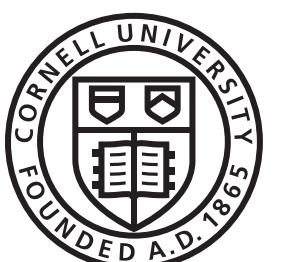


Active Documentation: Helping Developers Follow Design Decisions

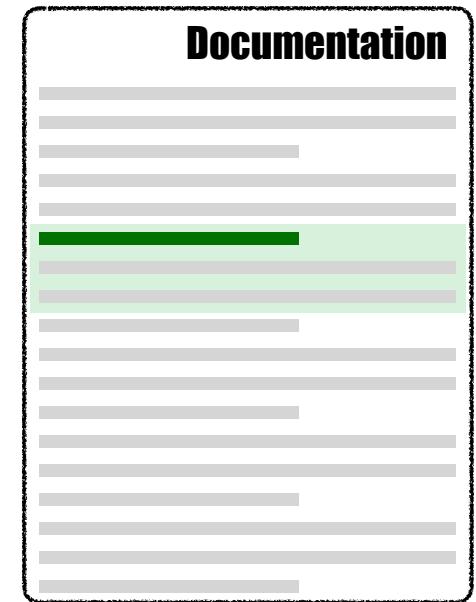
Sahar Mehrpour George Mason University

Thomas D. LaToza George Mason University

Rahul K. Kindi Cornell University



Scenario: Using Documentation Today



All interaction among *Artifact* classes must be done through *Command* classes to apply *sharding*.

Each *Artifact* must have a *Command* class.

- ▶ Alice is a developer in a Company.
- ▶ She is working to implement of a small feature (a new *Artifact B*) in the codebase.
- ▶ Alice starts reading the *documentation* ... But the documentation is **too long**.
- ▶ Alice reads one of the **Design Decisions** describing what alternative was chosen and why.
- ▶ Looking at the description of the design decision, she reads one of the **Design Rules** describing how to implement the design decision.
- ▶ She tries to **connect** the design rule to the code ... But the documentation and the source code are large and hard to connect.

Scenario: Using Documentation Today



checkstyle



- ▶ After some time, Alice finds that she believes to be an **Example** illustrating how to implement an Artifact. Following this example, She tries to re-implement her new class.
- ▶ She writes some code and wants to know if it follows the design rules. But she is not sure that she is following the examples correctly, and that there aren't other rules she missed.
- ▶ She looks at the **rule checkers** the company is using, but they only report defects about her use of Java and do not help with understanding these design decisions.
- ▶ Frustrated, she commits her code and waits for **code reviews** from other developers.

Active Documentation

Our solution: **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.



ACTIVE DOCUMENTATION



IntelliJ IDE plugin

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 isApiArtifact, 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.

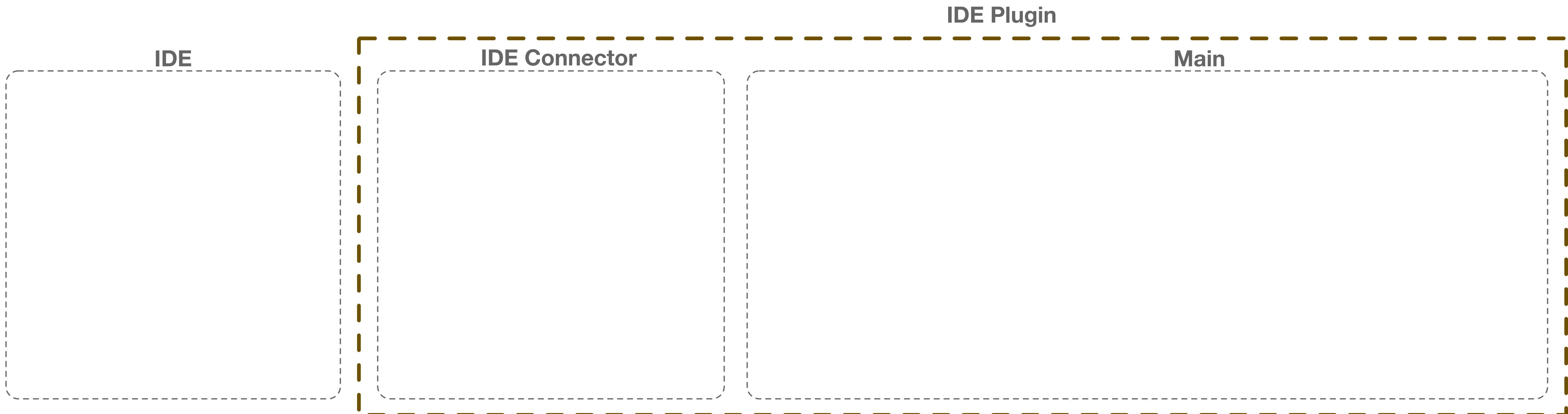
Microtask **Command** **Sharding**

Examples 0 out of 53 Violated 0 out of 0

```
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 isApiArtifact, boolean isReadOnly) {  
        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 find design doc with id " + DesignDocId);  
            else {  
                execute(designDoc, projectId);  
            }  
        } else  
            execute(null, projectId);  
    }  
  
    public abstract void execute(DesignDoc DesignDoc, String projectId);  
}
```

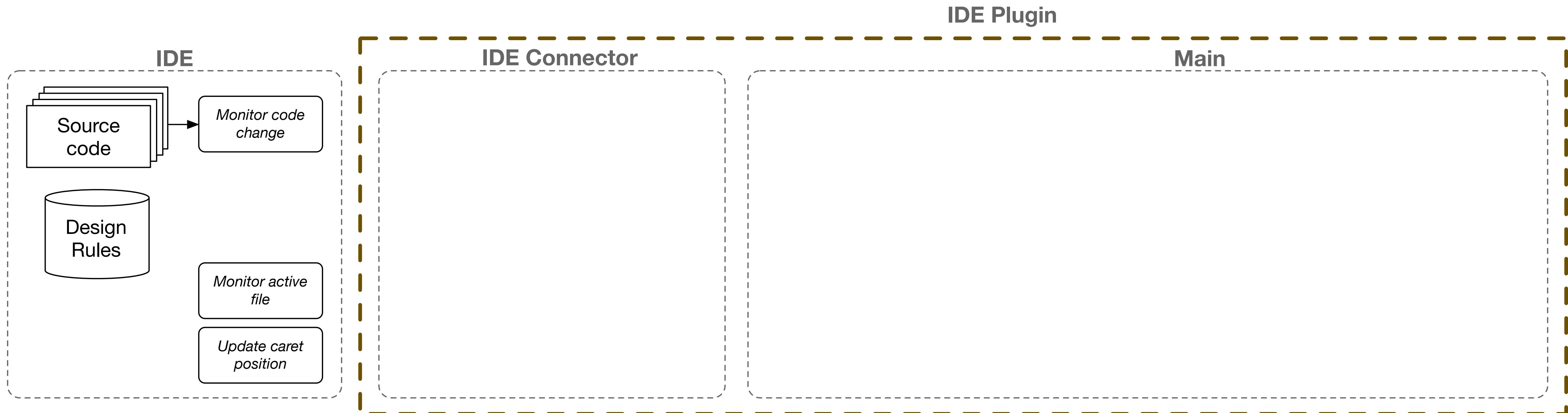
ACTIVE DOCUMENTATION System Architecture

- ▶ Independent from IDEs
- ▶ Two main components: IDE Connector and Main
- ▶ IDE Connector transfers data to/from the IDE



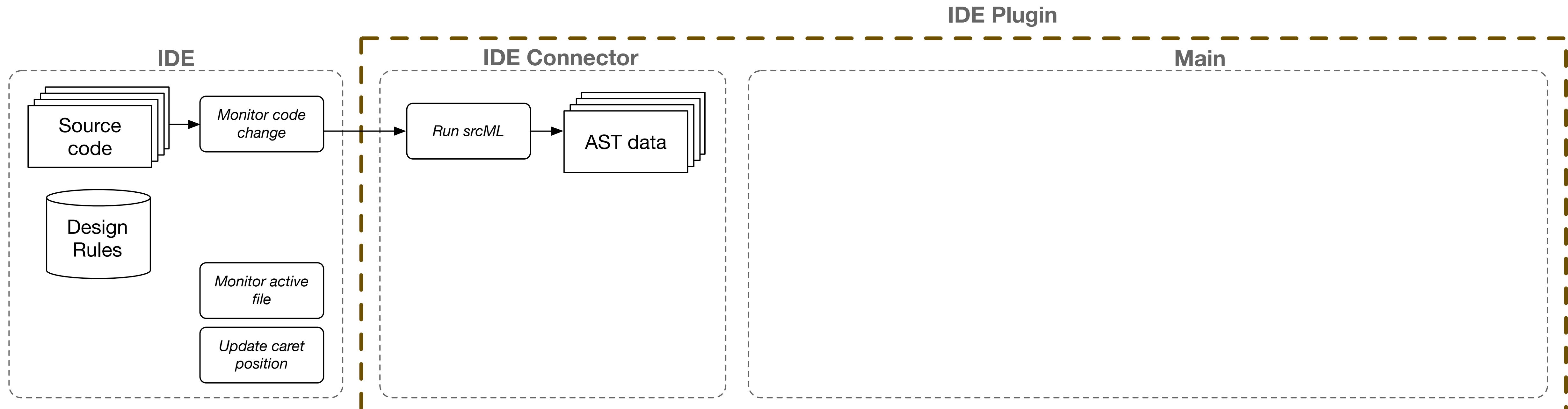
ACTIVE DOCUMENTATION System Architecture

- ▶ IDE is responsible for reporting code change, active file in the editor, and updating the caret position
- ▶ Stored design rules (stored as .json) are accessible in the IDE



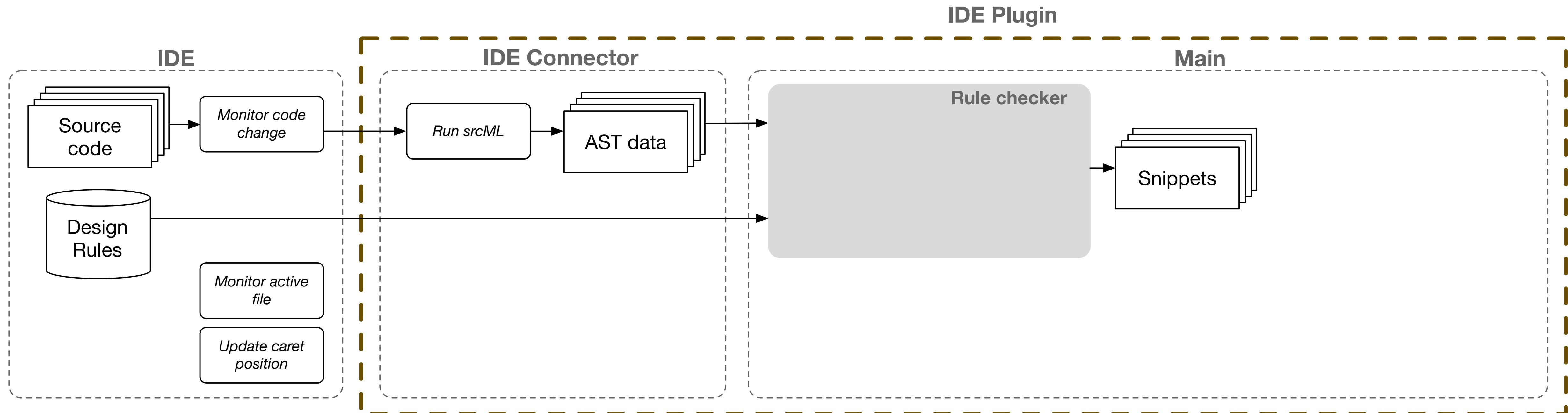
ACTIVE DOCUMENTATION System Architecture

- ▶ IDE connector creates the AST of the source code.
- ▶ XML representation of the ASTs are easier to work with.
- ▶ We used srcML to create the AST.  [Maletic et al. 2002]



ACTIVEDOCUMENTATION System Architecture

- ▶ The rule checker uses the design rules and the AST of code to extract snippets from code.
- ▶ ACTIVEDOCUMENTATION is agnostic to the underlying rule checker.



Rule Checker In ACTIVEDOCUMENTATION

- ▶ Existing rule checkers *only* find **violations** of rules.
- ▶ Developers need to search code to know *how* a rule is followed.
- ▶ ACTIVEDOCUMENTATION shows snippets from code that **satisfy** or **violate** the rule.

WHEN and HOW the rule should apply...

Quantifier WHEN the rule should apply

Constraint HOW the rule should apply

- ▶ Each Artifact must have a Command class.

IF a class is an artifact → **Quantifier**
THEN it should have a Command class → **Constraint**

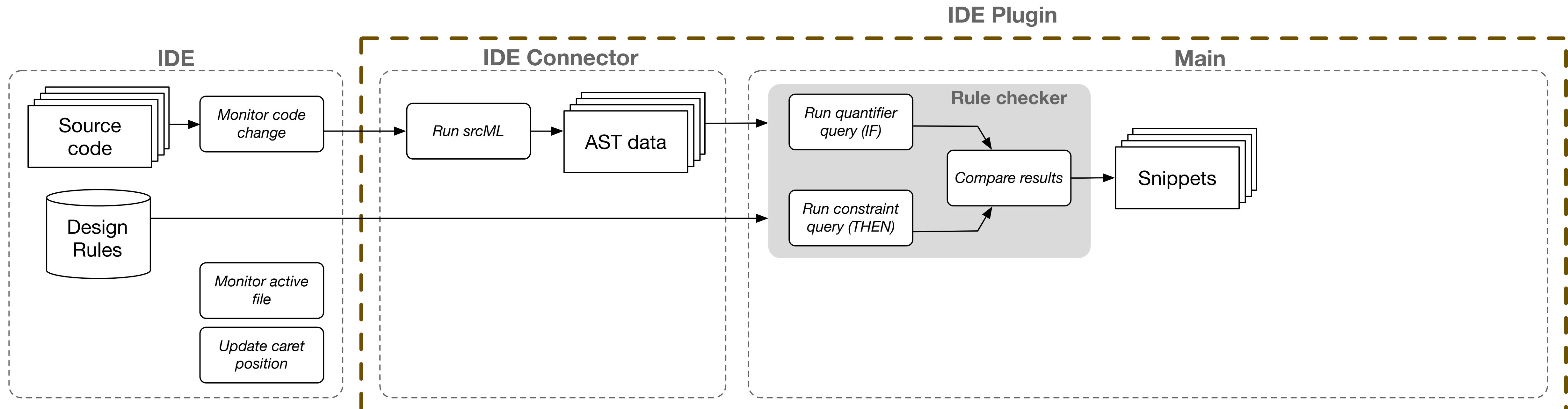
Rule Checker In ACTIVE DOCUMENTATION

In an IF/THEN structure of a rule:

IF part → Quantifier Query

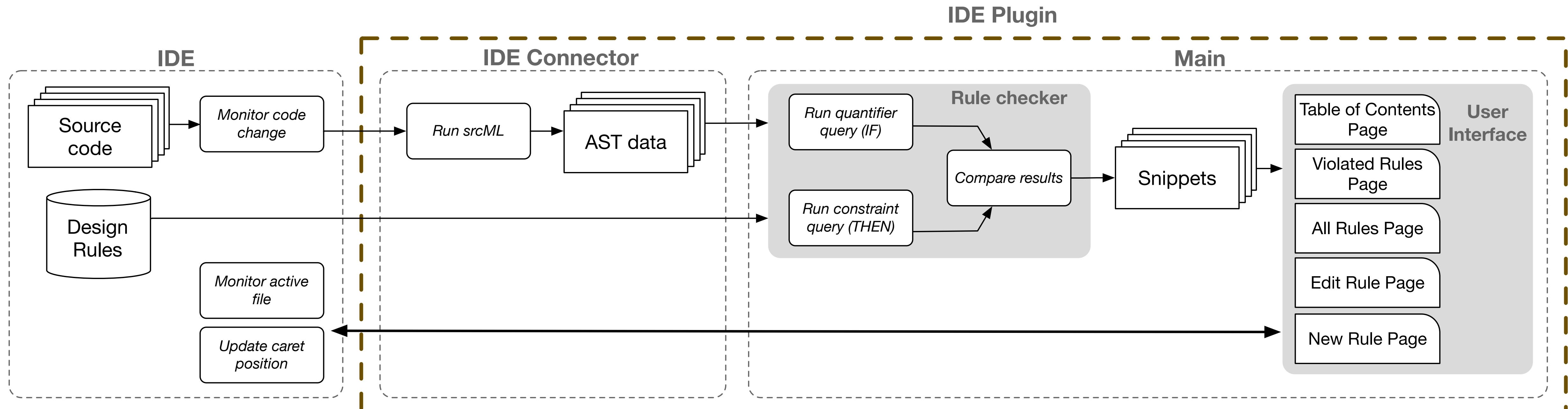
THEN part → Constraint Query

Compare the results of queries → **Satisfied** and **Violated** Snippets



ACTIVE DOCUMENTATION System Architecture

- After generating code snippets, they are visualized in the user interface through different pages.
- The User Interface sends and receives tasks to and from the IDE



Rule Organization

1

The screenshot shows a Java development environment with two open files: `DesignDocCommand.java` and `ADTCommand.java`. The `DesignDocCommand.java` file contains code for a command class that interacts with a `DesignDoc` entity. The `ADTCommand.java` file is currently closed.

On the right side, there is a separate window titled "Active Documentation". This window includes a "Tags" section with a grid of letters from A to U, where "All" is selected. Below the tags is a "Rules" section containing a list of eight numbered rules:

- 1 All Microtask commands must be handled by Command subclasses
- 2 Commands must implement execute
- 3 Artifacts should be marked as a data region with an @Entity annotation
- 4 Microtasks must have a reference to the Artifact that it belongs to
- 5 Communication between artifacts should be indirected through a Command
- 6 Objects to be sent to an external service should have a corresponding DTO that needs to be transferred has a DTO, i.e. ObjectMapper should not be in Entity.

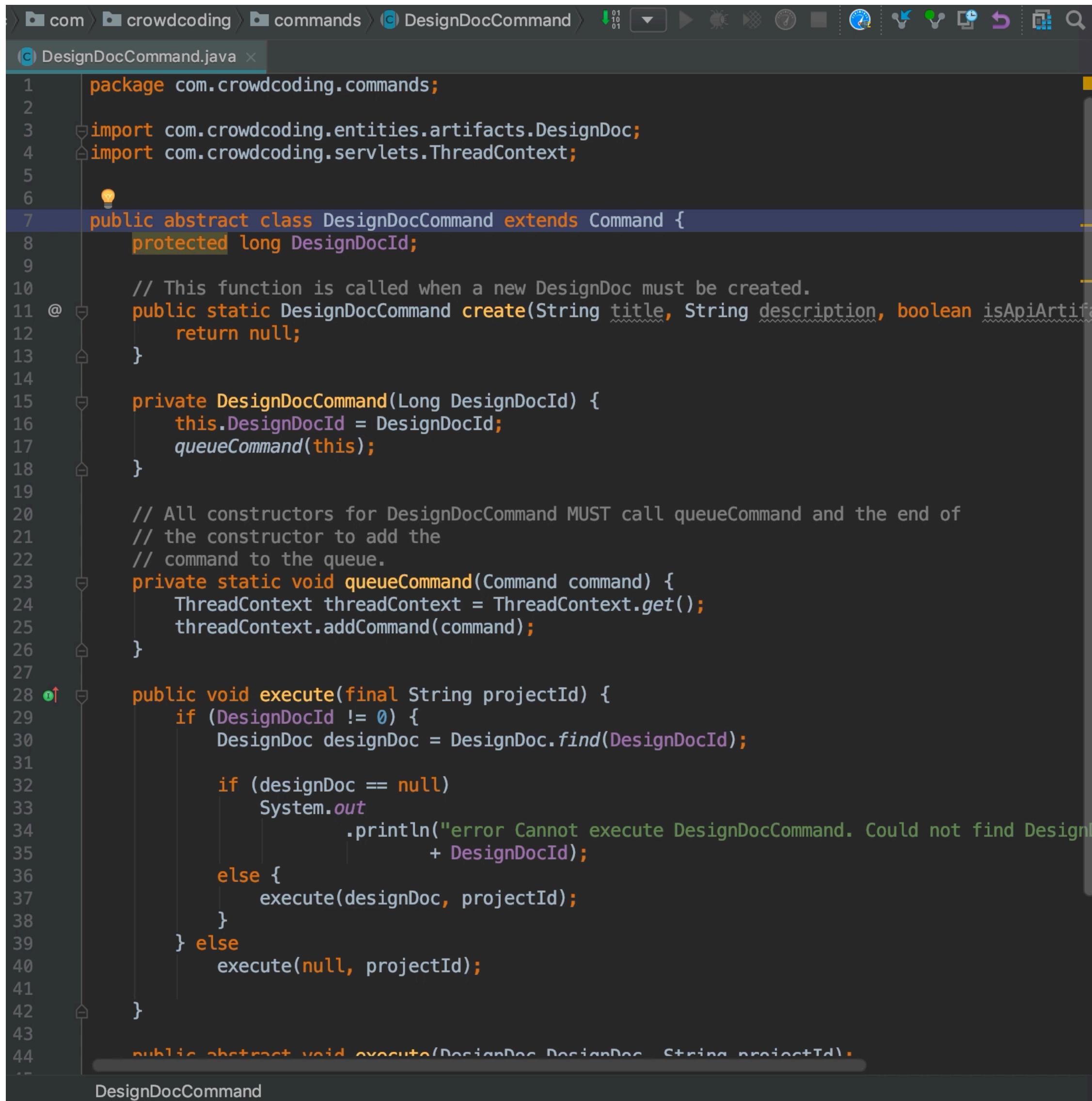
The "Active Documentation" window also lists several categories: Command, Entity, Objectify, Data Transfer Objects, Microtask, Persistence, Serialization, and Sharding. The "Entity" category is highlighted with a circular selection.

```
1 package com.crowdcoding.commands;
2
3 import com.crowdcoding.entities.artifacts.DesignDoc;
4 import com.crowdcoding.servlets.ThreadContext;
5
6 public abstract class DesignDocCommand extends Command {
7     protected long DesignDocId;
8
9     // This function is called when a new DesignDoc must be created.
10    @ ...
11    public static DesignDocCommand create(String title, String description, boolean isApiArtifact)
12        return null;
13    }
14
15    private DesignDocCommand(Long DesignDocId) {
16        this.DesignDocId = DesignDocId;
17        queueCommand(this);
18    }
19
20    // All constructors for DesignDocCommand MUST call queueCommand and the end of
21    // the constructor to add the
22    // command to the queue.
23    private static void queueCommand(Command command) {
24        ThreadContext threadContext = ThreadContext.get();
25        threadContext.addCommand(command);
26    }
27
28    public void execute(final String projectId) {
29        if (DesignDocId != 0) {
30            DesignDoc designDoc = DesignDoc.find(DesignDocId);
31
32            if (designDoc == null)
33                System.out.println("error Cannot execute DesignDocCommand. Could not find DesignDoc"
34                                + DesignDocId);
35            else {
36                execute(designDoc, projectId);
37            }
38        } else
39            execute(null, projectId);
40
41    }
42
43    public abstract void execute(DesignDoc designDoc, String projectId);
44}
```

DesignDocCommand

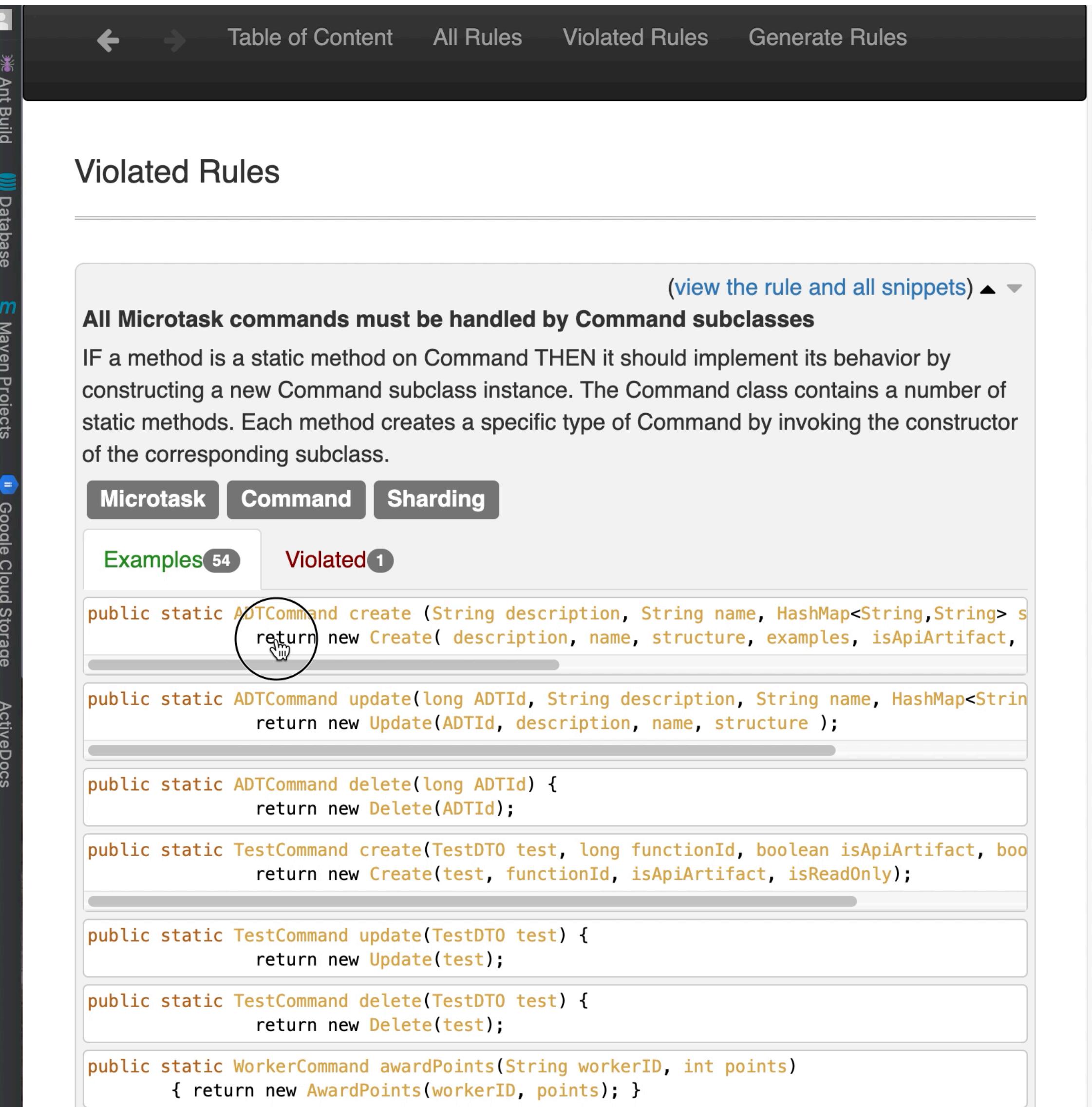
Using Example Code Snippets

2



The screenshot shows an IDE interface with a Java file named `DesignDocCommand.java` open. The code implements the Command design pattern. It includes a static factory method `create` that returns a new instance of a concrete command subclass. The `execute` method checks if a design document exists and executes it. The code uses annotations like `@Table` and `@Column` from the `com.crowdcoding.entities.artifacts` package.

```
1 package com.crowdcoding.commands;
2
3 import com.crowdcoding.entities.artifacts.DesignDoc;
4 import com.crowdcoding.servlets.ThreadContext;
5
6
7 public abstract class DesignDocCommand extends Command {
8     protected long DesignDocId;
9
10    // This function is called when a new DesignDoc must be created.
11    @Table("DesignDoc")
12    public static DesignDocCommand create(String title, String description, boolean isApiArtifact) {
13        return null;
14    }
15
16    private DesignDocCommand(Long DesignDocId) {
17        this.DesignDocId = DesignDocId;
18        queueCommand(this);
19    }
20
21    // All constructors for DesignDocCommand MUST call queueCommand and the end of
22    // the constructor to add the
23    // command to the queue.
24    private static void queueCommand(Command command) {
25        ThreadContext threadContext = ThreadContext.get();
26        threadContext.addCommand(command);
27    }
28
29    public void execute(final String projectId) {
30        if (DesignDocId != 0) {
31            DesignDoc designDoc = DesignDoc.find(DesignDocId);
32
33            if (designDoc == null)
34                System.out.println("error Cannot execute DesignDocCommand. Could not find DesignDoc " + DesignDocId);
35            else {
36                execute(designDoc, projectId);
37            }
38        } else
39            execute(null, projectId);
40    }
41
42    public abstract void execute(DesignDoc designDoc, String projectId);
43}
44
```



The screenshot shows a tool interface for managing rules. The top navigation bar includes `Table of Content`, `All Rules`, `Violated Rules`, and `Generate Rules`. The main area is titled `Violated Rules` and displays a rule titled **All Microtask commands must be handled by Command subclasses**. Below the rule description, there are tabs for `Microtask`, `Command`, and `Sharding`. Under the `Command` tab, there are two tabs: `Examples` (54) and `Violated` (1). A single violated example is shown, with the problematic line of code highlighted and circled:

```
public static ADTCommand create (String description, String name, HashMap<String, String> structure, boolean examples, boolean isApiArtifact, boolean isReadOnly) {
    return new Create( description, name, structure, examples, isApiArtifact, isReadOnly );
}
```

The code snippet shows a static factory method `create` that returns a new instance of the `Create` class, which implements the `ADTCommand` interface.

Instant Feedback

3

```
com>crowdcoding>commands>DesignDocCommand.java (C) ADTCommand.java (C)
public void execute(DesignDoc designDoc, String projectId) {
    if (DesignDocId != 0) {
        DesignDoc designDoc = DesignDoc.find(DesignDocId);
        if (designDoc == null)
            System.out.println("error Cannot execute DesignDocCommand. Could not find DesignDoc with id " + DesignDocId);
        else {
            execute(designDoc, projectId);
        }
    } else
        execute(null, projectId);
}

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

protected static class Create extends DesignDocCommand {
    private String description;
    private String title;

    private boolean isApiArtifact;
    private boolean isReadOnly;

    public Create(String title, String description, boolean isApiArtifact, boolean isReadOnly) {
        super(0L);
        this.title = title;
        this.description = description;
        this.isApiArtifact = isApiArtifact;
        this.isReadOnly = isReadOnly;
    }

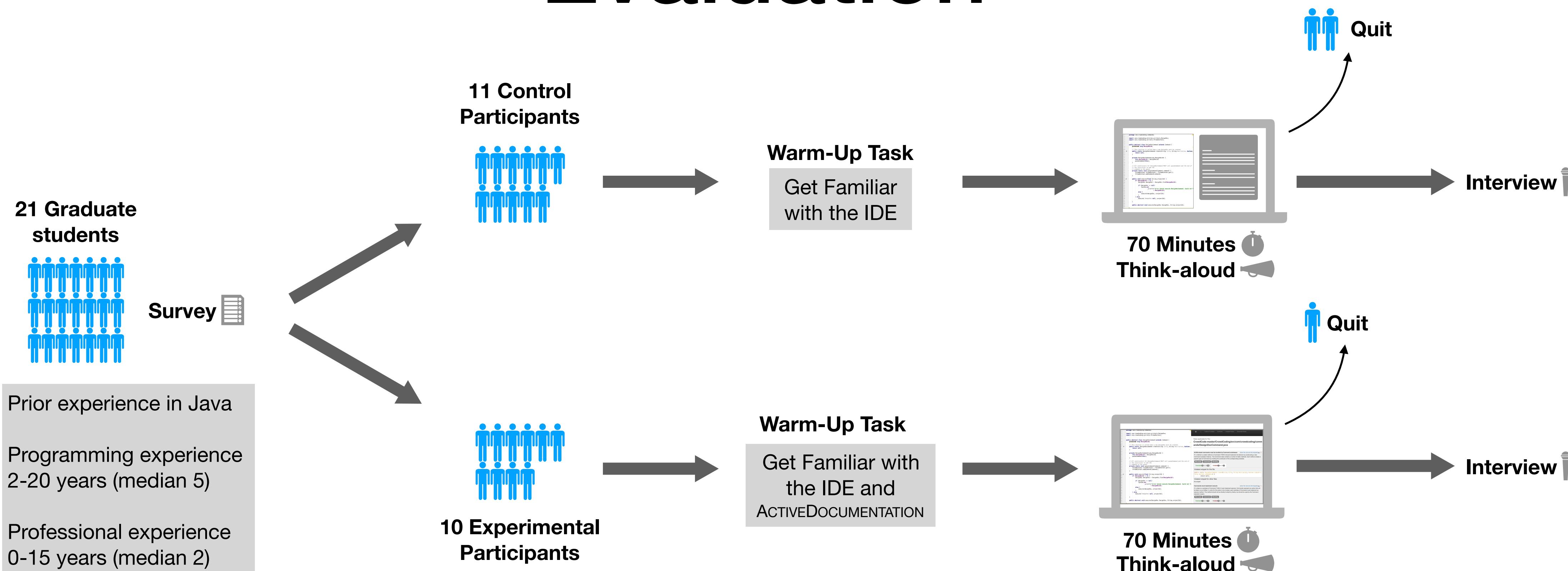
    public void execute(DesignDoc designDoc, String projectId) {
        new DesignDoc(title, description, isApiArtifact, isReadOnly, projectId);
    }
}
```

- (view the rule and all snippets) ▲ ▼
- 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. 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
- (view the rule and all snippets) ▲ ▼
- Commands must implement execute**
- 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 1 out of 54 Violated 0 out of 0
- (view the rule and all snippets) ▲ ▼
- Artifacts should be marked as a data region with an @Entity annotation**
- IF an object is an artifact subclass THEN it needs to be an entity. To signal that instances of a class constitute a separate data region, the class should have the @Entity annotation. All Artifact subclasses should be marked as a data region.
- (view the rule and all snippets) ▲ ▼
- Microtasks must have a reference to the Artifact that it belongs to**
- IF a class is a subclass of Microtask THEN it needs a field representing the reference to the associated entity. Each Microtask represents work to be done on an Artifact. As such, it needs to be connected back to its owning artifact through a reference to the Artifact. Without the reference, they need to have an ID of the artifact and for submitting they need to load the data beforehand.

Research Question

- ▶ Compared to traditional documentation, are developers able to use ACTIVEDOCUMENTATION to write code following design rules more quickly and successfully?
- ▶ In what ways does ACTIVEDOCUMENTATION support developers in writing code in an unfamiliar codebase?

Evaluation



Task: Add a small feature to an existing code

- Existing code: web-based IDE, 9K LOC, 107 Java classes, abstraction based on *artifacts* (persisted in a persistence framework)
- Requested code: add a new *artifact*, add 20 lines of code, edit 2 lines of code

Result - Quantitative

	Diff		Time (Minutes)		Submitted Lines of Code		
	Added	Removed	First Edit	Task Durat.	Missing	Incorrect	Task Irrlvnt
Control Group							
Mean	45.00	3.56	20.33	68.33	7.44	1.78	20.67
Median	36.00	3.00	12.00	70.00	1.00	1.00	8.00
Std. Dev.	39.64	3.40	19.82	3.39	8.37	2.54	27.76
Experimental Group							
Mean	29.67	4.44	6.33	48.89	1.89	0.11	5.89
Median	29.00	3.00	6.00	47.00	0.00	0.00	2.00
Std. Dev.	6.36	5.50	3.71	17.44	5.30	0.33	9.87
All Participants							
Mean	37.33	4.00	13.33	58.61	4.67	0.94	13.28
Median	29.00	3.00	8.50	70.00	0.00	0.00	3.00
Std. Dev.	28.65	4.46	15.59	15.77	7.37	1.95	21.59
p value	0.142	0.343	0.015	0.038	0.056	0.043	0.082

- ▶ Experimental participants were **3 times faster** in starting editing the code and **28% faster** in finishing the task.
- ▶ Experimental participants added few lines of code and removed more lines of code.
- ▶ Experimental participants submitted **98% fewer incorrect LOC**.

Result - Qualitative

Control Group

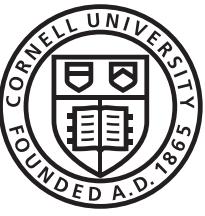
- ✗ Challenges in finding relevant design decisions within the design documentation
- ✗ Challenges in connecting code with design decisions
- ✗ Challenges in finding relevant pieces of code, scattered in different classes

Experimental Group

- ✓ Used *Violated Rules* page to find relevant design decisions.
- ✓ Used the violated snippets to identify relevant places to make changes.
- ✓ Used example snippets listed to compare examples of the rule and the faulty lines of code.
- ✓ Used real-time feedback to detect errors and violations early, immediately after changing the code without running the application.

Active Documentation: Helping Developers Follow Design Decisions

Sahar Mehrpour, Thomas D. LaToza, Rahul K. Kindi



Scenario

- After some time, Alice finds a class assuming being an Artifact and considers it as an **Example**. She tries to re-implement them in the new class.
- She writes some code and wants to know if it conforms with the design rules ... But she is not able to **verify** it herself.
- She looks at the **rule checkers** the company is using, but they are only reporting **universal defects** and not helpful.
- Frustrated, she commits her code and waits for **code reviews** from other developers.

3

Active Documentation

Our solution: **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.

7

ACTIVE DOCUMENTATION

8

Instant Feedback

15

Result - Quantitative

17

	Diff	Time (Minutes)	Submitted Lines of Code				
	Initial	Removed	Final Edit	Task Durat.	Missing	Inorrect	Task Infringe.
Control Group					7.44	1.78	20.67
Mean	45.00	3.56	20.33	68.33	1.00	1.00	8.00
Median	36.00	3.00	12.00	70.00	8.37	2.54	27.76
Std. Dev.	39.64	3.40	19.82	3.39			
Experimental Group							
Mean	29.67	4.44	6.33	48.89	1.89	0.11	5.89
Median	29.00	3.00	6.00	47.00	0.00	0.00	2.00
Std. Dev.	6.36	5.50	3.71	17.44	5.30	0.33	9.87
Participants							
Mean	37.33	4.00	13.33	58.61	4.67	0.94	13.28
Median	29.00	3.00	8.50	70.00	0.00	0.00	3.00
Std. Dev.	28.65	4.46	15.59	15.77	7.37	1.95	21.59
p value	0.142	0.343	0.015	0.038	0.056	0.043	0.082

- Experimental participants were **3 times** faster in starting editing the code and **28% faster** in finishing the task.
- Experimental participants added few lines of code and removed more lines of code.
- Experimental participants submitted **98% fewer** incorrect LOC.

Result - Qualitative

Control Group	Experimental Group
Challenges in finding relevant design decisions within the design document	Used Violated Rules page to find relevant design decisions.
Challenges in connecting code with design decisions	Used the violated snippets to identify relevant places to make changes.
Challenges in finding relevant pieces of code scattered in different classes	Used example snippets listed to compare examples of the rule and the faulty lines of code.

25

Used real-time feedback to detect errors and violations early, immediately after changing the code without running the application.

Thank You!

bit.ly/ActiveDocumentation

smehrpou@gmu.edu