Learning Natural Coding Conventions

Miltiadis Allamanis[†] Earl T. Barr[‡] Christian Bird[⋆] Charles Sutton[†]

[†]School of Informatics
University of Edinburgh
Edinburgh, EH8 9AB, UK
{m.allamanis, csutton}@ed.ac.uk

[‡]Dept. of Computer Science
University College London
London, UK
e.barr@ucl.ac.uk

[⋆]Microsoft Research
Microsoft
Redmond, WA, USA
cbird@microsoft.com

ABSTRACT

Every programmer has a characteristic style, ranging from preferences about identifier naming to preferences about object relationships and design patterns. Coding conventions define a consistent syntactic style, fostering readability and hence maintainability. When collaborating, programmers strive to obey a project's coding conventions. However, one third of reviews of changes contain feedback about coding conventions, indicating that programmers do not always follow them and that project members care deeply about adherence. Unfortunately, programmers are often unaware of coding conventions because inferring them requires a global view, one that aggregates the many local decisions programmers make and identifies emergent consensus on style. We present NATURALIZE, a framework that learns the style of a codebase, and suggests revisions to improve stylistic consistency. NATURALIZE builds on recent work in applying statistical natural language processing to source code. We apply NATURALIZE to suggest natural identifier names and formatting conventions. We present four tools focused on ensuring natural code during development and release management, including code review. NATURALIZE achieves 94% accuracy in its top suggestions for identifier names. We used NATURALIZE to generate 18 patches for 5 open source projects: 14 were accepted.

Categories and Subject Descriptors:
D.2.3 [Software Engineering]: Coding Tools and Techniques
General Terms: Algorithms
Keywords: Coding conventions, naturalness of software

1. INTRODUCTION

To program is to make a series of choices, ranging from design decisions — like how to decompose a problem into functions — to the choice of identifier names and how to format the code. While local and syntactic, the latter are important: names connect program source to its problem domain [13, 43, 44, 68]; formatting decisions usually capture control flow [36]. Together, naming and formatting decisions determine the readability of a program's source code, increasing a codebase's portability, its accessibility to newcomers, its reliability, and its maintainability [55, §1.1]. Apple's recent, infamous bug in its handling of SSL certificates [7, 40] exemplifies the impact that formatting can have on reliability. Maintainability is especially important since developers spend the majority (80%) of their time maintaining code [2, §6].

A convention is "an equilibrium that everyone expects in interactions that have more than one equilibrium" [74]. For us, coding conventions arise out of the collision of the stylistic choices of programmers. A *coding convention* is a syntactic restriction not imposed by a programming language's grammar. Nonetheless, these choices are important enough that they are enforced by software teams. Indeed, our investigations indicate that developers enforce such coding conventions rigorously, with roughly one third of code reviews containing feedback about following them (subsection 4.1).

Like the rules of society at large, coding conventions fall into two broad categories: *laws*, explicitly stated and enforced rules, and *mores*, unspoken common practice that emerges spontaneously. Mores pose a particular challenge: because they arise spontaneously from emergent consensus, they are inherently difficult to codify into a fixed set of rules, so rule-based formatters cannot enforce them, and even programmers themselves have difficulty adhering to all of the implicit mores of a codebase. Furthermore, popular code changes constantly, and these changes necessarily embody stylistic decisions, sometimes generating new conventions and sometimes changing existing ones. To address this, we introduce the *coding convention inference problem*, the problem of automatically learning the coding conventions consistently used in a body of source code. Conventions are pervasive in software, ranging from preferences about identifier names to preferences about class layout, object relationships, and design patterns. In this paper, we focus as a first step on local, syntactic conventions, namely, identifier naming and formatting. These are particularly active topics of concern among developers, for example, almost *one quarter* of the code reviews that we examined contained suggestions about naming.

We introduce NATURALIZE, a framework that solves the coding convention inference problem for local conventions, offering suggestions to increase the stylistic consistency of a codebase. NATURALIZE can also be applied to infer rules for existing rule-based formatters. NATURALIZE is descriptive, not prescriptive[1]: it learns what programmers actually do. When a codebase does not reflect consensus on a convention, NATURALIZE recommends nothing, because it has not learned anything with sufficient confidence to make recommendations. The naturalness insight of Hindle *et al.* [35], building on Gabel and Su [28], is that most short code utterances, like natural language utterances, are simple and repetitive. Large corpus statistical inference can discover and exploit this naturalness to improve developer productivity and code robustness. We show that coding conventions are *natural* in this sense.

[1]Prescriptivism is the attempt to specify rules for correct style in language, *e.g.*, Strunk and White [67]. Modern linguists studiously avoid prescriptivist accounts, observing that many such rules are routinely violated by noted writers.

FSE'14, November 16–21, 2014, Hong Kong, China
ACM 978-1-4503-3056-5/14/11
http://dx.doi.org/10.1145/2635868.2635883

Learning from local context allows NATURALIZE to learn syntactic restrictions, or sub-grammars, on identifier names like camelcase or underscore, and to *unify* names used in similar contexts, which rule-based code formatters simply cannot do. Intuitively, NATURALIZE works by identifying identifier names or formatting choices that are surprising according to a probability distribution over code text. When surprised, NATURALIZE determines if it is sufficiently confident to suggest a renaming or reformatting that is less surprising; it unifies the surprising choice with one that is preferred in similar contexts elsewhere in its training set. NATURALIZE is *not* automatic; it assists a developer, since its suggestions, both renaming and even formatting, as in Python or Apple's aforementioned SSL bug [7, 40], are potentially semantically disruptive and must be considered and approved. NATURALIZE's suggestions enable a range of new tools to improve developer productivity and code quality: 1) A pre-commit script that rejects commits that excessively disrupt a codebase's conventions; 2) A tool that converts the inferred conventions into rules for use by a code formatter; 3) An Eclipse plugin that a developer can use to check whether her changes are unconventional; and 4) A style profiler that highlights the stylistic inconsistencies of a code snippet for a code reviewer.

NATURALIZE draws upon a rich body of tools from statistical natural language processing (NLP), but applies these techniques to a different kind of problem. NLP focuses on *understanding* and *generating* language, but does not ordinarily consider the problem of improving existing text. The closest analog is spelling correction, but that problem is easier because we have strong prior knowledge about common types of spelling mistakes. An important conceptual dimension of our suggestion problems also sets our work apart from mainstream NLP. In code, rare names often usefully signify unusual functionality, and need to be preserved. We call this the *sympathetic uniqueness principle* (SUP): unusual names should be preserved when they appear in unusual contexts. We achieve this by exploiting a special token UNK that is often used to represent rare words that do not appear in the training set. Our method incorporates SUP through a clean, straightforward modification to the handling of UNK. Because of the Zipfian nature of language, UNK appears in unusual contexts and identifies unusual tokens that should be preserved. section 4 demonstrates the effectiveness of this method at preserving such names. Additionally, handling formatting requires a simple, but novel, method of encoding formatting.

As NATURALIZE detects identifiers that violate code conventions and assists in renaming, the most common refactoring [50], it is the first tool we are aware of that uses NLP techniques to aid refactoring.

The techniques that underlie NATURALIZE are language independent and require only identifying identifiers, keywords, and operators, a much easier task than specifying grammatical structure. Thus, NATURALIZE is well-positioned to be useful for domain-specific or esoteric languages for which no convention enforcing tools exist or the increasing number of multi-language software projects such as web applications that intermix Java, `css`, `html`, and JavaScript.

To the best of the authors' knowledge, this work is the first to address the coding convention inference problem, to suggest names and formatting to increase the stylistic coherence of code, and to provide tooling to that end. Our contributions are:

- We built NATURALIZE, the first framework to solve the *coding convention inference problem* for local conventions, including identifier naming and formatting, and suggests changes to increase a codebase's adherence to its own conventions;
- We offer four tools, built on NATURALIZE, all focused on release management, an under-tooled phase of the development process.
- NATURALIZE 1) achieves 94% accuracy in its top suggestions for identifier names and 2) never drops below a mean accuracy of 96% when making formatting suggestions; and
- We demonstrate that coding conventions are *important* to software teams, by showing that 1) empirically, programmers enforce conventions heavily through code review feedback and corrective commits, and 2) patches that were based on NATURALIZE suggestions have been incorporated into 5 of the most popular open source Java projects on GitHub — of the 18 patches that we submitted, 14 were accepted.

Tools are available at `groups.inf.ed.ac.uk/naturalize`.

2. MOTIVATING EXAMPLE

Both industrial and open source developers often submit their code for review prior to check-in [61]. Consider the example of the class shown in Figure 1 which is part of a change submitted for review by a Microsoft developer on February 17th, 2014. While there is nothing functionally wrong with the class, it violates the coding conventions of the team. A second developer reviewed the change and suggested that `res` and `str` do not convey parameter meaning well enough, the constructor line is much too long and should be wrapped. In the checked in change, all of these were addressed, with the parameter names changed to `queryResults` and `queryStrings`.

Consider a scenario in which the author had access to NATURALIZE. The author might highlight the parameter names and ask NATURALIZE to evaluate them. At that point it would have not only have identified `res` and `str` as names that are inconsistent with the naming conventions of parameters in the codebase, but would also have suggested better names. The author may have also thought to himself "Is the constructor on line 3 too long?" or "Should the empty constructor body be on it's own line and should it have a space inside?" Here again, NATURALIZE would have provided immediate, valuable answers based on the the conventions of the team. NATURALIZE would indicate that the call to the base constructor should be moved to the next line and indented to be consonant with team conventions and that in this codebase empty method bodies do not need their own lines. Furthermore it would indicate that some empty methods contain one space between the braces while others do not, so there is no implicit convention to follow. After querying NATURALIZE about his stylistic choices, the author can then be confident that his change is consistent with the norms of the team and is more likely to be approved during review. Furthermore, by leveraging NATURALIZE, fellow project members wouldn't need to be bothered by questions about conventions, nor would they need to provide feedback about conventions during review. We have observed that such scenarios occur in open source projects as well.

2.1 Use Cases and Tools

Coding conventions are critical during release management, which comprises committing, reviewing, and promoting (including releases) changes, either patches or branches. This is when a coder's idiosyncratic style, isolated in her editor during code composition, comes into contact with the styles of others. The outcome of this interaction strongly impacts the readability, and therefore the maintainability, of a codebase. Compared to other phases of the development cycle like editing, debugging, project management, and issue tracking, release management is under-tooled. Code conventions are particularly pertinent here, and lead us to target three use cases: 1) a developer preparing an individual commit or branch for review or promotion; 2) a release engineer trying to filter out needless stylistic diversity from the flood of changes; and 3) a reviewer wishing to consider how well a patch or branch obeys community norms.

Any code modification has a possibility of introducing bugs [3, 51]. This is certainly true of a system, like NATURALIZE, that is based on statistical inference, even when (as we always assume) all of NATURALIZE's suggestions are approved by a human. Because of

```
public class ExecutionQueryResponse : ExecutionQueryResponseBasic<QueryResults>
{
    public ExecutionQueryResponse(QueryResults res, IReadOnlyCollection<string> str, ExecutionStepMetrics metrics) : base(res, str, metrics) { }
}
```

Figure 1: A C# class added by a Microsoft developer that was modified due to requests by a reviewer before it was checked in.

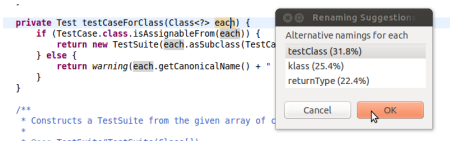


Figure 2: A screenshot of the `devstyle` Eclipse plugin. The user has requested suggestion for alternate names of the `each` argument.

this risk, the gain from making a change must be worth its cost. For this reason, our use cases focus on times when the code is already being changed. To support our use cases, we have built four tools:

`devstyle` A *plugin for Eclipse* IDE that gives suggestions for identifier renaming and formatting both for a single identifier or format point and for the identifiers and formatting in a selection of code.

`styleprofile` A *code review assistant* that produces a profile that summarizes the adherence of a code snippet to the coding conventions of a codebase and suggests renaming and formatting changes to make that snippet more stylistically consistent with a project.

`genrule` A *rule generator* for Eclipse's code formatter that generates rules for those conventions that NATURALIZE has inferred from a codebase.

`stylish?` A high precision *pre-commit script* for `Git` that rejects commits that have highly inconsistent and unnatural naming or formatting within a project.

The `devstyle` plugin offers two types of suggestions, single point suggestion under the mouse pointer and multiple point suggestion via right-clicking a selection. A screenshot from `devstyle` is shown in Figure 2. For single point suggestions, `devstyle` displays a ranked list of alternatives to the selected name or format. If `devstyle` has no suggestions, it simply flashes the current name or selection. If the user wishes, she selects one of the suggestions. If it is an identifier renaming, `devstyle` renames *all* uses, within scope, of that identifier under its previous name. This scope traversal is possible because our use cases assume an existing and compiled codebase. Formatting changes occur at the suggestion point. Multiple point suggestion returns a *style profile*, a ranked list of the top k most stylistically surprising naming or formatting choices in the current selection that could benefit from reconsideration. By default, $k = 5$ based on HCI considerations [23, 48]. To accept a suggestion here, the user must first select a location to modify, then select from among its top alternatives. The `styleprofile` tool outputs a style profile. `genrule` (subsection 3.5) generates settings for the Eclipse code formatter. Finally, `stylish?` is a filter that uses Eclipse code formatter with the settings from `genrule` to accept or reject a commit based on its style profile.

NATURALIZE uses an existing codebase, called a training corpus, as a reference from which to learn conventions. Commonly, the training corpus will be the current codebase, so that NATURALIZE learns domain-specific conventions related to the current project. Alternatively, NATURALIZE comes with a pre-packaged suggestion model, trained on a corpus of popular, vibrant projects that presumably embody good coding conventions. Developers can use this engine if they wish to increase their codebase's adherence to a larger community's consensus on best practice. Projects that are just starting and have little or no code written can also use as the training corpus a pre-existing codebase, for example another project in the same organization, whose conventions the developers wish to adopt. Here, again, we avoid normative comparison of coding conventions, and do not force the user to specify their desired conventions explicitly. Instead, the user specifies a training corpus, and this is used as an *implicit* source of desired conventions. The NATURALIZE framework and tools are available at `groups.inf.ed.ac.uk/naturalize`.

3. THE NATURALIZE FRAMEWORK

In this section, we introduce the generic architecture of NATURALIZE, which can be applied to a wide variety of different types of conventions and is language independent. NATURALIZE is general and can be applied to any language for which a lexer and a parser exist, as token sequences and abstract syntax trees (ASTs) are used during analysis. Figure 3 illustrates its architecture. The input is a code snippet to be naturalized. This snippet is selected based on the user input, in a way that depends on the particular tool in question. For example, in `devstyle`, if a user selects a local variable for renaming, the input snippet would contain all AST nodes that reference that variable (subsection 3.3). The output of NATURALIZE is a short list of suggestions, which can be filtered, then presented to the programmer. In general, a suggestion is a set of snippets that may replace the input snippet. The list is ranked by a *naturalness score* that is defined below. Alternately, the system can return a binary value indicating whether the code is natural, so as to support applications such as `stylish?`. The system makes no suggestion if it deems the input snippet to be sufficiently natural, or is unable to find good alternatives. This reduces the "Clippy effect" where users ignore a system that makes too many bad suggestions[2]. In the next section, we describe each element in the architecture in more detail.

Terminology A language model (LM) is a probability distribution over strings. Given any string $x = x_0, x_1 \ldots x_M$, where each x_i is a token, a LM assigns a probability $P(x)$. Let G be the grammar of a programming language. We use x to denote a snippet, that is, a string x such that $\alpha x \beta \in \mathscr{L}(G)$ for some strings α, β. We primarily consider snippets that are dominated by a single node in the file's AST. That is, there is a node within the AST whose subtree comprises the entire snippet and nothing else. We use x to denote the input snippet to the framework, and y, z to denote arbitrary snippets[3].

3.1 The Core of NATURALIZE

The architecture contains two main elements: proposers and the scoring function. The *proposers* modify the input code snippet to produce a list of *suggestion candidates* that can replace the input snippet. In the example from Figure 1, each candidate replaces all occurrences of `res` with a different name used in similar contexts elsewhere in the project, such as `results` or `queryResults`. In principle, many implausible suggestions could ensue, so, in practice, proposers contain filtering logic.

A *scoring function* sorts these candidates according to a measure of naturalness. Its input is a candidate snippet, and it returns a real number measuring naturalness. Naturalness is measured with respect to a training corpus that is provided to NATURALIZE— thus allowing us to follow our guiding principle that naturalness must be measured with respect to a particular codebase. For example,

[2]In extreme cases, such systems can be so widely mocked that they are publicly disabled by the company's CEO in front of a cheering audience: `http://bit.ly/pmHCwI`.

[3]The application of NATURALIZE to academic papers in software engineering is left to future work.

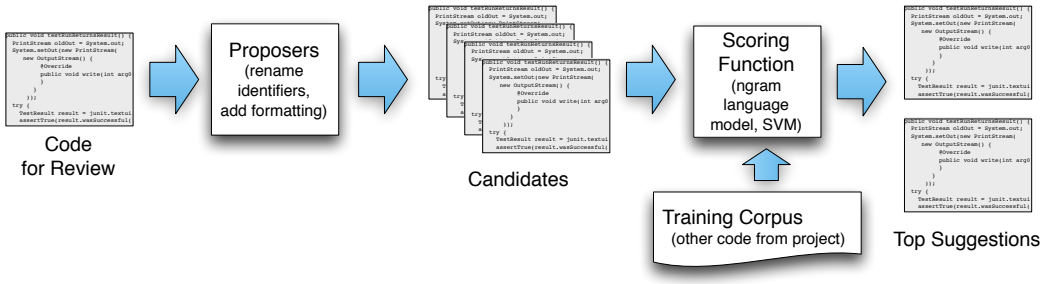


Figure 3: The architecture of NATURALIZE: a framework for learning coding conventions. A contiguous snippet of code is selected for review through the user interface. A set of *proposers* returns a set of candidates, which are modified versions of the snippet, e.g., with one local variable renamed. The candidates are ranked by a *scoring function*, such as an n-gram language model, which returns a small list of top suggestions to the interface, sorted by naturalness.

the training corpus might be the set of source files A from the current application. A powerful way to measure the naturalness of a snippet is provided by statistical language modeling. We use $P_A(y)$ to indicate the probability that the language model P, which has been trained on the corpus A, assigns to the string y. The key intuition is that an LM P_A is trained so that it assigns high probability to strings in the training corpus, *i.e.*, snippets with higher log probability are more like the training corpus, and presumably more natural. There are several key reasons why statistical language models are a powerful approach for modeling coding conventions. First, probability distributions provide an easy way to represent *soft* constraints about conventions. This allows us to avoid many of the pitfalls of inflexible, rule-based approaches. Second, because they are based on a learning approach, LMs can flexibly adapt to the conventions in a new project. Intuitively, because P_A assigns high probability to strings $t \in A$ that occur in the training corpus, it also assigns high probability to strings that are *similar to* those in the corpus. So the scoring function s tends to favor snippets that are stylistically consistent with the training corpus.

We score the naturalness of a snippet $y = y_{1:N}$ as

$$s(y, P_A) = \frac{1}{N} \log P_A(y); \tag{1}$$

that is, we deem snippets that are more probable under the LM as more natural in the application A. Equation 1 is cross-entropy multiplied by -1 to make s a score, where $s(x) > s(y)$ implies x is more natural than y. Where it creates no confusion, we write $s(y)$, eliding the second argument. When choosing between competing candidate snippets y and z, we need to know not only which candidate the LM prefers, but how "confident" it is. We measure this by a *gap function* g, which is the difference in scores $g(y,z,P) = s(y,P) - s(z,P)$. Because s is essentially a log probability, g is the log ratio of probabilities between y and z. For example, when $g(y,z) > 0$ the snippet y is more natural — *i.e.*, less surprising according to the LM — and thus is a better suggestion candidate than z. If $g(y,z) = 0$ then both snippets are equally natural.

Now we define the function $\mathsf{suggest}(x, C, k, t)$ that returns the top candidates according to the scoring function. This function returns a list of top candidates, or the empty list if no candidates are sufficiently natural. The function takes four parameters: the input snippet x, the list $C = (c_1, c_2, \ldots c_r)$ of candidate snippets, and two thresholds: $k \in \mathbb{N}$, the maximum number of suggestions to return, and $t \in \mathbb{R}$, a minimum confidence value. The parameter k controls the size of the ranked list that is returned to the user, while t controls the *suggestion frequency*, that is, how confident NATURALIZE needs to be before it presents any suggestions to the user. Appropriately setting t allows NATURALIZE to avoid the Clippy effect by making no suggestion rather than a low quality one. Below, we present an automated method for selecting t.

The suggest function first sorts $C = (c_1, c_2, \ldots c_r)$, the candidate list, according to s, so $s(c_1) \geq s(c_2) \geq \ldots \geq s(c_r)$. Then, it trims the list to avoid overburdening the user: it truncates C to include only the top k elements, so that $\text{length}(C) = \min\{k, r\}$. and removes candidates $c_i \in C$ that are not sufficiently more natural than the original snippet; formally, it removes all c_i from C where $g(c_i, x) < t$. Finally, if the original input snippet x is the highest ranked in C, *i.e.*, if $c_1 = x$, suggest ignores the other suggestions, sets $C = \emptyset$ to decline to make a suggestion, and returns C.

Binary Decision If an accept/reject decision on the input x is required, *e.g.*, as in `stylish?`, NATURALIZE must collectively consider all of the locations in x at which it could make suggestions. We propose a score function for this binary decision that measures how good is the best possible improvement that NATURALIZE is able to make. Formally, let L be the set of locations in x at which NATURALIZE is able to make suggestions, and for each $\ell \in L$, let C_ℓ be the system's set of suggestions at ℓ. In general, C_ℓ contains name or formatting suggestions. Recall that P is the language model. We define the score

$$G(x, P) = \max_{\ell \in L} \max_{c \in C_\ell} g(c, x). \tag{2}$$

If $G(x,P) > T$, then NATURALIZE rejects the snippet as being excessively unnatural. The threshold T controls the sensitivity of NATURALIZE to unnatural names and formatting. As T increases, fewer input snippets will be rejected, so some unnatural snippets will slip through, but as compensation the test is less likely to reject snippets that are in fact well-written.

Setting the Confidence Threshold The thresholds t in the suggest function and T in the binary decision function are on log probabilities of strings, which can be difficult for users to interpret. Fortunately, these can be set *automatically* using the *false positive rate (FPR)*, *i.e.* the proportion of snippets x that in fact follow convention but that the system erroneously rejects. We would like the FPR to be as small as possible, but, unless we wish the system to make no suggestions at all, we must accept some false positives. So we set a maximum acceptable FPR α, and search for a threshold t or T that ensures that NATURALIZE's FPR is at most α. The principle is similar to statistical hypothesis testing. To make this work, we estimate the FPR for a given t or T. To do so, we select a random set of snippets from the training corpus, *e.g.*, random method bodies, and compute the proportion of these snippets that are rejected using T. Again leveraging our assumption that our training corpus contains natural code, this proportion estimates the FPR. We use a grid search [11] to find the greatest value of $T < \alpha$ ($t < \alpha$), the user-specified acceptable FPR bound.

3.2 Choices of Scoring Function

The generic framework described in subsection 3.1 can, in principle, employ a wide variety of machine learning or NLP methods for

its scoring function. Indeed, a large portion of the statistical NLP literature focuses on probability distributions over text, including language models, probabilistic grammars, and topic models. Very few of these models have been applied to code; exceptions include [4, 35, 46, 49, 53]. We choose to build on statistical language models, because previous work of Hindle *et al.*. [35] has shown that they are particularly able to capture the naturalness of code.

The intuition behind language modeling is that since there is an infinite number of possible strings, obviously we cannot store a probability value for every one. Different LMs make different simplifying assumptions to make the modeling tractable, and will determine the types of coding conventions that we are able to infer. One of the most effective practical LMs is the n-gram language model. N-gram models make the assumption that the next token can be predicted using only the previous $n-1$ tokens. Formally, the probability of a token y_m, conditioned on all of the previous tokens $y_1 \dots y_{m-1}$, is a function only of the previous $n-1$ tokens. Under this assumption, we can write

$$P(y_1 \dots y_M) = \prod_{m=1}^{M} P(y_m | y_{m-1} \dots y_{m-n+1}). \quad (3)$$

To use this equation we need to know the conditional probabilities $P(y_m | y_{m-1} \dots y_{m-n+1})$ for each possible n-gram. This is a table of V^n numbers, where V is the number of possible lexemes. These are the *parameters* of the model that we learn from the training corpus. The simplest way to estimate the model parameters is to set $P(y_m | y_{m-1} \dots y_{m-n+1})$ to the proportion of times that y_m follows $y_{m-1} \dots y_{m-n+1}$. In practice, this simple estimator does not work well, because it assigns zero probability to n-grams that do not occur in the training corpus. Instead, n-gram models are trained using *smoothing* methods [22]. In our work, we use Katz smoothing.

Implementation When an n-gram model is used, we can compute the gap function $g(y,z)$ very efficiently. This is because when g is used within `suggest`, ordinarily the strings y and z will be similar, *i.e.*, the input snippet and a candidate revision. The key insight is that in an n-gram model, the probability $P(y)$ of a snippet $y = (y_1 y_2 \dots y_N)$ depends only on the multiset of n-grams that occur in y, that is,

$$NG(y) = \{y_i y_{i+1} \dots y_{i+n-1} \,|\, 0 \le i \le N-(n-1)\}. \quad (4)$$

An equivalent way to write a n-gram model is

$$P(y) = \prod_{a_1 a_2 \dots a_n \in NG(y)} P(a_n | a_1, a_2, \dots a_{n-1}). \quad (5)$$

Since the gap function is $g(y,z) = \log[P(y)/P(z)]$, any n-grams that are members both of $NG(y)$ and $NG(z)$ cancel, so to compute g, we only need to consider those n-grams not in $NG(y) \cap NG(z)$. Intuitively, this means that, to compute the gap function $g(y,z)$, we need to examine the n-grams around the locations where the snippets y and z differ. This is a very useful optimization if y and z are long snippets that differ in only a few locations.

When training an LM, we take measures to deal with *rare* lexemes, since, by definition, we do not have much data about them. We use a preprocessing step — a common strategy in language modeling — that builds a vocabulary with all the identifiers that appear more than once in the training corpus. Let $\text{count}(v,b)$ return the number of appearances of token v in the codebase b. Then, if a token has $\text{count}(v,b) \le 1$ we convert it to a special token, which we denote UNK. Then we train the n-gram model as usual. The effect is that the UNK token becomes a catchall that means the model expects to see a rare token, even though it cannot be sure which one.

3.3 Suggesting Natural Names

In this section, we instantiate the core NATURALIZE framework for the task of suggesting natural identifier names. We start by describing the single suggestion setting. For concreteness, imagine a user of the `devstyle` plugin, who selects an identifier and asks `devstyle` for its top suggestions. It should be easy to see how this discussion can be generalized to the other use cases described in subsection 2.1. Let v be the lexeme selected by the programmer. This lexeme could denote a variable, a method call, or a type.

When a programmer binds a name to an identifier and then uses it, she implicitly links together all the locations in which that name appears. Let L denote this set of locations, that is, the set of locations in the current scope in which the lexeme v is used. For example, if v denotes a local variable, then L_v would be the set of locations in which that local is used. Now, the input snippet is constructed by finding a snippet that subsumes all of the locations in L_v. Specifically, the input snippet is constructed by taking the lowest common ancestor in AST of the nodes in L_v.

The proposers for this task retrieve a set of alternative names to v, which we denote A_v, by retrieving other names that have occurred in the same contexts in the training set. To do this, for every location $\ell \in L_v$ in the snippet x, we take a moving window of length n around ℓ and copy all the n-grams w_i that contain that token. Call this set C_v the context set, *i.e.*, the set of n-grams w_i of x that contain the token v. Now we find all n-grams in the training set that are similar to an n-gram in C_v but that have some other lexeme substituted for v. Formally, we set A_v as the set of all lexemes v' for which $\alpha v \beta \in C_v$ and $\alpha v' \beta$ occurs in the training set. This guarantees that if we have seen a lexeme in at least one similar context, we place it in the alternatives list. Additionally, we add to A_v the special UNK token; the reason for this is explained in a moment. Once we have constructed the set of alternative names, the candidates are a list S_v of snippets, one for each $v' \in A_v$, in which all occurrences of v in x are replaced with v'.

The scoring function can use any model P_A, such as the n-gram model (Equation 3). N-gram models work well because, intuitively, they favors names that are common *in the context* of the input snippet. As we demonstrate in section 4, this does *not* reduce to simply suggesting the most common names, such as `i` and `j`. For example, suppose that the system is asked to propose a name for `res` in line 3 of Figure 1. The n-gram model is highly unlikely to suggest `i`, because even though the name `i` is common, the trigram "`QueryResults i ,`" is rare.

An interesting subtlety involves names that actually *should be* unique. Identifier names have a long tail, meaning that most names are individually uncommon. It would be undesirable to replace every rare name with common ones, as this would violate the sympathetic uniqueness principle. Fortunately, we can handle this issue in a subtle way: recall from subsection 3.1 that, during training of the n-gram LM, we convert rare names into the special UNK token. When we do this, UNK exists as a token in the LM, just like any other name. We simply allow NATURALIZE to return UNK as a suggestion, just like any other name. Returning UNK as a suggestion means that the model expects that it would be natural to use a rare name in the current context. The reason that this preserves rare identifiers is that the UNK token occurs in the training corpus specifically in unusual contexts where more common names were not used. Thus, if the input lexeme v occurs in an unusual context, this context is more likely to match that of UNK than of any of the more common tokens.

Multiple Point Suggestion It is easy to adapt the system above to the multiple point suggestion task. Recall (subsection 2.1) that this task is to consider the set of identifiers that occur in a region x of code selected by the user, and highlight the lexemes that are least natural in context. For single point suggestion, the problem is to rank different alternatives, *e.g.*, different variable names, for the same code location, whereas for multiple point suggestion, the problem is to rank different code locations against each other according to how

```
5       @Override public void
6       write(int arg0) throws IOException {
7       }
8   }
```

```
5   INDENT^{3s}_{1n} @ SPACE^0 ID SPACE^{1s} public SPACE^{1s} void
6   INDENT^0_{1n} ID SPACE^0 ( SPACE^0 ID SPACE^{1s} ID SPACE^0 ) SPACE^{1s}
    throws SPACE^{1s} ID SPACE^{1s} {
7   INDENT^0_{1n} }
8   INDENT^{-3s}_{1n} }
```

Figure 4: A code snippet from `TextRunnerTest.java` in `JUnit` and the corresponding formatting tokenization.

much they would benefit from improvement. In principle, a score function could be good at the single source problem but bad at the multiple source problem, e.g., if the score values have a different dynamic range when applied at different locations.

We adapt NATURALIZE slightly to address the multiple point setting. For all identifier names v that occur in x, we first compute the candidate suggestions S_v as in the single suggestion case. Then the full candidate list for the multiple point suggestion is $S = \cup_{v \in x} S_v$; each candidate arises from proposing a change to one name in x. For the scoring function, we need to address the fact that some names occur more commonly in x than others, and we do not want to penalize names solely because they occur more often. So we normalize the score according to how many times a name occurs. Formally, a candidate $c \in S$ that has been generated by changing a name v, we use the score function $s'(c) = |C_v|^{-1} s(c)$.

3.4 Suggesting Natural Formatting

We apply NATURALIZE to build a *language-agnostic code formatting suggester* that automatically and adaptively learns formatting conventions and generates rules for use by code formatters, like the Eclipse formatter. An n-gram model works over token streams; for the n-gram instantiation of NATURALIZE to provide formatting suggestions, we must tokenize whitespace. We change the tokenizer to encode *contiguous* whitespace into tokens using the grammar

$$\texttt{S} ::= \texttt{T W S} \mid \varepsilon$$
$$\texttt{W} ::= \texttt{SPACE}^{space/tab} \mid \texttt{INDENT}^{space/tabs}_{lines}$$
$$\texttt{T} ::= \texttt{ID} \mid \texttt{LIT} \mid \texttt{\{} \mid \texttt{\}} \mid \texttt{.} \mid \texttt{(} \mid \texttt{)} \mid \texttt{<keywords>}.$$

Figure 4 shows a code snippet drawn from `JUnit` followed by our tokenization of it. We collapse all identifiers to a single `ID` token and all literals to a single `LIT` token because we presume that the actual identifier and literal lexemes do not convey formatting information. Starting whitespace determines the indentation level and usually signifies nesting. We replace it with a special `INDENT` token, along with metadata encoding the increase of whitespace (that may be negative or zero) *relative* to the previous line. We also annotate `INDENT` with the number of new lines before any proceeding (non-whitespace) token. This captures code that is visually separated with at least one empty line. In Figure 4, line 5 indents by 3 spaces from the previous line. Whitespace within a line controls the appearance of operators and punctuation: "`if(` *vs.* `if (`" and "`x<y` *vs.* `x < y`". We encode this whitespace into the special token `SPACE`, along with the number of spaces/tabs that it contains. When no space occurs between two non-whitespace tokens, we inject $\texttt{SPACE}^0$. To capture the increasing probability of an `INDENT` in a long line, we annotate all tokens of type `T` with their size (number of characters) and the current column of that token. To reduce sparsity, we quantize these annotations into buckets of size q.

In the original code listing, an empty method is straddling lines 6–7 in Figure 4. If a programmer asks "Is this conventional?", NATURALIZE considers alternatives for the underlined token. We train the LM over the whitespace-modified tokenizer, then, since the vocabulary of whitespace tokens (assuming bounded numbers in the metadata) is small, we rank all whitespace token alternatives according to the scoring function (subsection 3.2).

3.5 Converting Conventions into Rules

NATURALIZE can convert the conventions it infers from a codebase into rules for a code formatter. We formalize a code formatter's rule as the set of settings $S = \{s_1, s_2, \ldots, s_n\}$ and C, a set of constraints over the elements in S. For example s_i could be a boolean that denotes "`{` must be on the same line as its function signature" and s_j might be the number of spaces between the closing `)` and the `{`. Then C might contain $(s_j \geq 0) \rightarrow s_i \wedge (s_j < 0) \rightarrow \neg s_i$. To extract rules, we handcraft a set of minimal code snippets that exhibit each different setting of s_i, $\forall s_i \in S$. After training NATURALIZE on a codebase, we apply it to these snippets. Whenever NATURALIZE is confident enough to prefer one to the other, we infer the appropriate setting, otherwise we leave the default untouched, which may well be to ignore that setting when applying the formatter.

4. EVALUATION

We now present an evaluation of the value and effectiveness of NATURALIZE. While our evaluation is on a Java corpus, NATURALIZE is language-agnostic. We first present two empirical studies that show NATURALIZE solves a real world problem that programmers care about (subsection 4.1). These studies demonstrate that 1) programmers do not always adhere to coding conventions and yet 2) that project members care enough about them to correct such violations. Then, we move on to evaluating the suggestions produced by NATURALIZE. We perform an extensive automatic evaluation (subsection 4.2) which verifies that NATURALIZE produces natural suggestions that matches real code. Automatic evaluation is a standard methodology in statistical NLP [56, 45], and is a vital step when introducing a new research problem, because it allows future researchers to test new ideas rapidly. This evaluation relies on perturbation: given code text, we perturb its identifiers or formatting, then check if NATURALIZE suggests the original name or formatting that was used. Furthermore, we also employ the automatic evaluation to show that NATURALIZE is robust to low quality corpora (subsection 4.3).

Finally, to complement the automatic evaluation, we perform two qualitative evaluations of the effectiveness of NATURALIZE suggestions. First, we manually examine the output of NATURALIZE, showing that even high quality projects contain many entities for which other names can be reasonably considered (subsection 4.4). Finally, we submitted patches based on NATURALIZE suggestions (subsection 4.5) to 5 of the most popular open source projects on GitHub — of the 18 patches that we submitted, 12 were accepted by the core members of these projects.

Methodology Our corpus is a set of well-known open source Java projects. From GitHub[4], we obtained the list of all Java projects that are not forks and scored them based on their number of "watchers" and forks. The mean number of watchers and forks differ, so, to combine them, we assumed these numbers follow the normal distribution and summed their z-scores. For these evaluations reported here, we picked the top 10 scoring projects that are *not* in the training set of the GitHub Java Corpus [4]. Our original intention was to also demonstrate cross-project learning, but have no space to report these findings. Table 1 shows the selected projects.

Like any experimental evaluation, our results are susceptible to the standard threat of external validity. Interestingly, this threat does not extend to the training corpus, because the whole point is to bias NATURALIZE toward the conventions that govern the training set. Rather, our interest is to ensure that NATURALIZE's performance on our evaluation corpus matches that of projects overall, which is why we took such care in constructing our corpus.

[4] `http://www.github.com`, on 21 August, 2013.

Table 1: Open-source Java projects used for evaluation. Ordered by popularity.

Name	Forks	Watchers	Commit	Pull Request	Description
elasticsearch	1009	4448	`af17ae55`	`#5075`$^{\text{merged}}$	REST Search Engine
libgdx	1218	1470	`a42779e9`	`#1400`$^{\text{merged}}$	Game Development Framework
netty	683	1940	`48eb73f9`	did not submit	Network Application Framework
platform_frameworks_base	947	1051	`a0b320a6`	did not submit	Android Base Framework
junit*	509	1796	`d919bb6d`	`#834`$^{\text{merged}}$	Testing Framework
wildfly	845	885	`9d184cd0`	did not submit	JBoss Application Server
hudson	997	215	`be1f8f91`	did not submit	Continuous Integration Server
android-bootstrap	360	1446	`e2cde337`	did not submit	Android Application Template
k-9	583	960	`d8030eaa`	`#454`$^{\text{merged}}$	Android Email Client
android-menudrawer	422	1138	`96cdcdcc`	`#216`$^{\text{open}}$	Android Menu Implementation

*Used as a validation project for tuning parameters.

Our evaluations use leave-one-out cross validation. We test on each file in the project, training our models on the remaining files. This reflects the usage scenario that we recommend in practice. We report the average performance over all test files. For an LM, we have used a 5-gram model, chosen via calibration on the validation project `JUnit`. We picked `JUnit` as the validation project because of its medium size.

4.1 The Importance of Coding Conventions

To assess whether obeying coding conventions, specifically following formatting and naming conventions, is important to software teams today, we conducted two empirical studies that we present in this section. But first, we posit that coding style is both an important and a contentious topic. The fact that many languages and projects have style guides is a testament to this assertion. For example, we found that the Ruby style guide has at least 167 un-merged forks and the Java GitHub Corpus [4] has 349 different `.xml` configurations for the Eclipse formatter.

Commit Messages We manually examined 1,000 commit messages drawn randomly from the commits of eight popular open source projects looking for mentions of renaming, changing formatting, and following other code conventions. We found that 2% of changes contained formatting improvements, 1% contained renamings, and 4% contained any changes to follow code conventions (which include formatting and renaming). We observed that not all commits that contain changes to adhere to conventions mention such conventions in the commit messages. Thus, our percentages likely represent lower bounds on the frequency of commits that change code to adhere to conventions.

Code Review Discussions We also examined discussions that occurred in reviews of source code changes. Code review is practiced heavily at Microsoft in an effort to ensure that changes are free of defects and adhere to team standards. Once an author has completed a change, he creates a code review and sends it to other developers for review. They then inspect the change, offer feedback, and either sign off or wait for the author to address their feedback in a subsequent change. As part of this process, the reviewers can highlight portions of the code and begin a discussion (thread) regarding parts of the change (for more details regarding the process and tools used, see Bacchelli *et al.* [10]).

We examined 169 code reviews selected randomly across Microsoft product groups during 2014. Our goal was to include enough reviews to examine at least 1,000 discussion threads. In total, these 169 reviews contained 1093 threads. We examined each thread to determine if it contained feedback related to a) code conventions in general, b) identifier naming, and c) code formatting. 18% of the threads examined provided feedback regarding coding conventions of some kind. 9% suggested improvements in naming and 2% suggested changes related to code formatting (subsets of the 18%).

Table 2: Percent commits with log messages and reviews containing feedback regarding code conventions, identifier naming, and formatting with 95% confidence intervals in parentheses.

Type	Reviews (CI)		Commits (CI)		$p-val$
Conventions	38%	(31%–46%)	4%	(3%–6%)	$p \ll 0.01$
Naming	24%	(17%–31%)	1%	(1%–2%)	$p \ll 0.01$
Formatting	9%	(6%–15%)	2%	(1%–3%)	$p \ll 0.01$

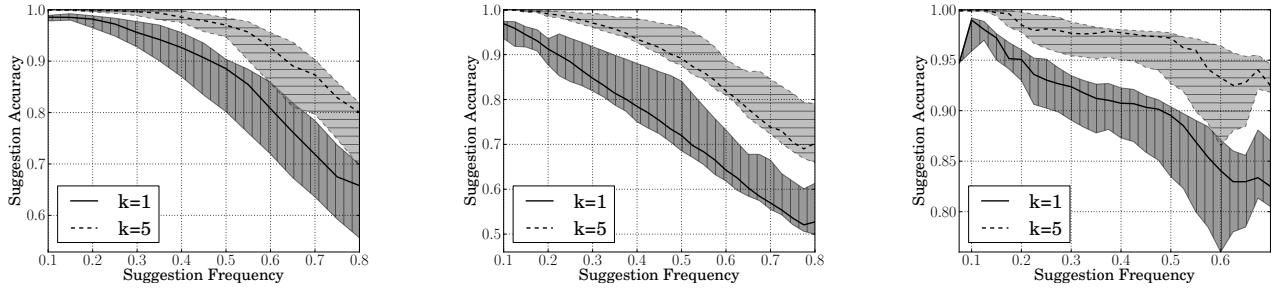
In terms of the reviews that contained feedback of each kind, the proportions are 38%, 24%, and 9%.

During February 2014, just over 126,000 reviews were completed at Microsoft. Thus, based on confidence intervals of these proportions, between 7,560 and 18,900 reviews received feedback regarding formatting changes that were needed prior to check-in and between 21,420 and 39,060 reviews resulted in name changes in just one month.

Table 2 summarizes our findings from examining commit messages and code reviews. We also present 95% confidence intervals based on the sampled results [25]. These results demonstrate that changes, to a nontrivial degree, violate coding conventions even after the authors consider them complete and also that team members expect that these violations be fixed. We posit that, like defects, many convention violations die off during the lifecycle of a change, so that few survive to review and fewer still escape into the repository. This is because, like defects, programmers notice and fix many violations themselves during development, prior to review, so reviewers must hunt for violations in a smaller set, and committed changes contain still fewer, although this number is nontrivial, as we show in subsection 4.4. Corrections during development are unobservable. However, we can compare convention corrections in review to corrections after commit. We used a one-sided proportional test to evaluate if more coding conventions are corrected during review than after commit. The last column in Table 2 contains the p-values for our tests, indicating that the null hypothesis can be rejected with statistically significant support.

4.2 Suggestion

In this section we present an automatic evaluation of NATURALIZE's suggestion accuracy. First we evaluate naming suggestions (subsection 3.3). We focus on suggesting new names for (1) locals, (2) arguments, (3) fields, (4) method calls, and (5) types (class names, primitive types, and enums) — these are the five distinct types of identifiers the Eclipse compiler recognizes [26]. Recall from subsection 3.3 that when NATURALIZE suggests a renaming, it renames *all* locations where that identifier is used at once. Furthermore, as described earlier, we always use leave-one-out cross validation, so we never train the language model on the files for which we are making suggestions. Therefore, NATURALIZE cannot pick up the correct name for an identifier from other occurrences in



(a) Variables **(b) Method Calls** **(c) Typenames**

Figure 5: Evaluation of single point suggestions of NATURALIZE, when it is allowed to suggest $k = 1$ and $k = 5$ alternatives. The shaded area around each curve shows the interquartile range of the suggestion accuracy across the 10 evaluation projects.

the same file; instead, it must generalize by learning conventions from other files in the project.

Single Point Suggestion First we evaluate NATURALIZE on the single point suggestion task, that is, when the user has asked for naming suggestions for a single identifier. To do this, for each test file, for each unique identifier we collect all of the locations where the identifier occurs and names the same entity, and ask NATURALIZE to suggest a new name, renaming all occurrences at once. We measure accuracy, that is, the percentage of the time that NATURALIZE correctly suggests the original name. This is designed to reflect the typical usage scenario described in section 2, in which a developer has made a good faith effort to follow a project's conventions, but may have made a few mistakes.

Figure 5 reports on the quality of the suggestions. Each point on these curves corresponds to a different value of the confidence threshold t. The x-axis shows the suggestion frequency, i.e., at what proportion of code locations where NATURALIZE is capable of making a suggestion does it choose to do so. The y-axis shows suggestion accuracy, that is, the frequency at which the true name is found in the top k suggestions, for $k = 1$ and $k = 5$. As t increases, NATURALIZE makes fewer suggestions of higher quality, so frequency decreases as accuracy increases. These plots are similar in spirit to precision-recall curves in that curves nearer the top right corner of the graph are better. Figure 5a, Figure 5b, and Figure 5c show that NATURALIZE performance varies with both project and the type of identifiers. Figure 5a combines locals, fields, and arguments because their performance is similar. NATURALIZE's performance varies across these three categories of identifiers because of the data hungriness of n-grams and because local context is an imperfect proxy for type constraints or function semantics. The results show that NATURALIZE effectively avoids the Clippy effect, because by allowing the system to decline to suggest in a relatively small proportion of cases, it is possible to obtain good suggestion accuracy. Indeed, NATURALIZE can achieve 94% suggestion accuracy across identifier types, even when forced to make suggestions at half of the possible opportunities.

Multiple Point Selection To evaluate NATURALIZE's accuracy at multiple point suggestion, e.g., in `devstyle` or `styleprofile`, we mimic code snippets in which one name violates the project's conventions. For each test snippet, we randomly choose one identifier and perturb it to a name that does not occur in the project, compute the style profile, and measure where the perturbed name appears in the list of suggestions. NATURALIZE's recall at rank $k = 7$, chosen because humans can take in 7 items at a glance [23, 48] is 64.2%. The mean reciprocal rank is 0.47: meaning that, on average, we return the bad name at position 2.

Single Point Suggestion for Formatting To evaluate NATURALIZE's performance at making formatting suggestions, we follow the same procedure as the single-point naming experiment to check if NATURALIZE correctly recovers the original formatting from the context. We train a 5-gram language model using the modified token stream ($q = 20$) discussed in subsection 3.4. We allow the system to make only $k = 1$ suggestions to simplify the UI. We find that the system is extremely effective (Figure 9) at formatting suggestions, achieving 98% suggestion accuracy even when it is required to reformat half of the whitespace in the test file. This is remarkable for a system that is not provided with any hand-designed rules about what formatting is desired. Obviously if the goal is to reformat *all* the whitespace an entire file, a rule-based formatter is called for. But this performance is more than high enough to support our use cases, such as providing single point formatting suggestions on demand, rejecting snippets with unnatural formatting and extracting high confidence rules for rule-based formatters.

Binary Snippet Decisions Finally, we evaluate the ability of `stylish?` to discriminate between code selections that follow conventions well from those that do not, by mimicking commits that contain unconventional names or formatting. Uniformly at random, we selected a set of methods from each project (500 in total), then uniformly ($\frac{1}{3}$), we either made no changes or perturb one identifier or whitespace token to a token in the n-gram's vocabulary V. This method for mimicking commits is probably a worst case for our method, because the perturbed methods will be very similar to existing methods, which are likely to be conventional.

We run `stylish?` and record whether the perturbed snippet is rejected because of its names or its formatting. `stylish?` is unaware of the perturbation (if any), identifier or whitespace, made to the snippet. Figure 6 reports NATURALIZE's rejection performance as ROC curves. In each curve, each point corresponds to a different choice of threshold T, and the x-axis shows FPR (estimated as in Equation 3.1), and the y-axis shows true positive rate, the proportion of the perturbed snippets that we correctly rejected. NATURALIZE achieves high precision, making it suitable for use as a filtering pre-commit script. When the FPR is at most 0.05, we are able to correctly reject 40% of the snippets. NATURALIZE is somewhat worse at rejecting snippets whose variable names have been perturbed; in part, this is because predicting identifier names is more difficult than predicting formatting. New advances in language models for code [46, 53] are likely to improve these results. Nonetheless, these results are promising: `stylish?` still rejects enough perturbed snippets that if deployed at 5% FPR, it would enhance convention adherence with minimal disruption to developers.

4.3 Robustness of Suggestions

We show that NATURALIZE avoids two potential pitfalls in its identifier suggestions: first, that it does not simply rename all tokens to common "junk" names that appear in many contexts, and second, that it retains unusual names that signify unusual functionality, adhering to the SUP.

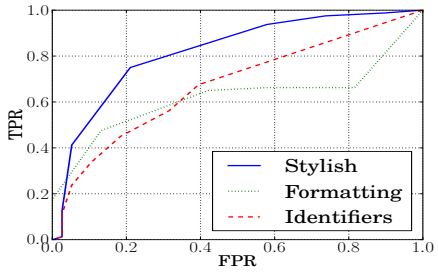


Figure 6: Evaluation of `stylish?` tool for rejecting unnatural changes. To generate unnatural code, we perturb one identifier or formatting point or make no changes, and evaluate whether NATURALIZE correctly rejects or accepts the snippet. The graph shows the receiver operating characteristic (ROC) of this process for `stylish?` when using only identifiers, only formatting or both.

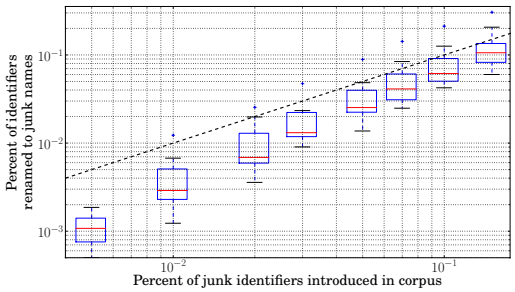


Figure 7: Is NATURALIZE robust to low-quality corpora? The x-axis shows percentage of identifiers perturbed to junk names to simulate low quality corpus. The y-axis is percentage of resulting low quality suggestions. Note log-log scale. The dotted line shows $y = x$. The boxplots are across the 10 evaluation projects.

Junk Names A junk name is a semantically uninformative name used in disparate contexts. It is difficult to formalize this concept: for instance, in almost all cases, `foo` and `bar` are junk names, while `i` and `j`, when used as loop counters, are semantically informative and therefore not junk. Despite this, most developers "know it when they see it." One might at first be concerned that NATURALIZE would often suggest junk names, because junk names appear in many different n-grams in the training set. We argue, however, that in fact the opposite is the case: NATURALIZE actually *resists* suggesting junk names. This is because if a name appears in too many contexts, it will be impossible to predict a unsurprising follow-up, and so code containing junk names will have lower probability, and therefore worse score.

To evaluate this claim, we randomly rename variables to junk names in each project to simulate a low quality project. Notice that we are simulating a low quality *training* set, which should be the worst case for NATURALIZE. We measure how our suggestions are affected by the proportion of junk names in the training set. To generate junk variables we use a discrete Zipf's law with slope $s = 1.08$, the slope empirically measured for all identifiers in our evaluation corpus. We verified the Zipfian assumption in previous work [4]. Figure 7 shows the effect on our suggestions as the evaluation projects are gradually infected with more junk names. The framework successfully avoids suggesting junk names, proposing them at a lower frequency than they exist in the perturbed codebase.

Sympathetic Uniqueness Surprise can be good in identifiers, where it signifies unusual functionality. Here we show that NATURALIZE preserves this sort of surprise. We find all identifiers in the test file that are unknown to the LM, *i.e.* are represented by an UNK. We then plot the percentage of those identifiers for which `suggest` does *not* propose an alternative name, as a function of the threshold t. As described in Equation 3.1, t is selected automatically, but it is useful to explore how adherence to the SUP varies as a function of t. Figure 8 shows that for reasonable threshold values, NATURALIZE suggests non-UNK identifiers for only a small proportion of the UNK identifiers (about 20%). This confirms that NATURALIZE does not cause the "heat death of a codebase" by renaming semantically rich, surprising names into frequent, low-content names.

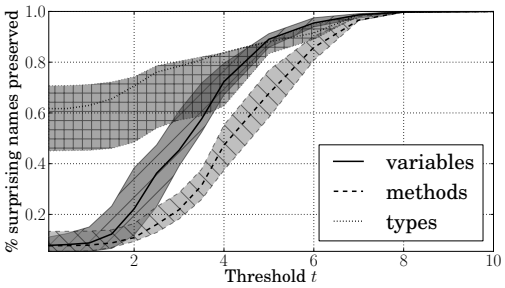


Figure 8: NATURALIZE does not cause the "heat death" of a codebase: we evaluate the percent of single suggestions made on UNK identifiers that preserve the surprising name. The threshold t on the x-axis controls the suggestion frequency of suggest; lower t gives suggest less freedom to decline to make low-quality suggestions.

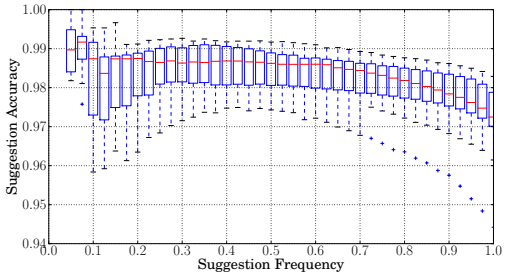


Figure 9: Evaluation of single point evaluation for formatting. Only $k = 1$ suggestions are allowed in the ranked list. The boxplots show the variance in performance across the 10 evaluation projects.

4.4 Manual Examination of Suggestions

As a qualitative evaluation of NATURALIZE's suggestions, three human evaluators (three of the authors) independently evaluated the quality of its suggestions. First, we selected two projects from our corpus, uniformly at random, then for each we ran `styleprofile` on 30 methods selected uniformly at random to produce suggestions for all the identifiers present in that method. We assigned 20 profiles to each evaluator such that each profile had two evaluators whose task was to *independently* determine whether any of the suggested renamings in a profile were reasonable. The evaluators attempted to take an evidence based approach, that is, not to simply choose names that they liked, but to choose names that were consistent with existing practice in the project. We provided access to the full source code of each project to the evaluators.

One special issue arose when suggesting changes to method names. Adapting linguistic terminology to our context, *homonyms* are two semantically equivalent functions with distinct names. We deem NATURALIZE's suggested renaming of a method usage to be reasonable, if the suggestion list contains a *partial*, not necessarily perfect, homonym; *i.e.*, if it draws the developer's attention

to another method that is used in similar contexts. Each evaluator had 15 minutes to consider each profile and 30 minutes to explore each project before starting on a new project. In fact, the evaluators required much less time than allocated, averaging about 5 minutes per example. The human evaluations can be found on our project's webpage. Surprisingly, given the quality of our evaluation codebases, 50% of the suggestions were determined to be useful by both evaluators. Further, no suggestion works well for everyone; when we consider NATURALIZE's performance in terms of whether at least one evaluator found a suggestion useful, 63% of the suggestions are useful, with an inter-rater agreement (Cohen's kappa [21]) of $\kappa = 0.73$. The quality of suggestions is strikingly high given that these projects are mature, vibrant, high-profile projects.

This provides evidence that the naming suggestions provided by NATURALIZE are qualitatively reasonable. Of course, an obvious threat to validity is that the evaluators are not developers of the test projects, and the developers themselves may well have had different opinions about naming. For this reason, we also provided the NATURALIZE naming suggestions to the project developers themselves, as described in the next section.

4.5 Suggestions Accepted by Projects

Using NATURALIZE's `styleprofile`, we identified high confidence renamings and submitted 18 of them as patches to the 5 evaluation projects that actively use GitHub. Table 1 shows the pull request ids and their current status. Four projects merged our pull requests (14 of 15 commits); the last ignored them without comment. Developers in the projects that accepted NATURALIZE's patches found the NATURALIZE useful: one said "Wow, that's a pretty cool tool!" [31]. JUNIT did not accept two of the suggested renamings as-is. Instead, the patches sparked a discussion. Its developers concluded that another name was more meaningful in one case and that the suggested renaming of another violated the project's explicit naming convention: "Renaming `e` to `t` is no improvement, because we should consistently use `e`." [30]. We then pointed them to the code locations that supported NATURALIZE's suggestion. This triggered them to change all the names that had caused the suggestion in the first place — NATURALIZE pointed out an inconsistency, previously unnoticed, that improved the naming in the project. Our project webpage has links to these discussions.

5. RELATED WORK

Coding conventions, readability, and identifiers have been extensively studied in the literature. Despite this, NATURALIZE is — to our knowledge — the first to infer coding conventions from a codebase, spanning naming and formatting.

Coding conventions are standard practice [15, 34]. They facilitate consistent measurement and reduce systematic error and generate more meaningful commits by eliminating trivial convention enforcing commits [73]. Some programming languages like Java and Python suggest specific coding styles [55, 64], while consortia publish guidelines for others, like C [8, 34].

High quality **identifier names** lie at the heart of software engineering [6, 16, 20, 24, 42, 66, 69, 54]; they drive code readability and comprehension [12, 19, 20, 41, 44, 68]. According to Deißenböck and Pizka [17], identifiers represent the majority (70%) of source code tokens. Eshkevari *et al.* [27] explored how identifiers change in code, while Lawrie *et al.* [41] studied the consistency of identifier namings. Abebe *et al.* [1] discuss the importance of naming to concept location [59]. Caprile and Tonella [20] propose a framework for restructuring and renaming identifiers based on custom rules and dictionaries. Gupta *et al.* present part-of-speech tagging on split multi-word identifiers to improve software engineering tools [33]. Høst and Østvold stem and tag method names, then learn a mapping from them to a fixed set of predicates over bytecode. Naming bugs are mismatches between the map and stemmed and tagged method names and their predicates in a test set [37]. In contrast, our work considers coding conventions more generally, and takes a flexible, data-driven approach. Several styles exist for engineering consistent identifiers [14, 20, 24, 65]. Because longer names are more informative [44], these styles share an agglutination mechanism for creating multi-word names [5, 60].

Many rule-based **code formatters** exist but, to our knowledge, are limited to constraining identifier names to obey constraints like CamelCase or underscore and cannot handle conventions like the use of `i` as a loop control variable. `Pylint` [58] checks if names match a simple set of regular expressions (*e.g.*, variable names must be lowercase); `astyle`, `aspell` and GNU `indent` [9, 32] only format whitespace tokens. `gofmt` formats the code aiming to "eliminating an entire class of argument" among developers [57, slide 66] but provides no guidance for naming. Wang *et al.* have developed a heuristic-based method to automatically insert blank lines into methods (vertical spacing) to improve readability [72]. NATURALIZE is unique in that it does not require upfront agreement on hard rules but learns soft rules that are implicit in a codebase. The soft rules about which NATURALIZE is highly confident can be extracted for a formatter.

API recommenders, code suggestion and completion systems aim to help during editing, when a user may not know the name of an API she needs [63], the parameters she should pass to the API [75], or that the API call she is making needs to preceded by another [29, 70, 71, 76]. Code suggestion and completion tools [18, 35, 52, 53, 62] suggest the next token during editing, often from a few initial characters. Essentially these methods address a search problem, helping the developer find an existing entity in code. Our focus is instead on release management and improving code's adherence to convention. For example, code completion engines will not suggest renaming parameters like `str` in Figure 1.

Language models are extensively used in **natural language processing**, especially in speech recognition and machine translation [22, 38]. Despite this extensive work, LMs have been under-explored for non-ambiguous (*e.g.* programming) languages, with only a few recent exceptions [4, 35, 46, 53]. The probabilistic nature of language models allows us to tackle the suggestion problem in a principled way. There is very little work on using NLP tools to suggest revisions to improve existing text. The main exception is spelling correction [39], to which LMs have been applied [47]. However, spelling correction methods often rely on strong assumptions about what errors are most common, a powerful source of information which has no analog in our domain.

6. CONCLUSION

We have presented NATURALIZE, the first tool that learns local style from a codebase and provides suggestions to improve stylistic consistency. We have taken the view that conventions are a matter of *mores* rather than *laws*: we suggest changes only when a codebase evinces stylistic consensus. We showed that NATURALIZE effectively makes natural suggestions, achieving 94% accuracy in its top suggestions for identifier names, and even suggests useful revisions to mature, high quality, open source projects.

7. ACKNOWLEDGEMENTS

This work was supported by Microsoft Research through its PhD Scholarship Programme. Charles Sutton was supported by the Engineering and Physical Sciences Research Council [grant number EP/K024043/1]. We acknowledge Mehrdad Afshari who first asked us about junk variables and the heat death of the codebase.

8. REFERENCES

[1] S. L. Abebe, S. Haiduc, P. Tonella, and A. Marcus. The effect of lexicon bad smells on concept location in source code. In *Source Code Analysis and Manipulation (SCAM), 2011 11th IEEE International Working Conference on*, pages 125–134. IEEE, 2011.

[2] A. Abran, P. Bourque, R. Dupuis, J. W. Moore, and L. L. Tripp. *Guide to the Software Engineering Body of Knowledge - SWEBOK*. IEEE Press, Piscataway, NJ, USA, 2004 version edition, 2004.

[3] E. N. Adams. Optimizing preventive service of software products. *IBM Journal of Research and Development*, 28(1):2–14, Jan. 1984.

[4] M. Allamanis and C. Sutton. Mining source code repositories at massive scale using language modeling. In *Proceedings of the Tenth International Workshop on Mining Software Repositories*, pages 207–216. IEEE Press, 2013.

[5] N. Anquetil and T. Lethbridge. Assessing the relevance of identifier names in a legacy software system. In *Proceedings of the 1998 Conference of the Centre for Advanced Studies on Collaborative Research*, page 4, 1998.

[6] N. Anquetil and T. C. Lethbridge. Recovering software architecture from the names of source files. *Journal of Software Maintenance*, 11(3):201–221, 1999.

[7] C. Arthur. Apple's SSL iPhone vulnerability: How did it happen, and what next? `bit.ly/1bJ7aSa`, 2014. Visited Mar 2014.

[8] M. I. S. R. Association et al. MISRA-C 2012: Guidelines for the Use of the C Language in Critical Systems. *ISBN 9781906400118*, 2012.

[9] astyle Contributors. Artistic style 2.03. `http://astyle.sourceforge.net/`, 2013. Visited September 9, 2013.

[10] A. Bacchelli and C. Bird. Expectations, outcomes, and challenges of modern code review. In *ICSE*, 2013.

[11] J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *The Journal of Machine Learning Research*, 13:281–305, 2012.

[12] T. J. Biggerstaff, B. G. Mitbander, and D. Webster. The concept assignment problem in program understanding. In *Proceedings of the 15th International Conference on Software Engineering*, pages 482–498. IEEE Computer Society Press, 1993.

[13] D. Binkley, M. Davis, D. Lawrie, J. Maletic, C. Morrell, and B. Sharif. The impact of identifier style on effort and comprehension. *Empirical Software Engineering*, 18(2):219–276, 2013.

[14] D. Binkley, M. Davis, D. Lawrie, and C. Morrell. To CamelCase or Under_score. In *IEEE International Conference on Program Comprehension (ICPC)*, pages 158–167, 2009.

[15] C. Boogerd and L. Moonen. Assessing the value of coding standards: An empirical study. In H. Mei and K. Wong, editors, *Proceedings of the 24th IEEE International Conference on Software Maintenance (ICSM 2008)*, pages 277 – 286. IEEE, October 2008.

[16] F. P. Brooks. *The Mythical Man-Month*. Addison-Wesley Reading, 1975.

[17] M. Broy, F. Deißenböck, and M. Pizka. A holistic approach to software quality at work. In *Proc. 3rd World Congress for Software Quality (3WCSQ)*, 2005.

[18] M. Bruch, M. Monperrus, and M. Mezini. Learning from examples to improve code completion systems. In *ESEC/SIGSOFT FSE*, pages 213–222. ACM, 2009.

[19] R. P. Buse and W. R. Weimer. Learning a metric for code readability. *Software Engineering, IEEE Transactions on*, 36(4):546–558, 2010.

[20] B. Caprile and P. Tonella. Restructuring program identifier names. In *International Conference on Software Maintenance (ICSM'00)*, pages 97–107, 2000.

[21] J. Carletta. Assessing agreement on classification tasks: the kappa statistic. *Computational Linguistics*, 22(2):249–254, 1996.

[22] S. Chen and J. Goodman. An empirical study of smoothing techniques for language modeling. In *Proceedings of the 34th Annual Meeting on Association for Computational Linguistics*, pages 310–318. Association for Computational Linguistics, 1996.

[23] N. Cowan. The magical number 4 in short-term memory: A reconsideration of mental storage capacity. *Behavioral and Brain Sciences*, 24(1):87–114, 2001.

[24] F. Deißenböck and M. Pizka. Concise and consistent naming [software system identifier naming]. In *Proceedings of the 13th International Workshop on Program Comprehension (IWPC'05)*, pages 97–106, 2005.

[25] S. Dowdy, S. Wearden, and D. Chilko. *Statistics for Research*, volume 512. John Wiley & Sons, 2011.

[26] Eclipse-Contributors. Eclipse JDT. `http://www.eclipse.org/jdt/`, 2013. Visited September 9, 2013.

[27] L. M. Eshkevari, V. Arnaoudova, M. Di Penta, R. Oliveto, Y.-G. Guéhéneuc, and G. Antoniol. An exploratory study of identifier renamings. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 33–42. ACM, 2011.

[28] M. Gabel and Z. Su. A study of the uniqueness of source code. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of software engineering*, FSE '10, pages 147–156, New York, NY, USA, 2010. ACM.

[29] M. G. Gabel. *Inferring Programmer Intent and Related Errors from Software*. PhD thesis, University of California, 2011.

[30] GitHub. JUnit Pull Request #834. `bit.ly/O8bmjM`, 2014. Visited Mar 2014.

[31] GitHub. libgdx Pull Request #1400. `bit.ly/O8aBqV`, 2014. Visited Mar 2014.

[32] gnu-indent Contributors. GNU Indent – beautify C code. `http://www.gnu.org/software/indent/`, 2013. Visited September 9, 2013.

[33] S. Gupta, S. Malik, L. Pollock, and K. Vijay-Shanker. Part-of-speech tagging of program identifiers for improved text-based software engineering tools. In *International Conference on Program Comprehension*, pages 3–12. IEEE, 2013.

[34] L. Hatton. Safer language subsets: an overview and a case history, MISRA C. *Information and Software Technology*, 46(7):465–472, 2004.

[35] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *International Conference on Software Engineering (ICSE)*, pages 837–847. IEEE, 2012.

[36] A. Hindle, M. W. Godfrey, and R. C. Holt. Reading beside the lines: Using indentation to rank revisions by complexity. *Science of Computer Programming*, 74(7):414–429, May 2009.

[37] E. W. Høst and B. M. Østvold. Debugging method names. In *In European Conference on Object-Oriented Programming (ECOOP)*, pages 294–317. Springer, 2009.

[38] D. Jurafsky and J. H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing,*

Computational Linguistics and Speech Recognition. Prentice Hall, 2nd edition, 2009.

[39] K. Kukich. Techniques for automatically correcting words in text. *ACM Computing Surveys*, 24(4):377–439, Dec. 1992.

[40] A. Langley. Apple's SSL/TLS bug. `bit.ly/MMvx6b`, 2014. Visited Mar 2014.

[41] D. Lawrie, H. Feild, and D. Binkley. Syntactic identifier conciseness and consistency. In *IEEE International Workshop on Source Code Analysis and Manipulation*, pages 139–148. IEEE, 2006.

[42] D. Lawrie, H. Feild, and D. Binkley. An empirical study of rules for well-formed identifiers: Research articles. *Journal of Software Maintenance Evolution: Research and Practice*, 19(4):205–229, July 2007.

[43] D. Lawrie, C. Morrell, H. Feild, and D. Binkley. What's in a Name? A Study of Identifiers. In *Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06)*, ICPC '06, pages 3–12, Washington, DC, USA, 2006. IEEE Computer Society.

[44] B. Liblit, A. Begel, and E. Sweetser. Cognitive perspectives on the role of naming in computer programs. In *Annual Psychology of Programming Workshop*, 2006.

[45] C.-Y. Lin. Rouge: A package for automatic evaluation of summaries. In *Text Summarization Branches Out: Proceedings of the ACL-04 Workshop*, pages 74–81, 2004.

[46] C. J. Maddison and D. Tarlow. Structured generative models of natural source code. *arXiv preprint arXiv:1401.0514*, 2014.

[47] E. Mays, F. J. Damerau, and R. L. Mercer. Context based spelling correction. *Information Processing and Management*, 27(5):517–522, 1991.

[48] G. A. Miller. The magical number seven, plus or minus two: some limits on our capacity for processing information. *Psychological review*, 63(2):81, 1956.

[49] D. Movshovitz-Attias and W. W. Cohen. Natural language models for predicting programming comments. In *Proc of the ACL*, 2013.

[50] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. *Software Engineering, IEEE Transactions on*, 38(1):5–18, 2012.

[51] N. Nagappan and T. Ball. Using software dependencies and churn metrics to predict field failures: An empirical case study. In *ESEM*, pages 364–373, 2007.

[52] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, A. Tamrawi, H. V. Nguyen, J. Al-Kofahi, and T. N. Nguyen. Graph-based pattern-oriented, context-sensitive source code completion. In *ACM/IEEE International Conference on Software Engineering (ICSE)*. IEEE, 2012.

[53] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. A statistical semantic language model for source code. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 532–542. ACM, 2013.

[54] M. Ohba and K. Gondow. Toward mining concept keywords from identifiers in large software projects. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 1–5. ACM, 2005.

[55] Oracle. Code Conventions for the Java Programming Language. `http://www.oracle.com/technetwork/java/codeconv-138413.html`, 1999. Visited September 2, 2013.

[56] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu. BLEU: a method for automatic evaluation of machine translation. In *Association for Computational Linguistics (ACL)*, pages 311–318, 2002.

[57] R. Pike. Go at Google. `http://talks.golang.org/2012/splash.slide`, 2012. Visited September 9, 2013.

[58] Pylint-Contributors. Pylint – code analysis for Python. `http://www.pylint.org/`, 2013. Visited September 9, 2013.

[59] V. Rajlich and P. Gosavi. Incremental change in object-oriented programming. *Software, IEEE*, 21(4):62–69, 2004.

[60] D. Ratiu and F. Deißenböck. From reality to programs and (not quite) back again. In *IEEE International Conference on Program Comprehension (ICPC)*, pages 91–102. IEEE, 2007.

[61] P. C. Rigby and C. Bird. Convergent software peer review practices. In *Proceedings of the the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE)*. ACM, 2013.

[62] R. Robbes and M. Lanza. How program history can improve code completion. In *Automated Software Engineering (ASE)*, pages 317–326. IEEE, 2008.

[63] M. Robillard, R. Walker, and T. Zimmermann. Recommendation systems for software engineering. *Software, IEEE*, 27(4):80–86, 2010.

[64] G. v. Rossum, B. Warsaw, and N. Coghlan. PEP 8–Style Guide for Python Code. `http://www.python.org/dev/peps/pep-0008/`, 2013. Visited September 8, 2013.

[65] C. Simonyi. Hungarian notation. `http://msdn.microsoft.com/en-us/library/aa260976(VS.60).aspx`, 1999. Visited September 2, 2013.

[66] E. Soloway and K. Ehrlich. Empirical studies of programming knowledge. *Software Engineering, IEEE Transactions on*, (5):595–609, 1984.

[67] W. Strunk Jr and E. White. *The Elements of Style*. Macmillan, New York, 3rd edition, 1979.

[68] A. Takang, P. Grubb, and R. Macredie. The effects of comments and identifier names on program comprehensibility: an experiential study. *Journal of Program Languages*, 4(3):143–167, 1996.

[69] A. A. Takang, P. A. Grubb, and R. D. Macredie. The effects of comments and identifier names on program comprehensibility: an experimental investigation. *J. Prog. Lang.*, 4(3):143–167, 1996.

[70] G. Uddin, B. Dagenais, and M. P. Robillard. Analyzing temporal API usage patterns. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 456–459. IEEE Computer Society, 2011.

[71] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang. Mining succinct and high-coverage API usage patterns from source code. In *Proceedings of the Tenth International Workshop on Mining Software Repositories*, pages 319–328. IEEE Press, 2013.

[72] X. Wang, L. Pollock, and K. Vijay-Shanker. Automatic segmentation of method code into meaningful blocks to improve readability. In *Working Conference on Reverse Engineering*, pages 35–44. IEEE, 2011.

[73] Wikipedia. Coding Conventions. `http://en.wikipedia.org/wiki/Coding_conventions`.

[74] H. P. Young. The economics of convention. *The Journal of Economic Perspectives*, 10(2):105–122, 1996.

[75] C. Zhang, J. Yang, Y. Zhang, J. Fan, X. Zhang, J. Zhao, and P. Ou. Automatic parameter recommendation for practical api usage. In *Proceedings of the 34th International Conference on*

Software Engineering, pages 826–836. IEEE Press, 2012.

[76] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. MAPO: Mining and recommending API usage patterns. In *ECOOP 2009–Object-Oriented Programming*, pages 318–343. Springer, 2009.