Understandability Smells in Regular Expressions

(blinded for review)

***Abstract*—The regular expression (regex) is a powerful tool employed in a large variety of tasks. Prior work has shown that regexes can be very complex and that it could be difficult for developers to compose and understand them.**

In order to identify the code smells which impact comprehension, we conducted an empirical study on dozens of pairs of behaviorally equivalent but syntactically different regexes among 180 participants and evaluated the understandability of various regex language features. We found that some regex expression representations are more understandable than others. For example, our study showed that in expressing one or more repetitions of a character the plus (e.g., `a+') operator is more understandable than the kleene star operator (e.g., `aa*'). We further analyzed regexes in GitHub to find the community standards, or the common usages of various features. Finally, we identified smelly and non-smelly regex representations based on a combination of community standards and understandability metrics. We also formed recommendations on how to transform regexes to enhance comprehension.

I. Introduction

Regular expressions (regexes) are used fundamentally in string searching and substitution tasks, such as word searching, text editing, file parsing, user input validation, and access controls. More advanced usages of regexes can be seen in search engines [1], database querying [2], and network security [3]–[5].

Recent research has suggested that regular expressions are hard to understand, hard to compose, and error prone [6]. Given their frequent appearances in software source code and the difficulty of working with them, some effort has been put into easing the burden on developers by providing environments that make regexes easier to understand. Some tools provide debugging environments which explain string matching results and highlight the parts of regex patterns which match a certain string [7], [8]. Other tools present graphical representations (e.g., finite automata) of the regular expressions [9], [10]. Still others can automatically generate strings according to the given regular expessions [11], [12] or automatically generate regexes according to the given list of strings [13], [14]. The commonality of such tools provides evidence that people need help with regex composition and understandability.

In software engineering, code smells have been found to hinder understandability of source code [15], [16]. Once removed through refactoring, the code becomes more understandable, easing the burden on the programmer. In regular expressions, such code smells have not yet been defined, perhaps in part because it is not clear what makes a regex difficult to understand or maintain.

In regular expressions as in source code, there are multiple ways to express the same semantic concept. For example, the regex, `aa*` matches an "a" followed by zero or more "a", and is is equivalent to `a+` , which matches one or more "a". That is, both regexes match the same *language* but are expressed differently. What is not clear is which representation, `aa*` or `a+`, is more easily understood.

In this work, we focus on identifying regex comprehension smells. We identify equivalence classes of regex representations that provide options for concepts such as double-bounds in repetitions (e.g., `a{1,2}`, `a|aa`) or character classes (e.g., `[0-9]`, `[\d]`). Based on an empirical study measuring regex comprehension on 35 pairs of regexes using 180 participants, as well as an empirical study of nearly 14,000 regexes and their features, we identify smelly and non-smelly regex representations. For example, `aa*` is more smelly than `a+`, based on feature usage frequency in source code (conformance to community standards) and understandability.

Our contributions are:

- An approach to, and evaluation with. 180 participants for studying regex understandability,
- Identification of equivalence classes for regular expressions,
- Identification of smelly and non-smelly regex representations to optimize 1) understandability and 2) conformance to community standards, backed by empirical evidence.

To our knowledge, this is the first work to explore regex comprehension and regex smells. We approach the problem of identifying preferred regex representations by looking at thousands of regexes in Python projects and measuring comprehension using human participants.

II. Research Questions

To explore regex comprehension and identify smells, we answer the following research questions:

RQ1: *Which regex representations are most understandable?* To answer RQ1, we conduct a study in which programmers are presented with a regex and asked comprehension questions about its matching behavior. By comparing accuracy between regexes that match the same language but are expressed differently (e.g., `tri[a-f]3` and `tri(a|b|c|d|e|f)3`), we can measure understandability and identify code smells. We also explore at the relationship between regex comprehension and regex size, as measured by string length and the DFA built for each regex.

This analysis requires identification of equivalence classes for regexes. By inspecting nearly 14,000 regexes extracted from Python projects in a publicly available dataset [17], we formed an initial set of five equivalence classes to explore.

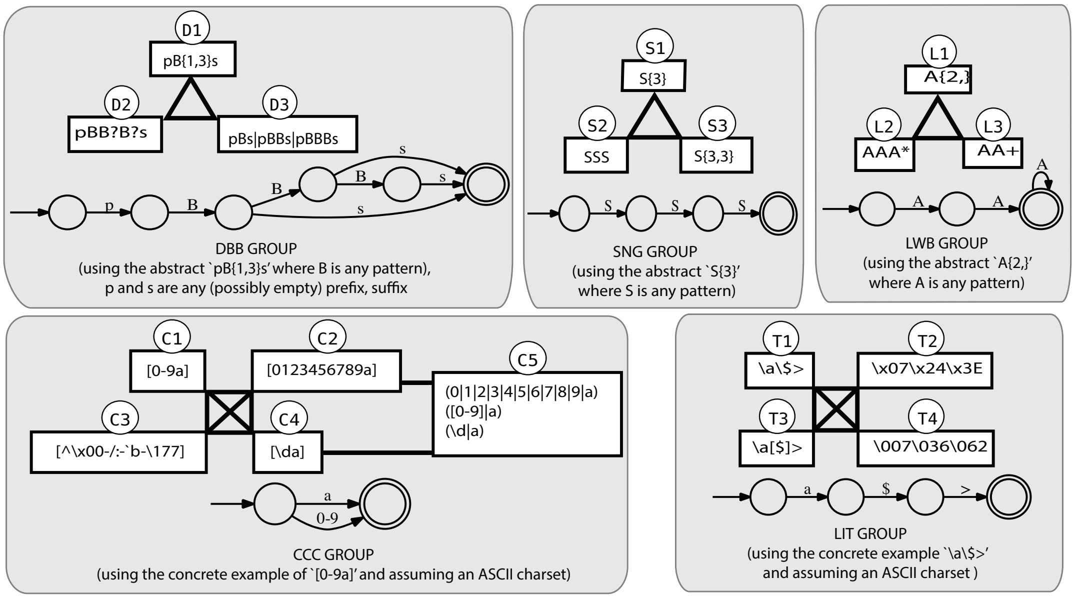


Fig. 1. Equivalence classes with various representations. DBB = Double-Bounded, SNG = Single Bounded, LWB = Lower Bounded, CCC = Custom Character Class and LIT = Literal. We use concrete regexes in the representations for illustration. However, the A's in the LWB group (or B's in DBB group, S's in SNG group, and so forth) abstractly represent any pattern that could be operated on by a repetition modifier (e.g., literal characters, character classes, or groups). The same is true for the literals used in all the representations.

RQ2: *Which regex representations have the strongest community support based on frequency?* To answer RQ2, we explore the publicly available regex dataset [17] and use the presence and absence of language features as a proxy for community support, where more frequently-used features are assumed to be more understandable.

RQ3: *Which regex representations are most desirable (i.e., least smelly) based on both community support and understandability?* Based on RQ1 and RQ2, we identify smelly and non-smelly regex features based on a combination of comprehension metrics and community support.

Next, we present the equivalence classes, analysis and results for each RQ, and a unified discussion.

III. Equivalence Classes

To explore understandability, we defined an initial set of equivalence classes for regexes. Using the publicly available behavioral clusters of Python regexes [17], we manually identified several representations that appeared in many of the larger clusters. While not a complete set of equivalence classes, this is the first work to explore regex understandability, and these equivalence classes provide an initial testbed for exploration.

Figure 1 shows five equivalence classes in grey boxes and semantically equivalent *representations* in white boxes with identifiers in white circles. For example, LWB is an equivalence class with representations L1, L2, and L3. Regexes `AAA*` and `AA+` map to L2 and L3, respectively. Next, we describe each equivalence class group.

A. Custom Character Class Group

The Custom Character Class (CCC) group has regex representations that use the custom character class language feature or can be represented by such a feature. A custom character class matches a set of alternative characters. For example, the regex `c[ao]t` will match strings "cat" and "cot" because, between the `c` and `t`, there is a custom character class, `[ao]`, that specifies either `a` or `o` (but not both) must be selected. We use the term *custom* to differentiate these classes from the default character classes, : `\d`, `\D`, `\w`, `\W`, `\s`, `\S` and `.`, provided by most regex libraries, though the default classes can be encapsulated in a custom character class.

C1: Any pattern that contains a (non-negative) custom character class with a range feature like `[a-f]` as shorthand for all of the characters between 'a' and 'f' (inclusive) belongs to C1.
C2: Any pattern that contains a (non-negative) custom character class without any shorthand representations, specifically ranges or defaults (e.g., `[012]` is in C2, but `[0-2]` is not).
C3: Any pattern with a character class expressed using negation, indicated by a caret (i.e., `^`) followed by a custom character class. For example, the pattern `[^ao]` matches every character *except* `a` or `o`.
C4: Any pattern using a default character class such as `\d` or `\W` within a (non-negative) character class.
C5: These can be transformed into custom character classes by removing the ORs and adding square brackets (e.g., `(\d|a)` in C5 is equivalent to `[\da]` in C4). All custom character classes expressed as an

OR of length-one sequences, including defaults or other custom classes, are in C5[1].

Note that a pattern can belong to multiple representations. For example, `[a-f\d]` belongs to both C1 and C4.

B. Double-Bounded Group

The Double-Bounded (DBB) group contains all regex patterns that use some repetition defined by a (non-equal) lower and upper boundary. For example, `pB{1,3}s` represents a `p` followed by one to three sequential `B` patterns, then followed by a single `s`. This matches "pBs", "pBBs", and "pBBBs".

D1: Any pattern that uses the curly brace repetition with a lower and upper bound, such as `pB{1,3}s`.

D2: Any pattern that uses the questionable (i.e., `?`) modifier implies a lower-bound of zero and an upper-bound of one (and hence is double-bounded).

D3: Any pattern that has a repetition with a lower and upper bound and is expressed using ORs (e.g., `pB{1,3}s` becomes `pBs|pBBs|pBBs` by expanding on each option in the boundaries).

Patterns can belong to multiple representations (e.g., `(a|aa)X?Y{2,4}` belongs to all three nodes: `Y{2,4}` maps to D1, `X?` maps to D2, and `(a|aa)` maps to D3).

C. Literal Group

In the Literal (LIT) group, all patterns that are not purely default character classes must use literal tokens. We use the ASCII charset in which all characters can be expressed using hex and octal codes such as `\xF1` and `\0108`, respectively.

T1: Patterns that do not use any hex, wrapped, or octal characters, but use at least one literal character. Special characters are escaped using backslash.

T2: Any pattern using a hex token, such as `\x07`.

T3: Any pattern with a literal wrapped in square brackets. This style is used most often to avoid using a backslash for a special character in the regex language, for example, `[{]` which must otherwise be escaped like `\{`.

T4: Any pattern using an octal token, such as `\007`.

Patterns often fall in multiple of these representations, for example, `abc\007` includes literals `a`, `b`, and `c`, and also octal `\007`, thus belonging to T1 and T4. Not all transformations are possible in this group, for example, if a hex representation is used for a character not on the keyboard, a transformation to T1 or T3 is infeasible.

D. Lower-Bounded Group

The Lower-Bounded (LWB) group contains patterns that specify only a lower boundary on repetitions. This can be expressed using curly braces with a comma after the lower bound but no upper bound, for example `A{2,}` which will match "AA", "AAA", "AAAA", and any number of A's greater or equal to 2. In Figure 1, we chose the lower bound repetition threshold of 2 for illustration; in practice this could be any number, including zero.

L1: Any pattern using this curly braces-style lower-bounded repetition belongs to node L1.

L2: Any pattern using the kleene star, which means zero-or-more repetitions.

L3: Any pattern using the additional repetition, for example `T+` which means one or more `T`'s.

Patterns often fall into multiple nodes in this equivalence class. For example, with `A+B*`, `A+` maps it to L3 and `B*` maps it to L2.

E. Single-Bounded Group

The Single-Bounded (SNG) equivalence class contains three representations in which each regex has a fixed number of repetitions of some element. The important factor distinguishing this group from DBB and LWB is that there is a single finite number of repetitions, rather than a bounded range on the number of repetitions (DBB) or a lower bound on the number of repetitions (LWB).

S1: Any pattern with a single repetition boundary in curly braces belongs to S1. For example, `S{3}`, states that S appears exactly three times in sequence.

S2: Any pattern that is explicitly repeated two or more times and could use repetition operators.

S3: Any pattern with a double-bound in which the upper and lower bounds are same belong to S3. For example, `S{3,3}` states `S` appears a minimum of 3 and maximum of 3 times.

The pattern `fa[lmnop][lmnop][lmnop]` is a member of S2 as `[lmnop]` is repeated three times, and it could be transformed to `fa[lmnop]{3}` in S1 or `fa[lmnop]{3,3}` in S3.

Missing: explain the process of getting DFA size Todo: resize Fig 1 to proper space

IV. Understandability Study (RQ1)

This study presents programmers with regexes and asks comprehension questions. By comparing the understandability of semantically equivalent regexes that match the same language but have different syntactic representations, we aim to identify understandability code smells. This study was implemented on Amazon's Mechanical Turk with 180 participants. A total of 35 pairs of regex were evaluated. Each regex pattern was evaluated by 30 participants.

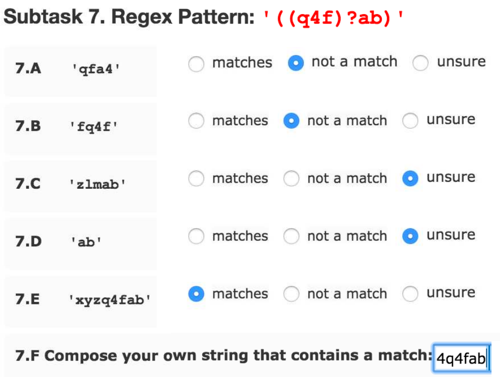
A. Metrics

We measure the understandability of regexes using two complementary metrics, *matching* and *compostition*.

Matching: Given a pattern and a set of strings, a participant determines by inspection which strings will be matched by the pattern. There are four possible responses for each string, *matches*, *not a match*, *unsure*, or blank. An example from our study is shown in Figure 2.

The percentage of correct responses, disregarding blanks and unsure responses, is the matching score. For example,

[1] An OR cannot be directly negated, it there is no edge between C3 and C5



Subtask 7. Regex Pattern: '((q4f)?ab)'

7.A 'qfa4' ○ matches ● not a match ○ unsure

7.B 'fq4f' ○ matches ● not a match ○ unsure

7.C 'zlmab' ○ matches ○ not a match ● unsure

7.D 'ab' ○ matches ○ not a match ● unsure

7.E 'xyzq4fab' ● matches ○ not a match ○ unsure

7.F Compose your own string that contains a match: 4q4fab

Fig. 2. Questions from one pattern in one HIT

TABLE I
MATCHING METRIC EXAMPLE

String	`'RR*'`	Oracle	P1	P2	P3	P4
1	"ARROW"	✓	✓	✓	✓	✓
2	"qRs"	✓	✓	✗	✗	?
3	"R0R"	✓	✓	✓	?	-
4	"qrs"	✗	✓	✗	✓	-
5	"98"	✗	✗	✗	✗	-
	Score	1.00	0.80	0.80	0.50	1.00

✓ = match, ✗= not a match, ? = unsure, – = left blank

consider regex pattern `'RR*'`, tge five strings shown in Table I, and the responses from four participants in the *P1*, *P2*, *P3* and *P4* columns. The *Oracle* indicates the first three strings match and the last two do not. *P1* answers correctly for the first three strings and the fifth, but incorrectly on the fourth, so the matching score is $4/5 = 0.80$. *P2* incorrectly thinks that the second string is not a match, so the score is also $4/5 = 0.80$. *P3* marks 'unsure' for the third string and so the total number of attempted matching questions is 4. *P3* is incorrect about the second and fourth string, so they score $2/4 = 0.50$. For *P4*, we only have data for the first and second strings, since the other three are blank. *P4* marks 'unsure' for the second string so only one matching question has been attempted; the matching score is $1/1 = 1.00$.

Blanks were incorporated into the metric because questions were occasionally left blank in the study. Unsure responses were provided as an option so not to bias the results through blind guessing. These situations did not occur very frequently. Out of 1800 questions (180 participants * 10 questions each), only 1.8%(32) were impacted by a blank or unsure response (never more than four out of 30 responses per pattern).

Composition: Given a pattern, a participant composes a string they think it matches (question 7.F in Figure 2). If the participant is accurate, a composition score is 1, otherwise 0. For example, given the pattern `(q4fab|ab)` from our study, the string, "xyzq4fab" matches and gets a score of 1, but the string, "acb" does not match and gets a score of 0.

To determine a match, each pattern was compiled using the *java.util.regex* library. A *java.util.regex.Matcher* `m` object was created for each composed string using the compiled pattern. If `m.find()` returned true, then that composed string was a match and scored 1, otherwise it scored 0.

B. Design

We implemented this study on Amazon's Mechanical Turk (MTurk), a crowdsourcing platform where requesters create human intelligence tasks (HITs) for completion by workers.

1) Worker Qualification: Workers qualified to participate by answering questions regarding some basics of regex knowledge. These questions were multiple-choice and asked the worker to analyze the following patterns: `a+`, `(r|z)`, `\d`, `q*`, and `[p-s]`. To pass the qualification test, workers had to answer four of the five questions correctly.

2) Tasks: Using the patterns in the corpus as a guide, we created 60 regex patterns that were grouped into 26 semantic equivalence groups. There were 18 groups with two regexes that target various edges in the equivalence classes. The other eight semantic groups had three regexes each, forming 42 total pairs. These semantic groups were intended to explore edges in the equivalence classes. In this way, we can draw conclusions by comparing between representations since the regexes evaluated were semantically equivalent.

To form the semantic groups, we took a regex from the corpus, matched it to a representation in Figure 1, trimmed it down so it contained little more than just the feature of interest, and then created other regexes that are semantically equivalent but belong to other nodes in the equivalence class. For example, a semantic group with regexes `((q4f){0,1}ab`, `((q4f)?ab)`, and `(q4fab|ab)` belong to D1, D2, and D3, respectively. A group with regexes `([0-9]+)\.([0-9]+)` and `(\d+)\.(\d+)` is intended to evaluate the edge between C1 and C4. We note that if we only used regexes from the corpus, we would have had regexes with different semantics at each node, or with additional language features, which would make the comparisons of the targeted features difficult.

For each of the 26 semantic groups, we created five strings for the study, where at least one matched and at least one did not match. These were used to compute the matching metric.

Once all the patterns and matching strings were collected, we created tasks for the MTurk participants as follows: randomly select a pattern from 10 of the 26 semantic groups. Randomize the order of these 10 patterns, as well as the order of the matching strings for each pattern. After adding a question asking the participant to compose a string that each pattern matches, this creates one task on MTurk, such as the example in Figure 2. This process was completed until each of the 60 regexes appeared in 30 HITs, resulting in a total of 180 total unique HITs.

3) Implementation: Workers were paid $3.00 for successfully completing a HIT, and were only allowed to complete one HIT. The average completion time for accepted HITs was 682 seconds (11 mins, 22 secs). A total of 54 HITs were rejected: 48 had too many blank responses, four were double-submissions by the same worker, one did not answer the composition questions, and one was missing data for

	P_CR1	P_CR2	CR1	CR2	AvgAccur1	AvgAccur2	wilcox_sig	CompAccur1	CompAccur2	pro
13	C1	C2	tri[a-f]3	tri[abcdef]3	0.94	0.93		83.33	83.33	
15	C1	C5	tri[a-f]3	tri(a|b|c|d|e|f)3	0.94	0.86		83.33	80.00	
16	C1	C2	no[w-z]5	no[wxyz]5	0.93	0.87		86.67	86.67	
18	C1	C3	no[w-z]5	no(w|x|y|z)5	0.93	0.94		86.67	96.67	
31	C1	C4	([0-9]+)\.([0-9]+)	(\d+)\.(\d+)	0.90	0.94		83.33	93.33	
32	C1	C4	xg1([0-9]{1,3})%	xg1(\d{1,3})%	0.83	0.81		76.67	66.67	
33	C1	C4	[a-f]([0-9]+)[a-f]	[a-f](\d+)[a-f]	0.91	0.83		80.00	70.00	
34	C1	C4	&([A-Za-z0-9_]+);	&(\w+);	0.82	0.83		56.67	66.67	
35	C1	C4	1q[A-Za-z0-9_][A-Za-z0-9_]	1q\w\w	0.86	0.78		83.33	70.00	
36	C1	C4	tuv[A-Za-z0-9_]	tuv\w	0.89	0.86		83.33	70.00	
14	C2	C5	tri[abcdef]3	tri(a|b|c|d|e|f)3	0.93	0.86		83.33	80.00	
17	C2	C5	no[wxyz]5	no(w|x|y|z)5	0.87	0.94		86.67	96.67	
30	C2	C4	[\t\r\f\n]	[\s]	0.83	0.92	.	3.33	0.00	
28	C3	C4	[^0-9A-Za-z]	[\W_]	0.65	0.61		46.67	53.33	
29	C3	C4	[^0-9]	[\D]	0.58	0.73		63.33	73.33	
19	C4	C5	([}{])	(\{|\})	0.79	0.70		50.00	86.67	**
22	C4	C5	([:;])	(:|;)	0.81	0.94		46.67	46.67	
8	D1	D2	((q4f){0,1}ab)	((q4f)?ab)	0.83	0.79		50.00	40.00	
9	D1	D3	((q4f){0,1}ab)	(q4fab|ab)	0.83	0.84		50.00	60.00	
10	D1	D3	(dee(do){1,2})	(deedo|deedodo)	0.85	0.90		66.67	63.33	
11	D1	D2	(dee(do){1,2})	(deedo(do)?)	0.85	0.77		66.67	60.00	
7	D2	D3	((q4f)?ab)	(q4fab|ab)	0.79	0.84		40.00	60.00	
12	D2	D3	(deedo(do)?)	(deedo|deedodo)	0.77	0.90	*	60.00	63.33	
38	L2	L3	zaa*	za+	0.87	0.91		70.00	50.00	
39	L2	L3	RR*	R+	0.86	0.92		66.67	66.67	
25	S1	S2	%([0-9A-Fa-f]{2})	%([0-9a-fA-F][0-9a-fA-F])	0.78	0.73		50.00	60.00	
26	S1	S2	&d([aeiou]{2})z	&d([aeiou][aeiou])z	0.91	0.95		83.33	83.33	
27	S1	S2	fa[lmnop]{3}	fa[lmnop][lmnop][lmnop]	0.87	0.88		83.33	73.33	
1	T1	T4	xyz[_\[\]‘\^\\]	xyz[\0133-\0140]	0.78	0.71		86.67	33.33	**
3	T1	T2	xyz[_\[\]‘\^\\]	xyz[\x5b-\x5f]	0.78	0.79		86.67	56.67	*
4	T1	T4	t[:;]+p	t[\072-\073]+p	0.94	0.90		80.00	70.00	
6	T1	T2	t[:;]+p	t[\x3a-\x3b]+p	0.94	0.89		80.00	63.33	
20	T1	T4	(\{|\})	([\0175\0173])	0.70	0.54	.	86.67	30.00	**
21	T1	T4	([}{])	([\0175\0173])	0.79	0.54	**	50.00	30.00	
23	T1	T4	(:|;)	([\072\073])	0.94	0.66	**	46.67	23.33	
24	T1	T4	([:;])	([\072\073])	0.81	0.66	.	46.67	23.33	
40	T1	T3	(\$\{)\d+(:[^}]+\})	([$][{])\d+(:[^}]+[}])	0.82	0.75		63.33	73.33	
41	T1	T3	t\.\$+\d+*	t[.][$]+\d+[*]	0.89	0.94		56.67	73.33	
42	T1	T3	\{\$(\d+\.\d)\}	[{][$](\d+[.]\d)[}]	0.93	0.89		70.00	66.67	
2	T2	T4	xyz[\x5b-\x5f]	xyz[\0133-\0140]	0.79	0.71		56.67	33.33	
5	T2	T4	t[\x3a-\x3b]+p	t[\072-\073]+p	0.89	0.90		63.33	70.00	

3 questions. Rejected HITs were returned to MTurk to be completed by others.

4) Participants: In total, there were 180 participants. A majority were male (83%). Most had at least an Associates degree (72%), were at least somewhat familiar with regexes (87%), and have prior programming experience (84%).

C. Analysis

For each of the 180 HITs, we computed a matching and composition score for each of the 10 regexes, using the metrics described in Section IV-A. This allowed us to compute and then average 26-30 values for each metric for each of the 60 regexes (fewer than 30 values were used if all the responses in a matching question were a combination of blanks and unsure).

Of the original 42 pairs, we report scores for only 35 pairs. Due to design flaws, seven pairs were dropped. In one, the regexes evaluated, `\..*` and `\.+`. are not semantically equivalent (the former is missing an escape and should be `\.\.*`). In six, the transformations were from multiple equivalence classes and confounded. For example, `([\072\073])` is in C2 and T4 and `(:|;)` in C5 and T1; it was not clear if any differences were due to the CCC or LIT groups, or both. However, `([:;])`, could be compared with each, since it is a member of T1 and C2, so comparing it to `([\072\073])` evaluates T1 and T4, and comparing to `(:|;)` evaluates C2 and C5. In the end, we covered 14 edges from Figure 1.

To compute DFA size, we **TODO.NOW: add details on how we built the DFAs for each regex**. We use the original set of 60 regexes for the size analysis. **TODO.NOW: justify this either intuitively or using the ANOVA**

D. Results

Todo: update Table and its description, add annova and correlation analysis about representation, dfa size, and string length

Missing: Heatmap about the positive and negative correlation of dfa size or string length with accuracy

Table **??** presents the results. *Index* enumerates the edges evaluated in this experiment (per Figure 1), *Nodes* lists the representations, *Pairs* shows the number of comparisons, *Example Preferred Regex* shows a regex from the preferred node (bolded in *Pairs* column), *Match1* and *Match2* give the matching scores for the first and second representations, respectively, and $H_0 : \mu_{match1} = \mu_{match2}$ uses the Mann-Whitney test of means to compare the matching scores, and presents the p-values. The last three columns list the average composition scores for the representations and the composition p-value.

For example, consider row E5 in Table **??**. One pair of regexes was `RR*` and `R+` in L2 and L3, respectively, with average matching scores of 86% and 92% and average composition scores of 97% and 100%, respectively. Thus, the community found `R+` from L3 more understandable. This is not the only regex pair in E5, however, The other is `zaa*` and `za+`. In aggregate, considering both regex pairs, the overall matching average for the regexes belonging to L2 was 0.86 and 0.91 for L3. The overall composition score for L2 was 0.97 and 1.00 for L3. The community found L3 to be more understandable than L2, from the perspective of both metrics, suggesting that L2 is generally smelly, though the differences are not significant.

In E1 through E4, there is a statistically significant difference between the representations for at least one of the metrics with $\alpha = 0.10$. These represent the strongest evidence for code smells, suggesting that T4, D2, and C2 are less understandable.

Recall that participants were able to select *unsure* for whether a string is matched by a pattern. From a comprehension perspective, this indicates some level of confusion. For each pattern, we counted the number of responses containing at least one unsure. Overall, the highest number of unsure responses came from T4 and T2, which have octal and hex representations of characters. The least number of unsure responses were in L3 and D3. These results mirror the understandability analysis.

V. Community Support Study (RQ2)

The goal of this study is to understand how frequently each of the regex representations appears in source code, as a way to identify community-based smells.

A. Artifacts

We analyzed an existing corpus of regexes collected from Python code in GitHub projects [17]. This dataset has 13,597 distinct (non-duplicate) regex patterns from 1,544 projects.

This corpus was created by analyzed static invocations to the Python `re` library. Consider the Python snippet:

```
r1 = re.compile('(0|−?[1−9][0−9]*)$', re.MULTILINE)
```

The *function* `re.compile` returns a regex object `r1`. The *pattern*, `(0|-?[1-9][0-9]*)$,` defines what strings will be matched and the *flag* `re.MULTILINE` modifies the matching behavior. This particular regex will match strings with a zero at the end of a line, or an integer at the end of a line (i.e., the `-?` sequence indicates the integer may be negative).

B. Metrics

We measure community support by matching regexes in the corpus to representations in Figure 1 and counting the *patterns* and *projects*. A regex can belong to multiple representations and to multiple projects since the corpus tracked duplicates.

C. Analysis

To match patterns to representations, we used the PCRE parser or treated the regexes as token streams, depending on the characteristics of the representation. Our analysis code is available on GitHub[2]. Next, we describe the process in detail:

1) Presence of a Feature: For representations that require a particular feature, we used the PCRE parser to decide membership. This applies to D1 (double-bounded repetition with different bounds), D2 (question-mark), S1 (single-bounded repetition), S3 (double-bounded repetition with the same bounds), L1 (lower-bound repetition), L2 (kleene star), L3 (add repetition), and C3 (negated custom character class).

2) Features and Pattern: Identifying D3 requires an OR containing at least two entries with a sequence present in one entry repeated N times and the same sequence present in another entry repeated N+1 times. We first looked for a sequence of N repeating groups with an OR-bar (ie. `|`) next to them on a side. This produced a list of 113 candidates which we narrowed down manually to 10 actual members.

T2 requires a literal with a hex structure. T4 requires a literal with a Python-style octal structure. T3 requires that a single literal character is wrapped in a custom character class (a member of T3 is always a member of C2). T1 requires that no characters are wrapped in brackets or are hex or octal characters, which matches over 91% of the patterns analyzed.

3) Token Stream : S2 requires a repeated element. This element could be a character class, a literal, or a collection of things encapsulated in parentheses; rather than a parser, we used a token stream to identify it. C1 requires that a non-negative character class contains a range. C2 requires that there exists a custom character class that does not use ranges or defaults. C4 requires the presence of a default character class within a custom character class. C5 requires an OR of length-one sequences (literal characters or any character class).

D. Results

Table II presents the results. *Node* references Figure 1, *description* briefly describes the representation, *example* provides a regex from the corpus. *nPatterns* counts the patterns that belong to the representation, followed by the percent of patterns out of 13,597. *nProjects* counts the projects that contain a regex belonging to the representation, followed by

[2] http://tinyurl.com/jmeeytk (this is a git repo - not anonymized)

TABLE II
HOW FREQUENTLY IS EACH ALTERNATIVE EXPRESSION STYLE USED?

Node	Description	Example	nPatterns	% patterns	nProjects	% projects
C1	char class using ranges	`'^[1-9][0-9]*$'`	2,479	18.2%	810	52.5%
C2	char class explicitly listing all chars	`'[aeiouy]'`	1,903	14.0%	715	46.3%
C3	any negated char class	`'[^A-Za-z0-9.]+'`	1,935	14.2%	776	50.3%
C4	char class using defaults	`'[-+\d.]'`	840	6.2%	414	26.8%
C5	an OR of length-one sub-patterns	`'(@|<|>|-|!)'`	245	1.8%	239	15.5%
D1	curly brace repetition like {M,N} with M¡N	`'^x{1,4}$'`	346	2.5%	234	15.2%
D2	zero-or-one repetition using question mark	`'^http(s)?://'`	1,871	13.8%	646	41.8%
D3	repetition expressed using an OR	`'^(Q|QQ)\<(.+)\>$'`	10	.1%	27	1.7%
T1	no HEX, OCT or char-class-wrapped literals	`'get_tag'`	12,482	91.8%	1,485	96.2%
T2	has HEX literal like `\xF5`	`'[\x80-\xff]'`	479	3.5%	243	15.7%
T3	has char-class-wrapped literals like [$]	`'[$][{]\d+:([^}]+)[}]'`	307	2.3%	268	17.4%
T4	has OCT literal like `\0177`	`'[\041-\176]+:$'`	14	.1%	37	2.4%
L1	curly brace repetition like {M,}	`'(DN)[0-9]{4,}'`	91	.7%	166	10.8%
L2	zero-or-more repetition using kleene star	`'\s*(#.*)?$'`	6,017	44.3%	1,097	71.0%
L3	one-or-more repetition using plus	`'[A-Z][a-z]+'`	6,003	44.1%	1,207	78.2%
S1	curly brace repetition like {M}	`'^[a-f0-9]{40}$'`	581	4.3%	340	22.0%
S2	explicit sequential repetition	`'ff:ff:ff:ff:ff:ff'`	3,378	24.8%	861	55.8%
S3	curly brace repetition like {M,M}	`'U[\dA-F]{5,5}'`	27	.2%	32	2.1%

the percentage of projects out of 1,544. For example, D1 appears in 346 (2.5%) of the patterns but only 234 (15.2%) of the projects. In contrast, T3 appears in 39 *fewer* patterns but 34 *more* projects, indicating that D1 is more concentrated in a few projects and T3 is more widespread across projects.

The pattern frequency is our guide for setting the community standards. For example, since C1 is more prevalent than C2 in both patterns and projects, we could say that C2 is smelly since it could better conform to community standards if expressed as C1. Based on patterns alone, the winning representations per equivalence class are C1, D2, T1, L2, and S2. With one exception, these are the same for recommendations based on projects; L3 appears in more projects than L2, so it is unclear which is smelly.

We note that our criteria for membership in a representation may overestimate the opportunities for refactoring to address the smells. For example, `[a-f]` in C1 cannot be refactored to C4 since there does not exist a default character class for that range of characters. A finer-grained analysis is needed to identify actual refactoring opportunities from the smells.

VI. DESIRABLE REPRESENTATIONS (RQ3)

To determine the overall trends in the data, we created and compared total orderings on the representations in each equivalence class (Figure 1) with respect to the understandability (RQ1) and community standards (RQ2) metrics.

A. Analysis

Total orderings were achieved by building directed graphs with the representations as nodes and edge directions determined by the metrics: matching and composition for understandability; patterns and projects for community standards. Within each graph, a topological sort created total orderings.

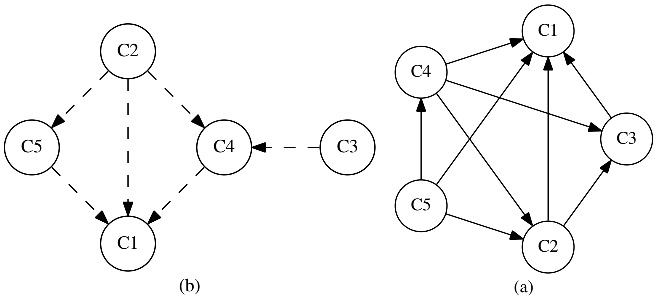


Fig. 3. Trend graphs for the CCC equivalence graph: (a) represent the understandability analysis (RQ1) and (b) represent the artifact analysis (RQ2)

The graphs for understandability are based on Table **??** and for community support are based on Table II and the graphs.

1) Building the Graphs: In the community standards graph, a directed edge $\overrightarrow{C2C1}$ is used when nPatterns(C1) > nPatterns(C2) *and* nProjects(C1) > nProjects(C2). When there is a conflict between nPatterns and nProjects, as is the case between L2 and L3, an undirected edge $\overline{L2L3}$ is used. The same process is used to identify directed and undirected edges based on community standards metrics. For example, Figure 3 shows the graphs for the CCC group; Figure 3a is based on understandability and Figure 3b is based on community standards. Nodes with fewer incoming edges are more smelly and nodes with more incoming edges are less smelly.

2) Topological Sorting: Once the graphs are built for each equivalence class and each set of metrics, we apply a modified version of Kahn's topological sorting algorithm to obtain a total ordering. One downside to Kahn's algorithm is that the total ordering is not unique and often multiple nodes with similar properties (e.g., no incoming edges) could be considered tied. Thus, we mark ties in order to identify when a

TABLE III
TOPOLOGICAL SORTING, WITH THE LEFT-MOST POSITION BEING HIGHEST (NON-SMELLY) AND THE RIGHT-MOST BEING MOST SMELLY

	Understandability	Community
CCC	C1 C5 C4 C2 C3	C1 C3 C2 C4 C5
DBB	D3 D1 D2	D2 D1 D3
LBW	L3 L2	L3 L2 L1
SNG	S2 S1	S2 S1 S3
LIT	T1 T2 T4 T3	T1 T3 T2 T4

tiebreaker is needed. Breaking ties on the community standards graph involves choosing the representation that appears in a larger number of projects, as it is more widespread across the community. Breaking ties in the understandability graph uses the RQ1 results. Based on Table **??**, we compute the average metrics for all instances of each representation. For example, C4 appears in E5, E12 and E13 with an overall average matching score of 0.81 and composition score of 24.3. C5 appears in E4 and E9 with an average matching of 0.87 and composition of 28.28. Thus, C5 is favored to C4 and appears higher in the sorting.

B. *Results*

The total orderings on nodes for each graph are shown in Table III. For example, given the graphs in Figure 3a and Figure 3b, the topological sorts are `C1 C5 C4 C2 C3` and `C1 C3 C2 C4 C5`, respectively.

Considering both topological sorts, there is a clear winner in each equivalence class, with the exception of DBB. This is C1 for CCC, L3 for LBW, S2 for SNG, and T1 for LIT. Having a consistent and clear winner is evidence of a preference with respect to community standards and understandability, and thus provides guidance for potential refactorings.

This positive result, that the most popular representation in the corpus is also the most understandable, makes sense as people may be more likely to understand things that are familiar or well documented. However, while L3 is the winner for the LBW group, we note that L2 appears in slightly more patterns. DBB is different as the orderings are completely reversed depending on the analysis, so the community standards favor D2 and understandability favors D3. Further study is needed on this, as well as LBW and SNG since not all nodes were considered in the understandability analysis.

VII. DISCUSSION

Based on our studies, we have identified preferred regex representations that may make regexes easier to understand and their smelly counterparts. Here, we summarize the results.

A. *Implications*

In the CCC equivalence class, C1 (e.g., `[0-9a]`) is more commonly found in the patterns and projects. C2 (e.g., `[0123456789a]`) and C3 (e.g., `` [^\x00-/:-`b-\x7F] ``) appear in similar percentages of patterns and projects but there is no significant difference in understandability considering two pairs of regexes tested (i.e., E13 in Table **??**). A small preference is shown for C1 over C2 (E7), leading this to to be the preference for RQ1–RQ3.

In DBB, D3 (e.g., `pBs|pBBs|pBBBs`) merits further exploration because it is the most understandable but least common node in the group. This may be because explicitly listing the possibilities with an OR is easy to grasp, but if the number of items in the OR is too large, understandability may suffer. Further analysis is needed to determine the optimal thresholds for representing a regex as D3 compared to D1 (e.g., `pB{1,3}s`) or D2 (e.g., `pBB?B?s`).

In the SNG group, S1 is compact (e.g., `S{3}`), but S2 was preferred (e.g., `SSS`). Similar to DBB, this may be due to the particular examples chosen in the analysis, as a large number of explicit repetitions may not be as preferred.

In LWB, L2 (e.g., `AAA*`) and L3 (e.g., `AA+`) appear in similar numbers of patterns and projects, but there is a significant difference in their understandability, favoring L3.

In the LIT group, T1 (e.g., `\a\$>`) is the typical way to list literals, but the reason to use hex (T2) or oct (T4) types is because some characters cannot be represented any other way, such as invisible chars. One main result of our work is that T4 (e.g., `\007\036\062`) is less understandable than T2 (e.g., `\x07\x24\x3E`), so if invisible chars are required, hex is more understandable. Also, given a choice between T1 and T3, the escape character was more understandable.

B. *Opportunities For Future Work*

We looked at five equivalence classes, each with three to five nodes. Future work could consider richer models with more or different classes, including:

- Multi line option: `(?m)G\n` ≡ `(?m)G$`
- Case insensitive: `(?i)[a-z]` ≡ `[A-Za-z]`
- Backreferences: `(X)q\1` ≡ `(?P<name>X)q\g<name>`

There also may exist critical comprehension differences within a representation. For example, between C1 (e.g., `[0-9a]`) and C4 (e.g., `[\da]`), it could be that `[0-9]` is preferred to `[\d]`, but `[A-Za-z0-9_]` is not be preferred to `[\w]`. By creating a more granular model of equivalence classes and carefully evaluating alternative representations of the most frequently used specific patterns, additional useful smells could be identified.

C. *Threats to Validity*

Internal: We measure understandability using two metrics, matching and composition. These measures may not reflect actual understanding of regex behavior. To mitigate, we used multiple metrics that require reading and writing regexes.

Participants evaluated regular expressions on MTurk, which may not be reflective enough of the context in which programmers would encounter regexes in practice. Further study is needed to determine the impact of the experimental context.

Some regex representations from the equivalence classes were not involved in the understandability analysis and that may have biased the results against those nodes. More compete coverage of the edges in the equivalence classes is needed.

External The regexes used in the evaluation were inspired by those found in Python code, which is just one language that has library support for regexes, and may not generalize to other languages.

Participants in our survey came from MTurk, which may not be representative of people who read and write regexes on a regular basis. However, all participants demonstrated knowledge of regexes through a qualification test.

The results of the understandability analysis may be closely tied to the particular regexes chosen for the experiment. For many of the representations, we had several comparisons. Still, replication with more regex patterns is needed.

VIII. Related Work

Regular expression understandability has not been studied directly, though prior work has suggested that regexes are hard to read and understand as there are tens of thousands of bug reports related to regexes [6]. To aid in regex creation and understanding, tools have been developed to support more robust creation [6] or to allow visual debugging [18]. Other research has focused on removing the human from the creation process by learning regular expressions from text [13], [14].

Code smells in object-oriented languages were introduced by Fowler [19]. Researchers have studied the impact of code smells on program comprehension [15], [16], finding that the more smells in the code, the harder the comprehension. Code smells have been extended to other language paradigms including end-user programming languages [20]–[23]. The code smells identified in this work are representations that are not common or not well understood by developers. Using community standards to define smells has been used in refactoring for end-user programmers [22], [23].

Regular expression refactoring has not been studied directly, though refactoring literature abounds [24]–[26]. The closest to regex refactoring comes from research toward expediting regular expressions processing on large bodies of text [27], similar to refactoring for performance.

Exploring language feature usage by mining source code has been studied extensively for Smalltalk [28], JavaScript [29], and Java [30]–[33], and more specifically, Java generics [32] and Java reflection [33]. Our prior work [17] was the first to mine and evaluate regular expression usages from software repositories. The intention of the prior work was to explore regex language features usage and surveyed developers about regex usage.

IX. Conclusion

In an effort to find smells that impact regex understandability, we created five equivalence class models and used these models to investigate the most common representations and most comprehensible representations per class. The high agreement between the community standards and understandability analyses suggests that one particular representation can be preferred over others in most cases. Based on these results, we recommend using hex to represent invisible characters in regexes instead of octal, and to escape special characters with slashes instead of wrapping them in brackets. Further research is needed into more granular models that treat common specific cases separately.

References

[1] H. Zhao, W. Meng, Z. Wu, V. Raghavan, and C. Yu, "Fully automatic wrapper generation for search engines," in *Proceedings of the 14th international conference on World Wide Web*. ACM, 2005, pp. 66–75.

[2] A. S. Yeole and B. B. Meshram, "Analysis of different technique for detection of sql injection," in *Proceedings of the International Conference &#38; Workshop on Emerging Trends in Technology*, ser. ICWET '11. New York, NY, USA: ACM, 2011, pp. 963–966. [Online]. Available: http://doi.acm.org/10.1145/1980022.1980229

[3] "The Bro Network Security Monitor," https://www.bro.org/, May 2015. [Online]. Available: https://www.bro.org/

[4] B. L. Hutchings, R. Franklin, and D. Carver, "Assisting network intrusion detection with reconfigurable hardware," in *Field-Programmable Custom Computing Machines, 2002. Proceedings. 10th Annual IEEE Symposium on*. IEEE, 2002, pp. 111–120.

[5] D. Ficara, S. Giordano, G. Procissi, F. Vitucci, G. Antichi, and A. Di Pietro, "An improved dfa for fast regular expression matching," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 5, pp. 29–40, 2008.

[6] E. Spishak, W. Dietl, and M. D. Ernst, "A type system for regular expressions," in *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs*, ser. FTfJP '12. New York, NY, USA: ACM, 2012, pp. 20–26. [Online]. Available: http://doi.acm.org/10.1145/2318202.2318207

[7] "Online regex tester, debugger with highlighting for php, pcre, python, golang and javascript," https://regex101.com/.

[8] "Regexr: Learn, build, and test regex," https://regexr.com/.

[9] "Regular expression visualizer using railroad diagrams," https://regexper.com/.

[10] "Rex @ rise4fun from microsoft," http://rise4fun.com/rex/.

[11] A. Kiezun, V. Ganesh, S. Artzi, P. J. Guo, P. Hooimeijer, and M. D. Ernst, "Hampi: A solver for word equations over strings, regular expressions, and context-free grammars," *ACM Trans. Softw. Eng. Methodol.*, vol. 21, no. 4, pp. 25:1–25:28, Feb. 2013. [Online]. Available: http://doi.acm.org/10.1145/2377656.2377662

[12] M. Veanes, P. De Halleux, and N. Tillmann, "Rex: Symbolic regular expression explorer," in *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*. IEEE, 2010, pp. 498–507.

[13] R. Babbar and N. Singh, "Clustering based approach to learning regular expressions over large alphabet for noisy unstructured text," in *Proceedings of the Fourth Workshop on Analytics for Noisy Unstructured Text Data*, ser. AND '10. New York, NY, USA: ACM, 2010, pp. 43–50. [Online]. Available: http://doi.acm.org/10.1145/1871840.1871848

[14] Y. Li, R. Krishnamurthy, S. Raghavan, S. Vaithyanathan, and H. V. Jagadish, "Regular expression learning for information extraction," in *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, ser. EMNLP '08. Stroudsburg, PA, USA: Association for Computational Linguistics, 2008, pp. 21–30. [Online]. Available: http://dl.acm.org/citation.cfm?id=1613715.1613719

[15] M. Abbes, F. Khomh, Y.-G. Gueheneuc, and G. Antoniol, "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension," in *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*. IEEE, 2011, pp. 181–190.

[16] B. Du Bois, S. Demeyer, J. Verelst, T. Mens, and M. Temmerman, "Does god class decomposition affect comprehensibility?" in *IASTED Conf. on Software Engineering*, 2006, pp. 346–355.

[17] C. Chapman and K. T. Stolee, "Exploring regular expression usage and context in python," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 2016, pp. 282–293.

[18] F. Beck, S. Gulan, B. Biegel, S. Baltes, and D. Weiskopf, "Regviz: Visual debugging of regular expressions," in *Companion Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE Companion 2014. New York, NY, USA: ACM, 2014, pp. 504–507. [Online]. Available: http://doi.acm.org/10.1145/2591062.2591111

[19] M. Fowler, *Refactoring: improving the design of existing code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[20] F. Hermans, M. Pinzger, and A. van Deursen, “Detecting code smells in spreadsheet formulas,” in *Proc. of ICSM ’12*, 2012.

[21] ——, “Detecting and refactoring code smells in spreadsheet formulas,” *Empirical Software Engineering*, pp. 1–27, 2014.

[22] K. T. Stolee and S. Elbaum, “Refactoring pipe-like mashups for end-user programmers,” in *International Conference on Software Engineering*, 2011.

[23] ——, “Identification, impact, and refactoring of smells in pipe-like web mashups,” *IEEE Trans. Softw. Eng.*, vol. 39, no. 12, pp. 1654–1679, Dec. 2013. [Online]. Available: http://dx.doi.org/10.1109/TSE.2013.42

[24] T. Mens and T. Tourwé, “A survey of software refactoring,” *IEEE Trans. Soft. Eng.*, vol. 30, no. 2, pp. 126–139, Feb. 2004. [Online]. Available: http://dx.doi.org/10.1109/TSE.2004.1265817

[25] W. F. Opdyke, “Refactoring object-oriented frameworks,” Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1992.

[26] W. G. Griswold and D. Notkin, “Automated assistance for program restructuring,” *ACM Trans. Softw. Eng. Methodol.*, vol. 2, no. 3, pp. 228–269, Jul. 1993. [Online]. Available: http://doi.acm.org/10.1145/152388.152389

[27] R. A. Baeza-Yates and G. H. Gonnet, “Fast text searching for regular expressions or automaton searching on tries,” *J. ACM*, vol. 43, no. 6, pp. 915–936, Nov. 1996. [Online]. Available: http://doi.acm.org/10.1145/235809.235810

[28] O. Callaú, R. Robbes, E. Tanter, and D. Röthlisberger, “How developers use the dynamic features of programming languages: The case of smalltalk,” in *Proceedings of the 8th Working Conference on Mining Software Repositories*, ser. MSR ’11. New York, NY, USA: ACM, 2011, pp. 23–32. [Online]. Available: http://doi.acm.org/10.1145/1985441.1985448

[29] G. Richards, S. Lebresne, B. Burg, and J. Vitek, “An analysis of the dynamic behavior of javascript programs,” *SIGPLAN Not.*, vol. 45, no. 6, pp. 1–12, Jun. 2010. [Online]. Available: http://doi.acm.org/10.1145/1809028.1806598

[30] R. Dyer, H. Rajan, H. A. Nguyen, and T. N. Nguyen, “Mining billions of ast nodes to study actual and potential usage of java language features,” in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 779–790. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568295

[31] M. Grechanik, C. McMillan, L. DeFerrari, M. Comi, S. Crespi, D. Poshyvanyk, C. Fu, Q. Xie, and C. Ghezzi, “An empirical investigation into a large-scale java open source code repository,” in *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM ’10. New York, NY, USA: ACM, 2010, pp. 11:1–11:10. [Online]. Available: http://doi.acm.org/10.1145/1852786.1852801

[32] C. Parnin, C. Bird, and E. Murphy-Hill, “Adoption and use of java generics,” *Empirical Softw. Engg.*, vol. 18, no. 6, pp. 1047–1089, Dec. 2013. [Online]. Available: http://dx.doi.org/10.1007/s10664-012-9236-6

[33] B. Livshits, J. Whaley, and M. S. Lam, “Reflection analysis for java,” in *Proceedings of the Third Asian Conference on Programming Languages and Systems*, ser. APLAS’05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 139–160. [Online]. Available: http://dx.doi.org/10.1007/11575467_11