Refactoring Regular Expressions

Carl Chapman
Department of Computer Science
Iowa State University
carl1976@iastate.edu

Kathryn T. Stolee
Department of Computer Science
North Carolina State University
ktstolee@ncsu.edu

ABSTRACT

Regular expressions (regexes) are powerful tools employed across many tasks and platforms. Regexes can be very complex so optimizing understandability of regexes is desirable for maintainers. Due to a rich feature set, there is more than one way to compose a regex to get the same desired behavior. In this work, we define five equivalence classes of regex representations where the same behavior can be achieved with multiple representations. With the goal of finding refactorings that improve understandability, we analyze regexes in GitHub to find community standards, or common usages of various features. We further obtain understandability metrics from an empirical study with 180 participants to discover which representations are more understandable to programmers. We found that, for example, patterns requiring one or more repetitions of a character are more understandable when expressed using the plus (e.g., `':+'`) operator than the kleene star operator (e.g., `'::*'`). We identify strongly preferred transformations in three of the five equivalence classes and identify opportunities for future work in regex refactoring.

1. INTRODUCTION

Regular expressions are used frequently by developers for many purposes, such as parsing files, validating user input, or querying a database. Regexes are also employed in MySQL injection prevention [1] and network intrusion detection [2]. However, recent research has suggested that regular expressions are hard to understand, hard to compose, and error prone [3]. Given the difficulties with working with regular expressions and how often they appear in software projects and processes, it seems fitting that efforts should be made to ease the burden on developers.

Tools have been developed to make regexes easier to understand, and many are available online. Some tools will, for example, highlight parts of regex patterns that match parts of strings to aid in comprehension.[1] Others will automatically generate strings that are matched by the regular expessions [4] or automatically generate regexes when given a list of strings to match [5, 6]. 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 [7,8]. 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 smelly.

As with source code, in regular expressions, there are multiple ways to express the same semantic concept. For example, the regex, `'aa*'` matches an `a` followed by zero or more `a`'s, and is is equivalent to `'a+'` , which matches one or more `a`'s. What is not clear is which representation, `'aa*'` or `'a+'`, is preferred. Preferences in regex refactorings could vary, including which is easier to maintain, easier to understand, or better conforms to community standards, depending on the goals of the programmer.

In this work, we introduce possible refactorings in regular expressions by identifying equivalence classes of regex representations and transformations between the representations. These equivalence classes provide options for how to represent double-bounds in repetitions (e.g., `'a{1,2}'` or `'a|aa'`), single-bounds in repetitions (e.g., `'a{2}'` or `'aa'`), lower bounds in repetitions (e.g., `'a{2,}'` or `'aaa*'`), character classes (e.g., `'[0-9]'` or `'[\d]'`), and literals (e.g., `'\a'` or `'\x07'`). We suggest directions for the refactorings, for example, from `'aa*'` to `'a+'`, based on two high-level concepts: which representation appears most frequently in source code (conformance to community standards) and which is more understandable by programmers, based on comprehension tests completed by 180 study participants. Our results identify preferred representations for four of the five equivalence classes based on mutual agreement between community standards and understandability, with three of those being statistically significant. For the fifth group on double-bounded repetitions, two recommendations are given depending on the programmer's goals. Our contributions are:

- Identification of equivalence classes for regular expressions with possible transformations within each class,
- Conducted an empirical study identifying opportunities for regex refactoring in Python projects based on how regexes are expressed,
- Conducted an empirical study with 180 participants evaluating regex understandability, and

[1] https://regex101.com/

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

- Identified preferred regex representations and refactorings for understandability and conformance to community standards, backed by empirical evidence.

To our knowledge, this is the first work to apply refactoring to regular expressions. We approach the problem of identifying preferred regex representations by looking at thousands of regexes in Python projects and measuring the understandability of various regex representations using human participants. The rest of the paper describes the equivalence classes, possible refactorings, and our two studies.

2. REFACTORINGS

We have defined a set of equivalence classes for regexes with refactorings that can transform among members in the classes. Figure 1 displays the five equivalence classes in grey boxes and various semantically equivalent *representations* of a regex are shown in white boxes. For example, LWB is an equivalence class with representations that all have a lower bound on repetitions. Regexes `AAA*` and `AA+` are both members of this class mapping to representations L2 and L3, respectively, along with the L1 representation, `A{2,}`. The undirected edges between the representations define possible refactorings. Identifying the best direction for each arrow in the possible refactorings is discussed in Section 6.

We use concrete regexes in the representations to more clearly illustrate examples of the representations. However, the `A`'s in the LWB group 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. Next, we describe each group, the representations, and possible transformations in detail.

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 enables a programmer to specify a set of alternative characters, any of which can match. For example, the regex `'c[ao]t'` will match both the string "cat" and the string "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 created by the user 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, as is the case with the C4 representation. Next, we provide descriptions of each representation in this equivalence 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 the C1 node.

C2: Any pattern that contains a (non-negative) custom character class without any shorthand representations, specifically ranges or defaults. For example, `'[012]'` is in C2, but `'[0-2]'` is not.

C3: Any pattern with a character classes expressed using negation, which is indicated by a caret (i.e., `^`) followed by a custom character class specification. For example, the pattern `[^ao]` matches every character *except* `a` or `o`. If the applicable character set is known (e.g., ASCII, UTF-8, etc.), then any non-negative character class can be represented as a negative character class. For example, assuming an ASCII charset that has 128 characters: `\x00-\x7f`, a character class representing the lower half: `[\x00-\x3f]` can be represented by negating the upper half: `[^\x40-\x7f]`.

C4: Any pattern using a default character class such as `\d` or `\W` within a (non-negative) character class belongs to the C4 node.

C5: While not expressed using a character class, these representations 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 included in C5. Note that because an OR cannot be directly negated, it does not make sense to have an edge between C3 and C5 in Figure 1, though C3 may be able to transition to C1, C2 or C4 first and then to C5.

Note that a pattern can belong to multiple representations. For example, `[a-f\d]` belongs to both C1 and C4. The edge between C1 and C4 represents the opportunity to express the same pattern as `[a-f0-9]` by transforming the default digit character class into a range. This transformed version would only belong to the C1 node.

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 the pattern `pB{1,3}s` represents a `p` followed by one to three sequential `B` patterns, then followed by a single `s`. This will match "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`, belongs to the D1 node. Note that `pB{1,3}s` can become `pBB{0,2}s` by pulling the lower bound out of the curly braces and into the explicit sequence (or visa versa). Nonetheless, it would still be part of D1, though this within-node refactoring on D1 is not discussed in this work.

D2: Any pattern that uses the questionable (i.e., `?`) modifier implies a lower-bound of zero and an upper-bound of one, and belongs to D2. For example, when a double-bounded regex has zero on the lower bound, as is the case with `pBB{0,2}s` in D1, transforming it to D2 involves replacing the curly braces with n questionable modifiers, where n is the upper bound, creating `pBB?B?s`.

D3: Any pattern that has a repetition with a lower and upper boundary and is expressed using ORs is part of D3. The example, `pB{1,3}s` would become `pBs|pBBs|pBBs` by expanding on each option in the boundaries.

A pattern can also belong to multiple nodes in the DBB group, for example, `(a|aa)X?Y{2,4}` belongs to all three nodes: `Y{2,4}` maps it to D1, `X?` maps it to D2, and `(a|aa)` maps it to D3.

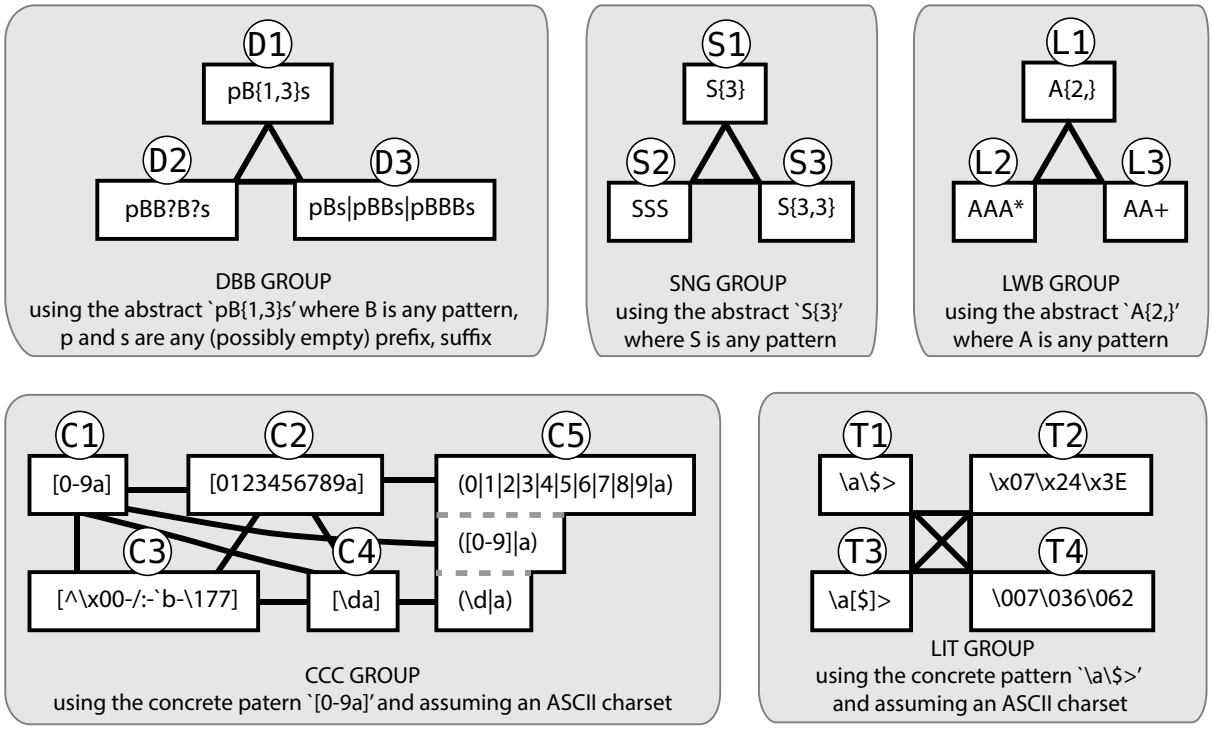

Figure 1: Equivalence classes with various representations. DBB = Double-Bounded, SNG = Single Bounded, LWB = Lower Bounded, CCC = Custom Character Class and LIT = Literal

Literal Group.

In the Literal (LIT) group, all patterns that are not purely default character classes must use literal tokens to specify characters to match. In most languages that support regex libraries, the programmer is able to specify literal tokens in a variety of ways. Here, we use the ASCII charset in which all characters can be expressed using hex and octal codes like `\xF1`, and `\0108`, respectively. This group defines transformations among various representations of literals.

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. Literal character can be wrapped in brackets to form a custom character classes of size one, such as `[x]`. 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.

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. Note that refactoring from L1 to L3 and L2 to L3 is not always possible when the lower bound is zero and the pattern is not repeated in sequence (e.g., `'A*'` or `'A{0,}'`).

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 is part of S2.

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.

Example.

Regular expressions will often belong to multiple representations in multiple equivalence classes described. Using an example from a Python project from our analysis, the regex `'[^ ]*\.[A-Z]{3}'` is a member of S1, L2, C1, C3, and T1. This is because `'[^ ]'` maps it to C3, `'[^ ]*'` maps it to L2, `'[A-Z]'` maps it to C1, `'\.'` maps it to T1, and `'[A-Z]{3}'` maps it to S1. As examples of refactorings, moving from S1 to S2 would be possible by replacing `'[A-Z]{3}'` with `'[A-Z][A-Z][A-Z]'` and moving from L2 to L1 would replace `'[^ ]*'` with `'[^ ]{0,}'`, resulting in a refactored regex of: `'[^ ]{0,}\.[A-Z][A-Z][A-Z]'`.

3. RESEARCH QUESTIONS

After defining the equivalence classes and potential regex refactorings as described in Section 2, we wanted to know which representations in the equivalence classes are considered desirable and which might be smelly. Desirability for regexes can be defined many ways, including maintainable, understandable, and performance. We focus on refactoring for understandability.

We define understandability two ways. First, assuming that common programming practices are more understandable than uncommon practices, we explore the frequencies of each representation from Figure 1 using thousands of regexes scraped from Python projects. Second, we then present people with regexes exemplifying some of the more common characteristics and ask them comprehension questions along two directions: determine which of a list of strings are matched by the regex, and compose a string that is matched by the regex. Our research questions are:

RQ1: Which refactorings have the strongest *community support* based on how frequently each representation appears in regexes in open source Python projects?

RQ2: Which refactorings have the strongest support based on *understandability* as measured by matching strings and composing strings?

RQ3: Which regex representations are most desirable based on both community support and understandability?

Next, we present the analysis and results for each question in turn, followed by a unified discussion in Section 7.

4. COMMUNITY SUPPORT STUDY (RQ1)

The goal of this study is to understand how frequently each of the regex representations appears in source code. Based on the results, we identify preferred representations using popularity in source code.

4.1 Artifacts

To determine how common each regex representations is, we collected and analyzed regexes from GitHub projects. We targeted Python as it is a popular programming language with a strong presence on GitHub, being the fourth most common language after Java, Javascript and Ruby. Further, Python's regex pattern language is close enough to other regex libraries that our conclusions are likely to generalize.

We collected and analyzed static invocations to the Python `re` library. Figure 2 presents an example with key components labeled. The *function* called is `re.compile`. The

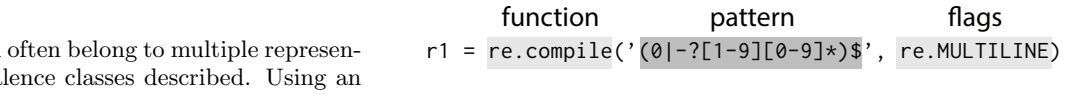

Figure 2: Example of one regex library invocation

pattern defines what strings will be matched and the *flag* `re.MULTILINE` modifies the rules used by the regex engine when matching. When executed, a regex object `r1` is created and it will match if it finds a zero at the end of a line, or a (possibly negative) integer at the end of a line (i.e., due to the `-?` sequence denoting zero or one instance of the `-`).

Our goal was to collect regex patterns from a variety of projects to represent the breadth of how developers use regexes. We scraped 3,898 projects containing Python code using the GitHub API. This was done by systematically selecting repository IDs, checking the repository for Python files, and retaining the project if Python was found. After dividing eight million repository IDs into 32 groups, we scanned from the beginning until we had collected approximately four thousand Python projects. At that point, we felt we had enough data to do an analysis without further perfecting our mining techniques.

To identify invocations of the `re` module, we built the AST of each Python file in each project. In most projects, almost all `re` invocations are present in the most recent version of a project, but to be more thorough, we also scanned up to 19 earlier versions. All regex patterns were retained, sans duplicates. In the end, we observed and recorded 16,088 non-duplicate patterns in 1,645 projects.

In collecting the set of distinct patterns for analysis, we ignore the 12.7% of `re` invocations using flags, which can alter regex behavior. An additional 6.5% of `re` invocations contained patterns that could not be compiled because the pattern was non-static (e.g., used some runtime variable). This parser was unable to support 0.8% (114) due to error. After removing all problematic patterns as described and collapsing on duplicates, we ended up with 13,597 distinct patterns from 1,544 projects.

4.2 Metrics

We measure community support by matching each regex in the corpus to the representations in Figure 1 and counting the number of *patterns* that contain the representation and the number of *projects* that contain the representation. Note that a regex can belong to multiple representations, and a regex can belong to multiple projects since we collapsed duplicates.

4.3 Analysis

To determine how many of the representations match patterns in the corpus, we performed an analysis using the PCRE parser and by representing 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:

4.3.1 Presence of a Feature

For the representations that only require a particular feature to be present, such as the question-mark in D2, the features identified by the PCRE parser were used to decide membership of patterns in nodes. These feature-requiring

[2]https://github.com/softwarekitty/regex_readability_study

Table 1: 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%

nodes are as follows: D1 requires double-bounded repetition with different bounds, D2 requires the question-mark repetition, S1 requires single-bounded repetition, S3 requires double-bounded repetition with the same bounds, L1 requires a lower-bound repetition, L2 requires the kleene star (`*`) repetition, L3 requires the add (`+`) repetition, and C3 requires a negated custom character class.

4.3.2 Features and Pattern

For some representations, the presence of a feature is not enough to determine membership. Identifying D3 requires an OR containing at least two entries with a sequence present in one entry repeated N times, and then the same sequence present in another entry repeated N+1 times. This is a hard pattern to detect directly, but we identified candidates by looking for a sequence of N repeating groups with an OR-bar (ie. `|`) next to them on one side (either side). This produced a list of 113 candidates which we narrowed down manually to 10 actual members.

Identifying T2 requires a literal feature that matches the regex `(\\x[a-f0-9A-F]{2})` which reliably identifies hex codes within a pattern. Similarly T4 requires a literal feature and must match the regex `((\\0\d*)|(\\\d{3}))` which is specific to Python-style octal, requiring either exactly three digits after a slash, or a zero and some other digits after a slash. Only one false positive was identified which was actually the lower end of a hex range using the literal `\0`.

Identifying 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 actually matches over 91% of the total patterns analyzed.

4.3.3 Token Stream

The rest of the representations were identified by representing the regex patterns as a sequence of tokens. Identifying S2 requires any element to be repeated at least twice in sequence. This element could be a character class, a literal, or a collection of things encapsulated in parentheses. Identifying C1 requires that a non-negative character class contains a range. Identifying C2 requires that there exists a custom character class that does not use ranges or defaults. Identifying C4 requires the presence of a default character class within a custom character class, specifically, `\d`, `\D`, `\w`, `\W`, `\s`, `\S` and `.`. Identifying C5 requires an OR of length-one sequences (literal characters or any character class).

4.4 Results

Table 1 presents the frequencies with which each representation appears in a regex pattern and in a project scraped from GitHub. The *node* column references the representations in Figure 1 and the *description* column briefly describes the representation, followed by an *example* from the corpus. The *nPatterns* column counts the patterns that belong to the representation, followed by the percent of patterns out of 13,597. The *nProjects* column counts the projects that contain a regex belonging to the representation, followed by the percentage of projects out of 1,544. Recall that the patterns are all unique and could appear in multiple projects, hence the project support is used to show how pervasive the representation in across the whole community. For example, 2,479 of the patterns belong to the C1 representation, representing 18.2% of the patterns. These appear in 810 projects, representing 52.5%. Representation D1 appears in 346 (2.5%) of the patterns but only 234 (15.2%) of the projects. In contrast, representation 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.

Using the pattern frequency as a guide, we can create refactoring recommendations based on community frequency. 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 the community standard if ex-

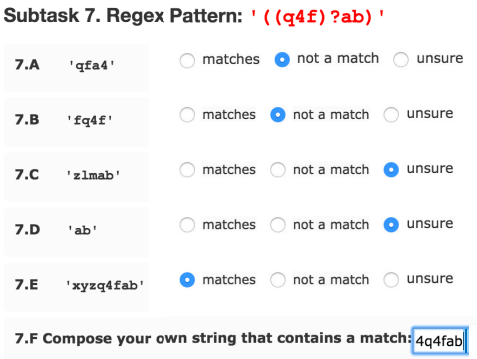

Figure 3: Example of one HIT Question

Table 2: 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

pressed as C1. Thus, we might recommend a $\overrightarrow{C2C1}$ refactoring. 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. The difference is that L3 appears in more projects than L2, so it is not clear which desirable based on community standards metrics. However, we note that our criteria for membership in a representation may overestimate the opportunities for refactoring. 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. Our analysis simply suggests a direction for a refactoring (in this case, from C4 to C1).

5. UNDERSTANDABILITY STUDY (RQ2)

The overall idea of this study is to present programmers with one of several representations of semantically equivalent regexes and ask comprehension questions. By comparing the understandability of semantically equivalent regexes that have different representations, we aim to understand which transformations are desirable. This study was implemented on Amazon's Mechanical Turk with 180 participants. Each regex pattern was evaluated by 30 participants.

5.1 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 3.

The percentage of correct responses, disregarding blanks and unsure responses, is the matching score. For example, consider regex pattern `'RR*'` and five strings shown in Table 2, and the responses from four participants in the *P1*, *P2*, *P3* and *P4* columns. The oracle has the first three strings matching since they each contain at least one `R` character. *P1* answers correctly for the first three strings but incorrectly on the fourth string, 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. It was answered correctly so 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 when participants were honestly unsure of the answer. These situations did not occur very frequently. Only 1.1% of the responses were left blank and only 3.8% of the responses were marked as unsure. Out of 1800 questions (180 participants * 10 questions each), 1.8%(32) had a blank or unsure (never more than 4 out of 30 per pattern).

Composition: Given a pattern, a participant composes a string they think it matches. If the participant is accurate, a composition score of 1 is assigned, 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.

5.2 Design

This study was implemented on the Amazon's Mechanical Turk (MTurk), a crowdsourcing platform in which requesters can create human intelligence tasks (HITs) for completion by workers. Each HIT is designed to be completed in a fixed amount of time and workers are compensated with money if their work is satisfactory. Requesters can screen workers by requiring each to complete a qualification test prior to completing any HITs.

5.2.1 *Worker Qualification*

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

5.2.2 *Tasks*

Using the patterns in the corpus as a guide, we created 60 regex patterns that were grouped into 26 semantic equivalence groups. These semantic groups were intended to explore edges in the equivalence classes. In this way, we can draw conclusions about transformations between representations since the regexes evaluated were semantically equivalent. For example, a group with regexes `'([0-9]+)\.([0-9]+)'` and `'(\d+)\.(\d+)'` is intended to evaluate the edge

Table 3: Averaged Info About Edges (sorted by lowest of either p-value)

Index	Representations	Pairs	Match1	Match2	$H_0 : \mu_{match1} = \mu_{match2}$	Compose1	Compose2	$H_0 : \mu_{comp1} = \mu_{comp2}$
E1	T1 – T4	2	0.80	0.60	0.001	0.87	0.37	**<0.001**
E2	D2 – D3	2	0.78	0.87	**0.011**	0.88	0.97	0.085
E3	L2 – L3	3	0.86	0.91	**0.032**	0.91	0.98	0.052
E4	C2 – C5	4	0.85	0.86	0.602	0.88	0.95	0.063
E5	C2 – C4	1	0.83	0.92	0.075	0.60	0.67	0.601
E6	D1 – D2	2	0.84	0.78	0.120	0.93	0.88	0.347
E7	C1 – C2	2	0.94	0.90	0.121	0.93	0.90	0.514
E8	T2 – T4	2	0.84	0.81	0.498	0.65	0.52	0.141
E9	C1 – C5	2	0.94	0.90	0.287	0.93	0.93	1.000
E10	T1 – T3	3	0.88	0.86	0.320	0.72	0.76	0.613
E11	D1 – D3	2	0.84	0.87	0.349	0.93	0.97	0.408
E12	C1 – C4	6	0.87	0.84	0.352	0.86	0.83	0.465
E13	C3 – C4	2	0.61	0.67	0.593	0.75	0.82	0.379
E14	S1 – S2	3	0.85	0.86	0.776	0.88	0.90	0.638

between C1 and C4. There were 18 groups with two regexes that target various edges in the equivalence classes. The other eight semantic groups had three regexes each. For example, a semantic group with regexes `'((q4f){0,1}ab)'`, `'((q4f)?ab)'`, and `'(q4fab|ab)'` is intended to explore the edges among D1, D2, and D3.

For each of the 26 groups of patterns, we created five strings, 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 each 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 3. This process was completed until each of the 60 regexes appeared in 30 HITs, resulting in a total of 180 total unique HITs.

5.2.3 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 55 HITs were rejected, and of those, 48 were rushed through by leaving many answers blank, four were rejected because a worker had submitted more than one HIT, one was rejected for not answering composition sections, and one was rejected because it was missing data for 3 questions. Rejected HITs were returned to MTurk to be completed by others.

5.2.4 Participants

In total, there were 180 participants. A majority were male (83%) with an average age of 31. Most had at least an Associates degree (72%) and most were at least somewhat familiar with regexes (87%). On average, participants compose 67 regexes per year with a range from 0 to 1000.

5.3 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 5.1. 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 unsure or a combination of blanks and unsure).

Each regex was a member of one of 26 groupings of equivalent regexes. These groupings allow pairwise comparisons of the metrics values to determine which representation of the regex was most understandable and the direction of a refactoring for understandability. Among all the groups, we performed 42 pairwise comparisons of the matching and composition scores (i.e., one comparison for each group of size two and three comparisons within each group of size three). For example, one group had regexes, `RR*` and `R+`, which represent a transformation between L2 and L3. The former had an average matching of 86% and the latter had an average matching of 92%. The average composition score for the former was 97% and 100% for the latter. Thus, the community found `R+` from L3 more understandable. There were two other pairwise comparisons performed between the L2 and L3 group, using regexes pair `zaa*` and `za+'`, and regexes pair `\..*` and `\.+'`. Considering all three of these 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 0.91 and 0.98 for L3. Thus, the community found L3 to be more understandable than L2, from the perspective of both understandability metrics, suggesting a refactoring from L2 to L3.

This information is presented in summary in Table 3, with this specific example appearing in the E3 row. The *Index* column enumerates the edges evaluated in this experiment, *Representations* lists the two representations, *Pairs* shows how many comparisons were performed, *Match1* gives the overall matching score for the first representation listed, *Match2* gives the overall matching score for the second representation listed, 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 p-value, also using the Mann-Whitney test of means.

Although we had 42 pairwise comparisons, we had to drop six comparisons due to a design flaw since the patterns performed transformations from multiple equivalence classes. For example, pattern `([\072\073])` is in C2 and T4, and

was grouped with pattern `(:|;)` in C5, T1, so it was not clear if any differences in understandability were due to the transformation between C2 and C5, or T4 and T1. However, the third member of the group, `([:;])`, could be compared with both, since it is a member of T1 and C2, so comparing it to `([\072\073])` evaluates the transformation between T1 and T4, and comparing to `(:|;)` evaluates the transformation between C2 and C5. The end result is 36 pairwise comparisons across 14 edges from Figure 1.

5.4 Results

Table 3 presents the results of the understandability analysis. A horizontal line separates the first three edges from the bottom 11. In E1 through E3, there is a statistically significant difference between the representations for at least one of the metrics considering $\alpha = 0.05$. These represent the strongest evidence for suggesting the directions of refactoring based on the understandability metrics we defined. Specifically, $\overrightarrow{T4T1}$, $\overrightarrow{D2D3}$, and $\overrightarrow{L2L3}$ are likely to improve understandability.

We note again that participants were able to select *unsure* when they were not sure if a string would be matched by a pattern (Figure 3). From a comprehension perspective, this indicates some level of confusion and is worth exploring.

For each pattern, we counted the number of responses containing at least one unsure, representing confusion. We then grouped the patterns into their representation nodes and computed an average of unsures per pattern. A higher number may indicate difficulty in comprehending a pattern from that node. Overall, the highest number of unsure responses came from T4 and T2, which present octal and hex representations of characters. The least number of unsure responses were in L3 and D3. These results also corroborate the refactorings suggested by the understandability analysis for the LIT group (i.e., $\overrightarrow{T4T1}$), the DBB group (i.e., $\overrightarrow{D2D3}$), and the LWB group (i.e., $\overrightarrow{L2L3}$) because the more understandable node has the least unsures of its group.

6. DESIRABLE REPRESENTATIONS (RQ3)

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

6.1 Analysis

At a high level, the total orderings were achieved by building directed graphs with the representations as nodes and edge directions determined by the metrics: patterns and projects for community standards and matching and composition for understandability. Then, within each graph, we performed a topological sort to obtain total node orderings.

The graphs for community support are based on Table 1 and the graphs for understandability are based on Table 3. The following sections describe the processes for building and and topologically sorting the graphs.

6.1.1 Building the Graphs

In the community standards graph, we represent a directed edge $\overrightarrow{C2C1}$ 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 where L2 is found in more patterns and L3 is found in more projects, an undirected edge $\overline{L2L3}$ is used. This represents that there was no winner based on the two metrics. After considering all pairs of nodes in each equivalence class that also have an edge in Figure 1, we have created a graph, for example Figure 4a, that represents the frequency trends among the community artifacts. A node with no incoming edges is less common and a node with many incoming edges is more common.

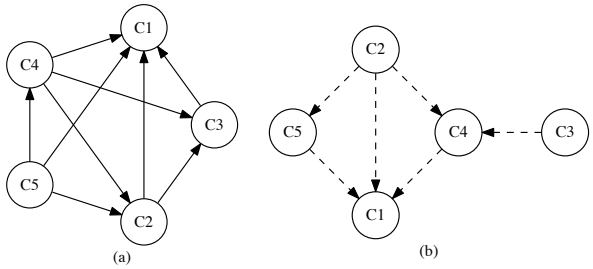

Figure 4: Trend graphs for the CCC equivalence graph: (a) represent the artifact analysis, (b) represent the understandability analysis.

In the understandability graph, we represent a directed edge $\overrightarrow{C2C1}$ when match(C1) > match(C2) *and* compose(C1) > compose(C2). When there is a conflict between match and compose, as is the case with T1 and T3 where match(T1) is higher but compose(T3) is higher, an undirected edge $\overline{T1T3}$ is used. When one metric has a tie, as is the case with composition in E9, we resort to the matching metric to determine $\overrightarrow{C5C1}$. An example understandability graph for the CCC is shown in Figure 4b. Nodes with few incoming edges are less understandable (or were not evaluated in our study), and nodes with more incoming edges were more understandable.

6.1.2 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 on the nodes, as shown in Algorithm 1.

The first modification is to remove all undirected edges since Kahn's operates over a directed graph (Line 3). To begin, any disconnected nodes are added to the end of the topologically sorted list L (Line 4). In Kahn's algorithm, all nodes without incoming edges are added to a set S (Line 5), which represents the order in which nodes are explored in the graph. For each n node in S (Line 6), all edges from n are removed and n is added to L (Line 8). If there exists a node m that has no incoming edges, it is added to S. In the end, L is a topologically sorted list.

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 tiebreaker is needed to enforce a total ordering on the nodes (though admittedly, it is not always unique). For example, on the understandability graph in Figure 4b, there is a tie between C3 and C2 since both have no incoming edges, so they are marked as a tie on Line 5. Further, if C3 is added to S first, when $n = C2$ on line 7, both C5 and C4 are added to S on Line 12, thus the tie between them is marked on line 15. In these cases, a tiebreaker is needed.

Table 4: Topological Sorting, with the left-most position being highest

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

Algorithm 1 Modified Topological Sort

1: $L \leftarrow []$
2: $S \leftarrow []$
3: Remove all undirected edges (creates a DAG)
4: Add all disconnected nodes to L and remove from graph. If there is more than one, mark the tie.
5: Add all nodes with no incoming edges to S. If there is more than one, mark the tie.
6: **while** S is non-empty **do**
7: remove a node n from S
8: add n to L
9: **for** node m such that e is an edge $\overrightarrow{nm}$ **do**
10: remove e
11: **if** m has no incoming edges **then**
12: add m to S
13: **end if**
14: **end for**
15: If multiple nodes were added to S in this iteration, mark the tie
16: remove n from graph
17: **end while**
18: For all ties in L, use a tiebreaker.

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 metrics. Based on Table 3, we compute the average matching score for all instances of each representation, and do the same for the composition score. 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.

6.2 Results

After running the topological sort in Algorithm 1 with tiebreakers, we have a total ordering on nodes for each graph, shown in Table 4. For example, given the graphs in Figure 4a and Figure 4b, the topological sorts are `C1 C3 C2 C4 C5` and `C1 C5 C4 C2 C3`, 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. After the top rank, the second rank varies depending on the metric, however, 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.

7. DISCUSSION

Based on our analyses of source code and our empirical study on the understandability of regex representations, we have identified preferred regex representations that may make regexes easier to understand and thus maintain. In this section, we describe the implications of these results.

7.1 Interpreting Results

In the CCC equivalence class, C1 (e.g., `[0-9a]`) is more commonly found in the patterns and projects. Representations 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 as part of E13 (Table 3). However, a small preference is shown for C1 over C2 (E7), leading this to to be the winner of both the community support and understandability analyses.

In the DBB group, D3 (e.g., `pBs|pBBs|pBBBs`) merits further exploration because it is the most understandable but least common node in DBB 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, the understandability may go down. 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 a compact representation (e.g., `S{3}`), but S2 was preferred (e.g., `SSS`). Similar to the DBB group, this may be do to the particular examples chosen in the analysis, as a large number of explicit repetitions may not be as preferred.

In the LWB group, representations 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 the more understandable representation. Regarding T3 (e.g., `\a[$]>`), initially we thought the square brackets would be more understandable than using an escape character, but we found the opposite. Given a choice between T1 and T3, the escape character was more understandable.

7.2 Opportunities For Future Work

There are several directions for future work related to regex study and refactoring.

Equivalence Class Models.

We looked at five equivalence classes, each with three to five nodes. Future work could consider richer models with more or different classes and nodes. Additional equivalence groups to consider may include:

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>`

It might also be the case that there exist critical comprehension differences within a representation. For example, between C1 (e.g., `[0-9a]`) and C4 (e.g., `[\da]`), it could be the case 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 refactorings could be identified.

Regex Migration Libraries.

We have identified opportunities to improve the understandability of regexes in existing code bases by looking for some of the less understandable regex representations, which can be thought of as antipatterns, and refactoring to the more common or understandable representations. Building migration libraries is a promising direction of future work to ease the manual burden of this process, similar in spirit to prior work on class library migration [9].

Regex Programming Standards.

Many organizations enforce coding standards in their repositories to ease understandability. Presently, we are not aware of coding standards for regular expressions, but this work suggests that enforcing standard representations for various regex constructs could ease comprehension.

Regex Refactoring for Performance.

The representation of regexes may have a strong impact on the runtime performance of a chosen regex engine. Prior work has sought to expedite the processing of regexes over large bodies of text [14]. Refactoring regexes for performance would complement those efforts.

7.3 Threats to Validity

Internal: We measure understandability of regexes using two metrics, matching and composition. However, these measures may not reflect actual understanding of the regex behavior. We chose to use multiple metrics in the context of reading and writing regexes, but the threat remains.

Participants evaluated regular expressions during tasks on MTurk, which may not be representative enough of the context in which programmers would encounter regexes in practive. Further study is needed to determine the impact of the experimentation context on the results.

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

External Participants in our survey came from MTurk, which may not be representative of people who read and write regexes on a regular basis.

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. Thus, we may have missed opportunities for other refactorings based on how programmers use regexes in other programming languages.

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.

8. RELATED WORK

Regular expression understandability has not been studied directly, though prior work has suggested that regexes are hard to read and understand since there are tens of thousands of bug reports related to regular expressions [3]. To aid in regex creation and understanding, tools have been developed to support more robust creation [3] or to allow visual debugging [10]. Other research has focused on removing the human from the creation process by learning regular expressions from text [5, 6].

Regular expression refactoring has also not been studied directly, though refactoring literature abounds [11–13]. The closest to regex refactoring comes from research toward expediting regular expressions processing on large bodies of text [14], similar to refactoring for performance.

Code smells in object-oriented languages were introduced by Fowler [15]. Researchers have studied the impact of code smells on program comprehension [7, 8], 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 [16–19]. 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 [18, 19].

Exploring language feature usage by mining source code has been studied extensively for Smalltalk [20], JavaScript [21], and Java [22–25], and more specifically, Java generics [24] and Java reflection [25]. Our prior work [26] 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.

9. CONCLUSION

In an effort to find refactorings that improve the understandability of regexes, we created five equivalence class models and used these models to investigate the most common representations and most comprehensible representations per class. We identified three strongly preferred transformations between representations (i.e., $\overrightarrow{T4T1}$, $\overrightarrow{D2D3}$, and $\overrightarrow{L2L3}$). 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.

Acknowledgements

This work is supported in part by NSF SHF-EAGER-1446932.

10. REFERENCES

[1] A. S. Yeole and B. B. Meshram, "Analysis of different technique for detection of sql injection," in *Proceedings of the International Conference & 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

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

[3] 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

[4] 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

[5] 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

[6] 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

[7] 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.

[8] 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.

[9] I. Balaban, F. Tip, and R. Fuhrer, "Refactoring support for class library migration," *SIGPLAN Not.*, vol. 40, no. 10, pp. 265–279, Oct. 2005. [Online]. Available: http://doi.acm.org/10.1145/1103845.1094832

[10] 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

[11] T. Mens and T. Tourwé, "A survey of software refactoring," *IEEE Trans. Soft. Eng.*, vol. 30, no. 2, pp. 126–139, Feb. 2004.

[12] W. F. Opdyke, "Refactoring object-oriented frameworks," Ph.D. dissertation, Champaign, IL, USA, 1992, uMI Order No. GAX93-05645.

[13] 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

[14] 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

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

[16] F. Hermans, M. Pinzger, and A. van Deursen, "Detecting code smells in spreadsheet formulas," in *Proc. of ICSM '12*, 2012.

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

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

[19] ——, "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

[20] 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

[21] 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

[22] 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

[23] 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

[24] 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

[25] 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

[26] C. Chapman and K. T. Stolee, "Exploring regular expression usage and context in python," January 2016, under review.