Refactoring Code Smells in Google Spreadsheets: A User Response Study

Divyansh Kankariya - 200352

Undergraduate Project (Faculty Supervisor: Prof. Sruti Srinivasa Ragavan), IIT Kanpur

Abstract

This project aims to investigate how users react to real-time notifications of detected code smells in Google Spreadsheets. By providing users with suggested steps for refactoring in the form of real-time notifications, users can enhance their spreadsheet code quality. We presented users with suggested approaches to refactor complex formulas and a feature to transform them into a LET function. The process of breaking down formulas and presenting them to users poses a challenge in terms of human interaction. We implemented several alternatives, including low-fidelity prototypes to further assess their effectiveness. The implications of this project's findings can significantly impact designing tools that support code quality in spreadsheets and applications with similar functions.

Contents

1 Introduction

HCI is extremely significant in our present-day digital world, as it shapes the way we engage with technology. To meet the growing need for efficiency and ease-of-use, it is critical to prioritize the development of interactive computing systems with a focus on HCI. Detecting code smells and suggesting refactoring can improve the efficiency and user-friendliness of interactive computing systems. Code smells indicate deeper problems and can make code hard to read, understand, and maintain. Our project focuses on tackling code smell issues in Google Spreadsheets. We worked on an add-on that detects "multiple operations" code smells in Google Spreadsheets and offers recommendations for improved maintainability. The add-on employs Google Apps Script and a web node server to detect "multiple operations" code odour while the user inputs a formula in a cell. It displays a notification in the sidebar with advice and an option for refactoring. The user can promptly act on the suggestions with a single click. This add-on development and evaluation of different ways to display the refactoring suggestions to the user can serve as a

model for future HCI improvements in such tools and improve the overall user experience of spreadsheet users.

1.1 Spreadsheet code smells

Spreadsheets are widely used in the industry, with an estimate 90% of all analysts utilizing them for calculations. End-user programmers, who are often not formally trained software engineers, develop these spreadsheets. These programmers are actually more abundant than traditional programmers, and the artifacts they create hold similar importance to an organization as traditional software does. Spreadsheets, technically, share similarities with software since spreadsheet formulas and source codes both comprise of constants, variables, conditional statements, and references to other areas of the software. It seems reasonable, therefore, to explore software engineering principles that are also relevant to spreadsheets.

Code smells are hints or signals that something may be amiss in a software code. These are typically minor design weaknesses or breaches of good coding practices that can build up and render the code difficult to comprehend, prolong, or modify in the future. There are various ways in which code smells can impact code quality. They can make the code more complex, reduce its readability, hinder testing, and introduce bugs or errors. Additionally, they can slow down development and increase maintenance costs. Code smells must be refactored to prevent bigger problems like technical debt. Refactoring means improving design and readability without changing how it works. Doing so can reduce maintenance costs and improve code quality.

1.2 Our focus

There are numerous types of formula code smells existent in spreadsheet code. These include:

1. Multiple operations
2. Multiple references
3. Conditional Complexity
4. Long Calculation Chain
5. Duplicated Formulas

In this project, we have initially focused on refactoring of the Multiple operations code smell as it is the most common code smell. We tried implementing the refactoring model for this code smell as a Google spreadsheet add-on. Several design alternatives for showing the suggestion to the user were implemented for further evaluation.

2 Multiple operations code smell and its refactoring

2.1 Multiple operations code smell

The "multiple operations" formula code smell in spreadsheets refers to a situation where a single formula performs multiple calculations or operations in a single cell. This can make the formula hard to read and understand, and increase the likelihood of errors. Examples of this code smell include:

- A formula that calculates the commission on sales by multiplying the sales amount by a commission rate, and then subtracting a fixed amount:

  ```
  =B2*0.05-100
  ```

- A formula that calculates the average of a range of cells, and then applies a condition to the result:

```
=IF(AVG(A1:A10)>50, "Above
Average", "Below Average")
```

- A formula that performs multiple calculations within nested functions:

```
=SUM(IF(B2:B10>100,
C2:C10*D2:D10,0))
```

2.2 Refactoring the smell

To refactor this code smell, the multiple operations in the formula can be **separated into separate cells** or formulas, each with a clear and specific purpose. For example:

- Separate the commission rate and fixed amount into separate cells, and perform the calculation in a separate cell:

```
=B2 * CommissionRate
 - CommissionFee
```

- Calculate the average in one cell, and then apply the conditional formatting in a separate cell:

```
=AVERAGE(A1:A10)
=IF(B1>50,"Above Average",
 "Below Average")
```

- Break down the nested function into multiple cells or formulas, each with a specific purpose:

```
=IF(B2:B10>100,
 C2:C10*D2:D10,0)
=SUM(E2:E10)
```

Alternatively, we can replace these formulas with more readable forms in the same cell. One such form is the **LET** function. The LET function is a powerful feature in spreadsheets that allows you to define a name for a formula or expression that you can use repeatedly in your worksheet. This function is available in recent versions of Excel and Google Sheets.

```
=LET(name1, value1,
        name2, value2,
             ...,
                 calculation)
```

Here,

- "name" is the name you want to give to the variable.
- "value" is the value you want to assign to the variable.
- "calculation" is the formula or expression in which you want to use the variables.

The LET function evaluates the calculation using the assigned value and returns the result. It is useful for simplifying complex formulas and making them easier to read and understand.

Using this refactoring method in one of the examples discussed above,

- Separate the commission rate and fixed amount into separate variables and perform the calculation as shown:

```
=LET(CRate, 0.05,
        CFee, 100,
            B2 * CRate - CFee)
```

Both of these solutions help in:

- Simplifying the expression into smaller parts
- Reducing the cognitive load while revisiting this formula expression for debugging/modifying it
- Mapping different parts of the expression to appropriate semantic labels

3 Suggesting solutions to the user

3.1 Problem and challenges

Both the refactoring solutions discussed in the last section require splitting the formula into units. We need to design appropriate algorithms which determine the units in which the formula must be divided These splits must be such that each part carries a significant semantic to it but, at the same time, is not over-simplified. Predicting a single such split accurately is a difficult task because at times it might require field knowledge or knowing the context the formula is being used in.

Also, giving users multiple split suggestions to choose from can be a bad idea because too many suggestions can overwhelm and distract the user, resulting in cognitive overload and lower productivity. The user may focus on minor adjustments instead of the content, resulting in frustration and decreased creativity. Moreover, too many suggestions can lead to decision paralysis and tool dissatisfaction.

3.2 How did we present these suggestions

Avoiding the problems caused by showing multiple suggestions, we chose to present the user with just one solution for now. While the user is using the spreadsheet, every time more than one major operation is detected in the currently active cell, the user is given a sidebar notification presenting the solution.

Among various possible ways of presenting the solution, sidebar notification was chosen because of its un-obtrusiveness. A sidebar notification is typically less intrusive than alert dialogs because it appears to the side of the main content rather than taking over the entire screen. This means that users can continue to interact with the content on the main screen without being interrupted or having to dismiss the notification.

Inside the sidebar notification, we decided to incorporate the following details about the solution:

- Description of the problem detected (the smell) and suggestive action in short
- A coloured version of the formula with highlighting on the recommended positions to split at in the formula
- A LET formula that can replace the current formula. The variable names in the LET formula are preset as var_1, var_2, var_3 ...
- An option to convert their expression into a LET formula

As much as it is important to include all this information in the notification, it is also important to keep this information concise and short because such suggestions are more likely to engage users and encourage them to take action. Users are more likely to engage with clear and actionable suggestions rather than those that require a lot of time and effort to understand. Keeping the notification short also helps decrease the cognitive load for the user, which can otherwise lead to frustration and decreased productivity. Users have limited mental resources available for processing information, and lengthy suggestions can quickly deplete those resources, leading to decision fatigue and reduced satisfaction with the tool.

3.2.1 Describing the problem detected

We decided to keep this part as short as possible :

```
WARNING: This cell contains multiple
operations and is prone to
errors/misunderstandings. It is
suggested to split it into more cells.
```

3.2.2 Splitting the formula

Providing the highlighted version of the formula and generating a LET function requires determining the positions where the formula should be split. For this, we used an algorithm which parses the input formula and initially splits the formula at all points where operators are present. It looks for operators only outside any parenthesis or any nested function which might be present inside the given expression. Then among the units obtained, it finds the units which involve only constants and merges them with units that precede them. For example, the formula:

```
=sum(B6:B11)*C14/24*B7
```

will be split into "sum(B6:B11)", "C14/24" and "B7".

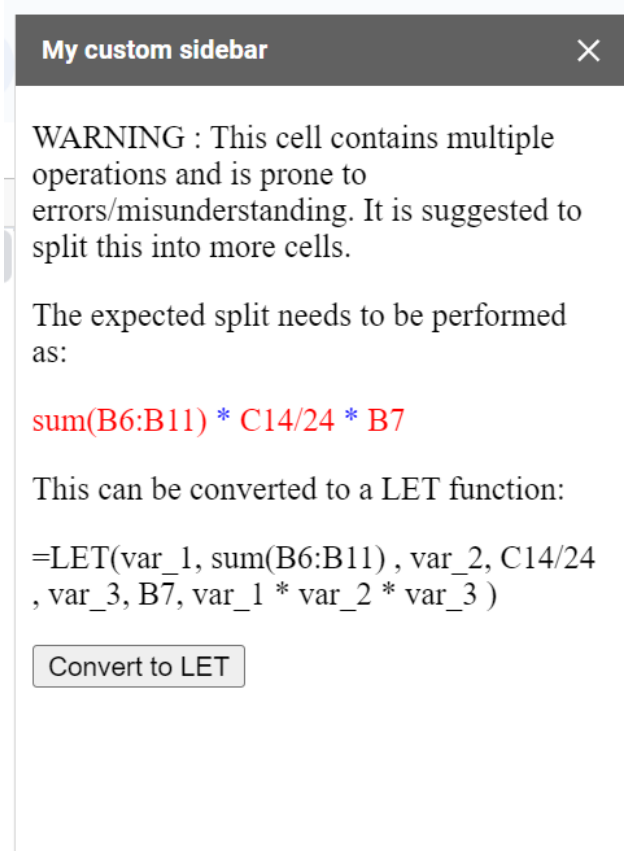

figure 1: Initially designed format of the notification

Once the formula is split, we present the user with a version of their formula which is alternatively coloured - highlighting the places at which the split is suggested. Also, a LET version of the formula is presented, which maps each of these units to different variables and performs the final calculation. Users can click a "Convert to LET" button which replaces the currently active cell content to this LET function.

3.2.3 Further design evolution

We realised that how we displayed the LET function was hard to read and, at the same time, not flexible in how it pre-sets the variable names. it doesn't allow the user to add variable names of their choice to the sub-expressions. We tried implementing quick working prototypes of designs that are more readable and flexible - provide a way of getting variable names from the user.

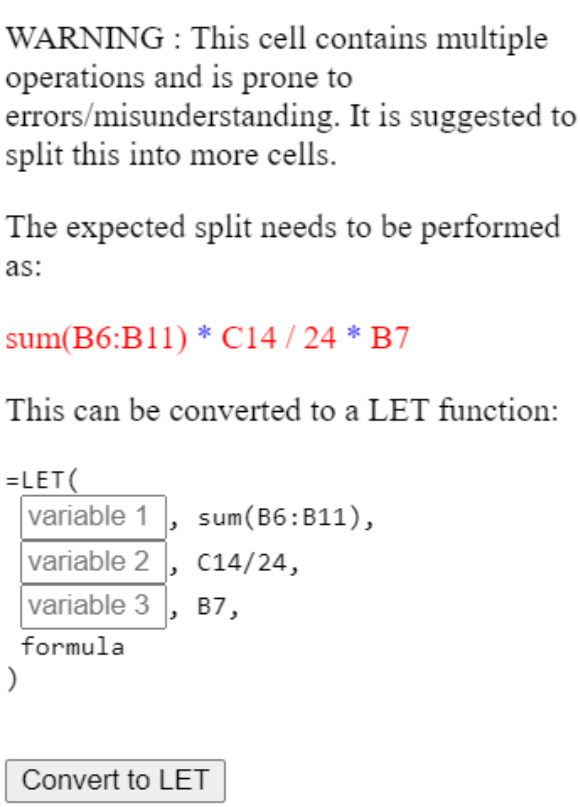

figure 2: New and more readable designed format of the notification. The names that the user enters in the input boxes are reflected in the final formula in real-time

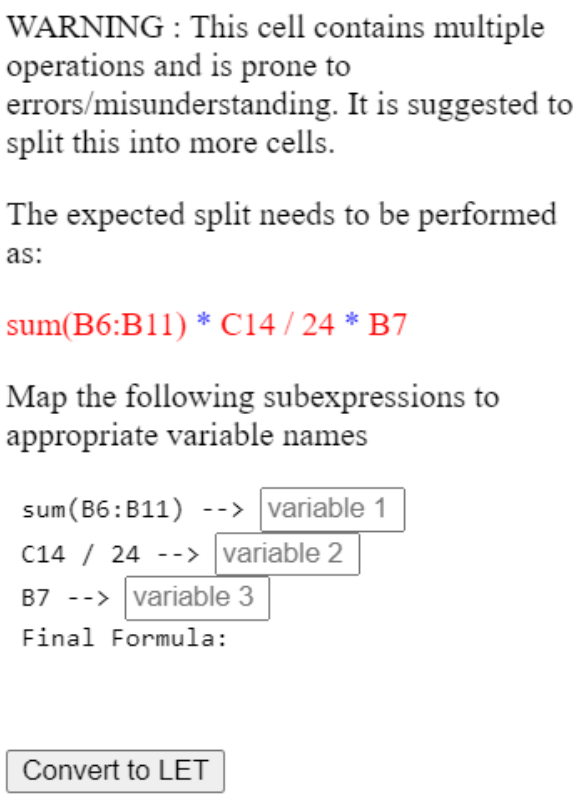

figure 3: New and more readable designed format of the notification. It shows user a mapping of variables instead of a function. The names that the user enters in the input boxes are reflected in the final formula in real-time

3.3 How our solution works

The add-on was built using Google Apps script API. This add-on provides additional features and tools that can help users to manipulate and analyze their data more efficiently within Google Sheets. This API helps read the currently active cell content and send it as a request to a separately built web server which does the smell detection and other computations.

This web server was built using node.Js and was hosted on railway.app service. Once the formula is processed by the server, it generates the code smell information, suggestion on how it can be split, and a LET function which can act as a replacement. This data is sent back to the apps script API as a JSON object, the Apps script API reads this JSON data structure and if needed, presents a sidebar notification to the user.

4 Conclusions

4.1 What did we achieve yet

We developed a Google Spreadsheet add-on that detects the "multiple operations" code smell and provides suggestions for refactoring. We analysed various ways in which the suggestion can be presented to the user and brainstormed ways of tackling the challenges such a tool might face. Working prototypes were made at pace to aid the further pending evaluation of our tool.

4.2 Persisting limitations

The currently developed solution, though at a good position, carries many limitations within it:

- Only the "Multiple Operations" code smell was addressed in our implementation; other code smell types still require attention.
- Our add-on requires an internet connection and cannot be used offline due to calls to a web server for formula processing.
- Our solution requires toggling the API repeatedly to start the task of detection. The formulas are not read and processed automatically.
- The split suggestions we provided are unary and non-flexible. The user cannot choose his own way of splitting the formula he has entered.
- The solution lacks an automated way of splitting the formula into multiple cells. it just provides a suggestion to do so.

4.3 Further Plan

Future work on this project will tentatively involve:

- Adding the ability to detect additional code smell types and present their refactoring solutions.
- Comparing and evaluating ways to present refactoring suggestions and norification designs
- Work on a path to create ideal design rules and guidelines for such refactory-help add-ons that are user-friendly and flexible.

Our project aims to inspire more research in real-time code smell detection and refactoring in various programming languages and platforms, including the design guidelines for such tools.

References

Hermans, F., Pinzger, M., & van Deursen, A. (2014, February 7). Detecting and refactoring code smells in spreadsheet formulas. Empirical Software Engineering, 20(2), 549–575. https://doi.org/10.1007/s10664-013-9296-2