JSNOSE: Detecting JavaScript Code Smells

Amin Milani Fard
University of British Columbia
Vancouver, BC, Canada
aminmf@ece.ubc.ca

Ali Mesbah
University of British Columbia
Vancouver, BC, Canada
amesbah@ece.ubc.ca

***Abstract*—JavaScript is a powerful and flexible prototype-based scripting language that is increasingly used by developers to create interactive web applications. The language is interpreted, dynamic, weakly-typed, and has first-class functions. In addition, it interacts with other web languages such as CSS and HTML at runtime. All these characteristics make JavaScript code particularly error-prone and challenging to write and maintain. Code smells are patterns in the source code that can adversely influence program comprehension and maintainability of the program in the long term. We propose a set of 13 JavaScript code smells, collected from various developer resources. We present a JavaScript code smell detection technique called JSNOSE. Our metric-based approach combines static and dynamic analysis to detect smells in client-side code. This automated technique can help developers to spot code that could benefit from refactoring. We evaluate the smell finding capabilities of our technique through an empirical study. By analyzing 11 web applications, we investigate which smells detected by JSNOSE are more prevalent.**

***Index Terms*—JavaScript, code smell, web applications, smell detection**

I. INTRODUCTION

JavaScript is a flexible popular scripting language for developing Web 2.0 applications. It is used to offload core functionality to the client-side web browser and mutate the Document Object Modelling (DOM) tree at runtime to facilitate smooth state transitions. Because of its flexibility JavaScript is a particularly challenging language to write code in and maintain.

The challenges are manifold: First, it is an interpreted language, meaning that there is typically no compiler in the development cycle that would help developers to spot erroneous or unoptimized code. Second, it has a dynamic, weakly-typed, asynchronous nature. Third, it supports intricate features such as prototypes [23], first-class functions, and closures [8]. And finally, it interacts with the DOM through a complex event-based mechanism [29].

All these characteristics make it difficult for web developers who lack in-depth knowledge of JavaScript, to write maintainable code. As a result, web applications written in JavaScript tend to contain many *code smells* [9]. Code smells are patterns in the source code that indicate potential comprehension and maintenance issues in the program. Code smells, once detected, need to be refactored to improve the design and quality of the code.

Detecting code smells manually is time consuming and error-prone. Automated smell detection tools can lower long-term development costs and increase the chances for success [28] by helping to make the code more maintainable.

978-1-4673-5739-5/13/$31.00 © 2013 IEEE

Current work on web application code smell detection is scarce [20] and tools [3], [4], [6], [20] available to web developers to maintain their code are mainly static analyzers and thus limited in their capabilities.

In this paper, we propose a list of code smells for JavaScript-based applications. In total, we consider 13 code smells: 7 are existing well-known smells adapted to JavaScript, and 6 are specific JavaScript code smell types, collected from various JavaScript development resources. We present an automated technique, called JSNOSE, to detect these code smells. Our approach uses a metric-based algorithm, and combines static with dynamic analysis to detect these smells in JavaScript code.

Our work makes the following main contributions:

- We propose a list of JavaScript code smells, collected from various web development resources;
- We present an automated metric-based approach to detect JavaScript code smells;
- We implement our approach in a tool called JSNose, which is freely available;
- We evaluate the effectiveness of our technique in detecting code smells in JavaScript applications;
- We empirically investigate 11 web applications using JSNOSE to find out which smells are more prevalent.

Our results indicate that amongst the smells detected by JSNOSE, lazy object, long method/function, closure smells, coupling between JavaScript, HTML, and CSS, and excessive global variables, are the most prevalent code smells. Further, our study indicates that there exists a strong and significant positive correlation between the types of smells and lines of code, number of functions, number of JavaScript files, and cyclomatic complexity.

II. MOTIVATION AND CHALLENGES

Although JavaScript is increasingly used to develop modern web applications, there is still lack of tool support targeting code quality and maintenance, in particular for automated code smell detection and refactoring.

JavaScript is a dynamic weakly typed and prototype-based scripting language, with first-class functions. Prototype-based programming is a class-free style of object-oriented programming, in which objects can inherit properties from other

objects directly. In JavaScript, prototypes can be redefined at runtime, and immediately affect all the referring objects.

The detection process for many of the traditional code smells [9], [12] in object-oriented languages is dependent on identifying objects, classes, and functions in the code. Unlike most object-oriented languages such as Java or C++, identification of such key language items is not straightforward in JavaScript code. Bellow we explain the major challenges in identifying objects and functions in JavaScript.

JavaScript has a very flexible model of objects and functions. Object properties and their values can be created, changed, or deleted at runtime and accessed via first-class functions. For instance, in the following piece of code a call to the function `foo()` will dynamically create a property *prop* for the object *obj* if *prop* does not already exist.

```
function foo(obj, prop, value){
    obj.prop = value;
}
```

Due to such dynamism, the set of all available properties of an object is not easily retrievable through static analysis of the code alone. Empirical studies [24] reveal that most dynamic features in JavaScript are frequently used by developers and cannot be disregarded in code analysis processes.

Furthermore, functions in JavaScript are first-class values. They can (1) be objects themselves, (2) contain properties and nested function closures, (3) be assigned dynamically to other objects, (4) be stored in variables, objects, and arrays, (5) be passed as arguments to other functions, and (6) be returned from functions. JavaScript also allows the creation (through `eval()`) and execution of new code at runtime, which again makes static analysis techniques insufficient.

Manual analysis and detection of code smells in JavaScript is time consuming, tedious, and error-prone in large code bases. Therefore, automated techniques are needed to support web developers in maintaining their code. Given the challenging characteristics of JavaScript, our goal in this work is to propose a technique that can handle the highly dynamic nature of the language to detect potential code smells effectively.

III. Related Work

Fowler and Beck [9] proposed 22 code smells in object-oriented languages and associated each of them with a possible refactoring. Although code smells in object-oriented languages have been extensively studied in the past, current work on smell detection for JavaScript code is scarce [20]. In this work we study a list of code smells in JavaScript, and propose an automated detection technique. The list of proposed JavaScript code smells in this paper is based on a study of various discussions in online development forums and JavaScript books [19], [20], [21], [22], [10].

Many tools and techniques have been proposed to detect code smells automatically in Java and C++ such as Checkstyle [2], Decor [17], and JDeodorant [27]. A common heuristic-based approach to code smell detection is the use of code metrics and user defined thresholds [17], [7], [18], [25], [13]. Similarly we adopt a metric-based smell detection strategy. Our approach is different from such techniques in the sense that due to the dynamic nature of JavaScript, we propose a code smell technique that combines static with dynamic analysis.

Cilla [15] is a tool that similar to our work applies dynamic analysis but for detecting unused CSS code in relation to dynamically mutated DOM elements. A case study conducted with Cilla revealed that over 60% of CSS rules are unused in real-world deployed web applications, and eliminating them could vastly improve the size and maintainability of the code.

A more closely related tool is WebScent [20], which detects client-side smells that exist in embedded code within scattered server-side code. Such smells can not be easily detected until the client-side code is generated. After detecting smells in the generated client-side code, WebScent locates the smells in the corresponding location in the server-side code. WebScent primarily identifies mixing of HTML, CSS, and JavaScript, duplicate code in JavaScript, and HTML syntax errors. Our tool, JSNose, can similarly identify JavaScript code smells generated via server-side code. However, we propose and target a larger set of JavaScript code smells and unlike the manual navigation of the application in WebScent, we apply automated dynamic exploration using a crawler. Another advantage of JSNose is that it can infer dynamic creation/change of objects, properties, and functions at runtime, which WebScent does not support.

A number of industrial tools exist that aim at assisting web developers with maintaining their code. For instance, WARI [6] examines dependencies between JavaScript functions, CSS styles, HTML tags and images. The goal is to statically find unused images as well as unused and duplicated JavaScript functions and CSS styles. Because of the dynamic nature of JavaScript, WARI cannot guarantee the correctness of the results. JSLint [4] is a static code analysis tool written in JavaScript that validates JavaScript code against a set of good coding practices. The code inspection tends to focus on improving code quality from a technical perspective. The Google Closure Compiler [3] is a JavaScript optimizer that rewrites JavaScript code to make it faster and more compact. It helps to reduce the size of JavaScript code by removing comments and unreachable code.

IV. JavaScript Code Smells

In this section, we propose a list of code smells for JavaScript-based applications. Of course a list of code smells can never be complete as the domain and projects that the code is used in may vary. Moreover, code smells are generally subjective and imprecise, i.e., they are based on opinions and experiences [28]. To mitigate this subjective nature of code smells, we have collected these smells by studying various online development resources [5], [10], [19], [21], [22], [26] and books [8], [31], [32] that discuss bad JavaScript coding patterns.

In total, we consider 13 code smells in JavaScript. Although JavaScript has its own specific code smells, most of the generic code smells for object-oriented languages [9], [12]

can be adapted to JavaScript as well. Since JavaScript is a class-free language and objects are defined directly, we use the notion of "object" instead of "class" for these generic smells. The generic smells include the following 7: *Empty catch blocks* (poor understanding of logic in the try), *Large object* (too many responsibilities), *Lazy object* (does too little), *Long functions* (inadequate decomposition), *Long parameter list* (need for an object), *Switch statements* (duplicated code and high complexity), and *Unused/dead code* (never executed or unreachable code).

In addition to the generic smells, we propose 6 types of JavaScript code smells in this section as follows.

A. Closure Smells

In JavaScript, it is possible to declare nested functions, called `closures`. Closures make it possible to emulate object-oriented notions such as `public`, `private`, and `privileged` members. Inner functions have access to the parameters and variables — except for `this` and `argument` variables — of the functions they are nested in, even after the outer function has returned [8]. We consider four smells related to the concept of function closures in JavaScript.

Long scope chaining. Functions can be multiply-nested, thus closures can have multiple scopes. This is called "scope chaining" [5], where inner functions have access to the scope of the functions containing them. An example is the following code:

```
function foo(x) {
  var tmp = 3;
  function bar(y) {
      ++tmp;
      function baz(z) {
        document.write(x + y + z + tmp);
      }
      baz(3);
  }
  bar(10);
}
foo(2);  // writes 19 i.e., 2+10+3+4
```

This nested function style of programming is useful to emulate privacy, however, using too many levels of nested closures over-complicates the code, making it hard to comprehend and maintain. Moreover, identifier resolution performance is directly related to the number of objects to search in the scope chain [31]. The farther up in the scope chain an identifier exists, the longer the search goes on and the longer time it takes to access that variable.

Closures in loops. Inner functions have access to the actual variables of their outer functions and not their copies. Therefore, creating functions within a loop can cause confusion and be wasteful computationally [8]. Consider the following example:

```
var addTheHandler = function (nodes) {
    for (i = 0; i < nodes.length; i++) {
        nodes[i].onclick = function (e) {
            document.write(i);
        };
    }
};
addTheHandler(document.getElementsByTagName("div"));
```

Assume that there are three `div` DOM elements present. If the developer's actual intention is to display the ordinal of the `div` nodes when a node is clicked, then the result will not be what she expected as the length of nodes, 3, will be returned instead of the node's ordinal position. In this example, the value of `i` in the `document.write` function is assigned when the `for` loop is finished and the inner anonymous function is created. Therefore, the variable `i` has the value of `nodes.length`, which is 3 in this case. To avoid this confusion and potential mistake, the developer can use a helper function outside of the loop that will deliver a function binding to the current local value of `i`.

Variable name conflict in closures. When two variables in the scopes of a closure have the same name, there is a name conflict. In case of such conflicts, the inner scope takes precedence. Consider the following example [5]:

```
function outside() {
   var a = 10;
   function inside(a) {
      return a;
   }
   return inside;
}
result = outside()(20);  \\ result: 20
```

In this example, there is a name conflict between the variable `a` in `outside()` and the function parameter `a` in `inside()` that takes precedence. We consider this a code smell as it makes it difficult to comprehend the actual intended value assignment. Due to scope chain precedence, the value of `result` is now 20. However, it is not evident from the code whether this is the intended result (or perhaps 10).

Moreover, dynamic typing in JavaScript makes it possible to reuse the *same* variable for *different* types at runtime. Similar to the variable name conflict issue, this style of programming reduces readability and in turn maintainability. Thus, declaring a new variable with a dedicated unique name is the recommended refactoring. This issue is not restricted to closures, or to nested functions.

Accessing the `this` reference in closures. Due to the design of JavaScript language, when an inner function in a closure is invoked, `this` becomes bounded to the global object and not to the `this` variable of the outer function [8]. We consider the usage of `this` in closures a code smell as it is a potential symptom for mistakes. As a refactoring workaround, the developer can assign the value of the `this` variable of the outer function to a new variable `that` and then use `that` in the inner function [8].

B. Coupling between JavaScript, HTML, and CSS

In web applications, HTML is meant for presenting content and structure, CSS for styling, and JavaScript for functional behaviour. Keeping these three entities separate is a well-known programming practice, known as *separation of concerns*. Unfortunately, web developers often mix JavaScript code with markup and styling code [20], which adversely influences program comprehension, maintenance and debugging efforts in web applications. We categorize the tight coupling

of JavaScript with HTML and CSS code into the following three code smells types:

JavaScript in HTML. One common way to register an event listener in web applications is via inline assignment in the HTML code. We consider this inline assignment of event handlers a code smell as it tightly couples the HTML code to the JavaScript code. An example of such a coupling is shown below:

```
<button onclick="foo();" id="myBtn"/>
```

This smell can be refactored by removing the `onclick` attribute from the button in HTML and using the `addEventListener` function of DOM Level 2 [29] to assign the event handler through JavaScript:

```
<button id="myBtn"/>

function foo() {
    // code
}
var btn = document.getElementById("myBtn");
btn.addEventListener("click", foo, false);
```

This could be further refactored using the jQuery library as `$("#myBtn").on("click", foo);`.

Note that JavaScript code within the `<script>` tag in HTML code can be seen as a code smell [20]. We do not consider this a code smell as it does not affect comprehension nor maintainability, although separating the code to a JavaScript file is preferable.

HTML in JavaScript. Extensive DOM API calls and embedded HTML strings in JavaScript complicate debugging and software evolution. In addition, editing markup is believed to be less error prone than editing JavaScript code [32]. The following code is an example of embedded HTML in JavaScript [10]:

```
// add book to the list
var book = doc.createElement("li");
var title = doc.createElement("strong");
titletext = doc.createTextNode(name);
title.appendChild(titletext);
var cover = doc.createElement("img");
cover.src = url;
book.appendChild(cover);
book.appendChild(title);
bookList.appendChild(book);
```

To refactor this code smell, we can move the HTML code to a template (`book_tpl.html`):

```
<li><strong>TITLE</strong><img src="COVER"/></li>
```

The JavaScript code would then be refactored as:

```
var tpl = loadTemplate("book_tpl.html");
var book = tpl.substitute({TITLE: name,  COVER: url});
bookList.appendChild(book);
```

Another example this smell is using long strings of HTML in jQuery function calls [22]:

```
$('\#news')
   .append('<div class="gall"><a href="javascript:void(0)↩
        ">Linky</a></div>')
   .append('<button onclick="app.doStuff()">Button</↩
       button>');
```

CSS in JavaScript. Setting the presentation style of DOM elements by assigning their `style` properties in JavaScript is a code smell [10]. Keeping styling code inside JavaScript is asking for maintenance problems. Consider the following example:

```
div.onclick = function(e) {
   var clicked = this;
   clicked.style.border = "1px solid blue";
}
```

The best way to change the style of an element in JavaScript is by manipulating CSS classes properly defined in CSS files [10], [32]. The above code smell can be refactored as follows:

```
\\ CSS file:
.selected{border: 1px solid blue;}
\\ JavaScript:
div.onclick = function(e) {
   this.setAttribute("class","selected");
}
```

C. Excessive Global Variables

Global variables are accessible from anywhere in JavaScript code, even when defined in different files loaded on the same page. As such, naming conflicts between global variables in different JavaScript source files is common, which affects program dependability and correctness. The higher the number of global variables in the code, the more dependent existing modules are likely to be; and dependency increases error-proneness, and maintainability efforts [21]. Therefore, we see the excessive use of global variables as a code smell in JavaScript. One way to mitigate this issue is to create a single global object for the whole application that contains all the global variables as its properties [8]. Grouping related global variables into objects is another remedy.

D. Long Message Chain

Long chaining of functions with the dot operator can result in complex control flows that are hard to comprehend. This style of programming happens frequently when using the jQuery library. One extreme example is shown bellow [22]:

```
$('a').addClass('reg-link').find('span').addClass('inner'↩
    ).end().find('div').mouseenter(mouseEnterHandler).↩
    mouseleave(mouseLeaveHandler).end().explode();
```

Long chains are unreadable specially when a large amount of DOM traversing is taking place [22].

Another instance of this code smell is too much cascading. Similar to object-oriented languages such as Java, in JavaScript many methods calls can be cascaded on the same object sequentially within a single statement. This is possible when the methods return the `this` object. Cascading can help to produce expressive interfaces that perform much work at once. However, the code written this way tends to be harder to follow and maintain. The following example is borrowed from [8]:

```
getElement('myBoxDiv').move(350, 150).width(100).height↩
    (100).color('red').border('10px outset').padding('4px↩
    ').appendText("Please stand by").on('mousedown', ↩
    function (m) {
    this.startDrag(m, this.getNinth(m));}).on('mousemove'↩
        , 'drag').on('mouseup', 'stopDrag').tip("This box↩
         is resizeable");
```

A possible refactoring to shorten the message chain is to break the chain into more general methods/properties for that object which incorporate longer chains.

E. Nested Callback

A callback is a function passed as an argument to another (parent) function. Callbacks are executed after the parent function has completed its execution. Callback functions are typically used in asynchronous calls such as timeouts and XMLHttpRequests (XHRs). Using excessive callbacks, however, can result in hard to read and maintain code due to their nested anonymous (and usually asynchronous) nature. An example of a nested callback is given below [26]:

```
setTimeout(function () {
    xhr("/greeting/", function (greeting) {
        xhr("/who/?greeting=" + greeting, function (who) ↩
            {
            document.write(greeting + " " + who);
        });
    });
}, 1000);
```

A possible refactoring to resolve unreadable nested callbacks is to split the functions and pass a reference to another function [1]. The above code can be rewritten as bellow:

```
setTimeout(foo,1000);
function foo() {
 xhr("/greeting/", bar);
}
function bar(greeting) {
 xhr("/who/?greeting=" + greeting, baz);
}
function baz(who) {
 document.write(greeting + " " + who);
}
```

F. Refused Bequest

JavaScript is a class-free prototypal inheritance language, i.e., an object can inherit properties from another object, called a prototype object. A JavaScript object that does not use/override many of the properties it inherits from its prototype object is an instance of a refused bequest [9] soft code smell. In the following example, the `student` object inherits from its prototype parent `person`. However, `student` only uses one of the five properties inherited from `person`, namely `fname`.

```
var person={fname:"John", lname:"Smith", gender:"male", ↩
     age:28, location:"Vancouver"};
var student = Object.create(person);
...
student.university = "UBC";
document.write(student.fname + " studies at " + student.↩
     university);
```

A simple refactoring, similar to the push down field/method proposed by Fowler [9], could be to eliminate the inheritance altogether and add the required property (`fname`) of the prototype to the object that refused the bequest.

V. Smell Detection Mechanism

In this section, we present our JavaScript code smell detection mechanism, which is capable of detecting the code smells discussed in the previous section.

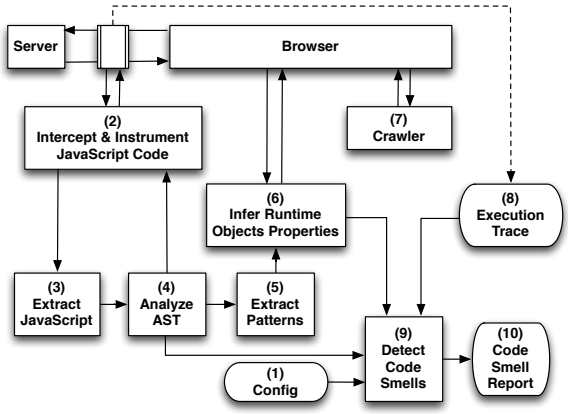

Fig. 1. Processing view of JSNose, our JavaScript code smell detector.

A common heuristic-based approach to detect code smells is the use of source code metrics and thresholds [7], [13], [17], [18], [25]. In this work, we adopt a similar metric-based approach to identify smelly sections of JavaScript code.

In order to calculate the metrics, we first need to extract objects, functions, and their relationships from the source code. Due to the dynamic nature of JavaScript, static code analysis alone will not suffice, as discussed in Section II. Therefore, in addition to static code analysis, we also employ dynamic analysis to monitor and infer information about objects and their relations at runtime.

Figure 1 depicts an overview of our approach. At a high level, (1) the configuration, containing the defined metrics and thresholds, is fed into the code smell detector. We automatically (2) intercept the JavaScript code of a given web application, by setting up a proxy between the server and the browser, (3) extract JavaScript code from all .js and HTML files, (4) parse the source code into an Abstract Syntax Tree (AST) and analyze it by traversing the tree. During the AST traversal, the analyzer visits all program entities, objects, properties, functions, and code blocks, and stores their structure and relations. At the same time, we (2) instrument the code to monitor statement coverage, which is used for unused/dead code smell detection. Next, we (7) navigate the instrumented application in the browser to produce an execution trace, through an automated dynamic crawler, and (8) collect and use execution traces to calculate code coverage. We (5) extract patterns from the AST such as names of objects and functions, and (6) infer JavaScript objects, their types, and properties dynamically by querying the browser at runtime. Finally, (9) based on all the static and dynamic data collected, we detect code smells (10) using the metrics.

A. Metrics and Criteria Used for Smell Detection

Table I presents the metrics and criteria we use in our approach to detect code smells in JavaScript applications. Some of these metrics and their corresponding thresholds have been proposed and used for detecting code smells in object-oriented languages [7], [13], [17], [18], [25]. In addition, we

TABLE I
METRIC-BASED CRITERIA FOR JAVASCRIPT CODE SMELL DETECTION.

Code smell	Level	Detection method	Detection criteria	Metric
Closure smell	Function	Static & Dynamic	$LSC > 3$	LSC: Length of scope chain
Coupling JS/HTML/CSS	File	Static & Dynamic	$JSC > 1$	JSC: JavaScript coupling instance
Empty catch	Code block	Static	$LOC(catchBlock) = 0$	LOC: Lines of code
Excessive global variables	Code block	Static & Dynamic	$GLB > 10$	GLB: Number of global variables
Large object	Object	Static & Dynamic	[30]: $LOC(obj) > 750$ or $NOP > 20$	NOP: Number of properties
Lazy object	Object	Static & Dynamic	$NOP < 3$	NOP: Number of properties
Long message chain	Code block	Static	$LMC > 3$	LMC: Length of message chain
Long method/function	Function	Static & Dynamic	[13], [30]: $MLOC > 50$	MLOC: Method lines of code
Long parameter list	Function	Static & Dynamic	[30]: $PAR > 5$	PAR: Number of parameters
Nested callback	Function	Static & Dynamic	$CBD > 3$	CBD: Callback depth
Refused bequest	Object	Static & Dynamic	[13]: $BUR < \frac{1}{3}$ and $NOP > 2$	BUR: Base-object usage ratio
Switch statement	Code block	Static	$NOC > 3$	NOC: Number of cases
Unused/dead code	Code block	Static & Dynamic	$EXEC = 0$ or $RCH = 0$	EXEC: Execution count RCH: Reachability of code

propose new metrics and criteria to capture the characteristics of JavaScript code smells discussed in Section IV.

Closure smell. We identify long scope chaining and accessing `this` in closures. If the length of scope chain (LSC) is greater than 3, or if `this` is used in an inner function closure, we report it as a closure smell instance.

Coupling JS/HTML/CSS. We count the number of occurrences of JavaScript within HTML tags, and CSS in JavaScript as described in Section IV-B. Our tool reports all such JavaScript coupling instances as code smell.

Empty catch. Detecting empty catches is straightforward in that the number of lines of code (LOC) in the catch block should be zero.

Excessive global variables. We extract global variables in JavaScript, which can be defined in three ways: (1) using a `var` statement outside of any function, such as `var x = value;`, (2) adding a property to the `window` global object, i.e., the container of all global variables, such as `window.foo = value;`, and (3) using a variable without declaring it by `var`. If the number of global variables (GLB) exceeds 10, we consider it as a code smell.

Large/Lazy object. An object that is doing too much or not doing enough work should be refactored. Large objects may be restructured or broken into smaller objects, and lazy objects maybe collapsed or combined into other classes. If an object's lines of code is greater than 750 or the number of its methods is greater than 20, it is identified as a large object [30]. We consider an object lazy, if the number of its properties (NOP) is less than 3.

Long message chain. If the length of a message chain (LMC), i.e., the number of items being chained by dots as explained in Section IV-D, in a statement is greater than 3, we consider it a long message chain and report it as a smell.

Long method/function. A method with more than 50 lines of code (MLOC) is identified as a long method smell [13], [30].

Long parameter list. We consider a parameter list long when the number of parameters (PAR) exceeds 5 [30].

Nested callback. We identify nested functions that pass a function type as an argument. If the callback depth (CBD) exceeds 3, we report it as a smell.

Refused bequest. If an object uses or specializes less than a third of its parent prototype, i.e., base-object usage ratio (BUR) is less than $\frac{1}{3}$, it is considered as refused parent bequest [13]. Further, the number of methods and the cyclomatic complexity of the child object should be above average since simple and small objects may unintentionally refuse a bequest. In our work, we slightly change this criteria to the constraint of NOP>2, i.e., not to be a lazy small object.

Switch statement. The problem with switch statements is duplicated code. Typically, similar switch statements are scattered throughout a program. If one adds or removes a clause in one switch, she often has to find and repair the others too [9], [12]. When the number of switch cases (NOC) is more than three, it is considered as a code smell. This can also be applied to `if-then-else` statements with more than three branches.

Unused/dead code. Unused/dead code has negative effects on maintainability as it makes the code unnecessarily more difficult to understand [21], [14]. Unlike languages such as Java, due to the dynamic nature of JavaScript it is quite challenging to reason about dead JavaScript code statically. Hence, if the execution count (EXEC) of an statement remains 0 after executing the web application, we report it as a candidate unused/dead code. Reachability of code (RCH) is another metric we use to identify unreachable code.

B. Combining Static and Dynamic Analysis

Algorithm 1 presents our smell detection method. The algorithm is generic in the sense that the metric-based static and dynamic smell detection procedures can be defined and used according to any smell detection criteria. Given a JavaScript application A, a maximum crawling time t, and a set of code smell criteria τ, the algorithm generates a set of code smells CS.

The algorithm starts by looking for inline JavaScript code embedded in HTML (line 3). All JavaScript code is then

Algorithm 1: JavaScript Code Smell Detection

```
   input : A JavaScript application A, the maximum exploration time
           t, the set of smell metric criteria τ
   output: The list of JavaScript code smells CS
1  CS ← ∅
   Procedure EXPLORE() begin
2      while TIMELEFT(t) do
3          CS ← CS ∪ DETECTINLINEJSINHTML(τ)
4          code ← EXTRACTJAVASCRIPT(A)
5          AST ← PARSTOAST(code)
6          VISITNODE(AST.root)
7          ASTinst ← INSTRUMENT(AST)
8          INJECTJAVASCRIPTCODE(A, ASTinst)
9          C ← EXTRACTCLICKABLES(A)
10         for c ∈ C do
11             dom ← browser.GETDOM()
12             robot.FIREEVENT(c)
13             new_dom ← browser.GETDOM()
14             CS ← CS ∪ DETECTDYNAMICALLY(τ)
15             if dom.HASCHANGED(new_dom) then
16                 EXPLORE(A)
17     CS ← CS ∪ DETECTUNUSEDCODE()
18     return CS
   Procedure VISITNODE(ASTNode) begin
19     CS ← CS ∪ DETECTSTATICALLY(node, τ)
20     for node ∈ ASTNode.getChildren() do
21         VISITNODE(node)
```

extracted from JavaScript files and HTML `<script>` tags (line 4). An AST of the extracted code is then generated using a parser (line 5). This AST is traversed recursively (lines 6, 19-21) to detect code smells using a static analyzer. Next the AST is instrumented (line 7) and transformed back to the corresponding JavaScript source code and passed to the browser (lines 8). The crawler then navigates (line 9-16) the application, and potential code smells are explored dynamically (line 9 14). After the termination of the exploration process, unused code is identified based on the execution trace and added to the list of code smells (line 17), and the resulting list of smells is returned (line 18).

Next, we present the relevant static and dynamic smell detection processes in detail.

Static Analysis. The static code analysis (Line 19) involves analyzing the AST by traversing the tree. During this step, we extract CSS style usage, objects, properties, inheritance relations, functions, and code blocks to calculate the smell metrics. If the calculated metrics violate the given criteria (τ), the smell is returned.

There are different ways to create objects in JavaScript. In this work, we only consider two main standard forms of using object literals, namely, through (1) the `new` keyword, and (2) `Object.create()`. To detect the prototype of an object, we consider both the non-standard form of using the `__proto__` property assignment, and the more general constructor functions through `Object.create()`.

In order to detect unreachable code, we search the AST nodes for `return`, `break`, `continue`, and `throw` statements. Whatever a statement is found right after these statements that is on the same node level in the AST, we mark it as potential unreachable code.

Dynamic Analysis. Dynamic analysis (Line 14) is performed for two reasons:

1) To calculate many of the metrics in Table I, we need to monitor the creation/update of functions, objects, and their properties at runtime. To that end, a combination of static and dynamic analysis should be applied. The dynamic analysis is performed by executing a piece of JavaScript code in the browser, which enables retrieving a list of all global variables, objects, and functions (own properties of the `window` object) and dynamically detecting prototypes of objects (using `getPrototypeOf()` on each object). However, local objects in functions are not accessible via JavaScript code execution in the global scope. Therefore, we use static analysis and extract the required information from the parsed AST. The objects, functions, and properties information gathered this way is then fed to the smell detector process.
2) To detect unused/dead code we need to collect execution traces for measuring code coverage. Therefore, we instrument the code and record which parts of it are invoked by exploring the application through automated crawling. However, this dynamic analysis can give false positives for non-executed, but reachable code. This is a limitation of any dynamic analysis approach since there is no guarantee of completeness (such as code coverage).

Note that our approach merely reports candidate code smells and the decision will always be upon developers whether or not to refactor the code smells.

C. Implementation

We have implemented our approach in a tool called JSNOSE, which is publicly available.[1] JSNOSE operates automatically, does not modify the web browser, is independent of the server technology, and requires no extra effort from the user. We use the WebScarab proxy to intercept the JavaScript/HTML code. To parse the JavaScript code to an AST and instrument the code, we use Mozilla Rhino.[2] To automatically explore and dynamically crawl the web application, we use CRAWLJAX [16]. The output of JSNOSE is a text file that lists all detected JavaScript code smells with their corresponding line numbers in a JavaScript file or an HTML page.

VI. EMPIRICAL EVALUATION

We have conducted an empirical study to evaluate the effectiveness and real-world relevance of JSNOSE. Our study is designed to address the following research questions:

RQ1: How effective is JSNOSE in detecting JavaScript code smells?

RQ2: Which code smells are more prevalent in web applications?

[1] http://salt.ece.ubc.ca/content/jsnose/

[2] https://github.com/mozilla/rhino/

TABLE II
EXPERIMENTAL OBJECTS.

ID	Name	#JS files	JS LOC	#Functions	Average CC	Average MI	Description	Resource
1	PeriodicTable	1	71	9	12	116	An AJAX-based periodic table of the elements	http://code.jalenack.com/periodic/
2	CollegeVis	1	177	30	11	119	A JavaScript-based visualization tool	https://github.com/nerdyworm/collegesvis
3	ChessGame	2	198	15	102	105	A JavaScript-based simple game	p4wn.sourceforge.net
4	Symbolistic	1	203	20	28	109	A JavaScript-based simple game	http://10k.aneventapart.com/2/Uploads/652
5	Tunnel	0	234	32	29	116	A JavaScript-based simple game	http://arcade.christianmontoya.com/tunnel
6	GhostBusters	0	278	26	45	97	A JavaScript-based simple game	http://10k.aneventapart.com/2/Uploads/657
7	TuduList	4	782	89	106	94	An AJAX-based todo lists manager in J2EE and MySQL	julien-dubois.com/tudu-lists/
8	FractalViewer	8	1245	125	35	116	A JavaScript-based fractal zoomer	http://onecm.com/projects/canopy
9	PhotoGallery	5	1535	102	53	102	An AJAX-based photo gallery in PHP without MySQL	sourceforge.net/projects/rephormer
10	TinySiteCMS	13	2496	462	54	115	An AJAX-based CMS in PHP without MySQL	tinysitecms.com
11	TinyMCE	174	26908	4455	67	101	A JavaScript-based WYSIWYG editor	tinymce.com

RQ3: Is there a correlation between JavaScript code smells and source code metrics?

Our experimental data along with the implementation of JSNOSE are available for download.[1]

A. Experimental Objects

We selected 11 web applications that make extensive use of client-side JavaScript, and fall under different application domains. The experimental objects along with their source code metrics are shown in Table II. In the calculation of these source code metrics, we included inline HTML JavaScript code, and excluded blank lines, comments, and common JavaScript libraries such as jQuery, DWR, Scriptaculous, Prototype, and google-analytics. Note that we also exclude these libraries in the instrumentation step. We use CLOC[3] to count the JavaScript lines of code (JS LOC). Number of functions (including anonymous functions), cyclomatic complexity (CC), and maintainability index (MI) are all calculated using complexityReport.js.[4] The reported CC and MI are across all JavaScript functions in each application.

B. Experimental Setup

We confine the dynamic crawling time for each application to 10 minutes, which is acceptable in a maintenance environment. Of course, the more time we designate for exploring the application, the higher statement coverage we may get and thus more accurate the detection of unused/dead code. For the crawling configuration, we set no limits on the crawling depth nor the maximum number of DOM states to be discovered. The criteria for code smell metrics are configured according to those presented in Table I.

To evaluate the effectiveness of JSNOSE (RQ1), we validate the produced results by JSNOSE against manual code inspection. Similar to [17], we measure precision and recall as follows:

Precision is the rate of true smells identified among the detected smells: $\frac{TP}{TP+FP}$

Recall is the rate of true smells identified among the existing smells: $\frac{TP}{TP+FN}$

where TP (true positives), FP (false positives), and FN (false negatives) respectively represent the number of correctly detected smells, falsely detected smells, and missed smells. To count TP, FP, and FN in a timely fashion while preserving accuracy, we only consider the first 9 applications since the last 2 applications have relatively larger code bases. In our manual validation process, we also consider runtime created/modified objects and functions that are inferred during JSNOSE dynamic analysis. It is worth mentioning that this manual process is a labour intensive task, which took approximately 6.5 hours for the 9 applications.

Note that the precision-recall values for detecting unused/dead code smell is calculated considering only "unreachable" code, which is code after an unconditional `return` statement. This is due to the fact that the accuracy of dead code detection depends on the running time and dynamic exploration strategy.

To measure the prevalence of JavaScript code smells (RQ2), we ran JSNOSE on all the 11 web applications and counted each smell instance.

To evaluate the correlation between the number of smells and application source code metrics (RQ3), we use R[5] to calculate the non-parametric Spearman correlation coefficients as well as the p-values. The Spearman correlation coefficient does not require the data to be normally distributed [11].

C. Results

Effectiveness (RQ1). We report the precision and recall in the first 5 rows of Table III. The reported TP_{total}, FP_{total}, and FN_{total}, are the sum of TP, FP, and FN values for the first 9 applications. Our results show that JSNOSE has an overall precision of 93% and an average recall of 98% in detecting the JavaScript code smells, which points to its effectiveness.

We observed that most false positives detected are related to large/lazy objects and refused bequest, which are primitive variables, object properties, and methods in jQuery. This is due to the diverse coding styles and different techniques in object manipulations in JavaScript, such as creating and initializing arrays of objects. There were a few false negatives

[3]http://cloc.sourceforge.net

[4]https://npmjs.org/package/complexity-report/

[5]http://www.r-project.org

TABLE III
PRECISION-RECALL ANALYSIS (BASED ON THE FIRST 9 APPLICATIONS), AND DETECTED CODE SMELL STATISTICS (FOR ALL 11 APPLICATIONS).

	S1. Closure smells	S2. Coupling JS/HTML/CSS	S3. Empty catch	Number of global variables	S4. Excessive global variables	S5. Large object	S6. Lazy object	S7. Long message chain	S8. Long method/function	S9. Long parameter list	S10. Nested callback	S11. Refused bequest	S12. Switch statement	S13. Unreachable code	S13. Unused/dead code	Number of smell instances	Number of types of smells
TP_{total}	19	171	16	200	-	14	391	87	25	12	1	13	10	0	n/a	959	n/a
FP_{total}	0	0	0	0	-	4	73	0	0	0	0	6	0	0	n/a	83	n/a
FN_{total}	6	0	0	0	-	1	2	8	0	0	0	0	0	0	n/a	17	n/a
$Precision_{total}$	100%	100%	100%	100%	-	78%	85%	100%	100%	100%	100%	68%	100%	n/a	n/a	92%	n/a
$Recall_{total}$	76%	100%	100%	100%	-	94%	99%	92%	100%	100%	100%	100%	100%	n/a	n/a	98%	n/a
PeriodicTable	1	2	0	6	-	4	10	0	0	0	0	0	0	0	28%	23	4
CollegeVis	1	0	0	17	+	0	32	0	2	0	1	0	0	0	22%	53	5
ChessGame	0	7	0	**39**	+	3	9	4	0	2	0	0	0	0	36%	64	6
Symbolistic	0	0	0	4	-	0	17	0	1	0	0	1	0	0	20%	23	3
Tunnel	**9**	0	0	15	+	0	28	0	2	0	0	0	0	0	44%	54	4
GhostBusters	2	0	0	4	-	0	38	0	2	3	0	0	0	0	45%	49	4
TuduList	**6**	**47**	0	**45**	+	**6**	**138**	**78**	**12**	2	0	2	**7**	0	**65%**	343	10
FractalViewer	0	16	0	**40**	+	**7**	**117**	4	5	5	0	**16**	2	0	36%	212	9
PhotoGallery	0	**99**	**16**	30	+	0	73	1	1	0	0	0	1	0	**64%**	221	7
TinySiteCMS	2	7	0	**82**	+	3	13	4	3	**58**	0	0	0	0	22%	172	8
TinyMCE	3	3	1	4	-	5	23	4	0	2	1	3	3	0	**63%**	52	10
Average	2.2	16.5	1.5	26	+	2.5	45.2	8.6	2.6	6.5	0.2	2	1.2	0	40%	115	6.4
#Smelly apps	7	7	2	n/a	7	6	11	6	8	6	2	4	4	0	n/a	n/a	n/a
%Smelly apps	**64%**	**64%**	18%	n/a	**64%**	55%	**100%**	55%	**73%**	55%	18%	36%	36%	0%	n/a	n/a	n/a

in closure smells and long message chain, which are due to the permissive nature of jQuery syntax, complex chains of methods, array elements, as well as jQuery objects created via `$()` function.

Code smell prevalence (RQ2). Table III shows the frequency of code smells in each of the experimental objects. The results show that among the JavaScript code smells detected by JSNOSE, lazy object, long method/function, closure smells, coupling JS/HTML/CSS, and excessive global variables, are the most prevalent smells (appeared in 100%-64% of the experimental objects).

Tunnel and TuduList use many instances of `this` in closures. Major coupling smells in TuduList and PhotoGallary are with the use of CSS in JavaScript. Refused bequest are most observed in FractalViewer in objects inheriting from geometry objects. The high percentage of unused/dead code reported for TuduList, PhotoGallary, and TinyMCE is in fact not due to dead code per se, but is mainly related to the existence of admin pages and parts of the code which require precise data inputs that were not provided during the crawling process. On the other hand, TinyMCE has a huge number of possible actions and features that could not be exercised in the designated time of 10 minutes.

Correlations (RQ3). Table IV shows the Spearman correlation coefficients between the source code metrics and the total number of smell instances/types. The results show that there exists a strong and significant positive correlation between the *types of smells* and LOC, number of functions, number of JavaScript files, and cyclomatic complexity. A weak correlation is also observed between the *number of smell instances* and the aforementioned source code metrics. Surprisingly, for the maintainability index (MI) we do not see any significant correlation with the types or number of code smells.

TABLE IV
SPEARMAN CORRELATION COEFFICIENTS BETWEEN NUMBER OF CODE SMELLS AND CODE QUALITY METRICS.

Metric	Total number of smell instances	Total number of types of smells
Lines of code	$(r = 0.53, p = 0.05)$	$(r = 0.70, p = 0.01)$
# Functions	$(r = 0.57, p = 0.03)$	$(r = 0.76, p = 0.00)$
# JavaScript files	$(r = 0.53, p = 0.05)$	$(r = 0.85, p = 0.00)$
Cyclomatic complexity	$(r = 0.63, p = 0.02)$	$(r = 0.70, p = 0.01)$
Maintainability index	$(r = -0.25, p = 0.74)$	$(r = -0.35, p = 0.86)$

D. Discussion

Here, we discuss some of the limitations and threats to validity of our results.

Implementation Limitations. The current implementation of JSNOSE is not able to detect all various ways of object creation in JavaScript. Also it does not deal with various syntax styles of frameworks such as jQuery. For the dynamic analysis part, JSNOSE is dependent on the crawling strategy and execution time, which may affect the accuracy if certain JavaScript files are never loaded in the browser during the execution since the state space of web applications is typically huge. Since JSNOSE is using Rhino to parse the JavaScript code and generate the AST, if there exists a syntax error in a JavaScript file, the code in that file will not be parsed to an AST and thus any potential code smells within that file will

be missed. Note that these are all implementation issues and not related to the design of our approach.

Threats to Validity. A threat to the external validity of our evaluation is with regard to the generalization of the results to other web applications. We acknowledge that more web applications should be evaluated to support the conclusions. To mitigate this threat we selected our experimental objects from different application domains, which exhibit variations in design, size, and functionality.

One threat to the internal validity of our study is related to the metrics and criteria we proposed in Table I. However, we believe these metrics can effectively identify code smells described in Section IV. The designated 10 minutes time for crawling could also be increased to get more accurate results, however, we believe that in most maintenance environments this is acceptable considering frequent code releases. The validation and accuracy analysis performed by manual inspection can be incomplete and inaccurate. We mitigated this threat by focusing on the applications with smaller sets of code smells so that manual comparison could be conducted accurately.

With respect to reliability of our evaluation, JSNOSE and all the web-based systems are publicly available, making the results reproducible.

VII. CONCLUSIONS AND FUTURE WORK

In this paper, we discussed a set of 13 JavaScript code smells and presented a metric-based smell detection technique, which combines static and dynamic analysis of the client-side code. Our approach, implemented in a tool called JSNOSE, can be used by web developers during development and maintenance cycles to spot potential code smells in JavaScript-based applications. The detected smells can be refactored to improve their code quality.

Our empirical evaluation shows that JSNOSE is effective in detecting JavaScript code smells; Our results indicate that lazy object, long method/function, closure smells, coupling between JavaScript, HTML, and CSS, and excessive global variables are the most prevalent code smells. Further, our study indicates that there exists a strong and significant positive correlation between the types of smells and LOC, cyclomatic complexity, and the number of functions and JavaScript files.

For future work, we intend to extend our list of code smells, increase the accuracy of JSNOSE, add smell detection support for jQuery syntax, and design and implement an automated tool for refactoring detected JavaScript code smells.

Acknowledgment: This work was supported by the National Science and Engineering Research Council of Canada (NSERC) through its Strategic Project Grants programme. Amin Milani Fard is also supported by an Alexander Graham Bell Canada Graduate Scholarship (CGS-D) from NSERC.

REFERENCES

[1] Callback hell: A guide to writing elegant asynchronous JavaScript programs. http://callbackhell.com/.
[2] Checkstyle. http://checkstyle.sourceforge.net/.
[3] Google closure compiler. https://developers.google.com/closure/.
[4] Jslint: The JavaScript code quality tool. http://www.jslint.com/.
[5] Mozilla developer network's JavaScript reference. https://developer.mozilla.org/en-US/docs/JavaScript/Reference.
[6] WARI: Web application resource inspector. http://wari.konem.net.
[7] Y. Crespo, C. López, R. Marticorena, and E. Manso. Language independent metrics support towards refactoring inference. In *9th ECOOP Workshop on QAOOSE*, volume 5, pages 18–29, 2005.
[8] D. Crockford. *JavaScript: the good parts*. O'Reilly Media, Incorporated, 2008.
[9] M. Fowler and K. Beck. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.
[10] F. Galassi. Refactoring to unobtrusive JavaScript. JavaScript Camp 2009.
[11] S. H. Kan. *Metrics and Models in Software Quality Engineering*. Addison-Wesley, 2002.
[12] J. Kerievsky. *Refactoring to patterns*. Pearson Deutschland GmbH, 2005.
[13] M. Lanza and R. Marinescu. *Object-oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer, 2006.
[14] M. Mäntylä, J. Vanhanen, and C. Lassenius. A taxonomy and an initial empirical study of bad smells in code. In *Proc. International Conference on Software Maintenance (ICSM)*, pages 381–384. IEEE Computer Society, 2003.
[15] A. Mesbah and S. Mirshokraie. Automated analysis of CSS rules to support style maintenance. In *Proc. International Conference on Software Engineering (ICSE)*, pages 408–418. IEEE Computer Society, 2012.
[16] A. Mesbah, A. van Deursen, and S. Lenselink. Crawling Ajax-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web (TWEB)*, 6(1):3:1–3:30, 2012.
[17] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. Le Meur. DECOR: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1):20–36, 2010.
[18] M. J. Munro. Product metrics for automatic identification of "bad smell" design problems in Java source-code. In *Proc. International Symposium Software Metrics*, pages 15–15. IEEE, 2005.
[19] R. Murphey. JS minty fresh: Identifying and eliminating JavaScript code smells. http://fronteers.nl/congres/2012/sessions/js-minty-fresh-rebecca-murphey.
[20] H. V. Nguyen, H. A. Nguyen, T. T. Nguyen, A. T. Nguyen, and T. N. Nguyen. Detection of embedded code smells in dynamic web applications. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 282–285. ACM, 2012.
[21] V. Özçelik. o2.js JavaScript conventions & best practices. https://github.com/v0lkan/o2.js/blob/master/CONVENTIONS.md.
[22] J. Padolsey. jQuery code smells. http://james.padolsey.com/javascript/jquery-code-smells/.
[23] S. Porto. A plain english guide to JavaScript prototypes. http://sporto.github.com/blog/2013/02/22/a-plain-english-guide-to-javascript-prototypes/.
[24] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of JavaScript programs. In *Proc. of the conf. on Programming language design and implementation (PLDI'10)*, pages 1–12. ACM, 2010.
[25] F. Simon, F. Steinbruckner, and C. Lewerentz. Metrics based refactoring. In *Proc European Conference on Software Maintenance and Reengineering (CSMR)*, pages 30–38. IEEE, 2001.
[26] K. Simpson. Native JavaScript: sync and async. http://blog.getify.com/native-javascript-sync-async.
[27] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou. JDeodorant: Identification and removal of type-checking bad smells. In *Proc. European Conference on Software Maintenance and Reengineering (CSMR)*, pages 329–331, 2008.
[28] E. Van Emden and L. Moonen. Java quality assurance by detecting code smells. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pages 97–106. IEEE Computer Society, 2002.
[29] W3C. Document Object Model (DOM) level 2 events specification. http://www.w3.org/TR/DOM-Level-2-Events/, 13 November 2000.
[30] L. Williams, D. Ho, and S. Heckman. Software metrics in eclipse. http://realsearchgroup.org/SEMaterials/tutorials/metrics/.
[31] N. C. Zakas. Writing efficient JavaScript. In S. Souders, editor, *Even Faster Web Sites*. O'Reilly, 2009.
[32] N. C. Zakas. *Maintainable JavaScript - Writing Readable Code*. O'Reilly, 2012.