Togglge Smell Detection in Google Chrome

Md Tajmilur Rahman, Dharani Kumar, Sumit Sarkar, Omar Faruk
Concordia University, Montreal, QC, Canada

***Abstract*—Identifying code smell is the first step to re-factor a software system. In Google Chrome developers use feature toggles to control their features and keep developing on a single trunk. This paper investigates code smell caused by improper use of feature toggles in Google Chrome that we call as toggle smell. Using feature toggles to hide/unhide features is an old concept but recently being practised in many software companies. Improper use of toggles might cause code smell including duplicate code, dead code, coupling and lack of cohesion. We classified the smells caused by toggles in six different patterns Dead Toggles, Tested Toggles, Spaghetti Toggles, Mix of Compile-Time and Runtime Toggles, Legacy toggles and Spread Toggles. We used AST parser tool as an eclipse plug-in to identify these toggle smells in major modules of Google Chrome source code. We found that TODO nested toggles smells, TODO spaghetti toggles and TODO spread toggles in "chrome" and "base" modules of Google Chrome.**

***Index Terms*—Software Refactoring, Software Re-Engineering, Software Design, Feature Toggles, Code Smell, Technical Debt, Mining Repository**

I. Introduction

TODO

II. Related Works

In order to study about the code smells and feature toggle well we have studied many paper but four of them most influential.Separating features in source code: an exploratory study [1] In this paper the researchers investigate about the separation of features in existing code.For the the purpose of the investigation they conducted an exploratory study in the context of two existing systems: **gnu.regexp and jFTPd**.Their believe about the study feature is that the features of interest to strip out will not always be known a-prior but will emerge as systems evolve.They have proposed two methodologies to separate out the features from the source code which are either tangled within a single method or across several classes.In lightweight separation of concern mechanism is: how far can they get by improving the object-oriented structure that exists in the system to capture features explicitly? If the feature is explicit and modularized in an object-oriented structure, then they could use naming conventions (and other simple tools) to help decompose a feature out of a system.The Hyper/J tool allows one to state declaratively what code in an existing set of class files contributes to a particular dimension, and within a dimension to particular concerns. AspectJ is an extension to Java to support aspect-oriented programming developed at Xerox PARC. They considered two common kinds of feature encodings in two existing Java systems. One is tangling of a feature within a method in the gnu.regexp system and another is tangling of a feature between classes in the jFTPd system.These entangling types In first case, probably lost cohesion of Lightweight SOC approach.but the splitting and restructuring was done by manually: moving fields, moving methods, forming methods, modifying inheritance and interface associations.Then they tried to separate these features using their different mechanisms (Hyper/J, AspectJ and lightweight separation of concern mechanism).Tangling in a Single Method is the method they worked with from the gnu.regexp system. This method performs part of the regular expression matching functionality. It supports several different kinds of matches, including matches across lines, matches where carat characters do not need to match at the beginning of the line and so on. Each of these kind of matches corresponds to a different feature. They wanted to separate these features that were originally tangled in a single method so as to produce a system that might support, for instance, only multi-line matches. Partial Dead Code Elimination [2]The paper presents an algorithm to eliminate partial dead code and to improve the speed of execution of program by moving the unnecessary variable assignment to the farthest location in the control flow graph, without changing the semantics of the program.The algorithm maintains a Graph whose nodes are basic block of statements and edges represents the branching information. The partial dead code elimination is a sequence/repetition of assignment sinking and further followed by dead code eliminations. The algorithm uses the dead code elimination techniques to eliminate the dead code and repeatedly tries to sink the assignments. In that process there are certain second order effects are classified as follows: **Sinking - Elimination Effects**- This is a recursive definition where an assignment statement can be sunk until it can be eliminated by dead code elimination **Sinking - Sinking Effects -** One assignment sinking can lead to sink other assignment statements. **Elimination - Sinking Effects -** Eliminating one or more dead assignments might lead to sinking of other assignment statements.**Elimination - Elimination Effects -** One dead assignment elimination might lead to subsequent assignment elimination if the control flow graph leading to the end node doesnt use the variable used in the assignment statement which is being eliminated. The algorithm he program addresses all the above second order effects caused by the movement of the assignment statements and also covers arbitrary control flow structures, distinguishes between the profitable code movement across the loop and fatal code movement into the loops.The dead code elimination program runs the following two functions:**dce**which eliminates the dead assignments by a dead variable analysis data and **ask** which does assignment sinking in the program by the delayability analysis.Dead variables are identified using a backward directed bit-vector

based dataflow analysis technique. The dead assignments are identified using definition-use graphs.The delayability analysis is a technique to identify the location in the block of code to hoist a variable assignment.Eliminating dead code from Xquery programs [3]In this paper they propose a technique for performing static basic dead-code analysis and eliminate the dead code from an XQuery program.XQuery is a query and functional programming language basically takes one (or possibly several) XML document as input, performs some computation based on its tree view, and finally outputs a result in the form of another XML document. The core of the XQuery language is composed of XPath expressions that make it possible to navigate in the document tree and extract nodes that satisfy some conditions.They use two methodology **Path-Error Detection** and **Static Code Refactoring and Highlighting**.In the path-error detection they gave a scema as input to an XQuery program and For each XPath expression occurring in XQuery program, they just check that it is meaningful or not with respect to the constraints described in scema. If navigational information contained in a given path contradicts the constraints described in Scema then the path will always return an empty sequence of nodes.All XQuery instructions that depend on that path are dead code and maybe they should remove.For Static Code Refactoring they build an AST that consists in extracting all the path expressions from the program and checking their satisfiability individually.Then, in a second step, these paths are combined with the schema, and checked again for satisfiability.Each kind of unsatisfiable path is marked differently in the AST.each path is considered as a sequence of basic navigation steps possibly with qualifiers. The first step is analyzed. Then each additional step is successively appended to this initial step and the resulting path is analyzed in turn. This makes it possible to identify precisely where the error has been introduced in the path and remove them.An Eclipse Plugin to Support Code Smells using Binary Logistic Regression Model[4]This paper presents an eclipse plugin tool to detect code smells using a technique build upon Binary Logistic Regression Model by using the heuristic of expert knowledge to calibrate the model.BLRM is used for estimating the probability of occurrence of an event by fitting the data set on to a logistic curve.The binary logistic regression with dependent variable having two values (either code smell present or absent) has been used to detect the probability of code smell.The BLR is calibration is done using the statistical tools like R or SPSS and JRI (Java R Inteface) is used to interface with R language from the eclipse plugin.The plugin operates in two different modes (local and remote).The plugin allows the source code to be annotated using a UI action and then collects the metrics from the annotated source code. The following metrics are calculated by the plugin for the classes and methods: **1. LOC**,**2. Depth of inheritance tree**,**3. Number of methods**,**4. Lack of cohesion**,**5. Number of parameters** and **5. McCabe Cyclomatic Complexity**

III. Study Setup

In order to explore the feature toggle related smell we have searched for open source projects and found some projects with feature toggle implemented in those project.But we have choose Google Chome as our main code base because it has more then 5 years active development life cycle with feature toggle in it and it is released frequently.So there might be any issue with feature toggle which might not maintain properly.

IV. Case Study

TODO: Answer the research questions

V. Code Smells

TODO

VI. Results

TODO

A. Qualitative

TODO

B. Quantitative

TODO

VII. Conclusion

TODO