RefDetectorJS for Mining Refactoring Data from Version Controlled Repositories

Vincius Garcia, *Student, UFMG*

Abstract—This paper discuss the development of a system designed to detect refactorings on diferent versions of Javascript code files using the same heuristics used by RefDetector tool for Java programming language. RefDetectorJS use Objects and functions signatures to detect refactoring between two different versions of the code. The focus of the project is on Javascript modules designed to be compiled with Nodejs.

Keywords—Computer Science, Refactoring, UFMG, Javascript, RefDetector.

1 Introduction

Refactoring is a frequent and well characterized behavior that can be used to evaluate and try to understand human behavior and habits when developing computer programs. To help these studies is necessary to create tools that can detect refactoring and return useful information about it. The RefDetector tool was developed by M. Danilo Silva of Federal University of Minas Gerais (UFMG) and aim refactoring detection on programs written in the Java programming language. This project discuss the development of a second tool called RefDetectorJS that aims to detect refactoring in Javascript source files.

RefDetectorJS is coded on Javascript and uses the same heuristics of M. Danilo's RefDetector. The main approach used is to collect a set of signagures from two versions of a source file, compare the differences to identify refactoring candidates between the two versions and then selecting the better candidates with a similarity metric called CloneFraction, which was precisely defined by Weigerber in [2]. The main objective is to provide a tool for mining javascript refactoring history from Javascript version controlled repositories.

- *M. Danilo Silva is with the Department Computer Science, Federal University of Minas Gerais, Brazil.*

The tool is implemented in Javascript and interpreted with Nodejs. To execute the tool it expect as parameters two consecutive versions of a same Javascript project and outputs a log of refactorings detected between the two versions.

October 17, 2014

2 Approach

2.1 Restrictions

Since Javascript have no class concept it was needed to apply some restrictions on the type of objects that should be read as classes. Those objects signatures are read with the same criteria used on Java Classes by M. Danilo's RefDetector, the other are read just as attributes (e.g. an array or a dictionary). It was also necessary to apply some restrictions to simplify the first release of the tool.

The first restriction was on how to define a class. On javascript the "class" concept doesn't exist, but it can be simulated using constructor functions, which is a function that if attached to an object when called instantiate attributes and functions on this object. The right way to instantiate such a "class" is by calling the "new" operator before the function call, which causes the function to be attached to an empty object, allowing it to build the class instance from scratch. We will also refer to constructor functions as classes on the rest of this article.

Inside a constructor function there are two ways to declared variables that could be used by the internal functions of the class:

- Declare it using the "this" keyword (e.g. "this.attribute = value")
- Declare it using the "var" keyword (e.g. "var attribute = value")

The main difference between these two declarations is that the first one is accessible outside the class instance, working as a public member. And the second is only accessible by the functions declared inside the class, and thus work as a private member. Therefore we will threat as public members the first group, and as private the second, either group can contain functions and attributes. Declarations made inside a class without one of these two keywords are global variables, and thus won't be counted as a class member.
For simplicity purposes we also estabilished some other restrictions:

- The objects analysed as classes must be declared with a constructor function (explained below) so that RefDetectorJS may be instantiated with the "new" operator.
- If a constructor function is declared inside one other constructor function it will be read only as a normal function.
- Functions and attributes declared outside those objects are not being parsed yet. This way the first release of the tool will follow the same structure used on M. Danilo's RefDetector for Java, which was centered on the class concept.

2.2 The RefDetectorJS parser

RefDetectorJS has its own parser implemented. This parser was build with help of Esprima[3], a tool that builds an Abstract Syntax Tree (AST) from a Javascript code.
The parser collect tree kinds of objects:

- Constructor functions (class declarations).
- Functions declared as public or private members of these classes.
- Attributes also declared as members.

The tree sets of values are build once for each project. Thus the set of functions contain all functions from all classes on this project and the same applies for the attributes.
To detect constructor functions the parser expect a function with at least one public attribute or function declared with the "this" keyword. The reason is because it would not be a class if there were no public members, it would be just a normal function. The parser also expect this function to return no value, if it does, it can't be a constructor function.
The parser output returns a set of signatures for each kind of object (classes, functions and attributes). This set is build to allow RefDetectorJS to search for refactoring candidates.
The signature of the classes, functions and attributes is collected as a tuple of sets and was based on Biegel et al.[1] method:

- The objects read as classes have as signature the tuple: (p, c, m) the module name, the class name and the set of class members respectively.
- The class attributes have as signature the tuple: (c, a, t, v) where c is the class name, a is the attribute name, t is the attribute type and v is the visibility of the attribute (e.g. public or private).
- The class functions have as signature the tuple: (c, f, l, r, v) where c is the class name, f is the function name, l is the parameters list, r is the set of class members being accessed within the body of the function and v is the visibility of the function (e.g. public or private).

Note that since javascript has dynamic typing, the attribute type detected is not guaranteed to be the same during the lifetime of a class instance. The type is only the type of the object assigned to this attribute when its declared inside the class.

2.3 Refactoring candidates

Once we have the signatures we need to search for refactorings candidates between the objects of two different versions of a same

project. A refactoring candidate is a touple: (i1, i2, ref) where i1 and i2 are objects from the code and ref is a type of refactor. For candidate to be accepted we use the following criteria:

- i1, and i2 must of the same kind of object: class, function or attribute.
- i1, and i2 signatures must match the refactoring criteria associated with ref.
- i1, and i2 signatures must partially match each other (there must be some resemblance).

The criterias are built on a set of removed and added objects. For each kind of object we these two sets. We now create a new group of sets: (C-,C+,F-,F+,A-,A+) where C- is the set of removed classes from old version of the project, and C+ is the set of added classes from on the new version of the project. The same applies to F- and F+ for functions, and A- and A+ for attributes.
For each different refactor there is a different matching criteria. Some examples of refactors and its criteria are shown in TABLE (1). The criteria are similar but not equal to the criteria shown on [1].

Refactoring Kind	Criteria
MoveClass	$\exists(p, n, m) \in C-$ and $\exists(p', n, m) \in C+$
RenameClass	$\exists(p, n, m) \in C-$ and $\exists(p, n', m) \in C+$
HideAttribute	$\exists(c, a, t, v) \in A-$ and $\exists(c, a, t, v') \in A+$ and $v \neq v'$
UnhideFunction	$\exists(c, f, l, r, v) \in F-$ and $\exists(c, f, l, r, v') \in F+$ and $v \neq v'$
AddParameter	$\exists(c, f, l, r, v) \in F-$ and $\exists(c, f, l', r, v) \in F+$ and $l \subset l'$
RemoveParameter	$\exists(c, f, l, r, v) \in F-$ and $\exists(c, f, l', r, v) \in F+$ and $l \supset l'$

TABLE 1
Table showing the criteria for choosing refactoring candidates.

2.4 Desambiguating Candidates

There are two possible kinds of ambiguity with the described candidate selection system. If there are two objects from the new project version associated with one single object in the old project version we have a *Source Ambiguousness*, if we have two objects on the old project version matching with one object from the new project version, this is called *Target Ambiguousness*.
On the disambiguation step we aim to reduce the number of candidates with a similarity criteria between 0% and 100% explained on the next section.
The first step is to rank the candidates for refactoring by the similarity between the old object and the new object. This way it is possible to solve ambiguity choosing always the most likely candidates. The second step is to remove candidates that are not good enough, for this we use a threshold of 80% (chosen in an arbitrary manner) and cut out all the candidates touples that don't fit this criteria. This way we avoid assuming refactorings on functions with similar signatures but totally different bodies.

2.5 Similarity Metric

RefDetectorJS uses its own similarity method. The method was build with help of the tool jsdiff[4] used to check differences betweens two different texts. The comparison between any pair of objects is made with the body of the object, it might be a class, a function or an attribute. For getting more detailed information even about attributes for the text comparison to work with, we used as the body of the objects an string version of the Abstract Syntax Tree generated by the Esprima tool. And then used word level comparison to compare each object. This way the comparison evaluates not only the names of the objects, but also the type, and the syntax of the code. It's also a way to compare all the objects with a single generic method, that can be used with all of them.
The similarity checker returns a number between 0 and 1 which is a percentage of similarity in number of matched tokens.

3 TESTS

The RefDetectorJS was tested using sample codes containing all the refactors described on TABLE (1). The code is executed in a command line like:

```
$
$ node refdetector.js <arg1> <arg2>
$
```

Where arg1 is the address of the old project version, and arg2 is the address of the new project version.

The code return in a file called refactors.log. A sample file is as shown below:

```
{ refactor: 'moveClass',
  similarity: '0.9928',
  before:
   { name: 'myClass',
     module:
        './projetoTeste1/testCase1.js',
     memberNames:
      [ 'privAttr1',
        'privAttr2',
        'privFunc1',
        'privFunc2',
        'attr1',
        'attr2',
        'fun1',
        'fun2' ] },
  after:
   { name: 'myClass',
     module:
        './projetoTeste2/testCase1.js',
     memberNames:
      [ 'privAttr1',
        'privAttr2',
        'privFunc1',
        'privFunc2',
        'attr1',
        'attr2',
        'fun1',
        'fun2' ] } }
{ refactor: 'addParameter',
  similarity: '1',
  before:
   { type: 'Function',
     name: 'fun2',
     class: 'myClass',
     visible: true,
     params: [ 'value' ],
     ref: [ 'attr1', 'attr2' ] },
  after:
   { type: 'Function',
     name: 'fun2',
     class: 'myClass',
     visible: true,
     params: [ 'value', 'value2' ],
     ref: [ 'attr1', 'attr2' ] } }
{ refactor: 'removeParameter',
  similarity: '0.98',
  before:
   { type: 'Function',
     name: 'fun1',
     class: 'myClass',
     visible: true,
     params: [ 'value' ],
     ref: [] },
  after:
   { type: 'Function',
     name: 'fun1',
     class: 'myClass',
     visible: true,
     params: [],
     ref: [] } }
```

For now the output is formatted in JavaScript Object Notation (JSON) in the future we might change the output for a less verbose option.

4 Conclusion

The tool was capable of parse and analyse any javascript code thanks to Esprima tool. The similarity metric applied wasn't fully tested, but in the the few testes applied it has calculated correctly a similarity between 100% in equal files, and 0% in totally different files, and is promissing. The system is still limited to class abstractions, but it is not the most common programming method on Javascript, but with the first version ready and with the help of Esprima tool, it is possible to compare methods, and attributes from diferents projects even if they are not inside classes. Some problems that are expecting using such method on Javascript are due to the impossibility in an dynamic weak-typed language to verify variable types and the results of calls to the function "eval" in a static analysis.

References

[1] Benjamin Biegel, Quinten David Soetens, Willi Hornig, Stephan Diehl, and Serge Demeyer. *Comparison of similarity metrics for refactoring detection.* In Proceedings of the 8th Working Conference on Mining

[2] P. Weigerber. *Automatic Refactoring Detection in Version Archives.* PhD thesis, University of Trier, 2009.

[3] A. Hidayat. *Esprima: Ecmascript parsing infrastructure for multipurpose analysis.* http://esprima.org/.