method, `downloadWithHttpClient` [39].

Miscellaneous. Adaptation types in this category do not have a significant impact on the reliability and functionality of a



(a) Distribution of AST edits (b) Code size vs. AST edits (c) Vote score vs. AST edits

Fig. 2: Code size (LOC) and vote scores on the number of AST edits in a SO example

SO example. However, several interesting cases still worth attention. In 91 inspected examples, GitHub counterparts include comments to explain the reused code. Sometimes, annotations such as `@NotNull` or `@DroidSafe` appear in GitHub counterparts to document the constraints of code.

B. Automated Adaptation Categorization

Based on the manual inspection, we build a rule-based classification technique that automatically categorizes AST edit operations generated by GumTree to different adaptation types. GumTree supports four edit operations—**insert**, **delete**, **update**, and **move**, described in GumTree Edit Operation in Table I. Given a set of AST edits, our technique leverages both syntactic and semantic rules to categorize the edits to 24 adaptation types. Column Rule in Table I describes the implementation logic of categorizing each adaptation type.

Syntactic-based Rules. 16 adaptation types are detected based on syntactic information, e.g., edit operation types, AST node types and values, etc. Column Syntactic Predicate defines the syntactic information used by our technique and can be obtained based on the built-in functions provided by GumTree. For example, the rule of *insert a final modifier* checks for an edit operation that inserts a `Modifier` node whose value is `final` in a GitHub counterpart.

Semantic-based Rules. 8 adaptation types require leveraging semantic information to be detected (Column Semantic Predicate). For example, the rule of *declare an undeclared variable* checks for an edit operation that inserts a `VariableDeclaration` node in the GitHub counterpart and the variable name is *used* but not *defined* in the SO example. Our technique traverses ASTs to gather such semantic information. For example, our AST visitor keeps track of all declared variables when visiting a `VariableDeclaration` AST node, and all used variables when visiting a `Name` node.

C. Accuracy of Adaptation Categorization

We randomly sampled another 100 pairs of SO examples and their GitHub counterparts to evaluate our automated categorization technique. To reduce bias, the second author who was not involved in the manual inspection of the taxonomy design labelled the adaptation types in this validation set. The ground truth contains 449 manually labeled adaptation types in 100 examples. Overall, our technique infers 440 adaptation types with 98% precision and 96% recall. In 80% SO examples, our technique infers all adaptation types correctly. In another 20% SO examples, it infers some but not all expected adaptation types.

Our technique infers incorrect or missing adaptation types for two main reasons. First, our technique only considers 24 common adaptation types in Table I but does not handle infrequent ones such as refactoring using lambda expressions and rewriting `++i` to `i++`. Second, GumTree may generate sub-optimal edit scripts with unnecessary edit operations in about 5% file pairs, according to [13]. In such cases, our technique may mistakenly report incorrect adaptation types.

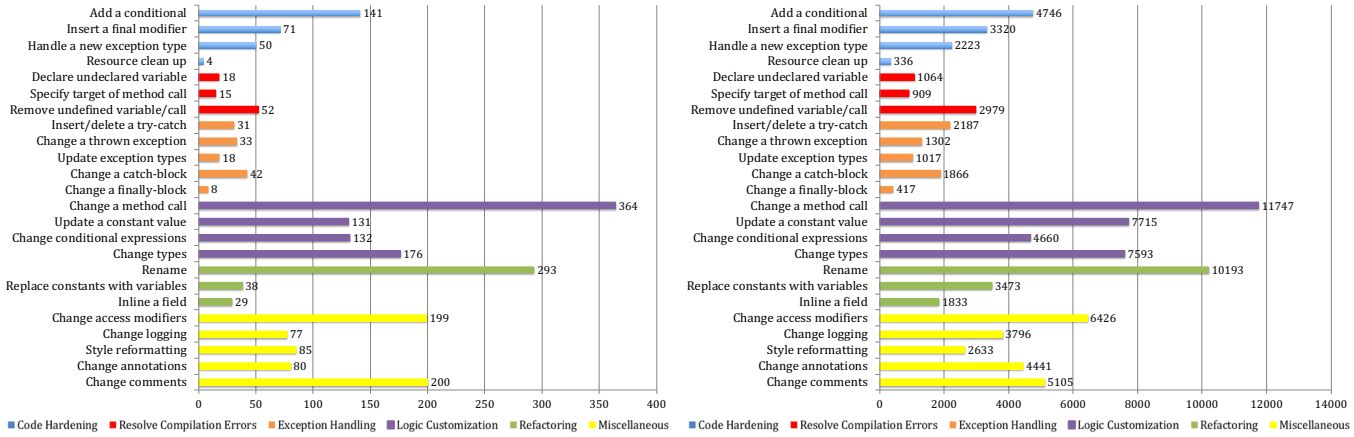
IV. Empirical Study

A. How many edits are potentially required to adapt a SO example?

We apply the adaptation categorization technique to quantify the extent of adaptions and variations in the two datasets of SO examples and their GitHub counterparts. We measure AST edits between a SO example to its GitHub counterpart. If a SO code example has multiple GitHub counterparts, we use the average number. Overall, 14,197 SO examples (96%) of the variation dataset include a median of 39 AST edits (mean 47). 556 SO examples (88%) of the adaptation dataset include a median of 23 AST edits (mean 33). Figure 2a compares the distribution of AST edits in these two datasets. In both datasets, most SO examples have variations from their counterparts, indicating that integrating them to production code may require some type of adaptations.

Figure 2b shows the median number of AST edits in SO examples with different lines of code. We perform a loess regression on the size of a SO example and the number of AST edits. As shown by the two lines in Figure 2b, there is a strong positive correlation between the number of AST edits and the SO example size in both datasets—long SO examples have more variations than short examples.

Stack Overflow allows users to upvote or downvote a post to indicate the applicability and usefulness of the post. Therefore,



(a) Adaptations: 629 explicitly attributed SO examples

(b) Variations: 14,124 potentially reused SO examples

Fig. 3: Frequencies of categorized adaptation types in two datasets

votes are often considered as the main quality metric of SO examples [40]. Figure 2c shows the median number of AST edits in SO examples with different vote scores. Despite the finding that the adaptation dataset has significantly higher votes than the variation dataset (Figure 1c), there is no strong positive or negative correlation between the AST edit and the vote score in both sets. This implies that highly voted SO examples do not necessarily have fewer edits from GitHub counterparts than those with low vote scores.

B. What are common adaptation and variation types?

To investigate the common adaptation types across different SO examples, we count the distinct types of adaptations applied to a code example. If a SO code example has multiple GitHub counterparts, we only consider the distinct types among all GitHub counterparts to avoid the inflation caused by repetitive variations among different counterparts. Figure 3 compares the frequencies of the 24 categorized adaptation types (Column Adaptation Type in Table I) for the adaptation and variation data sets. The frequency distribution is consistent in most adaptation types between the two sets.

In both sets, the most frequent adaptation type is *change a method call* in the logic customization category. Other logic change types in the same category also occur frequently. This is because SO examples are often designed for illustration purposes with contrived usage scenarios and input data, thus requiring further logic customization. *Rename* is the second most common adaptation type. It is frequently performed to make variable and method names more readable for the specific context of a GitHub counterpart. 35% and 14% of SO examples in the variation dataset and the adaptation dataset respectively include undefined variables or local method calls, leading to compilation errors. The majority of these compilation errors (60% and 61% respectively) could be resolved by simply removing the statements using these undefined variables or method calls. 34% and 22% of SO examples in the two datasets respectively include new conditionals (e.g., an `if` check) to handle corner cases or reject invalid input data.

To understand whether the same type of adaptations appears repetitively on the same SO example, we count the number of adaptation types shared by different GitHub counterparts. Multiple clones of the same SO example share at least one same adaptation type in the 70% of the adaptation dataset and 74% of the variation dataset. In other words, *the same type of adaptations is recurring among different GitHub counterparts*.

V. Tool Support and Implementation

Motivated by the finding in Section IV that prior adaptations and variations of a given SO example often share the same recurring adaptation type, we build a Chrome extension called ExampleStack, which visualizes similar GitHub code fragments alongside a SO code example and allows a user to explore variations of a given SO example in an adaptation-type aware code template.

A. ExampleStack Tool Features

This section describes the tool features of ExampleStack. Suppose Alice is new to Android and she wants to read some `json` data from the asset folder of her Android application. Alice finds a SO code example [41] that reads geometric data from a specific file, `locations.json` (① in Figure 4). ExampleStack helps Alice by detecting other similar snippets in real-world Android projects and by visualizing the hot spots where adaptations (/variations) occur.

Browse GitHub counterparts with differences. Given the SO example, ExampleStack displays five similar GitHub snippets and highlights their variations to the SO example (③ in Figure 4). It also surfaces the GitHub link and reputation metrics of the GitHub repository, including the number of stars, contributors, and watches (④ in Figure 4). By default, it ranks GitHub counterparts by the number of stars.

View hot spots with code options. ExampleStack lifts a code template to illuminate the unchanged code parts, while abstracting modified code as *hot spots* to be filled in (②

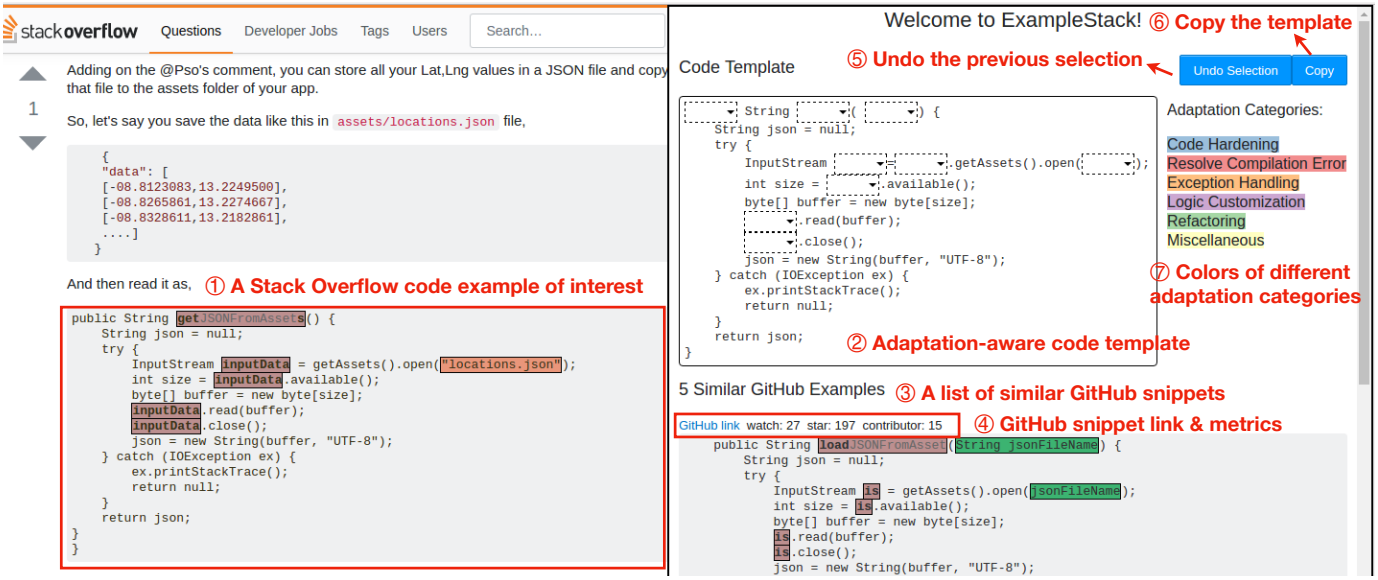


Fig. 4: In the lifted template, common unchanged code is retained, while adapted regions are abstracted with *hot spots*.

in Figure 4). The lifted template provides a bird's-eye view and also serves as a navigation model to explore a variety of code options used to customize the code example. As shown in Figure 5, Alice can click on each hot spot and view the code options along with their frequencies in a drop-down menu. Code options are highlighted in six distinct colors according to their underlying adaptation intent (⑦ in Figure 4). For example, the second drop-down menu in Figure 5 indicates that two GitHub snippets replace `locations.json` to `languages.json` to read the language asset resources for supporting multiple languages. This variation is represented as *update a constant value* in the *logic customization* category in our taxonomy in Section III-B.

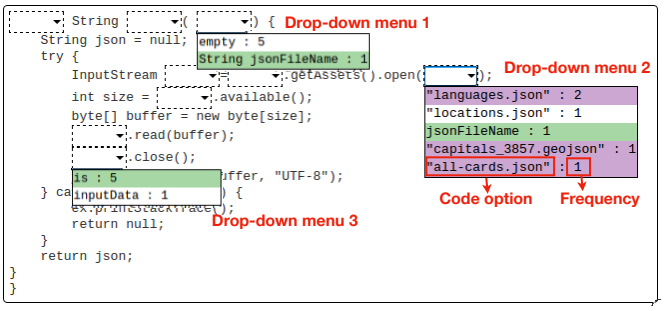


Fig. 5: Alice can click on a hot spot and view potential code options colored based on their underlying adaptation type.

Fill in hot spots with auto-selection. Instead of hard-coding the asset file name, Alice wants to make her program more general—being able to read asset files with any given file name. Therefore, Alice selects the code option, `jsonFileName`, in the second drop-down menu in Figure 5, which generalizes the hardcoded file name to a variable. EXAMPLESTACK automatically selects another code option, `String jsonFileName`, in the first drop-down menu in Figure 5, since this code option declares the `jsonFileName` variable as the method parameter. This auto-selection feature is enabled by *def-use* analysis, which correlates code options based on the definitions and uses of variables (Section V-B). By automatically relating code options in a template, Alice does not have to manually click through multiple drop-down menus to figure out how to avoid compilation errors. Figure 6 shows the customized template based on the selected `jsonFileName` option. The list of GitHub counterparts and the frequencies of other code options are also updated accordingly based on user selection. Alice can undo the previous selection (⑤ in Figure 4) or copy the customized template to her clipboard (⑥ in Figure 4).

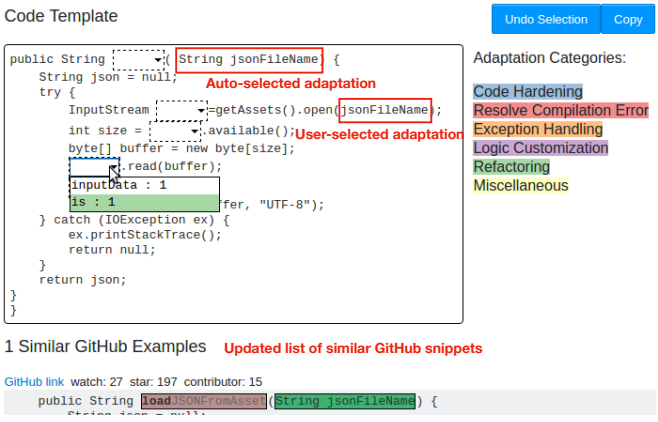


Fig. 6: EXAMPLESTACK automatically updates the code template based on user selection.

B. Template Construction

Diff generating and pruning. To lift an adaptation-aware code template of a SO code example, EXAMPLESTACK first computes the AST differences between the SO example and each GitHub counterpart using GumTree [13]. EXAMPLESTACK prunes the edit operations by filtering out *inner* operations that modify the children of other modified nodes. For example, if an insert operation inserts a tree node whose parent is also inserted by another insert, the first inner insert will be removed, since its adaptation is entailed by the second outer insert. Based on the resulting tree edits, EXAMPLESTACK keeps track of the change regions in the SO example and how each region is changed.

Diff grouping. EXAMPLESTACK groups change regions to decide where to place hot spots in a SO example and what code options to display in a hot spot. If two change regions are the same, they are grouped together. If two change regions overlap, EXAMPLESTACK merges the overlapping change locations into a bigger region enclosing both and groups them together. For example, consider a diff that changes `a=b` to `a=b+c`, and another diff that completely changes `a=b` to `o.foo()`. Simply abstracting the changed code in these two diffs without any alignment will overlay two hot spots in the template, `a=b` and the smaller diff is shadowed by the bigger diff in visualization. EXAMPLESTACK avoids this conflict by recalibrating the first change region from `a=b` to `a=b`.

Option generating and highlighting. For each group of change regions, EXAMPLESTACK replaces the corresponding location in the SO example with a hot spot and attaches a drop-down menu. EXAMPLESTACK displays both the original content in the SO example and contents of the matched GitHub snippet regions as options in each drop-down menu. EXAMPLESTACK then uses the adaptation categorization technique to detect the underlying adaptation types of code options. We use six distinct background colors to illuminate the categories in Table I, which makes it easier for developers to recognize different intent. The color scheme is generated using ColorBrewer [42] to ensure the primary visual differences between different categories in the template.

EXAMPLESTACK successfully lift code templates in all 14,124 SO examples. On average, a lifted template has 81 lines of code (median 41) with 13 hot spots (median 12) to fill in. On average, 4 code options (median 2) is displayed in the drop-down menu of each hot spot.

VI. USER STUDY

We conducted a within-subjects user study with twelve Java programmers to investigate the usefulness of viewing similar GitHub code alongside SO examples using EXAMPLESTACK. Ten participants were graduate students and two were undergraduate students from the Computer Science department in a research university. Seven participants had two to five years of Java programming experience, while the other five were novice Java programmers with one-year experience, showing a good mix of different levels of programming experience. All participants reported to use Stack Overflow for programming.

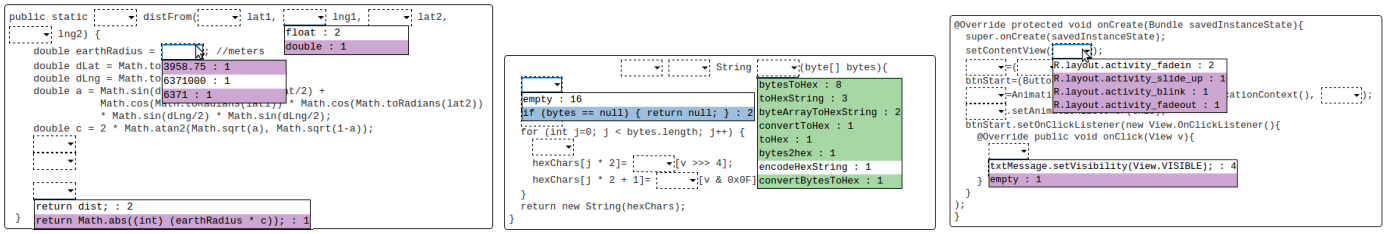
In each study session, we first gave a fifteen-minute tutorial of EXAMPLESTACK. Participants then did two code reuse tasks with and without EXAMPLESTACK. We stopped each task after fifteen minutes. Table II describes four code reuse tasks. In each task, we asked participants to annotate which parts of a SO example they would like to change and how they would change. Figure 7 shows the code templates generated by EXAMPLESTACK, not including the one in Task II due to its length (79 lines). When not using EXAMPLESTACK, participants were allowed to search online for other code examples and resources, which is commonly done in real-world programming workflow [3]. The order of assigned conditions and tasks were counterbalanced across participants through random assignment to mitigate the learning effect. We conducted a post survey to solicit feedback at the end.

TABLE II: Code reuse tasks and study results

ID	Desired Functions and SO Examples	LOC	Clone#	Adaptation#		Category#	
				w/o	with	w/o	with
I	Calculate the geographic distance between two GPS coordinates [43]	12	2	4.3	8.3	2.3	3.0
II	get the relative path between two files [46]	74	2	4.0	6.3	1.7	2.3
III	encode a byte array to a hex string [44]	12	17	3.3	6.0	2.0	3.0
IV	add animation to an Android view [45]	29	4	5.0	5.3	2.3	2.3
			Overall	4.2	6.5	2.1	2.7

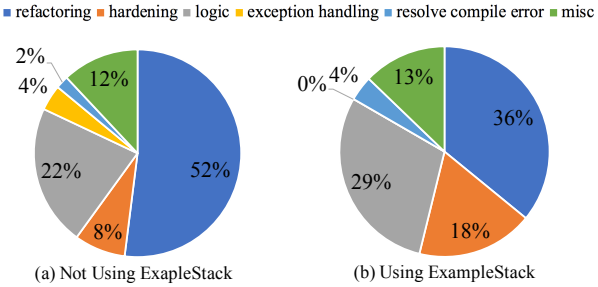
Overall, participants using EXAMPLESTACK grasped a more comprehensive view of adapting SO examples by giving more adaptation suggestions. Column Adaptation# in Table II shows that participants annotated 55% more code locations to be adapted in a SO code example with EXAMPLESTACK. The mean difference of 2.3 adaptations is statistically significant (paired t-test: t=3.56, df=11, p-value=0.004). Column Category# shows that participants also suggested more diverse types of adaptations (2.7 vs. 2.1) using EXAMPLESTACK (paired t-test: t=2.60, df=11, p-value=0.02). Figure 8 shows the total number of suggested adaptations in each category. When using EXAMPLESTACK, participants suggested more adaptations in the categories of code hardening (14 vs. 4) and logic customization (23 vs. 11) and accounted for different ways of using APIs and how to handle edge cases. Without EXAMPLESTACK, participants sometimes gave wrong or unnecessary adaptation suggestions. For example, P4 mistakenly tagged the name of an inherited method, `onCreate` to be changeable in Task IV, which broke the UI initialization flow in Android. Seeing how other developers adapted a code example could prevent users from modifying critical pieces of code that were not supposed to be modified.

How do you like or dislike viewing similar GitHub code alongside a SO example? In the post survey, all participants found being able to see similar GitHub code very useful for three main reasons. First, viewing the commonality among similar code examples helped users quickly understand the essence of a code example. P6 described this as "*the fast path to reach consensus on a particular operation.*" Second, the GitHub variants reminded users some points they may otherwise miss. P2 wrote, "*I learned different ways to implement something and understand the nuance better by seeing what*



(a) Compute distance between two coordinates [43] (b) Encode byte array to a hex string [44] (c) Add animation to an Android view [45]

Fig. 7: Code Template Examples



(a) Not Using ExapleStack (b) Using ExampleStack

Fig. 8: Annotated Adaptations in different categories

is and is not similar." Third, participants felt more confident of a SO code example after seeing how similar code was used in GitHub repositories. P9 stated that, "*[it is] reassuring to know that the same code is used in production systems and to know the common pitfalls.*"

How do you like or dislike interacting with a code template? Participants liked the code template, since it showed the essence of a code example and made it easier to see subtle changes, especially in lengthy code examples. Participants also found displaying the frequency count of different adaptations very useful. P5 explained, "*it highlights the best practices followed by the community and also prioritizes the changes that I should make first.*" However, we also observed that, when there were only a few GitHub counterparts, some participants inspected individual GitHub counterparts directly rather than interacting with the code template.

How do you like or dislike color-coding different adaptation types? Though the majority of participants confirmed the usefulness of color-coding adaptation types, four participants felt confused by the many colors used to distinguish different types of adaptations. They suggested to make the color scheme customizable, since it was difficult to remember these colors during navigation. Two of them considered some adaptation types (e.g., renaming) trivial and suggested to allow users to hide adaptation types of no interest to avoid distraction.

When would you use* EXAMPLESTACK*? Five participants mentioned that EXAMPLESTACK would be most useful when a SO code example was long and hard to understand. P8 said, "*I would use this tool when I am not sure of the credibility of the SO code.*" P4 wrote, "*the tool is very useful when the code is longer and hard to spot what to change at a glance.*" Four participants would like to use EXAMPLESTACK when learning APIs, since EXAMPLESTACK provided multiple code examples that use the same API in different contexts. Two participants would like to use EXAMPLESTACK to identify missing points and assess different solutions, especially when writing a large-scale robust project.

How would you like to improve our tool? In addition to making the color scheme customizable and hiding certain adaptation types, multiple participants suggested a meta-voting feature that allows users to rate the usefulness of individual GitHub examples and adaptations in EXAMPLESTACK.

VII. DISCUSSION

Threats to Validity. Despite code similarity and the timestamp order, SO examples and GitHub counterparts in the variation dataset could be coincidentally linked, e.g., the common code from official documentation. Though it does not weaken the proposal of guiding developers in identifying commonality and hot spots of variations between similar code, such coincidental linking could skew the analysis result from the variation dataset. In the dataset construction, identifying which SO example is attributed only based on SO links in a GitHub file may produce mismatches, if there are multiple similar examples in an answer post or a discussion thread. Clone detection might introduce false positive clones that are only structurally similar. We have made our best effort in conducting a comprehensive and unbiased dataset via extra quality control mechanisms such as time order checking, manual inspection, and cross-validation.

EXAMPLESTACK shows one possible instantiation of browser-based tool support but we do not argue that it is the only tool nor the best tool. Others can also design IDE-based tool support such as automating common adaptations discovered in this work and compare with existing code integration techniques [14, 16]. Investigating alternative tool designs and how such a tool fits the developer workflow can be future work. Since the user study is designed to understand whether developers can grasp a comprehensive view of adapting a SO example using EXAMPLESTACK, we do not ask users to perform code integration to avoid confounding factors such as the familiarity of IDEs and the complexity of target programs. Instead, we ask users to annotate where and how they would like to change a SO example and encourage them to think

about possible adaptations in different usage cases. Therefore, our findings do not imply code integration effectiveness.

Future Work. Existing code integration techniques can automatically rename variables and port related program statements based on the context of a target program during code reuse [14]–[16]. This paper highlights opportunities for extending such techniques. For example, commonalities shown in type conversion may suggest which type to convert which. Furthermore, developers extensively refactor and add code comments to improve the readability and maintainability of the adapted code. Therefore, automated name synthesis and comment generation could be beneficial. EXAMPLESTACK currently uses two semantic relationships to relate code options in different drop-down menus and to guide auto-selection. We plan to leverage other semantic relationships to enrich auto-selected options. For example, we can use mined API usage patterns to relate code options of API methods that are used together, such as `lock` and `unlock`.

VIII. RELATED WORK

Quality assessment of SO examples. Our work is inspired by previous studies that find SO examples are incomplete and inadequate [7]–[9, 12, 24, 25, 29]. Subramanian and Holmes find that the majority of SO snippets are free standing statements with no class or method headers [24]. Zhou et al. find that 86 of 200 accepted SO posts use deprecated APIs but only 3 of them are reported by other programmers [9]. Fischer et al. find that 29% security-related code in SO is insecure and could potentially be copied to 1 million Android apps [8]. Treude and Robillard conduct a survey to investigate comprehension difficulty of code examples in SO [10]. The responses from GitHub users indicate that less than half of the SO examples are self-explanatory and one main issue is code incompleteness. Zhang et al. constrast SO examples with API usage patterns mined from GitHub and detect potential API misuse in 31% SO posts [7]. These findings motivate our investigation of adaptations and variations of SO examples.

Stack Overflow usage and attribution. Our work is motivated by the finding that developers often resort to online Q&A forums such as Stack Overflow [3]–[6]. Despite the wide usage of SO, most developers are not aware of the SO licensing terms nor attribute to the code reused from SO [5, 6, 12]. Only 1.8% of GitHub repositories containing code from SO follow the licensing policy properly [5]. Almost one half developers admit copying code from SO without attribution and two thirds are not aware of the SO licensing implications. Based on these findings, we carefully construct a comprehensive dataset of reused code, including both explicitly attributed SO examples and potentially reused ones using clone detection, timestamp analysis, and URL references. Origin analysis can be potentially applied to match SO snippets with GitHub files [47]–[50]. Unlike clone detection, origin analysis leverages the LCS similarity between program entities such as variable names and expressions as well as call relations to identify the origin of a piece of code during software evolution.

SO snippet retrieval and code integration. Previous tool support for reusing code from SO mostly focuses on helping developers locate relevant posts or snippets from the IDE [16, 51]–[53]. For example, Prompter retrieves related SO discussions based on the program context in Eclipse. SnipMatch supports light-weight code integration by renaming variables in a SO snippet based on corresponding variables in a target program [16]. Code correspondence techniques [14, 54] match code elements (e.g., variables, methods) to decide which code to copy, rename, or delete during copying and pasting. Our work differs by focusing on analysis of common adaptations and variations of SO examples.

Change types and taxonomy. There is a large body of literature for source code changes during software evolution [55]–[57]. Fluri et al. present a fine-grained taxonomy of source code changes such as changing the return type and renaming a field, based on differences in abstract syntax trees [19]. Kim et al. analyze changes on "micro patterns" [58] in Java using software evolution data [18]. These studies investigate general change types in software evolution, while we quantify common adaptation and variation types using SO and GitHub code.

Program differencing and change template. Diff tools compute program differences between two programs [13, 59]–[62]. However, they do not support analysis of one example with respect to multiple counterparts simultaneously. Lin et al. align multiple programs and visualize their variations [63]. However, they do not lift a code template to summarize the commonality and variations between similar code. Several techniques construct code templates for the purpose of code search [64] or code transformation [15]. EXAMPLESTACK differs from these by identifying GitHub snippets similar to SO examples, annotating a lifted template with common adaptation types, and visualizing concrete variations in *hot spots*. Glassman et al. design an interactive visualization called EXAMPLORE to help developers comprehend hundreds of similar but different API usages in GitHub [65]. Given an API method of interest, EXAMPLORE instantiates a pre-defined API usage skeleton and fills in details such as varying guard conditions and succeeding API calls. EXAMPLESTACK is not limited to API usage and does not require a pre-defined skeleton.

IX. CONCLUSION

This paper reports common adaptations and variations of online code examples along with their types and frequencies using large-scale Stack Overflow and GitHub repository data. We construct a taxonomy of 24 common adaptation types through manual inspection and further develop an automated technique to categorize AST edits to these adaptation categories. Our automated adaptation categorization technique has 98% precision and 96% recall on the manually validated ground truth. By applying this automated analysis to the carefully constructed datasets of *adaptations* and *variations*, we find that the same type of adaptation appears repetitively between SO examples and GitHub counterparts, implying that developers should benefit from being informed of previous adaptations and variations. We design and implement a

Chrome extension called EXAMPLESTACK to render the commonality and differences between SO examples and similar GitHub snippets by lifting an adaptation-aware code template. User study participants find that crowd-sourced insights from prior adaptations and variations help improve confidence about how to reuse online code examples.

REFERENCES

[1] M. Umarji, S. E. Sim, and C. Lopes, "Archetypal internet-scale source code searching," in *IFIP International Conference on Open Source Systems*. Springer, 2008, pp. 257–263.

[2] R. E. Gallardo-Valencia and S. Elliott Sim, "Internet-scale code search," in *Proceedings of the 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*. IEEE Computer Society, 2009, pp. 49–52.

[3] J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer, "Two studies of opportunistic programming: interleaving web foraging, learning, and writing code," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2009, pp. 1589–1598.

[4] C. Sadowski, K. T. Stolee, and S. Elbaum, "How developers search for code: a case study," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 191–201.

[5] S. Baltes and S. Diehl, "Usage and attribution of stack overflow code snippets in github projects," *arXiv preprint arXiv:1802.02938*, 2018.

[6] Y. Wu, S. Wang, C.-P. Bezemer, and K. Inoue, "How do developers utilize source code from stack overflow?" *Empirical Software Engineering*, pp. 1–37, 2018.

[7] T. Zhang, G. Upadhyaya, A. Reinhardt, H. Rajan, and M. Kim, "Are code examples on an online q&a forum reliable?: a study of api misuse on stack overflow," in *Proceedings of the 40th International Conference on Software Engineering*. ACM, 2018, pp. 886–896.

[8] F. Fischer, K. Böttinger, H. Xiao, C. Stransky, Y. Acar, M. Backes, and S. Fahl, "Stack overflow considered harmful? the impact of copy&paste on android application security," in *Security and Privacy (SP), 2017 IEEE Symposium on*. IEEE, 2017, pp. 121–136.

[9] J. Zhou and R. J. Walker, "Api deprecation: a retrospective analysis and detection method for code examples on the web," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 266–277.

[10] C. Treude and M. P. Robillard, "Understanding stack overflow code fragments," in *Proceedings of the 33rd International Conference on Software Maintenance and Evolution*. IEEE, 2017.

[11] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: Scaling code clone detection to big-code," in *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. IEEE, 2016, pp. 1157–1168.

[12] L. An, O. Mlouki, F. Khomh, and G. Antoniol, "Stack overflow: a code laundering platform?" in *Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on*. IEEE, 2017, pp. 283–293.

[13] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 2014, pp. 313–324.

[14] R. Cottrell, R. J. Walker, and J. Denzinger, "Semi-automating small-scale source code reuse via structural correspondence," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. ACM, 2008, pp. 214–225.

[15] N. Meng, M. Kim, and K. S. McKinley, "Lase: locating and applying systematic edits by learning from examples," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 502–511.

[16] D. Wightman, Z. Ye, J. Brandt, and R. Vertegaal, "Snipmatch: Using source code context to enhance snippet retrieval and parameterization," in *Proceedings of the 25th annual ACM symposium on User interface software and technology*. ACM, 2012, pp. 219–228.

[17] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 2000.

[18] S. Kim, K. Pan, and E. J. Whitehead Jr, "Micro pattern evolution," in *Proceedings of the 2006 international workshop on Mining software repositories*. ACM, 2006, pp. 40–46.

[19] B. Fluri and H. C. Gall, "Classifying change types for qualifying change couplings," in *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 35–45.

[20] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The promises and perils of mining github," in *Proceedings of the 11th working conference on mining software repositories*. ACM, 2014, pp. 92–101.

[21] C. V. Lopes, P. Maj, P. Martins, V. Saini, D. Yang, J. Zitny, H. Sajnani, and J. Vitek, "Déjàvu: a map of code duplicates on github," *Proceedings of the ACM on Programming Languages*, vol. 1, p. 28, 2017.

[22] G. Gousios and D. Spinellis, "Ghtorrent: Github's data from a firehose," in *Mining software repositories (msr), 2012 9th ieee working conference on*. IEEE, 2012, pp. 12–21.

[23] *Stack Overflow data dump*, 2016, https://archive.org/details/stackexchange, accessed on Oct 17, 2016.

[24] S. Subramanian and R. Holmes, "Making sense of online code snippets," in *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press, 2013, pp. 85–88.

[25] D. Yang, A. Hussain, and C. V. Lopes, "From query to usable code: an analysis of stack overflow code snippets," in *Proceedings of the 13th International Workshop on Mining Software Repositories*. ACM, 2016, pp. 391–402.

[26] S. Subramanian, L. Inozemtseva, and R. Holmes, "Live api documentation," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 643–652.

[27] D. Yang, P. Martins, V. Saini, and C. Lopes, "Stack overflow in github: any snippets there?" in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 280–290.

[28] E. Torlak and S. Chandra, "Effective interprocedural resource leak detection," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 2010, pp. 535–544.

[29] B. Dagenais and M. P. Robillard, "Recovering traceability links between an api and its learning resources," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 47–57.

[30] *Get OS-level system information*, 2008, https://stackoverflow.com/questions/61727.

[31] *A GitHub clone that gets the CPU usage*, 2014, https://github.com/jomis/nomads/blob/master/nomads-framework/src/main/java/at/ac/tuwien/dsg/utilities/PerformanceMonitor.java\#L44-L63.

[32] *Adding new paths for native libraries at runtime in Java*, 2013, https://stackoverflow.com/questions/15409446.

[33] *A GitHub clone that adds new paths for native libraries at runtime in Java*, 2014, https://github.com/armint/firesight-java/blob/master/src/main/java/org/firepick/firesight/utils/SharedLibLoader.java\#L131-L153.

[34] *How to get IP address of the device from code?*, 2012, https://stackoverflow.com/questions/7899226.

[35] *JSlider question: Position after leftclick*, 2009, https://stackoverflow.com/questions/518672.

[36] *A GitHub clone about JSlide*, 2014, https://github.com/changkon/Pluripartite/tree/master/src/se206/a03/MediaPanel.java\#L329-L339.

[37] *Another GitHub clone about JSlide*, 2014, https://github.com/changkon/Pluripartite/tree/master/src/se206/a03/MediaPanel.java\#L343-L353.

[38] *A GitHub clone that downloads videos from YouTube*, 2014, https://github.com/instance01/YoutubeDownloaderScript/blob/master/IYoutubeDownloader.java\#L148-L193.

[39] *Youtube data API : Get access to media stream and play (JAVA)*, 2011, https://stackoverflow.com/questions/4834369.

[40] S. M. Nasehi, J. Sillito, F. Maurer, and C. Burns, "What makes a good code example?: A study of programming q&a in stackoverflow," in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. IEEE, 2012, pp. 25–34.

[41] *Add Lat and Long to ArrayList*, 2016, https://stackoverflow.com/questions/37273871.

[42] *ColorBrewer: Color Advice for Maps*, 2018, http://colorbrewer2.org.

[43] *Calculate distance in meters when you know longitude and latitude in java*, 2017, https://stackoverflow.com/questions/837957.

[44] *How to use SHA-256 with Android*, 2014, https://stackoverflow.com/questions/25803281.

[45] *How can I add animations to existing UI components?*, 2015, https://stackoverflow.com/questions/33464536.

[46] *Construct a relative path in Java from two absolute paths*, 2015, https://stackoverflow.com/questions/3054692.

[47] Q. Tu and M. W. Godfrey, "An integrated approach for studying architectural evolution," in *Program Comprehension, 2002. Proceedings. 10th International Workshop on*. IEEE, 2002, pp. 127–136.

[48] M. Godfrey and Q. Tu, "Tracking structural evolution using origin analysis," in *Proceedings of the international workshop on Principles of software evolution*. ACM, 2002, pp. 117–119.

[49] L. Zou and M. W. Godfrey, "Detecting merging and splitting using origin analysis," in *null*. IEEE, 2003, p. 146.

[50] M. W. Godfrey and L. Zou, "Using origin analysis to detect merging and splitting of source code entities," *IEEE Transactions on Software Engineering*, vol. 31, no. 2, pp. 166–181, 2005.

[51] A. Bacchelli, L. Ponzanelli, and M. Lanza, "Harnessing stack overflow for the ide," in *Proceedings of the Third International Workshop on Recommendation Systems for Software Engineering*. IEEE Press, 2012, pp. 26–30.

[52] L. Ponzanelli, A. Bacchelli, and M. Lanza, "Seahawk: Stack overflow in the ide," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 1295–1298.

[53] L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, and M. Lanza, "Mining stackoverflow to turn the ide into a self-confident programming prompter," in *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 2014, pp. 102–111.

[54] R. Holmes and R. J. Walker, "Supporting the investigation and planning of pragmatic reuse tasks," in *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 2007, pp. 447–457.

[55] S. Kim, K. Pan, and E. E. J. Whitehead, Jr., "Memories of bug fixes," in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. SIGSOFT '06/FSE-14. New York, NY, USA: ACM, 2006, pp. 35–45. [Online]. Available: http://doi.acm.org/10.1145/1181775.1181781

[56] M. Fischer, J. Oberleitner, J. Ratzinger, and H. Gall, "Mining evolution data of a product family," in *MSR '05: Proceedings of the 2005 International Workshop on Mining Software Repositories*. New York, NY, USA: ACM, 2005, pp. 1–5.

[57] D. Dig and R. Johnson, "How do APIs evolve? a story of refactoring," *Journal of software maintenance and evolution: Research and Practice*, vol. 18, no. 2, pp. 83–107, 2006.

[58] J. Y. Gil and I. Maman, "Micro patterns in java code," *ACM SIGPLAN Notices*, vol. 40, no. 10, pp. 97–116, 2005.

[59] W. Miller and E. W. Myers, "A file comparison program," *Software: Practice and Experience*, vol. 15, no. 11, pp. 1025–1040, 1985.

[60] W. Yang, "Identifying syntactic differences between two programs," *Software – Practice & Experience*, vol. 21, no. 7, pp. 739–755, 1991. [Online]. Available: citeseer.ist.psu.edu/yang91identifying.html

[61] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom, "Change detection in hierarchically structured information," in *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA: ACM, 1996, pp. 493–504.

[62] B. Fluri, M. Wuersch, M. PInzger, and H. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *IEEE Transactions on software engineering*, vol. 33, no. 11, 2007.

[63] Y. Lin, Z. Xing, Y. Xue, Y. Liu, X. Peng, J. Sun, and W. Zhao, "Detecting differences across multiple instances of code clones." in *ICSE*, 2014, pp. 164–174.

[64] T. Zhang, M. Song, J. Pinedo, and M. Kim, "Interactive code review for systematic changes," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 111–122.

[65] E. L. Glassman, T. Zhang, B. Hartmann, and M. Kim, "Visualizing api usage examples at scale," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. ACM, 2018, p. 580.