RulePad: Interactive Authoring of Checkable Design Rules

Sahar Mehrpour
smehrpou@gmu.edu
George Mason University
Fairfax, Virginia, USA

Thomas D. LaToza
tlatoza@gmu.edu
George Mason University
Fairfax, Virginia, USA

Hamed Sarvari
hsarvari@gmu.edu
George Mason University
Fairfax, Virginia, USA

ABSTRACT

Good documentation offers the promise of enabling developers to easily understand design decisions. Unfortunately, in practice, design documents are often rarely updated, becoming inaccurate, incomplete, and untrustworthy. A better solution is to enable developers to write down design rules which are checked against code for consistency. But existing rule checkers require learning specialized query languages or program analysis frameworks, creating a barrier to writing project-specific rules. We introduce two new techniques for authoring design rules: snippet-based authoring and semi-natural-language authoring. In snippet-based authoring, developers specify characteristics of elements to match by writing partial code snippets. In semi-natural language authoring, a textual representation offers a representation for understanding design rules and resolving ambiguities. We implemented these approaches in RulePad. To evaluate RulePad, we conducted a between-subjects study with 14 participants comparing RulePad to the PMD Designer, a utility for writing rules in a popular rule checker. We found that those with RulePad were able to successfully author 13 times more query elements in significantly less time and reported being significantly more willing to use RulePad in their everyday work.

CCS CONCEPTS

• **Software and its engineering** → **Software maintenance tools**; • **Human-centered computing** → *Interactive systems and tools.*

KEYWORDS

design rules, documentation, programming tools, bug finding, static analysis

ACM Reference Format:
Sahar Mehrpour, Thomas D. LaToza, and Hamed Sarvari. 2020. RulePad: Interactive Authoring of Checkable Design Rules. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3368089.3409751

1 INTRODUCTION

Writing good documentation has long been viewed as key to helping developers successfully follow and understand the rationale behind design decisions, helping to prevent software defects and code decay [24]. A design decision encompasses rationale explaining developers' reasoning which led to the decision as well as constraints imposed by the design decision on how code can be written. In this paper, we refer to the constraints imposed by a design decision as a *design rule* [4]. Traditional approaches to documentation rely on developers first writing decisions down in documents or comments and then consistently updating the documentation as the code changes. However, this often does not occur in practice [21], resulting in documentation which is inaccurate, incomplete, and untrustworthy [14], and encouraging developers to ignore documentation and reverse engineer design directly from code [20]. This makes questions about design rationale some of the most frequently reported hard-to-answer questions [19] and one of the most serious problems developers report facing [20].

A potential solution is to ensure that design rules are *checkable*, expressed in a representation where a program analysis tool can continuously check for inconsistencies and flag divergences between the documented rule and code [23]. To achieve this, developers might use static analysis tools such as CheckStyle [1], PMD [7], and FindBugs [15] to write custom, project-specific rules describing a defect pattern to avoid. However, existing tools require developers to either write program analyses in a general purpose programming language (e.g. FindBugs [15]) or use complex query notations to describe patterns (e.g. XPath in PMD [7]). This restricts authoring and changing design rules to those with specialized program analysis knowledge, preventing most developers from creating or maintaining design knowledge through checkable design rules.

To author checkable design rules, developers should be able to work with design rules in simple and expressive representations. We propose two techniques for authoring checkable design rules: *snippet-based authoring* in which design rules are represented in code-based templates and *semi-natural language authoring*. These techniques are complimentary, offering different levels of simplicity and expressiveness. Snippet-based authoring enables developers to easily express rule through templates that look like code, but may be ambiguous with complex rules. Semi-natural language authoring lets developers author design rules in an expressive and unambiguous textual representation, but while simpler than complex query notations, requires the developer to learn a semi-natural language.

We implemented these techniques in a system for authoring checkable design rules, RulePad, as a design rule Graphical Editor and Textual Editor. As developers edit design rules in the Graphical Editor, they receive immediate feedback and can view examples of code which satisfies and violates the rule. As a rule is constructed in the Graphical Editor, RulePad constructs a natural language textual representation of the rule using the grammar of the Textual Editor, which is bidirectionally synchronized. Developers may also edit or author new rules using the textual representation, including resolving ambiguities in situations where the graphical representation of a rule is ambiguous (e.g., elements are combined through either conjunction or disjunction).

To evaluate the ability of RulePad to enable developers to author checkable design rules more easily, we conducted a user study. 14 participants were asked to author design rules with either RulePad or PMD, an existing widely-used static rule checker. This study also offered a first usability evaluation of using PMD to author custom rules. Participants with RulePad were able to successfully author 13 times more query elements in significantly less time and reported being significantly more willing to use RulePad in their everyday work. Participants working with the PMD Designer experienced challenges with both the tool and its documentation which hindered them from making significant progress.

In the following sections, we first review design rules, present a motivating example, describe the two approaches for authoring design rules, describe the design of our system, and report the results from our evaluation study. We conclude with related work and a discussion of potential future directions.

2 DESIGN RULES

When writing code, developers make design decisions choosing between alternatives, formulating design rules specifying constraints to ensure that code is consistent with the design. Violating these constraints causes code to decay and drift from the originally intended design, which may alter the code's behavior, reduce its maintainability, or prevent extensibility.

Design rules vary in their scope and complexity. Some may manifest as statement-level constraints. For example, consider an application in which data is persisted. To ensure correct retrieval of persisted data in concurrently executing code, a design rule might impose immediate persistence, requiring a specific method call to immediately save data:

`Save()` calls should always be committed immediately.
IF the `save()` method of the persistence library is called *THEN* the `now()` method must be followed immediately.

```
ofy().save().entity(this).now();
```

Design rules may also manifest as constraints on how classes or methods are declared. For example, in order for specific data to be persisted as expected, a design rule might dictate that all subclasses must be persisted by requiring a specific annotation:

Artifacts should be marked for persistence with Entity annotations.
IF an object is an artifact subclass *THEN* it needs to be an entity marked with `@Subclass`.

```
@Subclass(index=true)
public class ADT extends Artifact { ... }
```

Design rules may also express higher-level constraints, such as an architectural style. High-level design rules may be decomposed into lower-level design rules that can then be directly checked. For example, adopting a sharded architectural style might be expressed through several rules. One might specify a constraint to create intermediate classes (a Command) to capture communication between shards and a second might specify how sharded commands are executed.

All microtask commands must be handled by `Command` subclasses.
IF a method is a `static` method on Command *THEN* it should implement its behavior by constructing a new `Command` subclass instance.

```
public abstract class FunctionCommand extends Command {
public static FunctionCommand create(...) {
return new Create(...); }
protected static class Create extends FunctionCommand { .. }
... }
```

`Commands` must implement *execute*.
IF a class is a subclass of `Command` *THEN* it must implement execute.

```
public abstract class ADTCommand extends Command {
protected static class Create extends ADTCommand {
public void execute(Function funct, String projectId) {
... } ... }
```

Design rules can be expressed through an IF/THEN structure. This structure specifies *when* and *how* it should apply. This can be expressed through a *quantifier* describing when the rule applies and *constraints* describing what must be true.

3 MOTIVATING EXAMPLES

We illustrate how RulePad enables developers to author design rules through three examples.

3.1 Basic Authoring

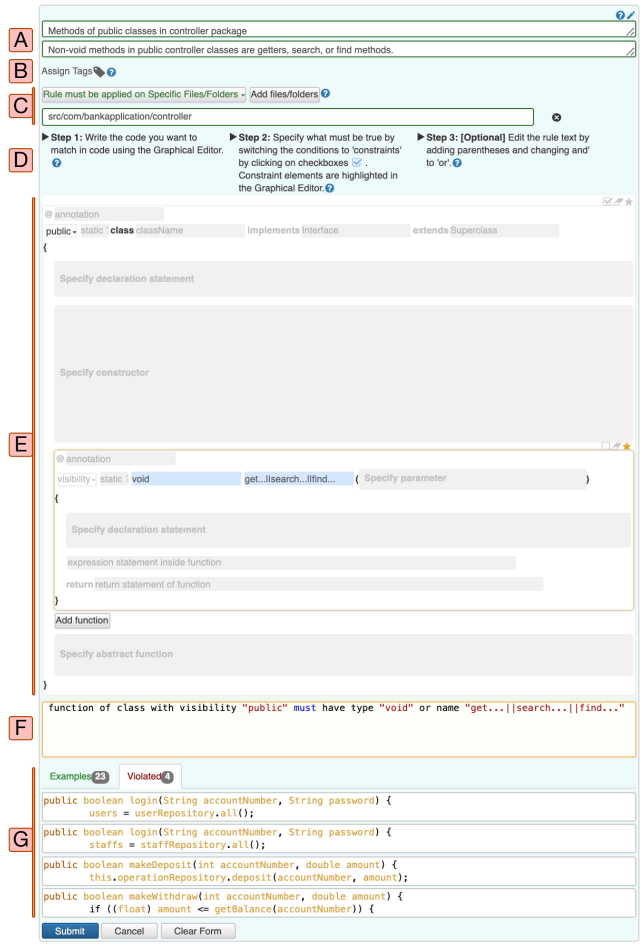
Alice is a developer working on a bank application[1]. While implementing a feature, she formulates a design rule about the responsibilities of classes in the `Controller` package by expressing a constraint on the allowed public methods.

> If a method is in a `public` class, then it must be `void` or have a name which begins with `get`, `search`, or `find`.

Alice opens *RulePad* and reads a brief tutorial explaining each element of the interface and offering an example of how to write a simple design rule. She writes a title and a description explaining the rationale, assigns tags, and specifies that the rule should apply only to files in the `controller` package (Figure 1.A-C). Alice notices the dynamic guide (Figure 1.D), which lists three steps, with the first highlighted. The first step instructs her to describe the code to matched in the GRAPHICAL EDITOR. Alice writes the IF part of the rule, `public` class, specifying `public` in the dropdown (Figure 1.E).

The dynamic guide now indicates that the first step has been finished and highlights the second step. The second step asks her to specify the constraints which matched code must satisfy by marking one or more of the elements as constraints. She specifies that the function return type should be `void`. Realizing she needs to match function names which start with a prefix, she writes `get...||search...||find...` to match identifiers which begin with `get`, `search`, or `find`. She then checks the boxes to indicate that both the function return type and identifier should be treated as constraints.

[1] https://github.com/derickfelix/BankApplication



Methods of public classes in controller package

Non-void methods in public controller classes are getters, search, or find methods.

src/com/bankapplication/controller

▶ **Step 1:** Write the code you want to match in code using the Graphical Editor.

▶ **Step 2:** Specify what must be true by switching the conditions to 'constraints' by clicking on checkboxes ✓. Constraint elements are highlighted in the Graphical Editor.

▶ **Step 3: [Optional]** Edit the rule text by adding parentheses and changing and' to 'or'.

```
function of class with visibility "public" must have type "void" or name "get...||search...||find..."
```

```
public boolean login(String accountNumber, String password) {
        users = userRepository.all();
```

```
public boolean login(String accountNumber, String password) {
        staffs = staffRepository.all();
```

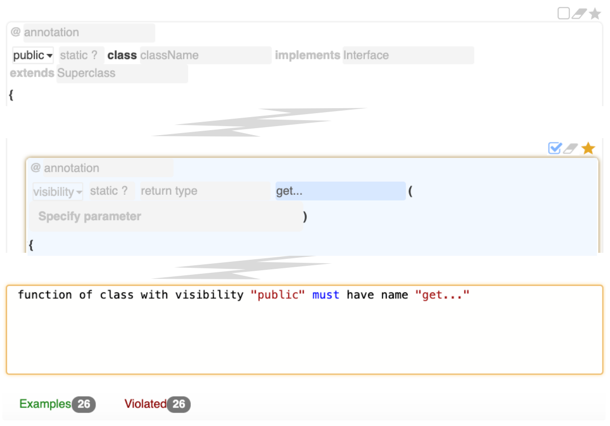
```
public boolean makeDeposit(int accountNumber, double amount) {
        this.operationRepository.deposit(accountNumber, amount);
```

```
public boolean makeWithdraw(int accountNumber, double amount) {
        if ((float) amount <= getBalance(accountNumber)) {
```

Figure 1: A developer using RulePad to create a new design rule may first specify a title and description (A), assign tags (B), and specify where the rule applies (C). Using the Graphical Editor (E), the developer may then write code snippets in a structured editor, interactively refining the rule to resolve ambiguity. Developers may verify the rule by (F) reading a description in the Textual Editor, which they may also optionally edit. While authoring rules, developers may (G) view a list of code snippets which satisfy and violate the rule to check its behavior.

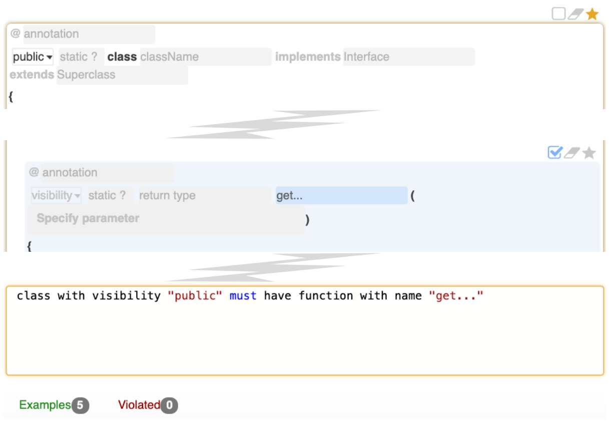
Returning to the dynamic guide, the third step suggests she consider editing the rule in the Textual Editor to change 'and' to 'or', if necessary. She reads the generated text: `class with visibility "public" must have function with (type "void" and name "get...|| search...||find...")`. Realizing this is not what she intended, she edits 'and' to 'or' (Figure 1.F).

To check if the rule behaves as she expects, Alice inspects the list of matches and violations (Figure 1.G). She sees that the rule is still not quite as she intended. Two additional method identifiers should be included: `login` and `make...`. She edits the method name pattern matching expression in the Graphical Editor. Glancing



```
function of class with visibility "public" must have name "get..."
```

Figure 2: By default, RulePad use the lowest common ancestor of constraint elements as the Element of Interest. In this example, there is a single constraint element, "get...". RulePad selects the containing element, a method, as the Element of Interest, indicated by a golden star.



```
class with visibility "public" must have function with name "get..."
```

Figure 3: Clicking the golden star on the class updates the Element of Interest to the class. This is reflected in the updated textual description and updated list of violations.

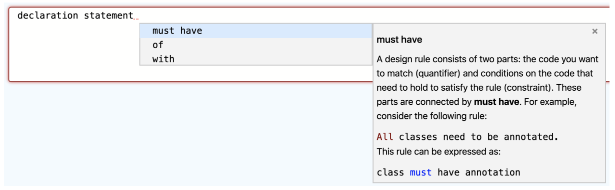
again at the updated list of matches and violations, she finds that the rule now works as she intends.

3.2 Advanced Authoring

Working on another feature, Alice formulates a rule:

> If a class is `public`, then it must have at least one method whose name begins with `get`.

While creating the design rule, she notices that the Textual Editor indicates that it is matching the *methods* in `public` classes rather than the `public` *classes* themselves. The tool indicates that there are 26 violations (Figure 2), which should not be violations. Recalling the tutorial, she realizes that she needs to set the class to be the element of interest. She clicks on the star on the right side of the class element. The text in the Textual Editor now indicates that `public` classes will be matched (Figure 3).



```
declaration statement
    must have
    of
    with
```

must have

A design rule consists of two parts: the code you want to match (quantifier) and conditions on the code that need to hold to satisfy the rule (constraint). These parts are connected by **must have**. For example, consider the following rule:

`All classes need to be annotated.`

This rule can be expressed as:

`class must have annotation`

Figure 4: Developers can write design rules using the Textual Editor. Autocomplete provides suggestions, including documentation and examples of its use.

3.3 Authoring Design Rules through the Textual Editor

Alice formulates a rule that

> If there is a field in a class, then it must be `private`.

After seeing the representation of the past rules she has authored in the Textual Editor, she decides to write the rule using the Textual Editor. Based on her prior experience, she begins the design rule with the Element of Interest, `declaration statements`. As she types, autocomplete provides suggestions (Figure 4). Alice reads the documentation of each suggestion and selects the next token by clicking on the suggestion.

4 TECHNIQUES FOR AUTHORING DESIGN RULES

Developers traditionally document design rules in design documents, such as by using example code snippets describing how to write code correctly or in prose describing constraints on how code should be written. RulePad enables developers to author and work with design rules in these forms.

A key challenge in supporting authoring design rules is the inherent tradeoff between simplicity and expressiveness in representing rules. We introduce snippet-based authoring and semi-natural language authoring as complimentary techniques for authoring checkable design rules. Representing the design rule in a representation close to what will be matched —code— enables a direct manipulation interaction, where developers can rapidly write and edit what they wish to see matched and immediately see what is matched. Using the Graphical Editor, developers can then begin to refine rules in increasingly complex ways, using code-snippets to express the essence of the rule and interactively refining it as necessary. In more complex rules, additional information is needed to resolve ambiguities. The Textual Editor enables developers to understand how the code snippets they have added are composed to form a design rule as well as enabling more experienced developers more control. Our key goal was to keep the design of of RulePad simple enough that developers inexperienced with program analysis and query notations are able to rapidly author design rules while making it expressive enough to cover most of the design rules expressible in existing AST-based rule checkers.

4.1 Snippet-Based Authoring

Enabling developers to author design rules using code snippets is a natural and appealing approach. Developers already know code, and can simply write down what they wish to match. However, code snippets by themselves are often ambiguous. Consider:

```
"Mapper String"
```

Mapper seems to be a name, but for what? And how is String related? Additional information is needed to determine what AST elements these code snippets should be matched against and how these elements are related. Using code-snippets entered in a traditional text editor is not straightforward, as the text is ambiguous.

To address this challenge, we introduce rule authoring through code-based templates. The code-based template enables developers to specify this information using a structured editing experience to select options and type text into boxes which correspond to specific AST elements that the developer wishes to match.

Other additional information is sometimes needed. Elements may define a quantifier or constraints of design rules (Section 2), describing either when the rule applies or what code must exist when it does apply. Design rules must also specify an Element of Interest (EoI) describing to which element it applies. Without specifying this, the rule may be ambiguous. For example, while the design rules in Figures 2 and 3 have identical partial code snippets, they have different meanings. The design rule in Figure 2 applies to functions, indicating that all public functions must be getters. The design rule in Figure 3 instead applies to classes, indicating that all public classes must have getters. To resolve this ambiguity, developers can interactively indicate the element of interest in the interface.

4.2 Semi-Natural-Language Authoring

Developers often author design rules in text, as it can contain explanations and other information about design rules. However, natural language text is difficult to use directly by tools, particularly as it may be ambiguous.

Traditionally, rule checkers have met this challenge by introducing specialized query notations, such as XPath queries. However, these notations are often unnatural, with little resemblance to natural language. This imposes a barrier to both rule authors and readers. To understand rules written in these notations, developers first need to know the syntax and meaning of the notation. To author rules, users must learn how to express the desired rule correctly in the notation. In particular, developers must learn how specific AST elements map into a corresponding representation in the query notation.

To make it easier for developers to work with design rules in textual form, we created a semi-natural language for design rules. Our goal was to create a language with easily understandable syntax and semantics, simplifying the complexity of the underlying mapping to the complex notations used by analysis tools. The language is translated into XPath queries by RulePad, but masks the complexity of this notation from the developer.

Design rules are expressed using an IF/THEN structure, with a quantifier followed by a constraint. In the design rule syntax, this take the form of text which first describes the quantifier, followed by 'must have', and then text describing one or more constraints. Rather than use an IF/THEN syntax, we chose to encode the IF/THEN structure using a must have syntax, as it is more compact. For example, compare '`class with visibility "private" must have function`

Table 1: The semi-natural language offers a more natural and compact representation of design rules. The top row lists a design rule and its corresponding representation in the semi-natural language (second row) and a traditional query language, XPath (bottom row).

Model classes should have 'private' fields and getters.
`class must have declaration statement with visibility "private" and function with name "get..."`
`//CompilationUnit[PackageDeclaration/Name[@Image="com.bankapplication.model"]]//ClassOrInterfaceDeclaration[count(ClassOrInterfaceBody/ClassOrInterfaceBodyDeclaration/FieldDeclaration[@Private="true"])=0 or count(ClassOrInterfaceBody/ ClassOrInterfaceBodyDeclaration/MethodDeclaration/MethodDeclarator[starts-with(@Image,"get")])=0]`

`with name "get..."`' and '`IF class has visibility "private" THEN it must have function with name "get..."`'. The must have syntax requires 10 words and only one reference to 'class', while in the IF/THEN syntax, there are 13 words and two references to 'class', one potentially ambiguous.

There are 6 main rules in the grammar.

(1) Values to match in the code are specified through double-quoted text, with pattern matching allowed.

```
"...Mapper"
```

(2) Conditions imposed on an element are listed after the element. AST children of an element may be specified using **'with'** and an AST parent specified using **'of'**.

```
function with parameter of class
```

(3) Quoted values are not prefixed with 'with', except for implementation and extension which are prefixed with 'of'.

```
name "...Mapper"
implementation of "I..."
extension of "...Repository"
```

(4) The quantifier is specified first, followed by **'must have'**, followed by the constraints.

```
function with type "void" of class must have name
"update||destroy"
```

(5) Multiple AST children may be specified using **'and'** and **'or'**.

```
function must have name "...Mapper" and visibility
"public" or type "void"
```

(6) Parentheses can be used around AST children to resolve ambiguity.

```
function must have ((name "...Mapper" and visibility
"public") or type "void")
```

5 RULEPAD

RulePad enables developers to work with design rules through a Textual Editor and Graphical Editor. RulePad integrates these into a unified interface for authoring design rules, built on top of the ActiveDocumentation system for working with design rules [23].

The Textual Editor and the Graphical Editor are bidirectionally synchronized, enabling developers to switch back and forth as they incrementally construct a design rule (Figure 5). Over time, this may facilitate a developer incrementally learning how to express specific rules in the semi-natural language. Code written in the Graphical Editor is processed by a graphical editor parser to generate a textual representation according to the grammar (**brown** path in Figure 5). If the text in the Textual Editor is changed, the

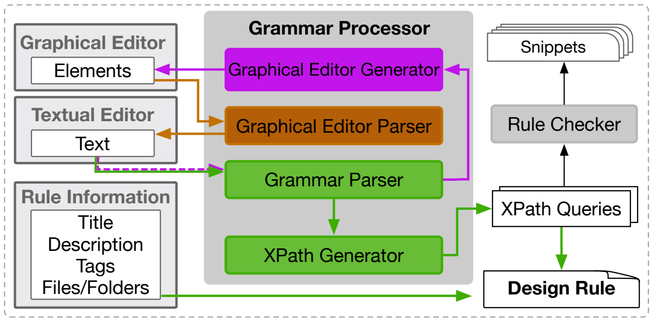


Figure 5: The grammar processor bidirectionally synchronizes the Graphical Editor with the Textual Editor and enables checking rules against code through XPath.

grammar parser, built on the ANTLR grammar parser [25], generates a parse tree. The parse tree is then traversed to create code in the Graphical Editor (**purple** path in Figure 5).

In RulePad, design rules are checked against the code for conformance. XPath queries are first generated by traversing the parse tree and mapping each node to an XPath element (**green** path in Figure 5). For example, the "*child* **of** *parent*" element in the grammar is mapped to `"parent/child"` in XPath, and "*element* **with** *conditions*" is mapped to `"element[conditions]"` in XPath. The **must have** construct enables creating two XPath queries, the quantifier and the constraints queries. As in many traditional rule checkers, RulePad checks the XPath queries against code through a custom rule checker. Each design rule is translated into two separate XPath queries, corresponding to the quantifier and constraint. By comparing the results from each query, RulePad is able to identify code snippets that satisfy the rule as well as violations.

5.1 The Graphical Editor

The Graphical Editor offers developers a design rule template to begin authoring, uses progressive disclosure to reduce visual clutter, supports pattern matching of identifiers through pattern matching expressions, supports specifying an element of interest, and supports labeling conditions as a quantifier or constraint.

5.1.1 Design Rule Template. Developers interact with design rules in the Graphical Editor through a code-based template which displays an outline of a class declaration as the main element. Within the template, common characteristics for each element are displayed, which the user may choose to specify in the design rule. For example, for a class, its annotations, visibility, specifier (e.g.

static), name, superclass, and interfaces it implements are displayed. Contained within the template of the class are additional elements which may also be specified and added to the rule: fields, constructors, methods, and abstract methods. Additional instances of an element may be added to the rule using the "Add [name of the element]" button.

5.1.2 *Progressive Disclosure.*

To reduce clutter and distraction, the Graphical Editor employs three levels of progressive disclosure, graying out text until needed. Elements may be inactive, of potential interest, or active. Inactive elements have no specified characteristics and have no role in the rule. As they are not yet relevant, they are indicated with a lighter font color or overlaid shading. When the user hovers over an inactive element, the element becomes of potential interest and the visibility of its color is increased. After elements are given at least one specified characteristic, they become active and are incorporated into the design rule. Active elements are highlighted with a darker font (Figure 1.E).

5.1.3 *Pattern Matching.*

The Graphical Editor supports pattern matching expressions with a notation that is code-like and may include simple wild cards. These expressions enable users to write patterns that match textual values including *ending with a sub-string*, *starting with a sub-string*, *containing a sub-string*, their negations, and combinations of patterns. To avoid confusion with the dereference operator in C++, three dots are used to denote *any sequence of characters*. For example, `...Repository` matches names ending with `Repository`, `!BaseRepository` matches names that are not `BaseRepository`, and `!BaseRepository&&...Repository` matches names that are not equal to `BaseRepository` and end with `Repository`.

5.1.4 *Constraints.*

Constraints specify what must be true about the quantifier to satisfy the rule. In the Graphical Editor, both the quantifier and the constraints are displayed in a single view. By default, elements are set as a quantifier when added. This enables users to then incrementally add constraints by using checkboxes to toggle elements from a quantifier to a constraint. If an element is selected as a constraint, all of its properties (the children of its AST node) are also marked as constraints. Constraints are indicated through a blue background color (Figure 1.E).

5.1.5 *Dynamic Guide.*

In evaluations of early prototypes, we found that users sometimes got stuck with how to make progress in authoring a design rule, particularly in initially discovering that each design rule must include a constraint. To help users understand where to start, the dynamic guide walks developers through the steps of authoring a design rule. As the developer completes each step, the next is highlighted. The user is prompted to write a quantifier, specify a constraint, and optionally edit the logical connectives between conditions.

5.1.6 *Elements of Interest.*

A default EoI is generated by computing the *lowest common ancestor* of all constraints, indicated through a golden star. For example, if the name of a function and its parameter type are both specified and marked as constraints, the function declaration is selected as the EoI for the rule. Elements which may be selected as the EoI are indicated through a grey star.

Table 2: The semi-natural language grammar. Bold indicates terminal nodes, italic non-terminals, brackets optional nodes, and curly brackets zero or more occurrences. *symbol* specifies valid Java symbols such as parentheses and dot, and *character* specifies valid characters for Java identifiers.

1	*DR* ::=	*c* **must have** *cExp* | *m* **must have** *mExp* | *am* **must have** *amExp* | *co* **must have** *coExp* | *d* **must have** *dExp* | *p* **must have** *pExp*
2	*c* ::=	**class** [**with** *cExp*] [**of** *c*]
3	*m* ::=	**function** [**with** *mExp*] [**of** *c*]
4	*am* ::=	**abstract function** [**with** *amExp*] [**of** *c*]
5	*co* ::=	**constructor** [**with** *coExp*] [**of** *c*]
6	*d* ::=	**declaration statement** [**with** *dExp*] [**of** (*c* | *m* | *co*)]
7	*p* ::=	**parameter** [**with** *pExp*]
8	*t* ::=	**type** [*pattern* | *expr*]
9	*e* ::=	**extension of** (*pattern* | **superclass**)
10	*im* ::=	**implementation of** (*pattern* | **interface**)
11	*ex* ::=	**expression statement** [*expr*] [**of** (*m* | *co*)]
12	*i* ::=	**initial value** [*expr*] [**of** *d*]
13	*r* ::=	**return value** [*expr*]
14	*a* ::=	**annotation** [*expr*]
15	*n* ::=	**name** [*pattern*]
16	*s* ::=	**specifier** [*pattern*]
17	*v* ::=	**visibility** [*pattern*]
18	*cExp* ::=	**(***cExp***)** | *cExp op cExp* | *cExp* | *a* | *s* | *v* | *n* | *e* | *im* | *m* | *am* | *co* | *d* | *c* | *r*
19	*mExp* ::=	**(***mExp***)** | *mExp op mExp* | *mExp* | *a* | *s* | *v* | *t* | *n* | *p* | *r* | *d* | *ex*
20	*amExp* ::=	**(***amExp***)** | *amExp op amExp* | *amExp* | *a* | *s* | *v* | *t* | *n* | *p*
21	*coExp* ::=	**(***coExp***)** | *coExp op coExp* | *coExp* | *a* | *s* | *v* | *p* | *r* | *d* | *ex*
22	*dExp* ::=	**(***dExp***)** | *dExp op dExp* | *dExp* | *a* | *s* | *v* | *t* | *n* | *i*
23	*pExp* ::=	**(***pExp***)** | *pExp op pExp* | *pExp* | *t* | *n*
24	*op* ::=	**and** | **or**
25	*pattern* ::=	**"** { *part* (**&&** | **||**) } *part***"**
26	*part* ::=	[**!**][**...**] *character*{*character*} [**...**]
27	*expr* ::=	**"** (*character* | *symbol*) {*character* | *symbol*} **"**

5.2 The Textual Editor

The Textual Editor enables developers to author and edit design rules in a semi-natural language. To help developers begin to understand and use the grammar, RulePad offers developers auto-complete suggestions. To help users understand the meaning of each potential completion, each is accompanied by context-specific documentation. In addition, developers may investigate the meaning of a token in the Textual Editor by following a visual link to the equivalent element in the Graphical Editor.

5.2.1 *Grammar.*

The semi-natural language is defined by a grammar, implemented using the ANTLR parser generator [25] (Table 2). The grammar currently supports 16 AST element types (rules 2-17), including class declarations, method and constructor declarations, declaration statements, and expression statements. Using the grammar, design rules are created with a structural construct of an AST element followed by **must have** and the expression construct of the same AST element (rule 1). The structural construct of an AST element starts with a terminal node (i.e., name of the element) which is followed by expression constructs, as children, ([**with** **Exp*]) or values (*pattern* and *expr*), and if applicable, followed by parent constructs ([**of** *]). Expression constructs (rules 18-23) allow combining child elements with conjunctions or disjunctions (*op*, rule 24) and parentheses. The *pattern* rule (rule 25) defines the structure for matching different parts of a pattern (rule 26) used for identifiers and keywords. The *expr* rule (rule 27) enables flexibly matching Java expressions. For example, a design rule might have the form:

class with name *pattern* **must have function with specifier** *pattern* **and return value** *expr*

such as:

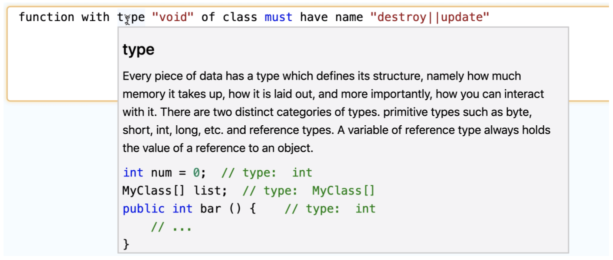
class with name "...Cls" must have function with specifier "static" and return value "new ArrayList<String>()"

The language can be easily extended additional rules to the grammar and describing how each rule is mapped to an equivalent XPath query.

5.2.2 Logical Connectives. One challenge with representing a design rule through partial code snippets is that there is no natural way to express how partial code snippets should be combined. In some cases, the Graphical Editor is able to resolve ambiguity by providing a default and enabling developers to change the default (e.g., by toggling a condition between a quantifier and constraint). However, in other cases it is clearer to express the meaning of the rule in a textual representation. One specific challenge is how to combine conditions: should including multiple conditions be interpreted as a conjunction or disjunction of conditions? By default, RulePad uses conjunction. The Textual Editor makes this choice explicit to the developer by inserting an 'and' terminal between conditions and enables them to edit the rule text to switch 'and' to 'or'. In addition, developers can use parentheses in the Textual Editor to resolve association ambiguities.

5.2.3 Autocomplete. Design rules written in the Textual Editor must follow the grammar. To enable developers to more easily learn the grammar, developers may first use the Graphical Editor to author design rules and then view the corresponding textual representation. In addition, the Textual Editor offers autocomplete. After writing a token, autocomplete suggests potential next tokens that are valid given the grammar. This enables developers to see a list of valid completions they may write next.

The parser is unable to parse unfinished design rule text. Consequently, the parse tree is not available for autocomplete. In these cases, the Textual Editor uses the information in the text and a set of rules to suggest valid next tokens. While providing suggestions for some cases is trivial (e.g. 'must' is always followed by 'have'), providing suggestions for other cases requires more logic. For example, the list of available tokens after 'must have' includes the AST children of the EoI, the first token in the text. However, if the EoI is a 2-part word (e.g. `declaration statement`), considering only the first token is not useful. In another example, 'with' may be followed by possible AST children of the second to the last token.



```
function with type "void" of class must have name "destroy||update"
```

type

Every piece of data has a type which defines its structure, namely how much memory it takes up, how it is laid out, and more importantly, how you can interact with it. There are two distinct categories of types. primitive types such as byte, short, int, long, etc. and reference types. A variable of reference type always holds the value of a reference to an object.

```
int num = 0;  // type:  int
MyClass[] list;  // type:  MyClass[]
public int bar () {    // type:  int
    // ...
}
```

Figure 6: After moving the cursor over the word *type*, in-context documentation describes its meaning and usage.

If there is no autocomplete suggestion for the input text, the Textual Editor checks the input text against the grammar, offering an error message for simple grammar violations. For example, if a developer writes a design rule with two instances of 'must', an error message indicates that only one 'must' is allowed, and the second 'must' is decorated with a red wavy underline.

5.2.4 In-Context Documentation. When the developer moves their cursor over a word in the design rule, the developer is offered in-context documentation with an explanation of the word (Figure 6). This enables learning the meaning of element names and keywords through descriptions and simple examples.

5.2.5 Linking the Graphical Editor to the Textual Editor. Moving the cursor over an element name in the Textual Editor highlights the corresponding element in the Graphical Editor with a corresponding color. This enables the developer to easily edit rules in either representation and facilitates using both together to understand the current design rule.

6 EVALUATION

To evaluate our approach for authoring design rules, we conducted an experiment comparing the experiences of developers authoring design rules in RulePad and in PMD[2]. The independent variable was the tool used by participants (RulePad or PMD), and the dependent variables included task time, errors, and reported willingness to use the tool in their own work. We chose to compare RulePad against PMD, a widely used rule checker. Rule checkers, such as PMD, offer developers the ability to author custom rules expressing constraints to be checked against code. We selected PMD as PMD rules can be written using XPath, which RulePad uses internally to check rules, enabling a direct comparison. In addition, no prior studies have examined the challenges developers may experience in authoring rules in PMD or other similar tools. PMD offers the Designer utility to write rules in XPath. Developers supply an example code snippet to which the rule can be applied, for which Designer then displays an AST. By referencing this generated AST, the developer may then write an XPath query expressing the desired rule (Table 1). The Designer utility then executes the XPath query against the supplied example code snippet, describing which parts, if any, of the supplied code snippet violate the rule.

[2]The tool and the study materials are available at https://github.com/devuxd/RulePad.

6.1 Method

To recruit participants, we invited students in relevant courses, posted flyers on bulletin boards, and sent email invitations to Computer Science masters and Ph.D. students. 14 senior undergraduate and graduate students at our institution participated in the study. All had prior experience in Java. Participants had a median of 5 years of programming experience (range 1.5 to 10) and 1.5 years of experience as a professional software developer (range 0 to 4). Participants reported using a variety of programming tools, including text editors (Vim, Atom, Sublime Text, Evernote), version control systems, and issue trackers (Git). Control participants reported a range of levels of experience with ASTs and XPath. Three (C3, C1, C4) reported no prior knowledge, two (C6 and C2) reported some knowledge of ASTs but none of XPath, one (C7) reported limited knowledge in XPath but none in ASTs, and one (C5) reported some knowledge in XPath and reported having enough knowledge in ASTs. Each participant was compensated with a $40 gift card for the 2-hour study session.

We used stratified random sampling to assign participants to one of two conditions, a control group (C1-C7) and an experimental group (E1-E7), balancing the levels of overall and professional experience reported in the pre-study survey. Participants in both conditions had a similar distribution of programming experience (median 5 years for both groups) and professional experience (median 1 years for control and 1.5 for experimental).

Participants in both groups were given an introduction about design rules and how design rules can be expressed using an IF/THEN structure. Then participants were asked to extract at least 3 design rules from the given source code. All were given 20 minutes to finish this task. Participants were then given a brief tutorial introducing RulePad or the PMD Designer (Version 6.12.0). Experimental participants were given access to RulePad and control participants were given access to Designer and allowed to use the Internet. Participants were given 80 minutes to complete the main task. As participants worked, we asked them to think aloud and captured a screen and audio recording. At the end of the study, we collected the final authored design rules and conducted a semi-structured interview. We asked participants about the features they found useful, their challenges, ideas for making the tool better fit their needs, and their willingness to use the tool in their everyday work.

6.2 Tasks

We adapted the source code used in the study from a simple Java project using the model-view-controller architecture[3]. We chose the project as it is complex enough to contain several design rules yet simple enough that the logic of the code can be easily understood.

To familiarize participants with the source code and practice expressing rules in the IF/THEN structure, participants first completed a training task. Participants in both groups were asked to extract at least 3 design rules from the source code and write a textual representation of them in an IF/THEN structure by hand, without the use of any tool. All participants completed the training task (median 16 minutes). In the training task, all except E1 and C5 successfully expressed design rules in the IF/THEN structure.

[3]https://github.com/derickfelix/BankApplication

In the main task, control and experimental participants were asked to use their tool to author 3 specified design rules, and one rule of their choice from the rules they extracted in the training task. The three specified design rules were given in increasing levels of complexity, evaluating the capability of each tool to support authoring rules of increasing complexity. The first rule exercised basic features, the second rule exercised features for authoring complex rules with several constraints, while the third rule required using pattern matching expressions as well as possibly changing the element of interest (Table 3).

6.3 Results

In the main task, all 7 experimental participants were able to begin authoring at least one design rule (containing at least one query element), while 4 of the 7 control participants were unable to begin authoring any design rule. Participants in both conditions had up to 80 minutes to complete the main task. Experimental participants finished the task in a median of 60 minutes (mean 58 minutes, SD=15 minutes). All control participants were still working at the end of the allotted task time, except C2 who gave up after 62 minutes. Using a Shapiro-Wilk test, we were not able to verify that task time data is normal, and we thus used a Mann Whitney U test. Participants with RulePad were significantly faster ($U = 3$, $p < 0.0074$, $r = 0.76$).

6.3.1 *Completeness and Correctness of Design Rules.*

To assess the completeness and correctness of the design rules participants wrote, we scored each design rule. To compare design rules written using the PMD designer to those written using RulePad, we first took each design rule authored in RulePad and extracted the resulting XPath query generated by RulePad. We then scored each response, identifying each required query element necessary in the AST paths defined by the XPath query. Table 3 lists the required query elements for each design rule. For each query element, if the design rule included the necessary AST nodes in the correct format and order, the query element was scored as correct. In addition, we marked design rules which included irrelevant query elements.

Across all required query elements in Rules I, II, and III, experimental participants on average were successful in correctly authoring 52% of the elements. In contrast, control participants succeeded in writing 4% of the elements (Table 3). We used a Shapiro-Wilk test to test the normality of the design rule scores. We were not able to verify the normality of our data, and we thus again used a Mann Whitney U test. Experimental participants authored significantly more correct query elements for Rules I ($U = 0$, $p < 0.0022$, $r = 0.86$) and II ($U = 2$, $p < 0.0004$, $r = 0.84$). Experimental participants wrote more correct query elements for Rule III, but the difference was not significant ($U = 11.5$, $p > 0.3371$, $r = 0.36$). Rule III was more complex, and the six rules experimental participants authored for it did not include all of the required elements.

As Rule IV was selected by each individual participant and varied between participants, we identified required query elements necessary for each separate design rule. On average, design rules authored by participants with RulePad received a score of 72%. Design rules authored by participants in PMD received a score of 0. No control participant succeeded in authoring any part of Rule IV.

Table 3: For each design rule (top), we identified a set of required query elements (numbered items in design rules and listed below) and scored each design rule on the correctness of each query element. The last row lists the mean score across all required query elements for all rules.

	Control Percentage Successful	Experimental Percentage Successful
Rule I: Model (1) classes must have (3) "private" (2) fields and (3)(4) getters.		
(1) class	14%	71%
(2) class field	14%	71%
(3) private class field	0	71%
(4) class method	0	57%
(5) class method name	0	57%
(6) class method name "...get"	0	57%
No Extra Query Elements	14%	57%
Overall Score	2%	63%
Rule II: Repository (1) classes except (2)(3) "BaseRepository" must (4) extend (5) "BaseRepository" and (6) implement an interface and they should have a (7)(8) "...Mapper" (9) function.		
(1) class	14%	100%
(2) class name	14%	14%
(3) class name except "BaseRepository"	0	14%
(4) class implements	0	86%
(5) class extends	0	86%
(6) class extends "BaseRepository"	0	71%
(7) class method name	0	100%
(8) class method name "...Mapper"	0	71%
(9) class method	0	100%
No Extra Query Elements	0	57%
Overall Score	3%	70%
Rule III: (1) Name of (2)(3) "void" (4) functions in Controller classes are either (5) "store", "update", "deposit", "withdraw", or "destroy".		
(1) class method name	0	14%
(2) class method return type	0	0
(3) class method return type "void"	0	0
(4) class method	14%	14%
(5) class method name "store, update, deposit, withdraw or destroy"	0	0
No Extra Query Elements	0	29%
Overall Score	2%	10%
Mean score of required query elements for Rules I, II, and III	4%	52%

6.3.2 *Perceived Value of Tools.*

In post-task interviews, we asked participants about their willingness to use the tool they used within the study in their everyday work. On a 7-point Likert scale (1 being the least preferred), experimental participants reported being significantly more willing to use RulePad (mean 5.2, SD=1.5) than control participants to use the PMD Designer (mean 3.7, SD=1.1)(one-tailed equal t-test, $p < 0.023$). Control participants reported that the main reasons they were willing to use PMD and PMD Designer were its perceived popularity and ability to find defects.

6.3.3 *Challenges using the PMD Designer.*

Only 3 of the 7 control participants were able to author any design rule at all, authoring a total of 4 design rules. Participants using the PMD Designer faced several significant challenges using the tool and its documentation which hindered their progress.

Unclear Expected Notation: All control participants experienced challenges determining the type of input (e.g. text, code snippets, class files, etc.) the tool expected them to enter to author a design rule. The Designer includes a main panel for writing example Java code. However, this was not discoverable by many participants. C3 expected the Designer to be an IDE and tried to write design rules in Java rather than XPath. C7 assumed the tool expected them to enter XML template text, as used in the official website. The Designer user interface consists of three read-only and three editable input boxes. This distinction was unclear. Only two participants (C1 and C4) successfully discovered the component for writing and wrote XPath queries. After using Designer to compute the AST of a sample code snippet, C6 instead directly authored the XPath query in the XML file executed by PMD. C3 tried to import the project into the tool.

Inadequate Documentation: Participants experienced challenges using the steps described in the official documentation due to a lack of detail. Participants reported they were confused about steps and required actions (C4). For example, after the AST is displayed, some participants assumed the rule had already been generated and were confused to see the fields of imported XML text not filled in. At the same time, some (C7, C6) reported that the official tutorial was too long and overwhelming for new users. This tutorial includes information about writing XPath queries. This information misled one participant, who spent considerable time looking for an answer to a specific question about XPath on the website (C1). Participants wished to instead see documentation which included a 'Hello World' example (C7), integrated tutorial (C2), description of what developers needed to do (C4), and a visual tour for new users (C6).

Poor Error Handling: In the PMD Designer, compilation errors for source code and XPath queries are reported through a small icon at the top of the component. In the post-task interview, C4 reported that these were not visible. XPath query matches as well as error messages are reported in the component adjacent to the XPath query field. Compilation error messages did not provide information about how to fix errors. C1 suggested the need for better error handling, in particular localizing the line where the error occurs.

6.3.4 *Experiences and Challenges using RulePad.*

The in-tool tutorial, dynamic guide, and walk-through tutorial helped familiarize participants with the steps for authoring a design rule. All except E4 used the dynamic guide to determine the requested actions in each step. In contrast, only three (E1, E4, E5) read the walk-through tutorial given when they first opened RulePad. All participants used the Textual Editor to verify design rules. E7 and E6 initially authored a design rule in the Textual Editor. But after receiving an error, they continued instead with the Graphical Editor. After completing the steps in the Graphical Editor, they used the Textual Editor to understand the grammar for expressing rules. None of the participants tried to author rules from scratch using the Textual Editor.

In the post-task interview, participants reported several features which helped them to author design rules, including the unambiguous nature of the design rule template (E2, E4, E6, E7), the ability to specify characteristics explicitly (E4), error handling (E6), the pop-up tutorials in the Graphical Editor (E2), automatically ensuring the structure of the rule in the code (E5), and presenting the design rules in a readable form (E1).

The study revealed several usability issues. Some participants misinterpreted the meaning of some of the AST elements in the GUI (E1, E6). 5 participants added extra query elements to at least one design rule. 3 of the 7 experimental participants were unable to successfully use pattern matching. To help users reuse data, RulePad initially populates the template with the data of the previously authored rule and offers a 'Clear Form' button at the bottom of the form. This confused participants, and the 'Clear Form' button was not discoverable by some participants. Some participants were confused by the first step of the dynamic guide, as it prompted them to specify conditions on quantifiers. However, the first design rule did not require a condition on the quantifier. The paths of the files and folders to which the design rule applies follows a standard format with slashes. As there is no validation for these paths, E5 and E6 entered incorrect paths, and E2 and E3 used an incorrect format for specifying paths without receiving any error. As a result, no matches or violations were listed, and participants incorrectly assumed that the tool did not check rules against the code or offer immediate feedback.

7 LIMITATIONS AND THREATS TO VALIDITY

As in all studies, our study has several important limitations and threats to validity. Participants were artificially limited in learning about the tools exclusively by working with the tools and their documentation. In practice, developers may have access to other resources or to experienced teammates. However, developers often rely first on building their own understanding, as they may be expected to make an effort to understand before engaging with teammates [20].

Unlike developers working in their own codebase, participants were unfamiliar with the code they worked with. In practice this situation may occur when developers are new to a team or find themselves working in part of a large codebase with which they are previously unfamiliar. Much of the task involved working with a specific persistence framework, with which developers were unfamiliar. Developers with more experience in a codebase or a framework might benefit less, as they might have internalized more of the design rules. At the same time, as participants wrote design rules and viewed the matching code, participants were unable to use feedback to refine their rules. None knew what the correct behavior of the rule should be, and were thus unable to use the list of matches and violations to judge the correctness of their rule. In practice, we would expect developers authoring a design rule to have a clear expectation of which code snippets should and should not be violations and use this feedback to identify issues with the rule to be resolved. This may have made writing correct rules artificially difficult.

Our selection of participants may threaten the generalization of the results. To mitigate this threat, we recruited participants with a range of experience levels, from one to ten years of industrial experience. In the control group, participants with prior experience working with AST and XPath queries performed slightly better than other participants. Participants with more prior experience with program analysis, or PMD in particular, would likely have performed better.

8 RELATED WORK

A variety of tools have been designed which use a catalog of rules to check for potential defects in code. PMD [7] and FindBugs [15] are widely used tools used for finding statement-level defects. CheckStyle helps developers identify coding style defects [1]. ActiveDocumentation checks code for conformance against design rules and reports matches and violations in code [23]. While most enable developers to write custom rules for their own project, they require developers to interact with specialized query notations or a program analysis framework to do so.

Other tools support analyzing the architectural structure of code [26] through various techniques, including visualizing or querying the architecture of the code. Visualization techniques such as dependency-structure matrices (e.g., Lattix DM [30]) and reflexion models (e.g., SAVE [18]) display the software architecture through matrices and graphs and allow developers to compare them against the intended architecture. Source code query languages (e.g., Semmle.QL [10]) allow developers to query the intended architecture of the code and report textual lists, graphs, and diagrams.

Source code query languages (SCQL) and tools enable developers to query source code to check conformance to potential design rules or find patterns in code. Some offer query languages which are close to natural language [2, 5]. srcQL is a code query language built upon the XML representation of the AST and XPath and is modeled on SQL [5]. Developers are able to query code using a semi-structured natural language in Browse-By-Query (BBQ) [2]. SOUL [11] and Rscript [17] provide functional languages. Unlike RulePad, many source code query tools require knowledge of the internal AST to write a query and full compilation of the code to execute it [10, 11, 17]. More broadly, SCQL tools are specialized for querying code rather than authoring rules.

Studies have investigated the barriers developers face when working with program analysis tools as well as offered approaches to mitigate those issues. Developers face a number of barriers that can lead to misuse of the tools and discourage them from using them [6, 16]. These include a high number of false positives, poor organization of warnings, lack of support for collaborative environments, disconnection from developers' workflow, and difficulties in interpreting the results. Developers report that the integration of tools into their development environment, organized warnings, and immediate feedback are helpful and motivate them to write better code during code reviews [16, 35]. Other approaches include customizing program analysis tools at the project level, which can be more successful than incorporating customization at the user level [29].

Natural language interfaces enable users to directly express queries without learning system commands or interfaces [32]. One early focus was in introducing natural language interfaces to generate database queries (e.g., [3, 28]). PRECISE [27] translates natural language inquiries into corresponding SQL queries by matching tokens discovered in the user query with the database schema. NaLIX takes an English sentence as a query and converts it to an XML query executable on XML databases [22]. Other work introduced natural language interfaces to support data exploration in information visualization applications [8]. One example is *Articulate*, which leverages natural language processing to translate users' crude verbal descriptions into actionable and valid visualization expressions, without requiring a user to learn a complex user interface or query language [34].

A variety of interactive systems have explored approaches for enabling users to interactively refine query text to add necessary details by continuously offering feedback as the user works. Eviza supports users in creating an interactive session with data using natural language and a probabilistic grammar [31]. It enhances natural language queries by first guessing the user's intent and then offering ambiguity widgets to change the guess. DataTone offers a mixed-initiative approach for handling ambiguity in natural language interfaces in the context of data visualization [12]. NLyze offers a robust natural language interface for creating formulas in spreadsheets [13]. Orko utilizes both natural language and direct manipulation of input for network visualization and uses graphical widgets such as dropdowns to resolve ambiguities [33]. FREyA reduces the need for customizing domain-specific languages for each domain by engaging the user in clarification dialogs [9].

9 DISCUSSION

In this paper, we introduced two techniques for authoring checkable design rules: snippet-based authoring and semi-natural-language authoring. These enable developers to work with design rules in more natural and easy to learn representations. Compared to authoring rules in PMD, we found that developers were able to successfully author 13 times more query elements in significantly less time and wrote query elements which were 48% more correct. Developers working with RulePad reported significantly more willingness to use RulePad in their everyday work than developers working with PMD. Developers using RulePad reported benefiting from the unambiguous nature of the design rule template and the ability to specify characteristics explicitly and interactively. Developers used the synchronized Textual Editor to review and better understand the rules they constructed.

Semi-natural language interfaces can sometimes lead to confusion, as it may be hard to learn the limits of what can be expressed in the language as well as constraints imposed by its syntax and grammar on how it should be expressed. RulePad offers several mechanisms to reduce these issues. First, developers may begin writing rules using the Graphical Editor. Second, autocomplete offers a way to discover what can be expressed. Third, the bidirectional connection of the editors enables developers to use rules they have already written using the Graphical Editor to understand how to express them using the Textual Editor, and vice versa. Finally, immediate feedback allows developers to see if the rule behaves as intended by examining the matches and violations generated by the rule.

Our study offers evidence for the challenges developers face in customizing a new program analysis tool for their work context. While we expected the difficulty of authoring XPath queries to be the primary challenge, we found that participants were often confused long before they reached that point. Usability issues working with the tool were often a significant challenge for participants. While diligent and dedicated developers may ultimately succeed, these issues offer a significant barrier dissuading developers from adapting it to their project.

There are a number of ways in which our semi-natural language might be made even closer to natural language. One approach might be to enable developers to define their own vocabulary, giving developers more flexibility in how rules are expressed. This might be achieved through macros. Developers might take a series of tokens in a design rule (e.g, a declaration statement with visibility "private") and offer a short form for this text (e.g., "private fields"). Design rules might then be written and displayed to the user with a mixture of grammar and short form text. Short form text might be saved and shared across design rules, creating a dictionary of alternative terms for expressing common ideas. Autocomplete might assist developers in discovering these short forms.

While our system focuses on authoring design rules which may be expressed as a pattern in an abstract syntax tree, our approach might be applied to authoring design rules which offer constraints over other representations of programs and their execution. The key insight of snippet-based authoring is that, instead of learning a complex notation, developers can instead give examples of what they wish to match and then interactively refine the rules generated from this to resolve any ambiguity. This method of interactive example refinement might be used to construct other forms of specification. For example, ordering relationships might be expressed by listing a sequence of method invocations which must occur. To reduce the ambiguity in interpreting a sequence of method invocations, developers might interactively indicate that some are optional or may occur multiple times. As usability is often a key barrier in adopting specification languages, we believe that interactive example refinement is an approach which may be more widely applicable.

Working with design rules might also be integrated into code transformation systems. If, for example, a developer edited a code snippet so that it no longer satisfies a design rule, the developer might receive feedback that their code has violated a rule. Developers might then be given a choice: fix their code to match the rule or update the rule itself. Each of these actions might be automated in whole or in part through a code transformation system. Updating the rule might initiate a refactoring, where the updated rule is used to transform every instance to which it applies.

ACKNOWLEDGEMENTS

We thank our study participants for their time. This work was supported in part by the National Science Foundation under grant NSF CCF-1703734.

REFERENCES

[1] 2004. CheckStyle. http://checkstyle.sourceforge.net.
[2] 2015. Browse-By-Query. http://browsebyquery.sourceforge.net/.
[3] Ion Androutsopoulos, Graeme D. Ritchie, and Peter Thanisch. 1995. Natural language interfaces to databases – an introduction. *Natural language engineering* 1, 1 (1995), 29–81.
[4] Carliss Young Baldwin and Kim B Clark. 2000. *Design rules: The power of modularity*. Vol. 1. MIT Press.
[5] Brian Bartman, Christian D. Newman, Michael L. Collard, and Jonathan I. Maletic. 2017. srcQL: A syntax-aware query language for source code. In *International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 467–471.

[6] Maria Christakis and Christian Bird. 2016. What developers want and need from program analysis: An empirical study. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 332–343.

[7] Tom Copeland. 2005. *PMD Applied*. Centennial Books.

[8] Kenneth Cox, Rebecca E. Grinter, Stacie L. Hibino, Lalita Jategaonkar Jagadeesan, and David Mantilla. 2001. A multi-modal natural language interface to an information visualization environment. *International Journal of Speech Technology* 4, 3-4 (2001), 297–314.

[9] Danica Damljanovic, Milan Agatonovic, and Hamish Cunningham. 2010. Natural Language Interfaces to Ontologies: Combining Syntactic Analysis and Ontology-based Lookup Through the User Interaction. In *International Conference on The Semantic Web: Research and Applications*. 106–120.

[10] Oege de Moor, Damien Sereni, Mathieu Verbaere, Elnar Hajiyev, Pavel Avgustinov, Torbjörn Ekman, Neil Ongkingco, and Julian Tibble. 2008. *.QL: Object-Oriented Queries Made Easy*. 78–133.

[11] Coen De Roover, Carlos Noguera, Andy Kellens, and Vivane Jonckers. 2011. The SOUL Tool Suite for Querying Programs in Symbiosis with Eclipse. In *International Conference on Principles and Practice of Programming in Java (PPPJ)*. 71–80.

[12] Tong Gao, Mira Dontcheva, Eytan Adar, Zhicheng Liu, and Karrie G. Karahalios. 2015. Datatone: Managing ambiguity in natural language interfaces for data visualization. In *ACM Symposium on User Interface Software and Technology (UIST)*. 489–500.

[13] Sumit Gulwani and Mark Marron. 2014. Nlyze: Interactive programming by natural language for spreadsheet data analysis and manipulation. In *ACM SIGMOD International Conference on Management of Data*. 803–814.

[14] Andrew Head, Caitlin Sadowski, Emerson Murphy-Hill, and Andrea Knight. 2018. When Not to Comment: Questions and Tradeoffs with API Documentation for C++ Projects. 2. Conference. In *International Conference on Software Engineering (ICSE)*. 643–653.

[15] David Hovemeyer and William Pugh. 2004. Finding bugs is easy. *ACM SIGPLAN Notices* 39, 12 (2004), 92–106.

[16] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why Don't Software Developers Use Static Analysis Tools to Find Bugs?. In *International Conference on Software Engineering (ICSE)*. 672–681.

[17] Paul Klint. 2003. How understanding and restructuring differ from compiling-a rewriting perspective. In *International Workshop on Program Comprehension*. 2–11.

[18] Jens Knodel and Daniel Popescu. 2007. A comparison of static architecture compliance checking approaches. In *Working IEEE/IFIP Conference on Software Architecture*. 12–12.

[19] Thomas D. LaToza and Brad A. Myers. 2010. Hard-to-answer Questions About Code. In *Evaluation and Usability of Programming Languages and Tools (PLATEAU)* (Reno, Nevada). Article 8, 6 pages.

[20] Thomas D. LaToza, Gina Venolia, and Robert DeLine. 2006. Maintaining mental models: a study of developer work habits. In *International Conference on Software Engineering (ICSE)*. 492–501.

[21] Timothy C. Lethbridge, Janice Singer, and Andrew Forward. 2003. How software engineers use documentation: The state of the practice. *IEEE Software* 6 (2003), 35–39.

[22] Yunyao Li, Huahai Yang, and HV Jagadish. 2005. NaLIX: an interactive natural language interface for querying XML. In *ACM SIGMOD International Conference on Management of Data*. 900–902.

[23] Sahar Mehrpour, Thomas D. LaToza, and Rahul K. Kindi. 2019. Active Documentation: Helping Developers Follow Design Decisions. In *Visual Languages and Human-Centric Computing (VL/HCC)*. 2–11.

[24] David L. Parnas and Paul C. Clements. 1986. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering* SE-12, 2 (1986), 251–257.

[25] Terence Parr. 2013. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf.

[26] Leonardo Passos, Ricardo Terra, Marco Tulio Valente, Renato Diniz, and Nabor Mendoça. 2010. Static Architecture-Conformance Checking: An Illustrative Overview. *IEEE software* 27, 5 (2010), 82–89.

[27] Ana-Maria Popescu, Alex Armanasu, Oren Etzioni, David Ko, and Alexander Yates. 2004. Modern natural language interfaces to databases: Composing statistical parsing with semantic tractability. In *International Conference on Computational Linguistics*. 141.

[28] Ana-Maria Popescu, Oren Etzioni, and Henry Kautz. 2003. Towards a Theory of Natural Language Interfaces to Databases. In *International Conference on Intelligent User Interfaces (IUI)*. 149–157.

[29] Caitlin Sadowski, Jeffrey van Gogh, Ciera Jaspan, Emma Söderberg, and Collin Winter. 2015. Tricorder: Building a Program Analysis Ecosystem. In *International Conference on Software Engineering (ICSE)*. 598–608.

[30] Neeraj Sangal, Ev Jordan, Vineet Sinha, and Daniel Jackson. 2005. Using Dependency Models to Manage Complex Software Architecture. In *SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)* (San Diego, CA, USA). 167–176.

[31] Vidya Setlur, Sarah E. Battersby, Melanie Tory, Rich Gossweiler, and Angel X. Chang. 2016. Eviza: A natural language interface for visual analysis. In *ACM Symposium on User Interface Software and Technology (UIST)*. 365–377.

[32] Arjun Srinivasan and John Stasko. 2017. Natural Language Interfaces for Data Analysis with Visualization: Considering What Has and Could Be Asked. In *Eurographics/IEEE VGTC Conference on Visualization: Short Papers*. 55–59.

[33] Arjun Srinivasan and John Stasko. 2018. Orko: Facilitating multimodal interaction for visual exploration and analysis of networks. *IEEE Transactions on Visualization and Computer Graphics* 24, 1 (2018), 511–521.

[34] Yiwen Sun, Jason Leigh, Andrew Johnson, and Sangyoon Lee. 2010. Articulate: A semi-automated model for translating natural language queries into meaningful visualizations. In *International Symposium on Smart Graphics*. Springer, 184–195.

[35] Yuriy Tymchuk, Mohammad Ghafari, and Oscar Nierstrasz. 2018. JIT Feedback: What Experienced Developers Like About Static Analysis. In *Conference on Program Comprehension*. 64–73.