Programming Tools for working with Design Decisions in Code

Sahar Mehrpour, Thomas D. LaToza

PLATEAU2021

Design Decisions in Code

- **Design decisions** are choices developers make between alternatives.
  Design rules are constraints on code imposed by design decisions.

- Design decisions define how functional or non-functional **requirements** are satisfied.

- Developers need to **understand** them to write correct and maintainable code.

- Traditionally, developers write design decisions in **documentation**.

- However, documentation is often **outdated** and untrustworthy, and developers reverse engineer design decisions from code.

Design Decisions in Code

- We propose a new vision for documentation: **Active Documentation**
- Documentation are viewed as **specifications** that can be **checked** against code.

✓ **Understandable** by developers

✓ **Editable** as the code or design changes

✓ Help in **reasoning about** design decisions

✓ Provide positive **examples** as well as **violations** in code

Overview

Background

Developer Goals when working with design decisions and **Existing Tool Support** for Helping Developers achieve their Goals

Design Decisions in Code Review

Potential of Tools in Detecting Violations of Design Decisions

Our Proposed Approach and Implemented Tools

Making design decisions **checkable** and documentation **“active"**

Writing Checkable Design Decisions

Suggesting Design Decisions from Code

Overview

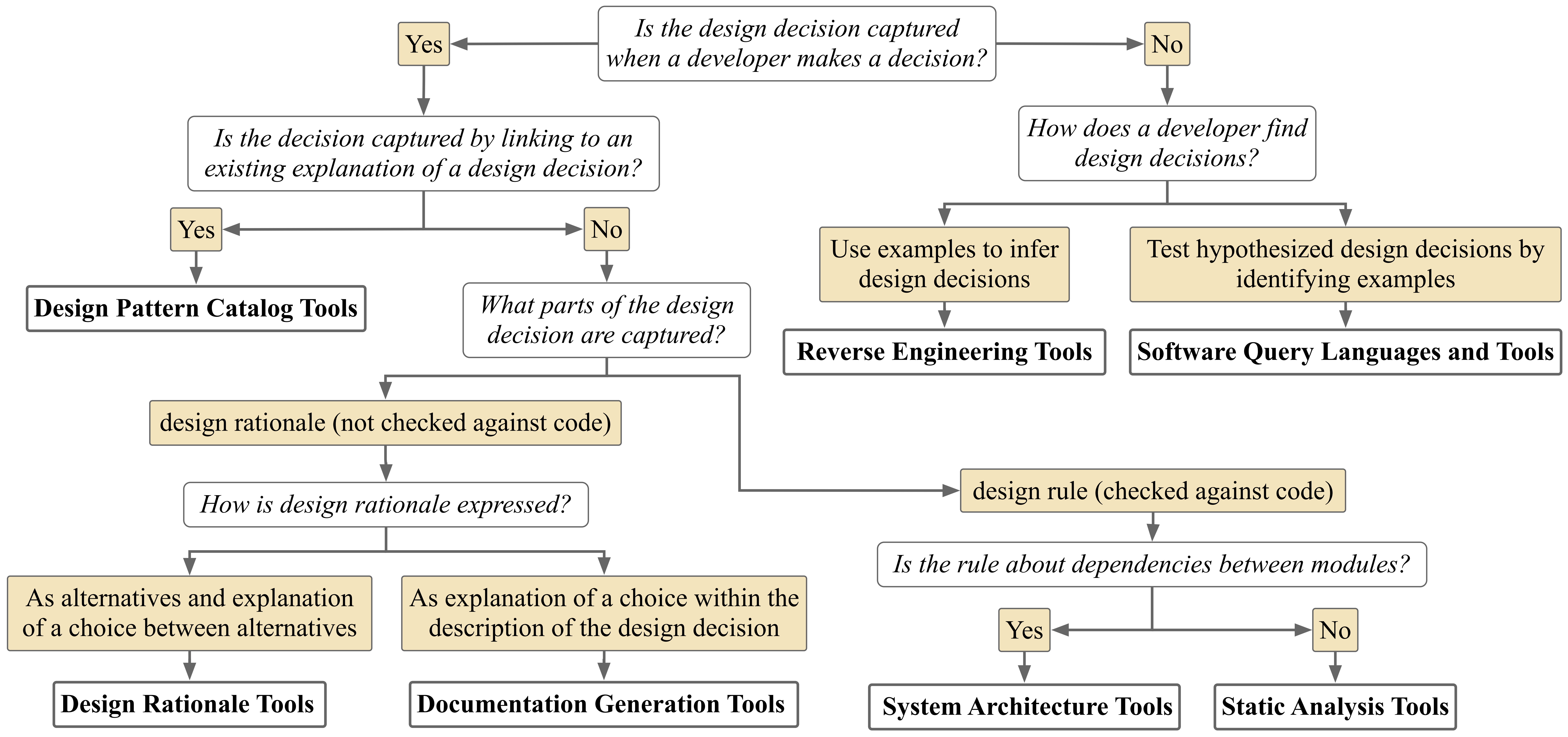
Background

Developer Goals
when working with
design decisions
and
Existing Tool Support for
Helping Developers
achieve their Goals

Background: Developer Goals when Working with Design Decisions

	Goal	Example
Goal 1	**Identify potential alternatives**	*How should functionality be decomposed into classes to achieve extensibility and maintainability?*
Goal 2	**Select an alternative as a design decision**	*Is the best alternative for this situation the Command Pattern or Publish/Subscribe?*
Goal 3	**Document the chosen alternative**	*Communicate the design decision of selecting the Command pattern to future developers through documentation.*
Goal 4	**Check hypothesized design decisions against code**	*After reading the code, a developer hypothesizes that the Command pattern is being used and seeks additional evidence to test this hypothesis.*
Goal 5	**Find and follow relevant design decisions**	*While creating a new class to implement a new user action, a developer tries to determine how it should be connected to existing functionality that captures user toolbar actions.*
Goal 6	**Determine why an alternative was selected**	*After seeing that communication is mediated through Command patterns, the developer tries to determine why it was selected instead of a Publish/Subscribe approach.*

Background: Existing Tool Support for Working with Design Decisions



Is the design decision captured when a developer makes a decision?
Yes
No
Is the decision captured by linking to an existing explanation of a design decision?
Yes
No
Design Pattern Catalog Tools
What parts of the design decision are captured?
design rationale (not checked against code)
How is design rationale expressed?
As alternatives and explanation of a choice between alternatives
As explanation of a choice within the description of the design decision
Design Rationale Tools
Documentation Generation Tools
design rule (checked against code)
Is the rule about dependencies between modules?
Yes
No
System Architecture Tools
Static Analysis Tools
How does a developer find design decisions?
Use examples to infer design decisions
Test hypothesized design decisions by identifying examples
Reverse Engineering Tools
Software Query Languages and Tools

Background: Existing Tool Support for helping Developers achieve their goals

	Identify Alternatives	**Select an Alternative**	**Document the Decision**	**Test Hypothesized Decisions**	**Find and Follow Decisions**	**Reason about Decisions**
Documentation Generation Tools	Partial	Partial	-	Partial	Partial	Partial
Static Analysis Tools	-	-	Partial	Partial	**Full**	Partial
Design Rationale Tools	Partial	**Full**	**Full**	Partial	Partial	**Full**
Design Pattern Catalogs	**Full**	**Full**	**Full**	Partial	Partial	**Full**
System Architecture Tools	-	-	Partial	Partial	**Full**	Partial
Reverse Engineering Tools	Partial	-	-	-	Partial	-
Software Query Languages and Tools	Partial	-	-	**Full**	Partial	-

Overview

Background

Developer Goals when working with design decisions and **Existing Tool Support** for Helping Developers achieve their Goals

Design Decisions in Code Review

Potential of Tools in Detecting Violations of Design Decisions

Our Proposed Approach and Implemented Tools

Making design decisions **checkable** and documentation **“active"**

Writing Checkable Design Decisions

Suggesting Design Decisions from Code

Overview

Background

Developer Goals when working with design decisions and Existing Tool Support for Helping Developers achieve their Goals

Design Decisions in Code Review

Potential of Tools in Detecting Violations of Design Decisions

Design Decisions in Code Review

- We studied the **types of decisions** developers **fail to follow** by analyzing code review defects.

- Prior studies analyzed how individual tools (e.g., FindBugs) can detect code review defects.

    - 35% to 95% of defects reported issue trackers could be found by FindBugs, JLint, and PMD. [Thung et al., ASE 2012]

    - 4.5% of defects in Defects4J could be detected by Error Prone, Infer, SpotBugs. [Habib and Pradel, ASE 2018]

    - 16% of issues in review comments can be detected by PMD. [Singh et al., VL/HCC 2018]

- Not much information on the potential for creating more effective tools.

Design Decisions in Code Review: Process

- We study the **potential** of tools in checking different types of design decisions by analyzing code review comments *qualitatively*.
- We systematically **collected and analyzed** more than 1300 review comments.
- We used all available information to **formulate** each defect as a violation of a design rule.
- We **mapped** the design rules to **existing types of program analysis tools** by comparing the underlying techniques of tools and properties of design rules.
- We found a **taxonomy of program analysis tools** focusing on the properties of rules they check.

Design Decisions in Code Review: Taxonomy of Program Analysis Tools

	Representation of Code			Origin of Defects			Consequences of defects	
Categories	**AST**	**Code Execution**	**Strings**	**Language**	**Specifications**	**Best Practices**	**Code Quality**	**Behavioral**
Style Checkers	✓		✓			✓	✓	
Continuous Integration Tools	✓	✓	✓		✓			✓
Data Flow Analyzers	✓	✓			✓		✓	✓
Architectural Style Checkers	✓	✓			✓		✓	
Test Suite Quality Checkers	✓	✓	✓		✓		indirect	indirect
Dead Code Detectors	✓	✓				✓	✓	
Code Clone Detectors	✓	✓				✓	✓	
Compilers	✓			✓				✓
String Compilers			✓		✓	✓	✓	✓
Code Smell Detectors	✓	✓				✓	✓	
Memory Leak Detectors		✓		✓				✓
AST Pattern Checkers	✓				✓		✓	✓

Design Decisions in Code Review: Results

- Existing program analysis tools may be able to **detect 76%** of code review defects.

- **Style Checkers** and **AST Pattern Checkers** are most broadly applicable, with each potentially able to detect **more than half defects**.

- Many defects are violations of **project-specific** design rules.

- Defects not detectable by program analysis tools **lack formalism** and require **human judgement**.

Overview

Background

Developer Goals when working with design decisions and **Existing Tool Support** for Helping Developers achieve their Goals

Design Decisions in Code Review

Potential of Tools in Detecting Violations of Design Decisions

Our Proposed Approach and Implemented Tools

Making design decisions **checkable** and documentation **“active"**

Writing Checkable Design Decisions

Suggesting Design Decisions from Code

Overview

Background

Developer Goals when working with design decisions and Existing Tool Support for Helping Developers achieve their Goals

Design Decisions in Code Review

Potential of Tools in Detecting Violations of Design Decisions

Our Proposed Approach and Implemented Tools

Making design decisions **checkable** and documentation **“active"**

Making Documentation Active

To help developers work with design decisions, we propose a new form of documentation:
Active Documentation

Design rules are translated into constraints and ***actively* checked** against code.

Wherever a design rule applies to code, an ***active* link** between the documentation and code is generated.

Developers can ***actively* update** the documentation.

Making Documentation Active

To help developers work with design decisions, we propose a new form of documentation:
Active Documentation

✓ Design rules are translated into constraints and ***actively* checked** against code.

Wherever a design rule applies to code, an ***active* link** between the documentation and code is generated.

Developers can ***actively* update** the documentation.

Making Documentation Active

To help developers work with design decisions, we propose a new form of documentation:
Active Documentation

- ✓ Design rules are translated into constraints and ***actively* checked** against code.
- ✓ Wherever a design rule applies to code, an ***active* link** between the documentation and code is generated.

Developers can ***actively* update** the documentation.

Mehrpour, S., LaToza, T. D., Kindi, R. K. Active Documentation: Helping Developers Follow Design Decisions, VL/HCC 2019

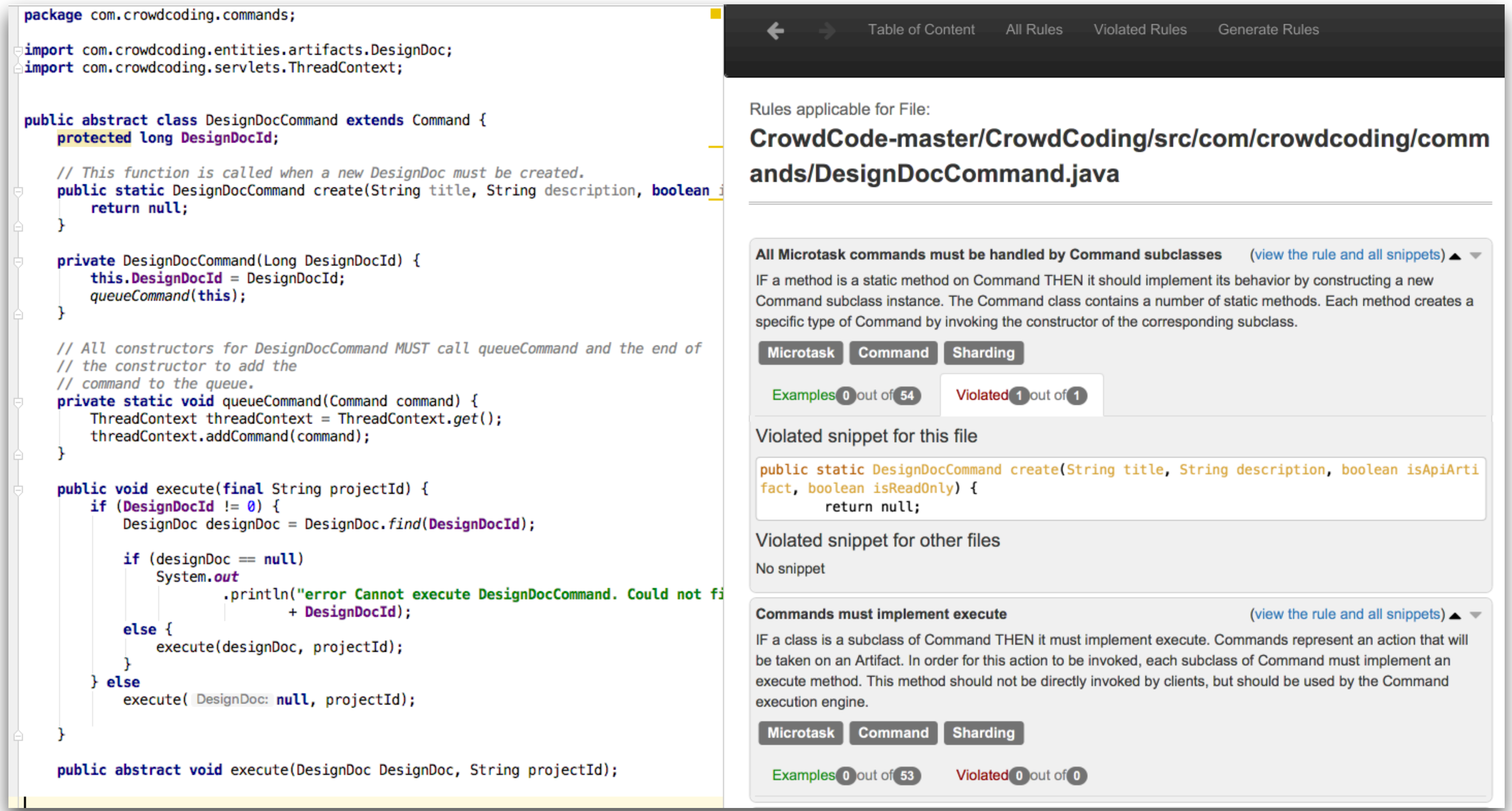
Making Documentation Active

To help developers work with design decisions, we propose a new form of documentation:
Active Documentation

- ✓ Design rules are translated into constraints and ***actively* checked** against code.
- ✓ Wherever a design rule applies to code, an ***active* link** between the documentation and code is generated.
- ✓ Developers can ***actively* update** the documentation.

Mehrpour, S., LaToza, T. D., Kindi, R. K. Active Documentation: Helping Developers Follow Design Decisions, VL/HCC 2019

Making Documentation Active: ActiveDocumentation



```java
package com.crowdcoding.commands;

import com.crowdcoding.entities.artifacts.DesignDoc;
import com.crowdcoding.servlets.ThreadContext;


public abstract class DesignDocCommand extends Command {
    protected long DesignDocId;

    // This function is called when a new DesignDoc must be created.
    public static DesignDocCommand create(String title, String description, boolean i
        return null;
    }

    private DesignDocCommand(Long DesignDocId) {
        this.DesignDocId = DesignDocId;
        queueCommand(this);
    }

    // All constructors for DesignDocCommand MUST call queueCommand and the end of
    // the constructor to add the
    // command to the queue.
    private static void queueCommand(Command command) {
        ThreadContext threadContext = ThreadContext.get();
        threadContext.addCommand(command);
    }

    public void execute(final String projectId) {
        if (DesignDocId != 0) {
            DesignDoc designDoc = DesignDoc.find(DesignDocId);

            if (designDoc == null)
                System.out
                        .println("error Cannot execute DesignDocCommand. Could not fi
                                + DesignDocId);
            else {
                execute(designDoc, projectId);
            }
        } else
            execute( DesignDoc: null, projectId);

    }

    public abstract void execute(DesignDoc DesignDoc, String projectId);
```

Rules applicable for File:

CrowdCode-master/CrowdCoding/src/com/crowdcoding/commands/DesignDocCommand.java

All Microtask commands must be handled by Command subclasses (view the rule and all snippets)

IF a method is a static method on Command THEN it should implement its behavior by constructing a new Command subclass instance. The Command class contains a number of static methods. Each method creates a specific type of Command by invoking the constructor of the corresponding subclass.

Microtask Command Sharding

Examples 0 out of 54 Violated 1 out of 1

Violated snippet for this file

```java
public static DesignDocCommand create(String title, String description, boolean isApiArti
fact, boolean isReadOnly) {
        return null;
```

Violated snippet for other files

No snippet

Commands must implement execute (view the rule and all snippets)

IF a class is a subclass of Command THEN it must implement execute. Commands represent an action that will be taken on an Artifact. In order for this action to be invoked, each subclass of Command must implement an execute method. This method should not be directly invoked by clients, but should be used by the Command execution engine.

Microtask Command Sharding

Examples 0 out of 53 Violated 0 out of 0

Design Decisions Related to a File

Tags

Examples

Violations

Making Documentation Active: ActiveDocumentation Evaluation

- We conducted a user study with 21 participants.
- We asked them to add a new feature in an unfamiliar codebase.
- We found ActiveDocumentation helped participants work **quickly** and **successfully** with design decisions.

Overview

Background

Developer Goals when working with design decisions and **Existing Tool Support** for Helping Developers achieve their Goals

Design Decisions in Code Review

Potential of Tools in Detecting Violations of Design Decisions

Our Proposed Approach and Implemented Tools

Making design decisions **checkable** and documentation **“active"**

Writing Checkable Design Decisions

Suggesting Design Decisions from Code

Overview

Background

Developer Goals when working with design decisions and Existing Tool Support for Helping Developers achieve their Goals

Design Decisions in Code Review

Potential of Tools in Detecting Violations of Design Decisions

Our Proposed Approach and Implemented Tools

Making design decisions **checkable** and documentation **"active"**

Writing Checkable Design Decisions

Helping Developers Write Checkable Design Decisions

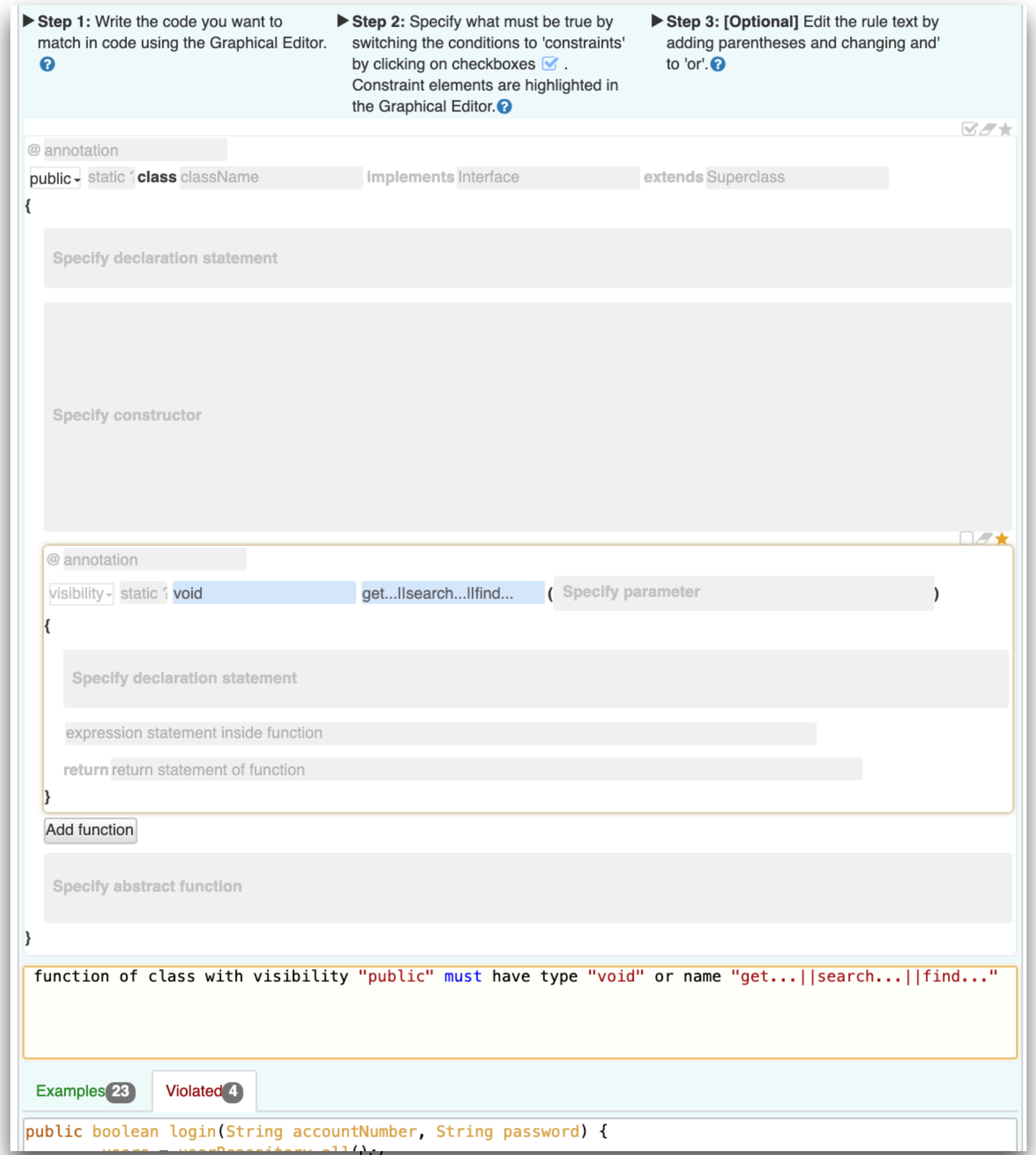
- To make documentation checkable, developers should be able to write checkable design decisions.
- Existing **extensible** tools like PMD or Error Prone enable developers to write **custom** rules.
- But they require specialized **knowledge** of program analysis or complex query notations.
- We introduce two complimentary techniques to write checkable design decisions:

✓ **Snippet-Based Authoring**: code-based templates, can be ambiguous

✓ **Semi-Natural Language Authoring**: expressive, natural

Mehrpour, S., LaToza, T. D., Sarvari, H., RulePad: Interactive Authoring of Checkable Design Rules, ESEC/FSE 2020

Helping Developers Write Checkable Design Decisions: RulePad



▶ **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'.

```
@ annotation
public  static  class className  implements Interface  extends Superclass
{
    Specify declaration statement

    Specify constructor

    @ annotation
    visibility  static  void  get...||search...||find...  ( Specify parameter )
    {
        Specify declaration statement
        expression statement inside function
        return return statement of function
    }
    Add function

    Specify abstract function
}
```

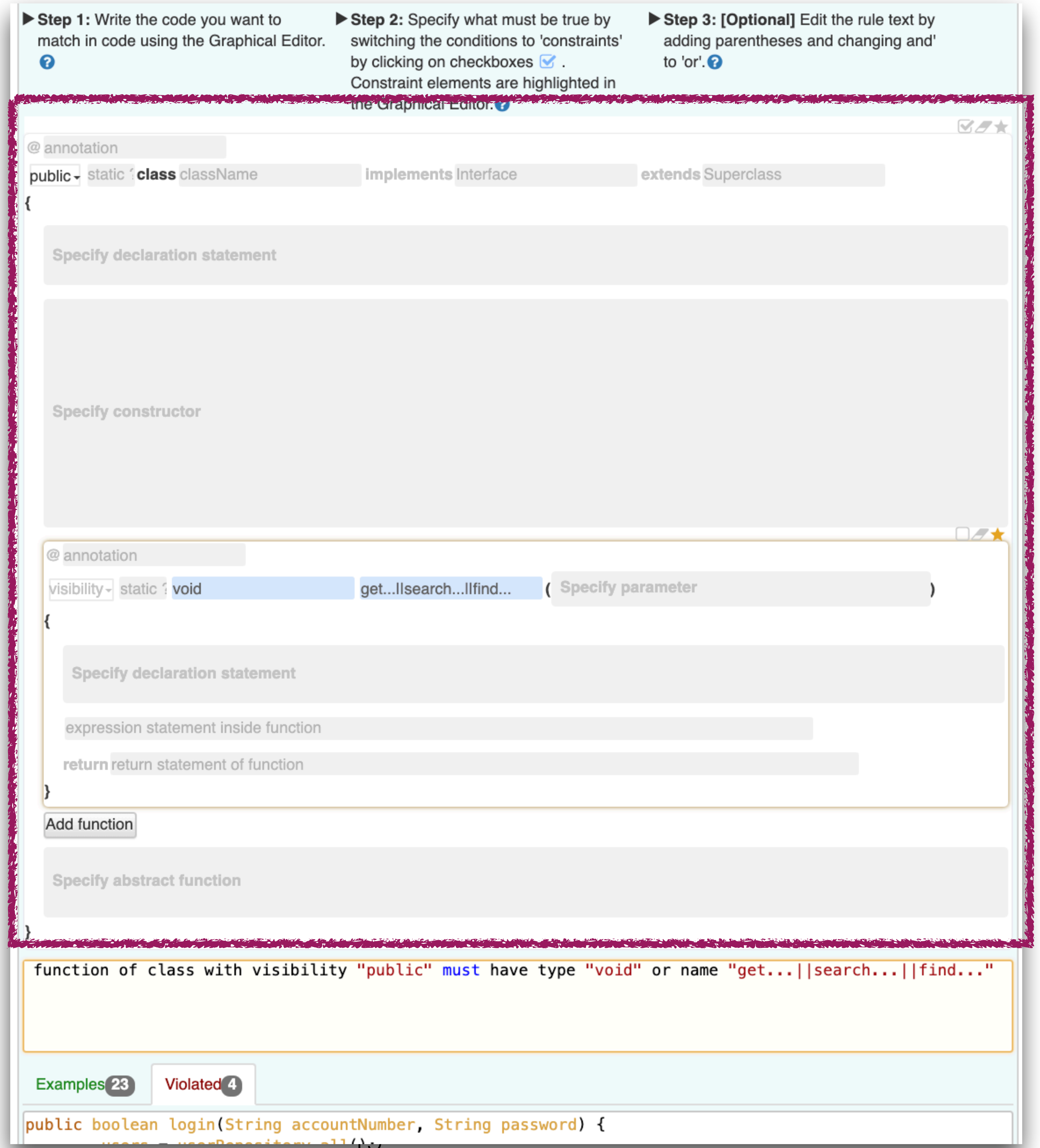
```
function of class with visibility "public" must have type "void" or name "get...||search...||find..."
```

Examples 23   Violated 4

```
public boolean login(String accountNumber, String password) {
```

Helping Developers Write Checkable Design Decisions: RulePad

The Graphical Editor



▶ **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'.

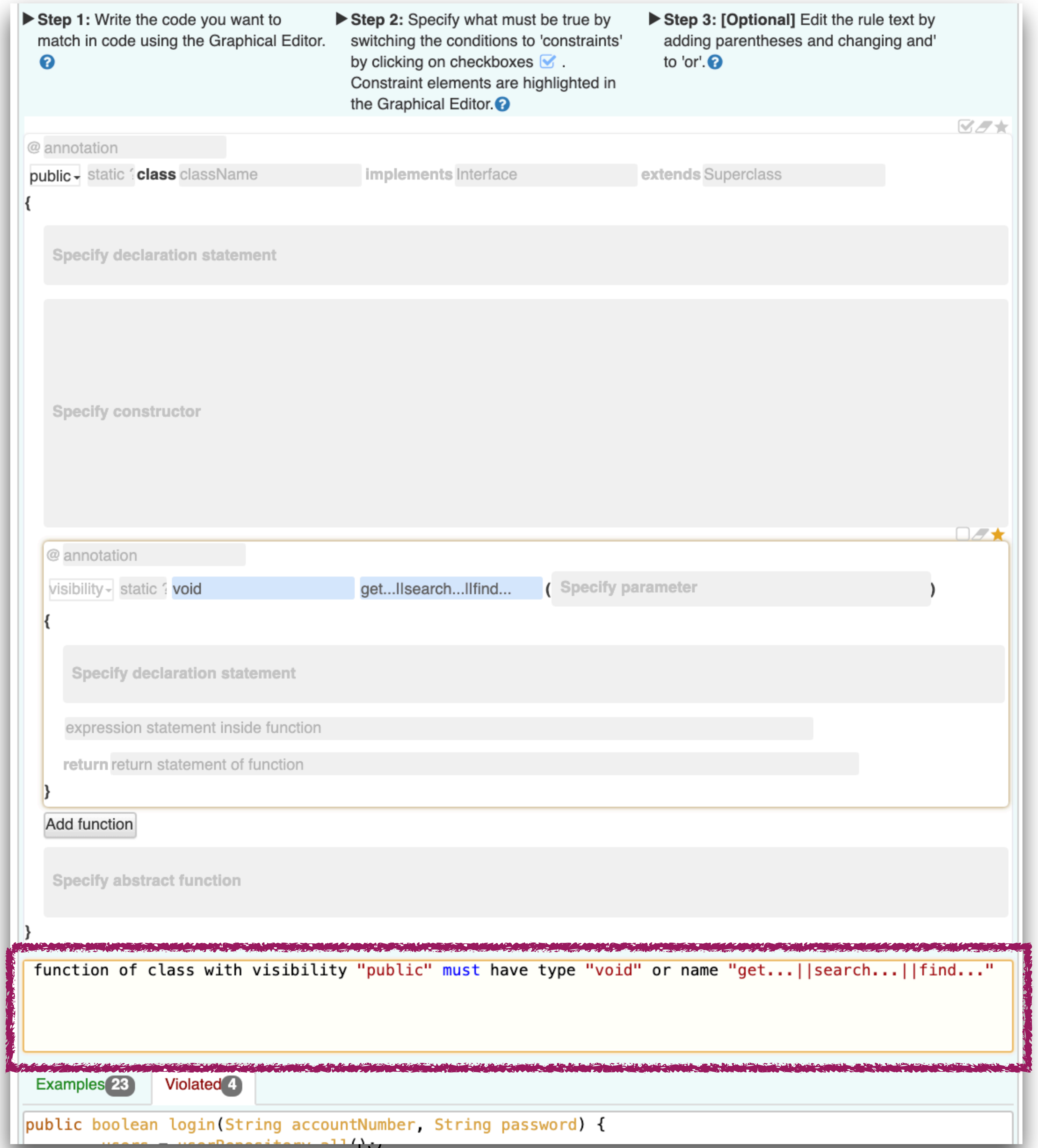
```
@ annotation
public static class className Implements Interface extends Superclass
{
    Specify declaration statement

    Specify constructor

    @ annotation
    visibility static void get...||search...||find... ( Specify parameter )
    {
        Specify declaration statement
        expression statement inside function
        return return statement of function
    }
Add function

    Specify abstract function
}
```

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

Examples 23 | Violated 4

```
public boolean login(String accountNumber, String password) {
```

Helping Developers Write Checkable Design Decisions: RulePad



▶ **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'.

```
@ annotation
public static class className  implements Interface  extends Superclass
{
    Specify declaration statement

    Specify constructor

    @ annotation
    visibility static void  get...||search...||find... ( Specify parameter )
    {
        Specify declaration statement
        expression statement inside function
        return return statement of function
    }
    Add function

    Specify abstract function
}
```

The Textual Editor

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

Examples 23   Violated 4

```
public boolean login(String accountNumber, String password) {
```

Helping Developers Write Checkable Design Decisions: RulePad Evaluation

- We conducted a **user study** with 14 participants, comparing authoring checkable design decisions in **RulePad** and **PMD**.

- We asked participants to **write a few design decisions** using RulePad (experimental group) or PMD (control group).

- Participants using RulePad were **more successful** and able to write **13 times more** query elements.

- Participants also reported they are **more willing** to use RulePad in their everyday work.

Overview

Background

Developer Goals when working with design decisions and **Existing Tool Support** for Helping Developers achieve their Goals

Design Decisions in Code Review

Potential of Tools in Detecting Violations of Design Decisions

Our Proposed Approach and Implemented Tools

Making design decisions **checkable** and documentation **“active"**

Writing Checkable Design Decisions

Suggesting Design Decisions from Code

Overview

Background

Developer Goals when working with design decisions and Existing Tool Support for Helping Developers achieve their Goals

Design Decisions in Code Review

Potential of Tools in Detecting Violations of Design Decisions

Our Proposed Approach and Implemented Tools

Making design decisions **checkable** and documentation **“active"**

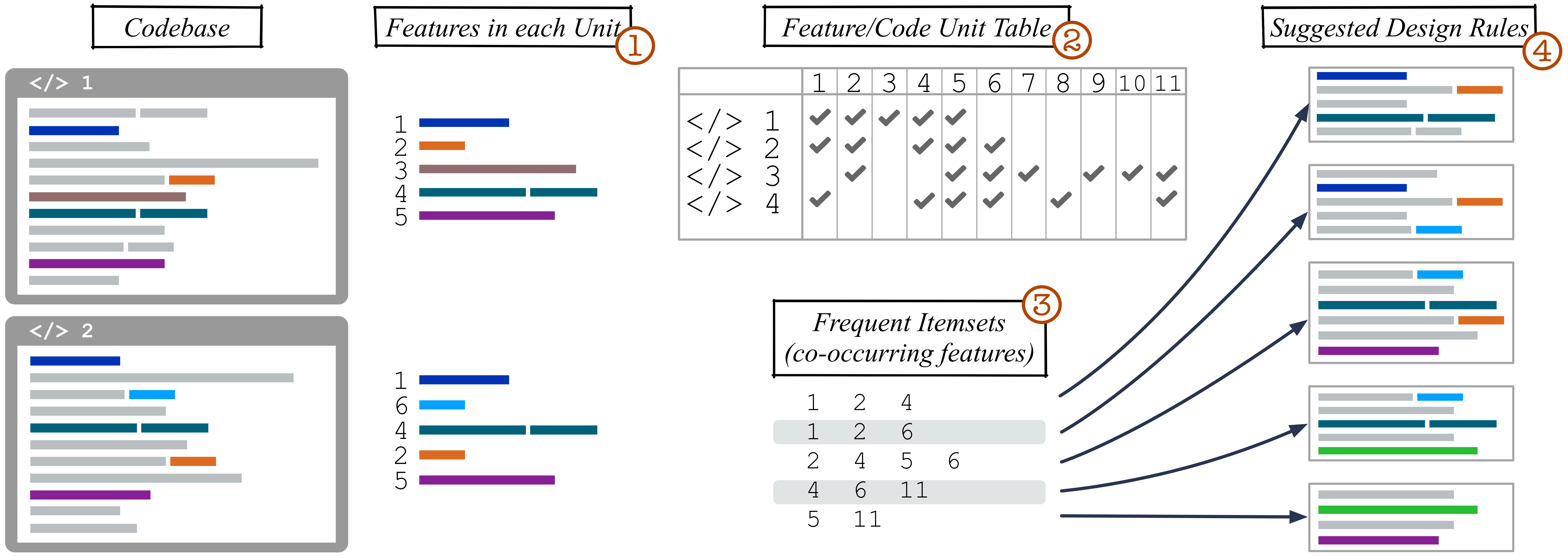
Writing Checkable Design Decisions

Suggesting Design Decisions from Code

Suggesting Design Decisions from Code

- When design decisions are not written, developers need to find them in code by reverse engineering.
- Developers either infer design decisions from code examples, or test hypothesized decisions against code.
- We envision a tool to help developers by suggesting design decisions relevant to the code.

Suggesting Design Decisions from Code: Approach



	1	2	3	4	5	6	7	8	9	10	11
</> 1	✓	✓	✓	✓	✓						
</> 2	✓	✓		✓	✓	✓					
</> 3		✓			✓	✓	✓		✓	✓	✓
</> 4	✓			✓	✓	✓		✓			✓

```
1   2   4
1   2   6
2   4   5   6
4   6   11
5   11
```

Suggesting Design Decisions from Code: Challenges and Evaluation

- What **features** should the tool select?
  - **Standard** features observed from examples in other codebases
  - **Extensibility**, allowing developers to add custom features.

- Suggested design decisions should be **important** to the developer.
  - Evaluate the tool by **comparing** the suggested decisions by the tool and a previously found corpus of design decisions.
  - **User study** to evaluate the techniques

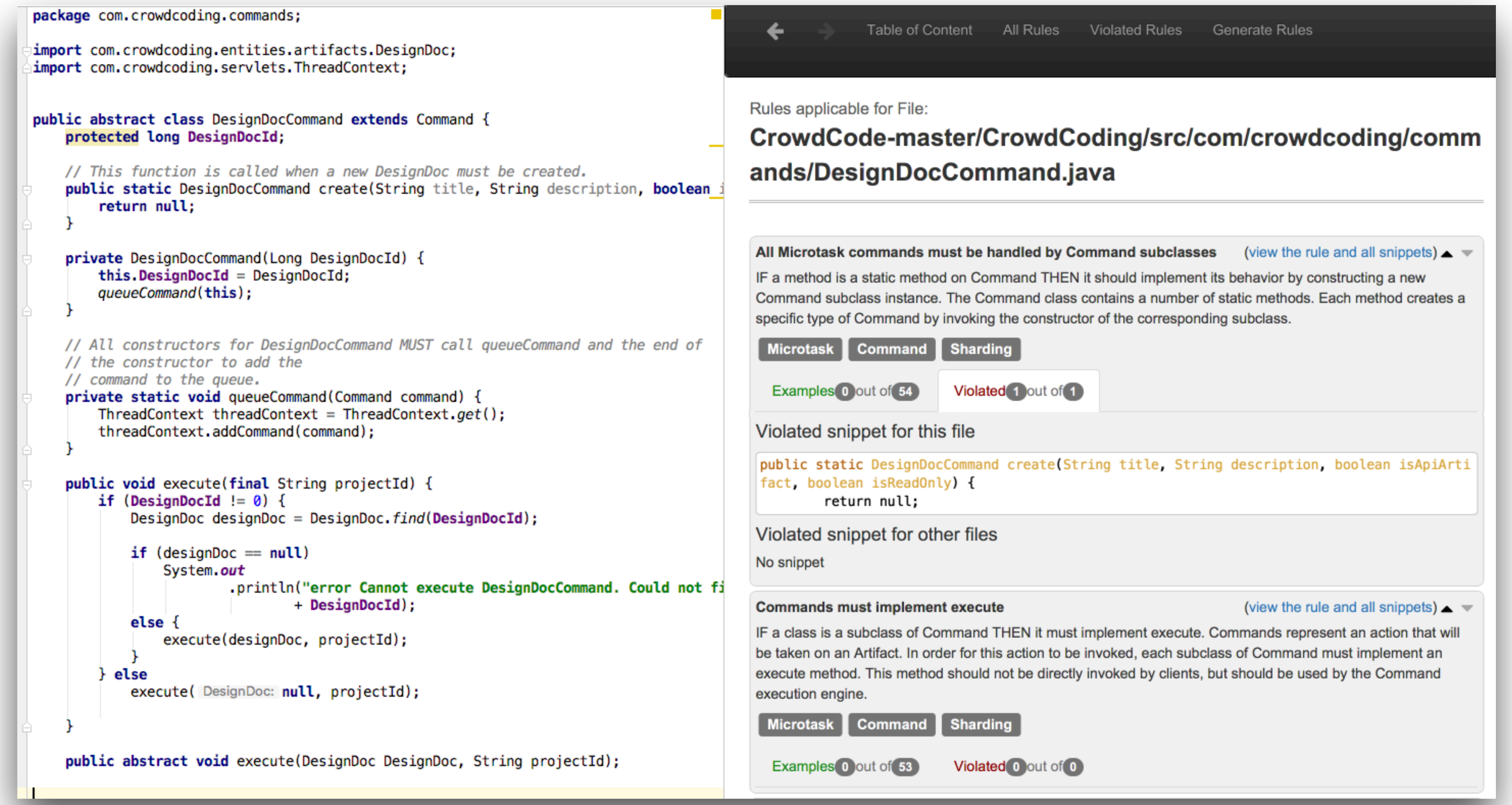
Programming Tools for working with Design Decisions in Code

Sahar Mehrpour, Thomas D. LaToza

Making design decisions **checkable** and documentation **"active"**

Writing Checkable Design Decisions

Suggesting Design Decisions from Code



```
package com.crowdcoding.commands;

import com.crowdcoding.entities.artifacts.DesignDoc;
import com.crowdcoding.servlets.ThreadContext;


public abstract class DesignDocCommand extends Command {
    protected long DesignDocId;

    // This function is called when a new DesignDoc must be created.
    public static DesignDocCommand create(String title, String description, boolean i
        return null;
    }

    private DesignDocCommand(Long DesignDocId) {
        this.DesignDocId = DesignDocId;
        queueCommand(this);
    }

    // All constructors for DesignDocCommand MUST call queueCommand and the end of
    // the constructor to add the
    // command to the queue.
    private static void queueCommand(Command command) {
        ThreadContext threadContext = ThreadContext.get();
        threadContext.addCommand(command);
    }

    public void execute(final String projectId) {
        if (DesignDocId != 0) {
            DesignDoc designDoc = DesignDoc.find(DesignDocId);

            if (designDoc == null)
                System.out
                        .println("error Cannot execute DesignDocCommand. Could not fi
                                + DesignDocId);
            else {
                execute(designDoc, projectId);
            }
        } else
            execute( DesignDoc: null, projectId);

    }

    public abstract void execute(DesignDoc DesignDoc, String projectId);
```

Rules applicable for File:

CrowdCode-master/CrowdCoding/src/com/crowdcoding/commands/DesignDocCommand.java

All Microtask commands must be handled by Command subclasses (view the rule and all snippets)

IF a method is a static method on Command THEN it should implement its behavior by constructing a new Command subclass instance. The Command class contains a number of static methods. Each method creates a specific type of Command by invoking the constructor of the corresponding subclass.

Microtask Command Sharding

Examples 0 out of 54 Violated 1 out of 1

Violated snippet for this file

```
public static DesignDocCommand create(String title, String description, boolean isApiArti
fact, boolean isReadOnly) {
        return null;
```

Violated snippet for other files

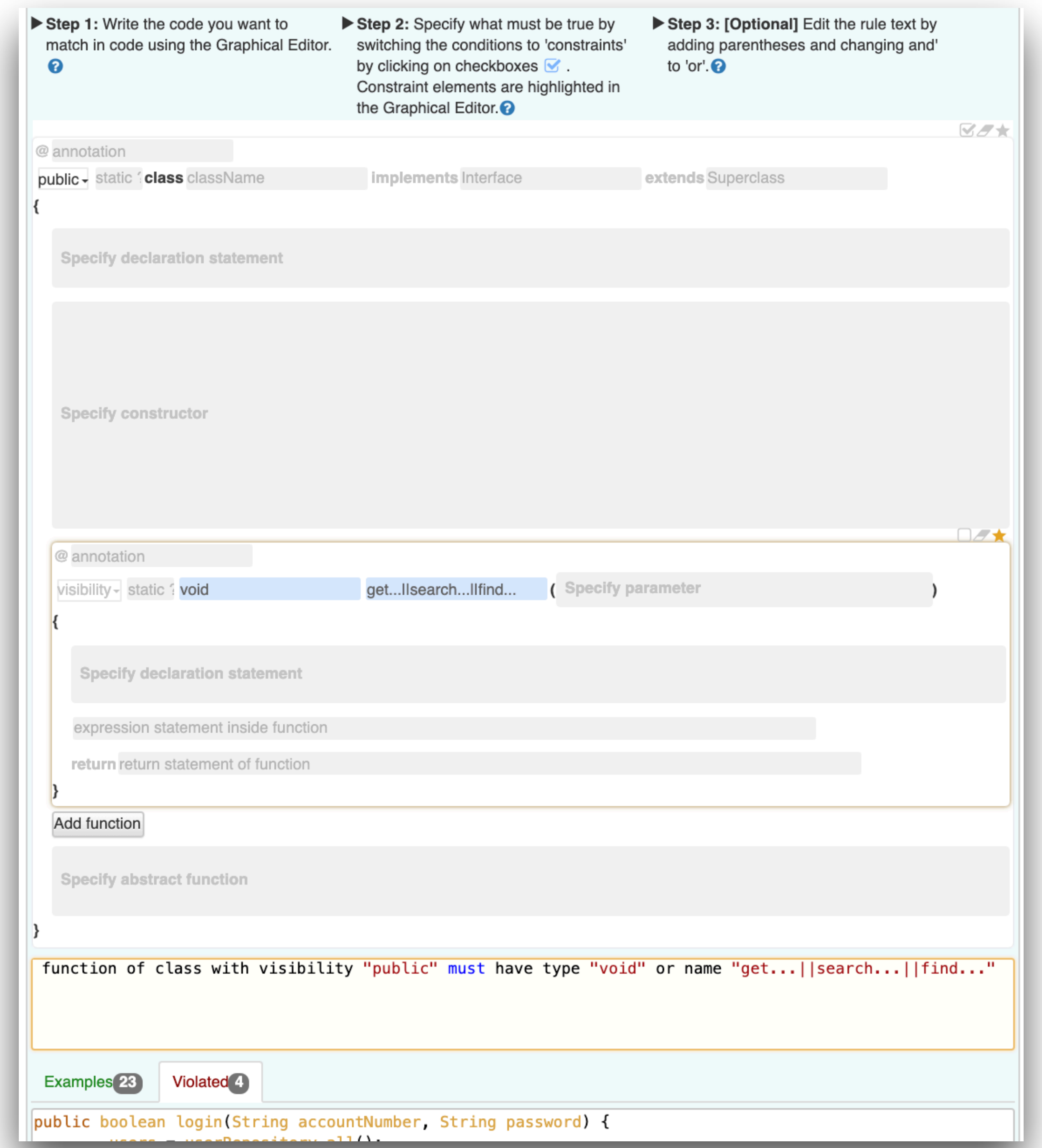
No snippet

Commands must implement execute (view the rule and all snippets)

IF a class is a subclass of Command THEN it must implement execute. Commands represent an action that will be taken on an Artifact. In order for this action to be invoked, each subclass of Command must implement an execute method. This method should not be directly invoked by clients, but should be used by the Command execution engine.

Microtask Command Sharding

Examples 0 out of 53 Violated 0 out of 0



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.

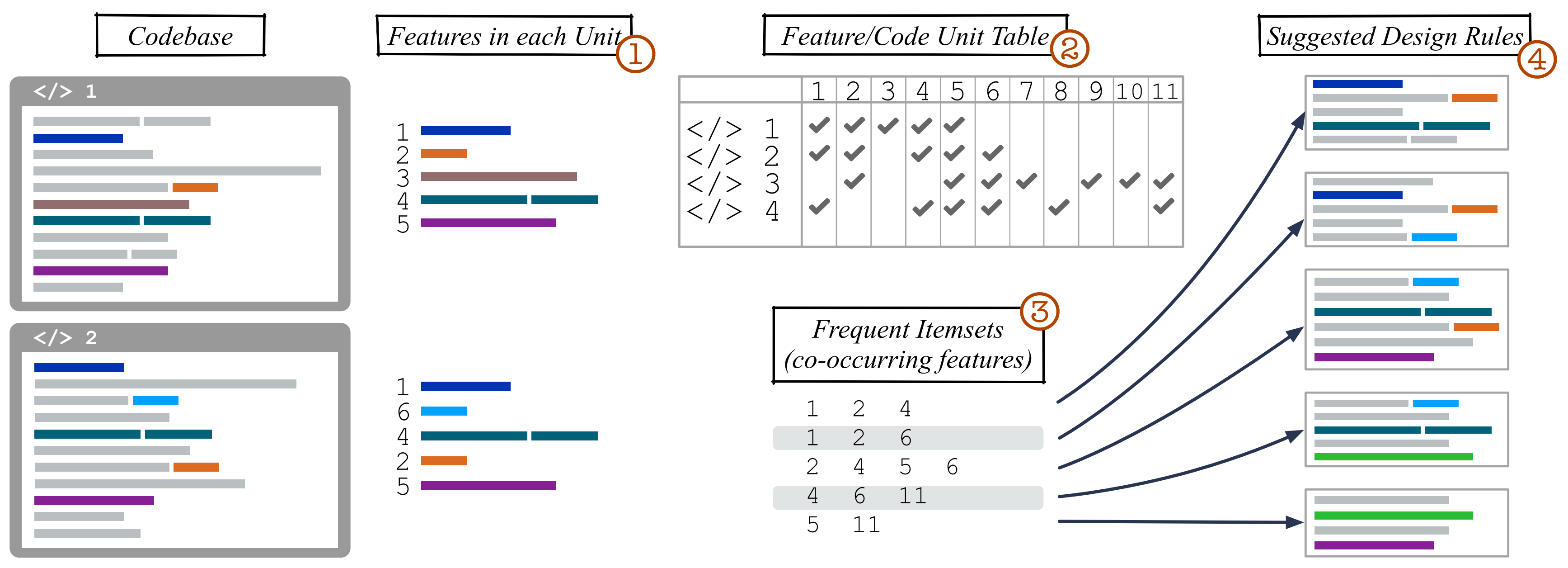
Step 3: [Optional] Edit the rule text by adding parentheses and changing and' to 'or'.

@ annotation
public static class className implements Interface extends Superclass
{
Specify declaration statement
Specify constructor
@ annotation
visibility static void get...||search...||find... (Specify parameter)
{
Specify declaration statement
expression statement inside function
return return statement of function
}
Add function
Specify abstract function
}

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

Examples 23 Violated 4

```
public boolean login(String accountNumber, String password) {
```


Codebase
</> 1
</> 2
Features in each Unit
1
Feature/Code Unit Table
2
Frequent Itemsets (co-occurring features)
3
Suggested Design Rules
4