

# Assignment 2 Report: Static Analysis using Infer

Luca Airaghi

April 27, 2025

## 1. Project Overview

**Project Name:** `emotionalSong`

**GitHub URL:** None, it's a personal project

**Description:** EmotionalSongs is a project I worked on during my bachelor's degree, along with three other students. It was my first-ever project, which makes it especially interesting to analyze, so I can clearly see the mistakes I made back then and how much I've improved since. The application uses a CSV file to load the names of 500,000 songs. After logging in, users can search for a specific song and express the emotions they felt while listening to it. Unfortunately, users can't listen to the music directly within the app; they can only search for songs and leave comments. There are nine emotions that users can associate with a song, each rated on a scale from 1 to 5. Users can also view how others have rated a song, including the average emotion ratings based on all feedback. The application was developed in Java using Swing for the graphical interface and PostgreSQL as the database. One of the constraints of the group project was that the application had to be distributed. To meet this requirement, we implemented synchronized methods and used serialized classes to share objects between the client and server.

## 2. Setup and Environment

**Environment:** Docker image `bugcounting/satools:y24`

**JDK Version:** Java 17+

**Build Tool:** Maven

To set up the environment, I first created two folders: `before` and `after`, which are used to compare the code before and after the refactoring.

Next, I configured Docker to be used within the project. By default, the Docker container was set to use Java 11, so I updated it to use Java 17 instead.

Finally, I created a shell script to run Infer in `Eradicate` mode. This mode focuses on detecting null-related issues in Java code, but since my project only have these types, captures also all the issues that the "normal" mode captures. Below is the content of the script and the command used to run it:

```
#!/bin/bash
# Clean previous Infer output
rm -rf infer-out
# Run Infer in Eradicate mode on the whole Maven project
infer --keep-going --jobs 1 -- mvn clean compile
```

This script does the following:

- Removes any previous Infer output to ensure a clean run.
- Runs Infer in `Eradicate` mode to detect issues like null dereferences.
- Analyzes the entire Maven project by compiling it with `mvn clean compile`.

### 3. Running Infer on the Original Code (Before)

To execute the script, I navigated to the `emotional_song` directory and ran the following command:

```
./run_infer.sh
```

Here is a summary of the findings reported by infer:

Issue Type (ID)	Occurrences
Field Not Initialized (ERADICATE_FIELD_NOT_INITIALIZED)	74
Parameter Not Nullable (ERADICATE_PARAMETER_NOT_NULLABLE)	70
Null Dereference (NULL_DEREFERENCE)	20
Field Not Nullable (ERADICATE_FIELD_NOT_NULLABLE)	3
Return Not Nullable (ERADICATE_RETURN_NOT_NULLABLE)	2
Resource Leak (RESOURCE_LEAK)	1
<b>Total</b>	<b>170</b>

Table 1: Summary of issues found using Infer in Eradicate mode

Running Infer without eradicate mode, produces less results, but the ones that produces are already included in the Eradicate mode.

Since my project is also concurrent and distributed I run infer using a more precises mode (`racerd` to detect race conditions), but again I get the same results as before.

### 4. Modifications Made (After)

Now in this section of the report, I would like to show how I fix 3 type of warnings:

- Parameter Not Nullable

```
/*
Before warning: Parameter Not Nullable
'ImageIcon(...): parameter #1('arg1') is declared non-nullable but
the argument 'logoOneUrl' is nullable: call to Class.getResource(...) at
line 46
(nullable according to nullsafe internal models).
*/

java.net.URL logoOneUrl =
    getClass().getResource("/icons/Imagine-emotional.jpg");
    System.out.println(logoOneUrl);
    Icon logoOne = new ImageIcon(logoOneUrl);
    System.out.println(logoOne);

//After
```

```

java.net.URL logoOneUrl =
    getClass().getResource("/icons/Imagine-emotional.jpg");
    if(logoOneUrl == null) {
        System.out.println("Resource not found at
            /icons/Imagine-emotional.jpg");
        return;
    }
    Icon logoOne = new ImageIcon(logoOneUrl);

```

### Explanation of the improvement:

In the original code, the `logoOneUrl` was directly passed to the `ImageIcon` constructor without a null check, which could result in a `NullPointerException` if the resource wasn't found.

To address this, I added a null check before the `ImageIcon` constructor:

- **Check if `logoOneUrl` is null:** If it is, an error message is printed, and the method returns early, preventing any further code execution with a null value.
- **If not null, proceed with `ImageIcon`:** Only if `logoOneUrl` is valid, the `ImageIcon` is created.

This improvement ensures that `ImageIcon` always receives a valid, non-null URL, preventing runtime exceptions and improving code stability.

### • Null Dereference

```

/*
Before error: Null Dereference
object 'result' last assigned on line 75 could be null and is dereferenced
at line 76
*/

public boolean queryControlloUser(DataBase db, String user) throws
    SQLException {
    String s="";
    boolean bol = true;
    ResultSet result = db.submitQuery("SELECT username FROM Utenti WHERE
        username = '"+user+"'");
    while(result.next()){ //INCRIMINATED LINE
        s = result.getString("username"); //vado a leggere tutte le righe
        corrispondenti alla colonna 'username'
        if(s.equals(user)){
            bol=false;
            break;
        }
    }
    return bol;
}

//After:

```

```
public boolean queryControlloUser(DataBase db, String user) throws
SQLException {
    String s = "";
    boolean bol = true;
    try (ResultSet result = db.submitQuery("SELECT username FROM Utenti
        WHERE username = '" + user + "'")) {
        if (result == null) {
            System.err.println("Query failed or returned null ResultSet");
            return false;
        }
        while (result.next()) {
            s = result.getString("username");
            if (s.equals(user)) {
                bol = false;
                break;
            }
        }
    }
    return bol;
}
```

### Explanation of the improvement:

In the original code, the `result` object, which is the `ResultSet` from the database query, was not checked for null before being used. If `db.submitQuery()` returned `null` (which could happen in case of a failed query or no results), the `result` object would be dereferenced in the `while(result.next())` loop, leading to a `NullPointerException`.

To fix this issue, I added the following improvements:

- **Null Check for result:** I wrapped the `ResultSet` in a `try-with-resources` block, which ensures that the resource is properly closed after the operation. Before entering the `while(result.next())` loop, I added a check to see if `result` is `null`. If it is, an error message is printed, and the method returns `false` immediately to avoid further processing with a null `ResultSet`.
- **Resource Management:** The `ResultSet` is now automatically closed at the end of the `try` block due to the use of `try-with-resources`.

By doing this improvement, I make sure that:

- The code now avoids dereferencing a `null` object, preventing a `NullPointerException` from being thrown.
- By checking the result before continuing with the rest of the logic, we ensure that the code behaves safely even if the query fails or returns no data.

- Resource Leak

```
/*
Before error: Resource Leak
resource of type 'java.net.HttpURLConnection' acquired by call to
'openConnection()' at line 541 is not released after line 541.
```

```

    */

public void queryPopolaDb (DataBase db) throws SQLException {
    ...
    try {
        Connection connection = (Connection) db.getConnection();
        CopyManager copyManager = new CopyManager((BaseConnection)
            connection);
        BufferedReader fileReader = new BufferedReader(new
            InputStreamReader(percorsoProgramma.openConnection()
                .getInputStream()));

        copyManager.copyIn(copyQuery, fileReader);
    } catch (SQLException | IOException e) {
        e.printStackTrace();
    }
}

//After:
public void queryPopolaDb(DataBase db) throws SQLException {
    ...
    HttpURLConnection connectionUrl = null;
    try (Connection connection = (Connection) db.getConnection())
    {
        connectionUrl = (HttpURLConnection)
            percorsoProgramma.openConnection();
        try (BufferedReader fileReader = new BufferedReader(new
            InputStreamReader(connectionUrl.getInputStream())) {
            CopyManager copyManager = new
                CopyManager((BaseConnection) connection);
            copyManager.copyIn(copyQuery, fileReader);
        }
    } catch (SQLException | IOException e) {
        e.printStackTrace();
    } finally {
        if (connectionUrl != null) {
            connectionUrl.disconnect();
        }
    }
}
}

```

### Explanation of the improvement:

In the original code, the `HttpURLConnection` opened with `openConnection()` was not closed, causing a resource leak. To fix this, I saved the connection in a variable and explicitly called `disconnect()` in a finally block. I also used try-with-resources for the `Connection` and

`BufferedReader` to ensure automatic resource management.

## 4.1 False Positive

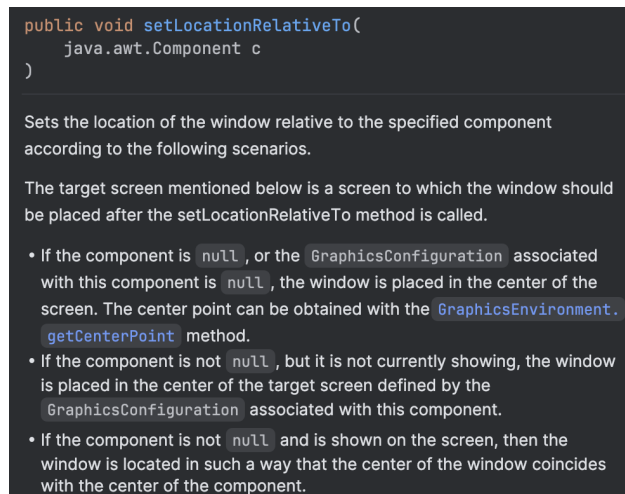
While searching for warnings / errors, I also found false positives, for instance:

```
warning: Parameter Not Nullable
'SchermataAvvio.setLocationRelativeTo(...)': parameter #1('arg1') is declared
non-nullable but the argument is 'null'.
```

but when I went in the class to see the issue, this was the line:

```
setLocationRelativeTo(null);
```

which clearly shows that the argument of the method is null, but is the method itself that requires a null argument, in order to set the windows in the center of the screen:



```
public void setLocationRelativeTo(
    java.awt.Component c
)
```

Sets the location of the window relative to the specified component according to the following scenarios.

The target screen mentioned below is a screen to which the window should be placed after the `setLocationRelativeTo` method is called.

- If the component is `null`, or the `GraphicsConfiguration` associated with this component is `null`, the window is placed in the center of the screen. The center point can be obtained with the `GraphicsEnvironment.getCenterPoint` method.
- If the component is not `null`, but it is not currently showing, the window is placed in the center of the target screen defined by the `GraphicsConfiguration` associated with this component.
- If the component is not `null` and is shown on the screen, then the window is located in such a way that the center of the window coincides with the center of the component.

Figure 1: Method's description

To avoid this warning appearing in the class, I used:

```
@SuppressWarnings("parameter.null")
```

after checking for other similar warning in the class, to avoid skip the correction of them.

## 4.2 After refactoring

Before this section I show how I refactor two kind of warnings, but of course the assignment required to fix a significant amount of issue, in order to gain knowledge with infer. To show that below you will find a summary of the findings after the refactoring:

Issue Type (ID)	Occurrences
Field Not Initialized (ERADICATE_FIELD_NOT_INITIALIZED)	68
Parameter Not Nullable (ERADICATE_PARAMETER_NOT_NULLABLE)	34
Null Dereference (NULL_DEREFERENCE)	19
Field Not Nullable (ERADICATE_FIELD_NOT_NULLABLE)	2
Return Not Nullable (ERADICATE_RETURN_NOT_NULLABLE)	1
<b>Total</b>	<b>124</b>

Table 2: Summary of issues found using Infer in Eradicate mode

## 5. PMD Vs. Infer

The results produced by PMD and Infer differ notably in their focus and nature of analysis. PMD mainly identifies style violations, design inefficiencies, and code smell patterns such as unnecessary constructors, unused private fields, or the use of implementation types like ArrayList instead of interfaces. For example, it flags naming conventions (e.g., uppercase letters in package names), redundant empty constructors, and unused imports or variables. On the other hand, Infer focuses more on potential runtime issues and memory/resource management bugs, such as null pointer exceptions, resource leaks, and concurrency problems. While PMD emphasizes clean, maintainable code and adherence to Java best practices, Infer aims to catch bugs that could lead to failures during program execution. Therefore, both tools complement each other, with PMD enhancing code quality and readability, and Infer helping ensure runtime correctness and safety.

Problems found	
Line	Problem
1	<a href="#">Package name contains upper case characters</a>
49	<a href="#">Avoid unnecessary constructors - the compiler will generate these for you</a>
49	<a href="#">Document empty constructor</a>
1	<a href="#">Package name contains upper case characters</a>
31	<a href="#">Avoid unused private fields such as 'nomeDato'.</a>
1	<a href="#">Package name contains upper case characters</a>

Figure 2: Example of PMD results

## 6. Conclusion

Using Infer for static analysis turned out to be a valuable experience that significantly improved my understanding of code quality and the importance of preventative tools in software development. The tool was effective in uncovering subtle but critical issues that could lead to bugs, runtime exceptions, or resource mismanagement. Among the most relevant problems detected were uninitialized fields, null dereferences, and violations of non-nullable constraints—issues that are often hard to catch during regular testing or manual code reviews.

What makes these types of bugs particularly dangerous in real-world applications is their potential to cause unexpected crashes or unstable behavior, especially in production environments where reliability is crucial. A null dereference, for example, might seem like a minor oversight, but

it can crash an entire service if not handled properly. Similarly, not releasing system resources like file streams or network connections can lead to memory leaks or system slowdowns over time.

One of the most interesting benefits of using Infer was how it forced me to adopt a more defensive and thoughtful approach to my code. Knowing that the analyzer would flag potential misuses pushed me to write cleaner, more robust code from the start.

The only real challenge I faced during this assignment was finding a suitable project to analyze. I initially downloaded several open-source repositories, but many of them either failed to build, contained no detectable issues, or were too minimal to offer meaningful insights. Eventually, I chose to analyze my own project, which not only fit the requirements but also provided a more personal and practical context for applying what I learned.

In conclusion, Infer is a powerful and practical tool that I would highly recommend integrating into any software development workflow. It serves not only as a bug detector but also as a guide for writing safer, higher-quality Java code.