A Live Environment for Inspection and Refactoring of Software Systems - Empirical Experiment

We worked on a **Live Refactoring Environment**. With this approach, software developers have access to several **refactoring suggestions** that aim to improve the quality of their software systems. This refactoring environment was developed for **Java** code on the **IntelliJ IDE**, and it focuses on analyzing specific quality attributes to identify particular refactorings.

Our tool is named **LiveRef**, and it includes the following refactorings:

- Extract Method
- Extract Class
- Extract Variable
- Move Method
- Inheritance to Delegation
- Introduce Parameter Object
- String Comparison

But, you may ask how this tool is different from the other existing refactoring tools. **LiveRef** allows developers to be guided to better code, through refactoring suggestions, while programming. They don't need to click on a button whenever they want to check their refactoring possibilities. Besides, with it, developers can visualize each refactoring in a more user-friendly way than the conventional and well-known refactoring tools.

Basically, this project was developed assuming that *software developers aim to create great software systems, with more quality, faster*.

To validate our beliefs and evaluate our tool, we should perform an empirical experiment with several participants. That's why we need your help!

To help us measure the quality of each project and its evolution, our tool collects several code quality metrics while you program and in each refactoring performed on your code. Then, these values are automatically saved on a Firebase database.

But don’t worry! Your identity will always be kept anonymous.

Also, notice that even with this tool, there is a chance that your code quality metrics may get worsen when applying each suggested refactoring. Each refactoring displayed by **LiveRef** is merely a suggestion and not a mandatory action that must be applied to your code.

If you have any question on this tool and how to use it, please send an email to sfcf@fe.up.pt .

User Guide

To start the experiment, you should launch the **IntelliJ IDE** and install our tool **LiveRef**. Please download the current version of **LiveRef** (depending on your IntelliJ version). To install it, you need to select your IDE preferences/settings. There you will find the "**Plugins**" menu, where you must select the option of "**Install Plugin from Disk**" (see the following image). Search the folder where you saved the attached .zip (file **LiveRef.zip**). Then, IntelliJ will install the plugin. Probably, you will need to restart IntelliJ to have full access to **LiveRef**.

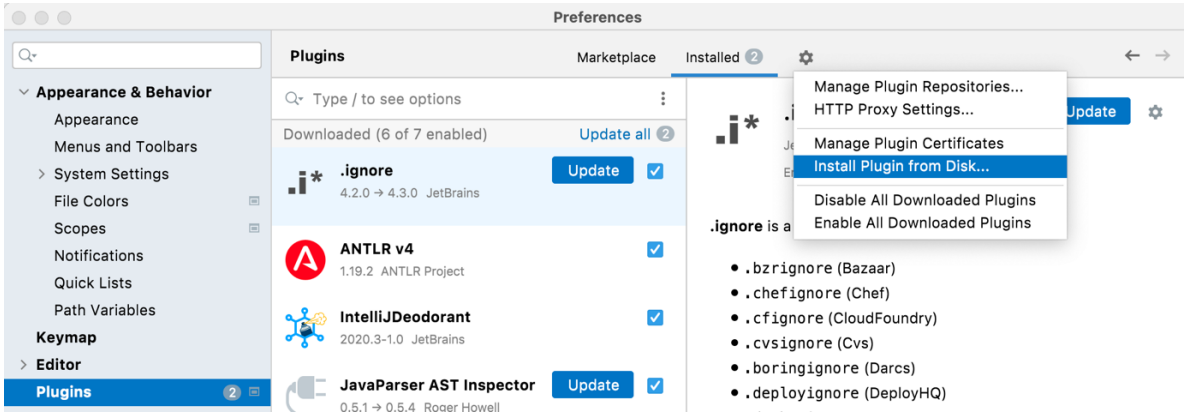


Figure 1 - How to install LiveRef.

After installing the plugin, you can activate the tool. The next steps depend on the IDE version you are using.

- For **IntelliJ 2021**, you should click on the option "**Live Refactoring**" on the main menu;
- For **IntelliJ 2022**, you will have a popup dialog showing when you open your project and a menu named "**Live Refactoring**" inside the **"Tools"** option on the main menu. If you activate the tool through the popup dialog, its starts automatically. If you choose to not activate it using the dialog, you can always start it through the "**Live Refactoring**" menu.

On the "**Live Refactoring**" menu, you will have access to two new options:

1. ***Start Analysis*** **–** It activates the tool and starts the analysis to find refactoring candidates;
2. ***Configure Tool*** **–** It allows the users to configure the tool and the thresholds for each refactoring.

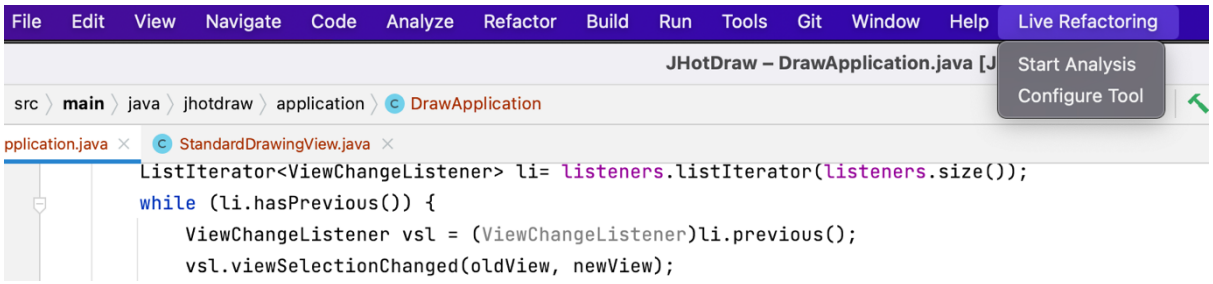


Figure 2 - "Live Refactoring" menu when using IntelliJ 2021.

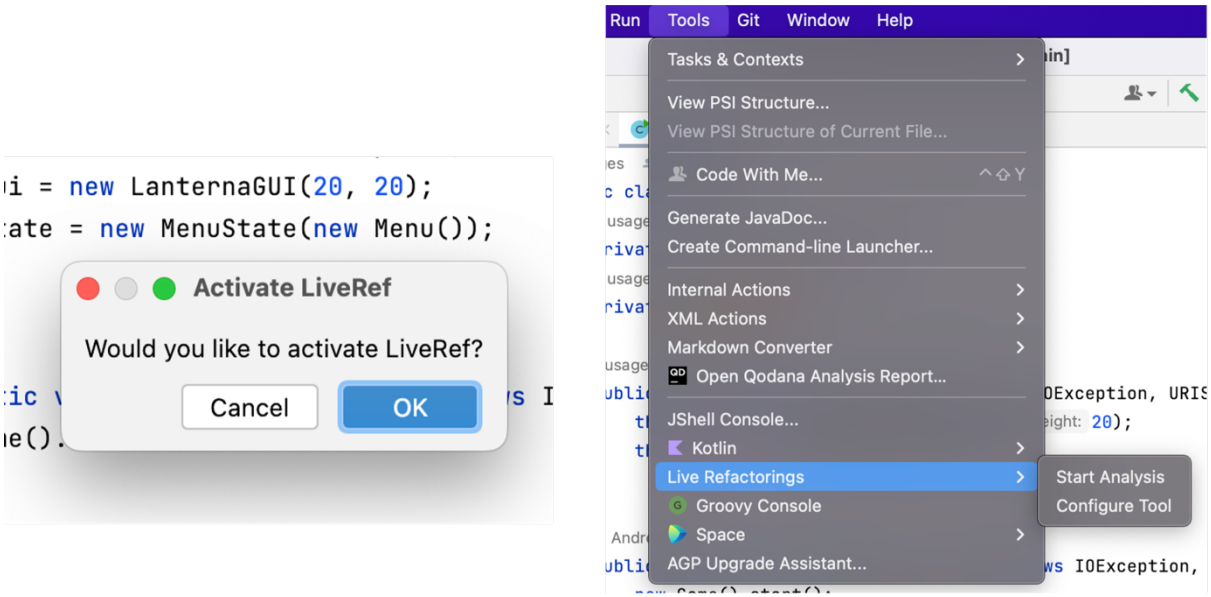


Figure 3 - Popup dialog and "Live Refactoring" menu when using IntelliJ 2022.

Start Analysis

After clicking on "***Start Analysis***" (or after activating the tool through popup dialog), the tool inspects the code and presents the different refactoring candidates. To simplify the suggestion of each refactoring, we display each candidate through different colors ranging from *light green* to *dark red*. The assigned colors were based on the number of candidates per refactoring type. **Light green** represents less impactful candidates and **dark red** highest impactful refactorings.

```
        ArrayList<Entity> entities = new ArrayList<>();
        entities.addAll(sourceClass.getAttributeList());
        entities.addAll(sourceClass.getMethodList());
        double[][] jaccardDistanceMatrix = new double[entities.size()][entities.size()];
        for (int i = 0; i < jaccardDistanceMatrix.length; i++) {
            for (int j = 0; j < jaccardDistanceMatrix.length; j++) {
                if (i != j) {
                    jaccardDistanceMatrix[i][j] = DistanceCalculator.getDistance(entities.get(i)
                } else {
                    jaccardDistanceMatrix[i][j] = 0.0;
                }
            }
        }
        return jaccardDistanceMatrix;
    }
```

Figure 4 - Displaying refactorings using multiple colors.

Then, if you click on a color, you will have access to a refactoring menu. This menu lists the refactoring candidates related to the line of code represented by that color. Imagine that there are overlapping refactorings related to that line of code. To simplify, we display the color of just one of those refactorings. But on the menu, the users have access to all refactorings that include that line.

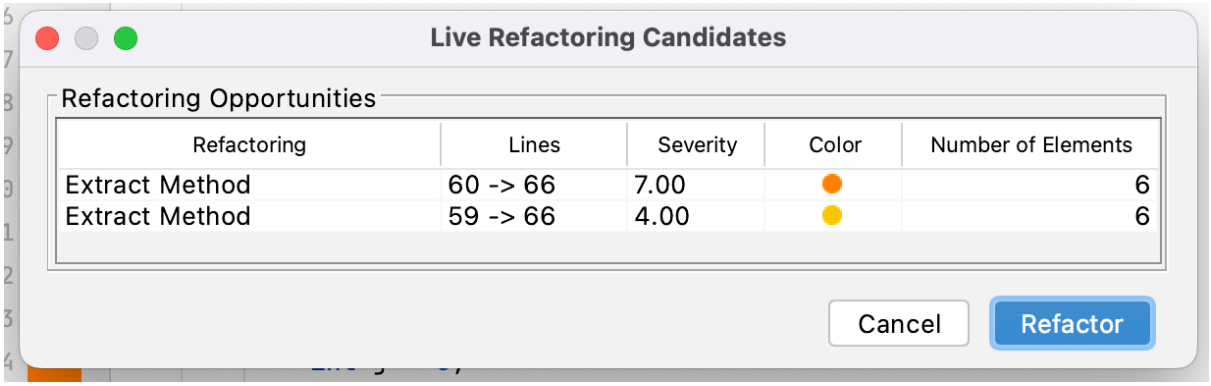


Figure 5 - Refactoring menu listing multiple refactoring candidates.

If you desire to perform one of the refactorings listed on the refactoring menu, you need to select it and click the "**Refactor**" button. Then, the refactoring activity will start.

After applying the refactoring, the tool sends the measured metrics to Firebase, starts a new code inspection, and displays the new refactoring candidates.

Note that the tool may not suggest all refactorings at the same time. As it identifies possible refactoring candidates by their type, it recommends them (e.g., there may be a time-lapse of seconds or less between listing Extract Variable and Extract classes candidates).

If the tool doesn't find any refactoring candidates, a message pops up in the bottom left corner of IntelliJ.

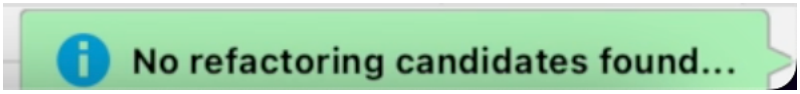


Figure 6 - Message displayed when no refactoring is found.

At any time, you can stop the tool from inspecting your code. To do so, you need to click on the "***Stop Analysis***" option, which is only available after you start the tool.

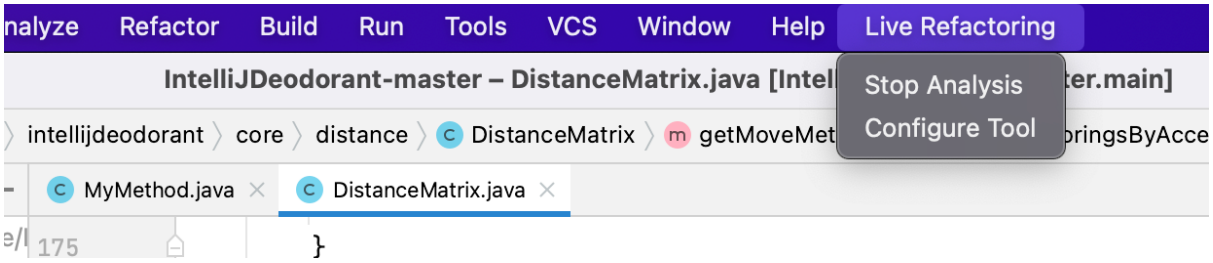


Figure 7 – "Stop Analysis" option displayed when using IntelliJ 2021.

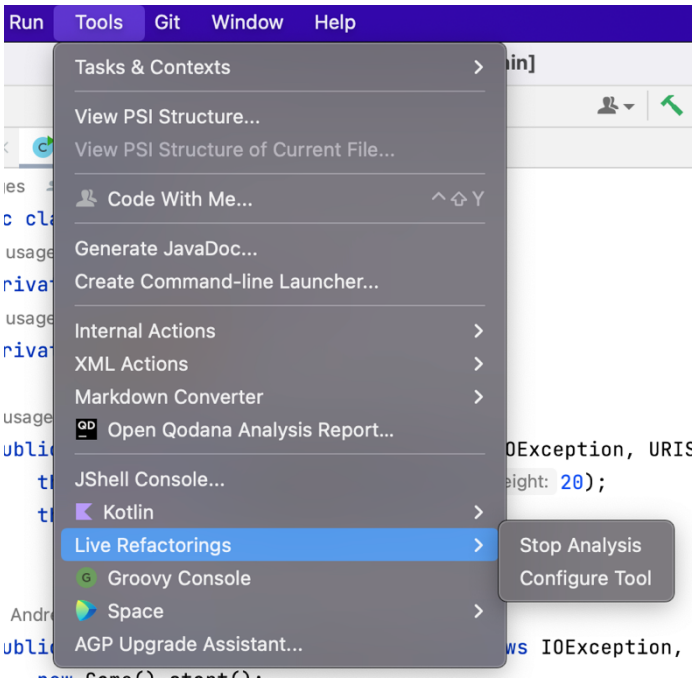
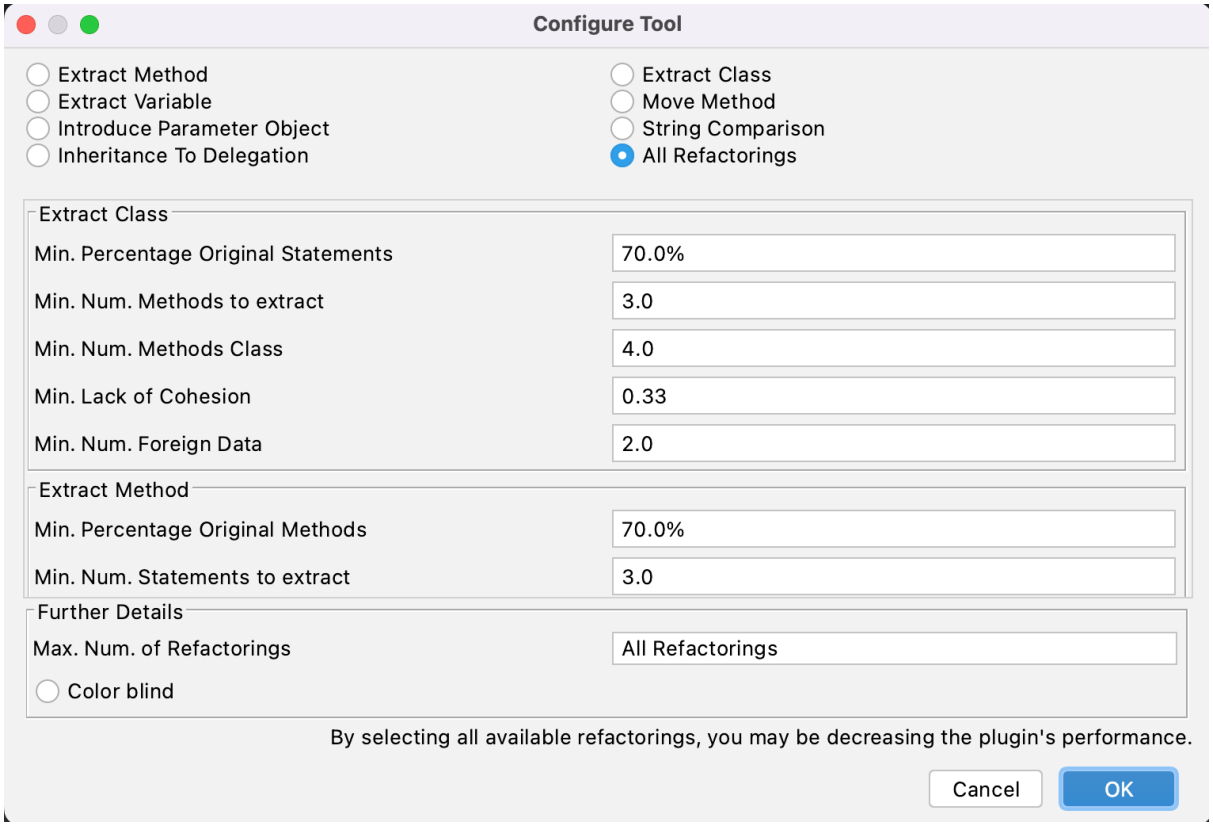


Figure 8 - "Stop Analysis" option displayed when using IntelliJ 2022.

Configure Tool

If you click on the "***Configure Tool***" option, you access the configuration menu. You can select the refactoring to include in the inspection process, change some **thresholds** used to detect or sort the refactoring candidates, **limit** the **number of refactorings** to be displayed (by refactoring type), or identify if you are color-blind. For color-blind people, the tool uses symbols instead of colors to list the refactoring candidates.



Extract Class

- ***Min. Original Methods:*** Minimum percentage of original methods to ensure that the original class maintains a percentage of its original methods, or it will be turned into an empty class;
- ***Min. Num. Methods to extract:*** Minimum number of methods to extract to ensure that the extracted class contains at least a specific number of methods;
- ***Min. Num. Methods Class:*** Minimum number of methods of the class being analyzed to ensure that the tool only inspects classes long enough and not simple and small ones;

- ***Min. Lack of Cohesion:*** Minimum lack of cohesion to ensure that the tool only analyzes low cohesive classes (highest lack of cohesion represents non-cohesive classes). Lack of cohesion should be a value between 0 and 1;
- ***Min. Num Foreign Data:*** Minimum number of foreign data to ensure that the tool only analyzes classes that use several foreign methods or fields from other classes.

Extract Method

- ***Min. Percentage Original Statements:*** Maximum percentage of original statements to ensure that the original method maintains a percentage of its original statements, or it will be turned into an empty method;
- ***Min. Num. Statements to extract:*** Minimum number of statements to extract to ensure that the tool doesn't identify too small and simple Extract method candidates;
- ***Min. Num. Lines of Code:*** Minimum number of lines of code of the method being analyzed to ensure that the tool only inspects methods long enough and not small ones;
- ***Min. Cyclomatic Complexity:*** Minimum cyclomatic complexity of the method being analyzed to ensure that the tool doesn't analyze not so complex methods;
- ***Min. Halstead Effort:*** Minimum Halstead effort of the method being analyzed to ensure that the tool doesn't inspect methods that are easy to understand.

Extract Variable

- ***Min. Length of Expressions:*** Minimum length of the expressions that should be inspected as possible Extract variable candidates to ensure that the tool doesn't explore small and simple expressions.

Introduce Parameter Object

- ***Min. Num. Parameters:*** Minimum number of parameters of the methods that should be analyzed to check if they are suitable to be an Introduce Parameter Object refactoring.

Inheritance To Delegation

- ***Max. Percentage inherited Methods:*** Maximum percentage of inherited methods that the subclass being analyzed should have to ensure that the tool only inspects classes that don't inherit enough methods from their superclass;
- ***Max. Percentage Override Methods:*** Maximum percentage of overridden methods that the subclass being analyzed should have to ensure that the tool only inspects classes that don't override enough methods from their superclass.

Included Refactorings:

- **Extract Method:** The *Extract Method* refactoring extracts some code from an existing method to a new one. Its purpose is to improve code by splitting a long and/or complex method into simpler ones;
- **Extract Class:** The *Extract Class* refactoring is applied when a class becomes overweight with too many methods, and its purpose becomes unclear. It involves creating a new class and moving methods and/or data to the new class;
- **Extract Variable:** The *Extract Variable* refactoring consists of placing the result of an expression in a variable to make it easier to understand;
- **Move Method:** The *Move Method* refactoring consists of moving a specific method from its original class to another one in which the method is used more often than its own original class;
- **Inheritance to Delegation:** The *Inheritance to Delegation* refactoring creates a superclass object, delegates methods to it, and gets rid of the inheritance. This refactoring is suitable for solving cases in which the subclass only uses a portion of the superclass methods or fields;
- **Introduce Parameter Object:** The *Introduce Parameter Object* refactoring replaces a long list of parameters of a class with a new object;
- **String Comparison:** The *String Comparison* refactoring allows to replace string comparisons using the operator *"=="* with method *equals().*