On the Use of Smelly Examples to Detect Code Smells in JavaScript

Ian Shoenberger[1], Mohamed Wiem Mkaouer[1(✉)], and Marouane Kessentini[2]

[1] Department of Software Engineering, Rochester Institute of Technology, Rochester, USA
{ian.shoenberger,mwmvse}@rit.edu
[2] Department of Computer and Information Science, University of Michigan, Ann Arbor, USA
marouane@umich.edu

Abstract. JavaScript has become one of the widely-used languages. However, as the size of JavaScript-based applications grows, the number of defects grows as well. Recent studies have produced a set of manually defined rules to identify these defects. We propose, in this work, the automation of deriving these rules to ensure scalability and potentially the detection of a wider set of defects without requiring any extensive knowledge on rules tuning. To this end, we rely on a base of existing code smells that is used to train the detection rules using Genetic Programming and find the best threshold of metrics composing the rules. The evaluation of our work on 9 JavaScript web projects has shown promising results in terms of detection precision of 92% and recall of 85%, with no threshold tuning required.

1 Introduction

JavaScript (JS) has been revolutionizing the web by combining services, libraries, and services from various third party providers. It was initially born to exclusively serve as a scripting standard at the browser level but it has drastically expanded to take over the lead in managing web N-tier architectures and as a result, 98% of the most visited websites incorporate JS [1]. The popularity of JS is issued from its dynamically-typed nature [2] and the wide variety of features it can quickly and dynamically include on the fly [3]. On the other hand, dynamically-typed languages, in general, have proven to be difficult to analyze, and thus, their catalog of development supporting tools suffers, and JavaScript is no exception [4]. With the rapid growth of JS applications in terms of size, rich interpolated functionalities at the expense of complexity and the lack of tools support, it is becoming a maintenance nightmare of developers [5]. Just like any other language, JS suffers from bad programming decisions, known as code smells [6], that can be introduced during the initial software development or during adding new features or applying debugging patches. The existence of code smells indicates the poor software quality and it increases the risk of introducing bugs. With the fact that JS is an interpreted language, the absence of a compiler that may raise warnings about potential runtime errors, adding to that the ability to include more code on the runtime through prototyping

© Springer International Publishing AG 2017
G. Squillero and K. Sim (Eds.): EvoApplications 2017, Part II, LNCS 10200, pp. 20–34, 2017.
DOI: 10.1007/978-3-319-55792-2_2

makes it hard to maintain a defect-free JS code base [7]. So code smells detection can be seen as a preventive task to minimize the number of bugs.

Unlike Object-Oriented (OO) languages, in which code smell detection and correction have been widely studied in the literature, there isn't much work tackling these smells in JS for various reasons, for instance, JS naturally does not necessarily comply with object oriented design rules, although it supports their implementation, furthermore, the lack of modeling support for JS prevents the automated identification of any high-level design anti-patterns. Still, several studies have approached the detection of bad development behavior in JS, in which, the most developed family of tools is for static analysis of programs, mostly for detecting very low-level errors in the code. Although tools such as JSLint [8], and JSHint [9] are great for syntactical issues like missing semi-colons and enforcing organizational policy, they do not support the high-level detection of structural defects within the source code that may not be as obvious.

The main challenge of detecting code smells in JS is the inability of statically calculate structural measurements e.g., coupling and cohesion, which can be combined to create code smells detection rules. To capture these interaction properties between JS objects, static analysis has to be augmented with dynamic analysis. On the flipside, dynamic analysis is more costly in terms of time and performance, also, in many statements and dependencies cannot be analyzed at runtime until triggered by a given input or scenario. Still, various studies have been conducted in the context of dynamic analysis to detect type inconsistencies [10], event related [11] cross-browser testing [12] and code smell detection [13]. The latter work labeled JSNOSE [13] extracts a set of static metrics from the static analysis of JS objects and monitor their behavior during the application runtime. Using this combination, it uses the rule-based approach to identify high-level disharmonies, which are similar to a subset of the code smells that exist in literature, as well as smells that exist specific to JavaScript. In addition, JSNOSE allows developers to extend the list of possible smells to detect. In this approach, rules are manually defined to identify the key symptoms that characterize a code-smell using a set of limited quantitative metrics. However, in a more concrete setting, the number of possible code-smells to manually characterize with rules can be large. Moreover, for each code-smell, rules that are expressed in terms of metrics require continuous calibration efforts to find the right threshold especially in a highly dynamic environment like JS.

To cope the above-mentioned limitations, we propose in this paper the automated tuning of code-smells detection rules using a dataset of existing code-smells. The process aims at finding, for each JS object, a subset of similar objects in the base of examples, then, using this subset of objects with their known smells, we use the Genetic Programming to tune the threshold of quantitative metrics in order to maximize the coverage of the detected smells in the subset. These calibrated metrics will be later on used to detect smells in JS. The evaluation of our work on 9 JS web projects has shown promising results in achieving a detection of a subset of JS smells that were reported by JSNOSE with 92% precision, with no metrics tuning needed.

The remainder of this paper is structured as follows. Section 2 provides the background required to understand our approach and the challenges in detecting code-smells in JS. In Sect. 3, we describe our approach and show how we deployed a GP to tune

metrics that will be used for the smells detection. Section 4 presents and discusses the results obtained by comparing our approach with JSNOSE. Related work is discussed in Sect. 5, while in Sect. 6 we conclude and suggest future research directions.

2 Background and Problem Statement

After nearly two decades since code smells were introduced by Fowler [6], there is no consensus on how to standardize the definition of code smells. Several studies have been characterizing smells by their symptoms at the source code level that are be measured by structural metrics [14] or using history of code changes [15] or even using textual information extracted from the code base internal and external documentation [16]. It is to note that dynamic analysis was not solicited in the detection process mainly due to its complexity and also because static analysis offers a rich catalog of metrics that be used to create detection rules. However, this does not apply to JS. JS revolution in the last decade was driven by the dynamic manipulation of the Document Object Model (DOM) and XML objects under several web protocols [17]. The tremendous growth of JS web application has negatively impacted their maintenance especially with the propagation of the JS technology to become the leading language for servers and web databases [18]. In this context, Mesbah et al. [13] discovered the existence of traditional code smells in JS and clustered them into "classic" smells that are derived from the literature and "JS-specific" smells that identify exclusive bad programming patterns in JS. Since we aim at using the knowledge from existing smells detection literature, this paper will only focus on detecting the first type of the smells. Table 1 summarizes the code smells detected by JSNOSE and studied in this work.

Table 1. Detection Rules for JS code smells [13].

Code smell	Level	Detection rule	Structural metrics
Many global variables	File	GLB > 10	GLB: Number of global variables NOP: Number of properties LOC: Lines of code MLOC: Method lines of code PAR: Number of parameters BUR: Base-Object usage ratio NOC: Number of cases
Large object	Object	LOC > 750 OR NOP > 20	
Lazy object	Object	NOP < 3	
Long method	Method	MLOC > 50	
Long parameter list	Method	PAR > 5	
Refused Bequest	Object	BUR < 1/3 AND NOP > 2	
Switch statement	Method	NOC > 3	

Although JSNOSE has given promising results, it suffers from scalability issues as the application of the above-mentioned rules requires the manual calibration of thresholds which tends to be subjective, human intensive and error prone. Furthermore, previous detection studies resulted in several approximations of each smell in terms a set of metrics that be deployed to its identification and addressing this limitation by

asking the developers to redefine their own rules is difficult as it needs an extensive knowledge about analysis and requires qualitative validation.

Recent studies [19, 20] have been investigating how developers rely on OO methodology when designing their JS code-base and have demonstrated the existence of OO-like structures and code-elements in JS [21]. Therefore, the main contribution of this paper is driven by the following research question: How to consider the existing knowledge in detecting bad programming practices in OO languages in the detection similar practices in JS. A straightforward approach would be to deploy existing detection techniques, defined for OO, to JS, but the detection process relies essentially on a profound static analysis that allows the definition of smells using a rich set of structural metrics, which are limited in the context of JS. Our contribution relies essentially on exploiting the similarity between JS and Java in order to identify smelly JS elements if they are similar to smelly Java elements. Also, we propose to address the above-mentioned limitations of JSNOSE by automatically generating thresholds for detection rules. Moreover, the base of smells examples can be generated using any developer-preferred existing detection tool in order to tune the rules to detect similar-smells in JS.

3 Approach Overview

The general workflow of this approach is decomposed into 4 main stages as shown Fig. 1. The 4 stages are described in this Section.

(1) **JS entities enumeration.** Our approach, built upon JSNOSE, inputting the web app, containing JS files, to the crawler. Once JS files extracted and parsed, the set of extracted JS Entities, called JSE, (e.g., objects, properties) is sent to the Similarity Calculator.

(2) **Similar Elements extraction.** This module takes as input JS entities and code elements from existing software systems. The purpose of this process is to extract code elements that are mostly similar in terms of structural properties, this set of similar extracted elements is labeled SEE. To avoid the exhaustive comparison between JS entities with all the code elements from the code base, which can become easily large, we apply an initial matching between them as follows [19] (Table 2):

Table 2. Initial mapping between JS entities and code elements

Code element	JS entity
Class	File/Object
Attribute	Property
Method	Function/Inner function

Then for each pair of entity/code element, we calculate their structural similarity. We consider this process similar to structural code clone detection and we extend this existing metric-based technique [22] that identifies exact elements by loosening

this exactitude to a certain threshold. Let MX = <$m1(X)$, …, $mn(X)$> be the set of metrics characterizing the entity X; $mi(X)$ ($i = 1 \ldots n$) stands for the i-th software metric chosen to describe X, and n is a fixed number metrics describing the entities. Any other entity Y is considered similar to X with threshold α *iff*:

$$Sim(X, Y) = \sum_{i=1}^{n} |m_i(X) - m_i(y)| < \sum_{i=1}^{n} \alpha_i \quad (1)$$

In order to automatically approximate α for each metrics, we use the box-plot technique to select near lower bound of the values space. Using these values, for each JS entity a subset of code elements will be traced.

(3) **Metrics thresholds calibration.** The purpose of this step is to update the thresholds of JS detection rules. To do so, we first extract the smelly elements, called SmellyE, from SEE. The detection of those elements can be done with any state-of-art detection tool, in this study we used infusion to detect smelly elements in our base of example projects. Also, this detection process may be done prior to running the calibration algorithm for performance purposes. Once the set of SmellyE is known, for each smell type a detection rule is generated. Therefore, for each generated rule, the metrics used by JSNOSE undergo the calibration process by inputting them in a tree structure. the tuning process aims in tuning metrics thresholds in order to maximize the number of detected smells in SmellyE, this way, the tuned rule can be used later on to detect smells in JS.

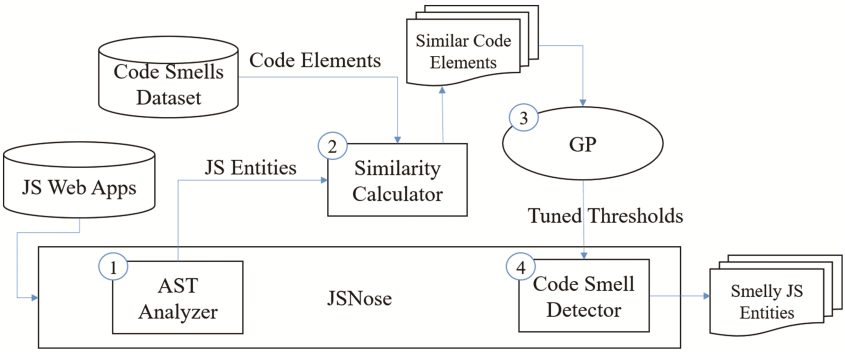


Fig. 1. Approach overview.

Solution Representation. Since our aim is to only tune thresholds for given rules, the individual initial setup was seeded with JSNose built in rules.

Solution Evaluation. The GP Initially takes as input the list of code elements, which were previously selected based on their similarity with a given JS entity. Initially, the calibrator tests whether a subset of elements contain smells. If the elements are smell free then the subset will be discarded and no calibration needed. Otherwise, the code smell types will be sorted based on their occurrence and the most occurring smell type

triggers the GP's evolutionary process of tuning its metrics' thresholds. The fitness function used evolves the rule towards maximizing the detection of the infected elements in SmellyE, the set of identified smells is labeled DSmellyE.

Solution Evolution. Since the rules structure was not to change, the crossover was not solicited. Also, the uniform subtree mutation [23] was constrained to act exclusively on nodes with values and the function was only allowed to update the value by randomly adding or subtracting a randomly selected number between 0 and 1. After several generations, once the stopping criterion is met, the best solution i.e., a rule with the best detection ranking is then sent to back to JSNose to be used for the detection process. The following table gives a summary of the terminals used in the GP algorithm (Table 3).

Table 3. GP Terminology.

Term	Definition
JSE	Set of JS extracted entities, subject to smell investigation
SEE	Set of extracted elements from existing projects as they exhibit a strong structural similarity with the elements in JSE
SmellyE	Subset of JSE that are infected with code smells
DSmellyE	Subset of SmellyE that were identified by the detection rule

The following pseudo-code highlights the adaptation of GP for the problem of detection rules generation.

Algorithm1. Metrics calibration using GP for each smell type

```
Input: Subset of JSE, infected with code smells (SmellyE)
Input: Metrics (R)
Output: Detection rule
1: Create a random Population (P) of Individuals (I)
2: Randomly create a rule using metrics I ← rule(R)
3: repeat
4:     for all I ∈ P do
5:             DSmellyE ← I.executeRule(SmellyE)
6:             I.fitness ← DSmellyE ∩ SmellyE
7:     end for
8:     BestIndividual ← Rank(P, BestIndividual)
9:     P ← reproducePopulation(P)
10:
11:    generation←generation+1;
12: until generation = maxGeneration
13: return BestIndividual
```

(4) **Code Smells Detection.** The GP returns an updated detection rule, for any smell type that was known in the subset of code elements. This rule is then executed to report any JS entities that their properties do not violate the updated metrics thresholds.

4 Initial Evaluation Study

4.1 Research Questions

We defined two research questions to address in our experiments.

RQ1: What is the detection performance of the auto-tuned rules in the detection of several JS smell types?

RQ2: What is the impact of the base of examples' size on the finding code elements with similar properties to the JS entities and how similar are the tuned rules to the ones defined by JSNose?

To answer RQ1, we assess the performance of our detection process by its ability to replicate the detected smells by JSNOSE for the same given projects. To do so, we run JSNose to re-generate the code smells that we consider as expected. Then we run our approach to generating our suggested smells. These expected and suggested smells are used to calculate the precision and recall as follows:

$$\text{PR}precision = \frac{|\text{ suggested smells} \cap \text{ expected smells}|}{|\text{suggested smells}|} \in [0,1] \quad \text{RC}recall = \frac{|\text{ suggested smells} \cap \text{expected smells}|}{|\text{expected smells}|} \in [0,1] \tag{2}$$

For RQ2 we want to study the impact of varying the size of the dataset on the performance of the similarity process. We also illustrate the results by manually verifying a set of the reported JS entities.

4.2 Experimental Setting

To build our dataset of example, we randomly sampled 100 small to early medium open source projects. We limited the size of our projects files based on our initial observation that the average size of the JS projects is small. The sampled projects were 61 Java Projects and 39 C++ projects, this helps in diversifying the set of examples especially that C++ is not purely object oriented. To detect smells within the dataset, we used two state-of-art code smell detectors namely InCode [14] and PMD [24]. To reduce the cost of parsing several projects, we performed the detection process prior to running the GP and saved the results (Code elements, their metrics and the list of infected ones) on separate log files[1]. We tried selecting the projects previously used in JSNose as the JS smells were manually validated on that study. We couldn't locate two of the projects. It is also to note that the TinyMCE project is still under continuous development and its size has increased compared to when it was tested using the original experiment of JSNose. Since the previously used releases were not mentioned and the number of added smells is relatively low, we included it in our benchmark (Table 4).

[1] For replication purposes, the dataset and tools used are located in: https://github.com/mkaouer/Code-Smells-Detection-in-JavaScript.

Table 4. Projects constituting the benchmark for our approach.

Systems	Number of JS files	JS LOC	Number of infected JS entities
PeriodicTable	1	71	23
CollegeVis	1	177	53
ChessGame	2	198	64
Tunnel	0	234	54
GhostBusters	0	278	49
FractalViewer	8	1245	212
PhotoGallery	5	1535	221
TinySiteCMS	13	2496	172
TinyMCE	191	26908	59

During this study, we use the same parameter setting for all executions of the GP. The parameter setting is specified in Table 5.

Table 5. Parameter tuning for GP.

GP parameter	Values
Population size/Max tree depth	50/2
Selection/Survival/K	Roulette-Wheel/K-Tournament/2
Mutation/Mutation rate/Range	Uniform-Sub-tree/0.1/[0…1]
Max iterations	1000/2500/5000

4.3 Results and Discussions

As an answer to RQ1, Table 6 reports the results of the empirical qualitative evaluation of the tuned detection rules in terms of precision and recall.

It is observed in Table 6 that our GP algorithm was able in most cases to replicate the results of JSNose. For reporting a high number of global variables, GP and JSNose performed the same except for *PeriodicTable* project, which GP's acceptance threshold was lower than JSNose. Since Global variables are highly discouraged especially in OO programs, having 6 global variables in *PeriodicTable* was considered high. For the large object defect, it is to note that, JSNose has not reported any instance for the *GhostBusters* and *PhotoGallery* project which was not the case for GP, since, in relational programming, several classes and files have the blob behavior, it is most likely to find several functions condensed in one entity and this increases the chance of detecting them as large entities. For the lazy object, GP has missed several smells that were reported by JSNose mainly in projects with limited size. However, in larger projects, GP reported more smells than JSNose. Both algorithms have almost agreed on the long method, long parameter list and refused bequest smells, this was expected especially that the definition of these smells that was adopted by JSNose matches the exact same rules used by InCode [14] since the authors of JSNose tuned the exact subtree, previously proposed by the authors of InCode. It can be also seen that; in general, the recall was usually lower than the precision for the small-sized JS projects while the precision was mostly lower for

Table 6. Median values of precision and recall for the JS code smells in 9 JS web projects over 31 runs. Differences are highlighted

Software	Many global variables		Large object		Lazy object		Long method		Long parameter list		Refused Bequest		Switch statement		Median per Project	
	GP (+/-)	JSNose (+/-)	PRE (%)	REC (%)	PRE (%)	REC (%)	PRE (%)	REC (%)	PRE (%)	REC (%)	PRE (%)	REC (%)	PRE (%)	REC (%)	PRE (%)	REC (%)
PeriodicTable	+	-	1	**0.25**	1	**0.40**	1	1	1	1	1	1	1	1	1	**0.77**
CollegeVis	+	+	1	1	1	**0.37**	1	**0.5**	1	1	1	1	1	1	1	**0.81**
ChessGame	+	+	1	**0.77**	1	**0.55**	1	1	1	1	1	1	1	1	1	**0.88**
Tunnel	+	+	1	1	1	**0.59**	1	1	1	1	1	1	1	1	1	**0.93**
GhostBusters	-	-	**0**	**0**	1	0.65	1	1	1	**0.33**	1	1	1	1	**0.83**	**0.66**
FractalViewer	+	+	1	**0.71**	1	1	**0.83**	**1**	1	**0.80**	1	**0.93**	1	1	**0.97**	**0.90**
PhotoGallery	+	+	**0**	**0**	1	1	1	1	1	1	1	1	1	1	**0.83**	**0.83**
TinySiteCMS	+	+	1	**0.77**	**0.86**	1	1	1	**0.51**	1	1	1	1	1	**0.89**	**0.96**
TinyMCE	-	-	**0.60**	1	**0.78**	1	1	1	**0.44**	1	1	**0.77**	1	**0.66**	**0.80**	**0.94**
Median per Smell	**N/A**	**N/A**	**0.73**	**0.61**	**0.96**	**0.72**	**0.98**	**0.94**	**0.88**	**0.90**	1	**0.93**	1	**0.96**	**0.92**	**0.85**

the projects which were larger like *TinySiteCMS* and *TinyMCE*. This induces that GP tends to be stricter on smaller projects while its rules become less selective. To better understand if this is due to the random sampling of the projects or due to the number of similar code elements that were selected during the training of GP rules, we conducted thought of tuning the rules with only smells that existed different releases within the same project. During multiple releases of a software, several features are added, bugs are fixed and refactorings conducted, this induces several changes in the software structure and any smell that would survive throughout these changes can be considered persistent. To do so, we picked, for each project, 4 previous releases and we conducted the same static analysis to generate all the structural metrics and we analyzed them with PMD and infusion, then we ruled out all the smells that didn't exist throughout the total of 5 releases. Then we re-conducted threshold of our rules. The following Fig. 2 shows the comparison of tuning with one release for each project (GP1R) with the one with 5 releases (GP5R).

As observed in Fig. 2, GP5R has increased the recall score of rules with multiples metrics (e.g., lazy object, refused bequest) and this can be explained by the fact that tuning the threshold with fewer but persistent smells makes the detection harder and fewer fittest rules evolve during the GP. On the other hand, the precision of GP5R for refused bequest has reduced.

For **RQ2**, Fig. 3 shows that the number of existing projects in this study has provided enough elements that are structurally similar to the JS elements under analysis. In general, it was easier to quickly find functions with equivalent size and number of parameters and the similarity has converged into an acceptable range earlier compared to the objects and files; it was harder to find similar classes in terms of number of lines of code and number of properties and functions, especially that the JS projects sizes were

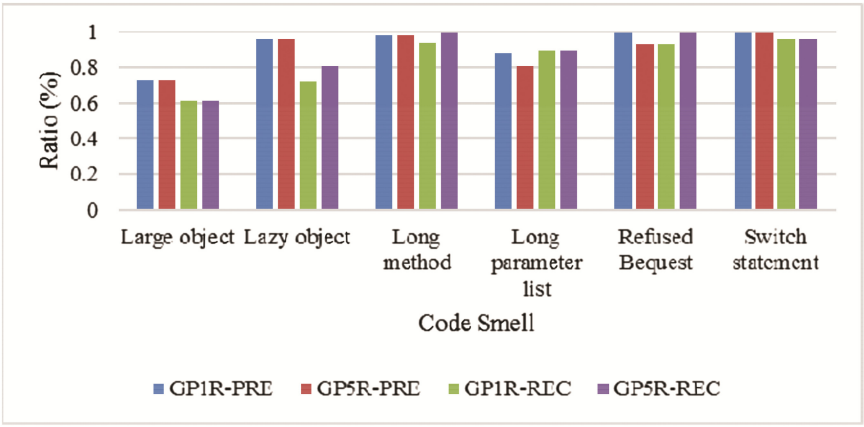


Fig. 2. Median values of precision and recall of GP1R and GP5R for the JS code smells in 9 JS web projects over 31 runs.

relatively small and so is for the projects that were also small and thus the number of classes was limited per project.

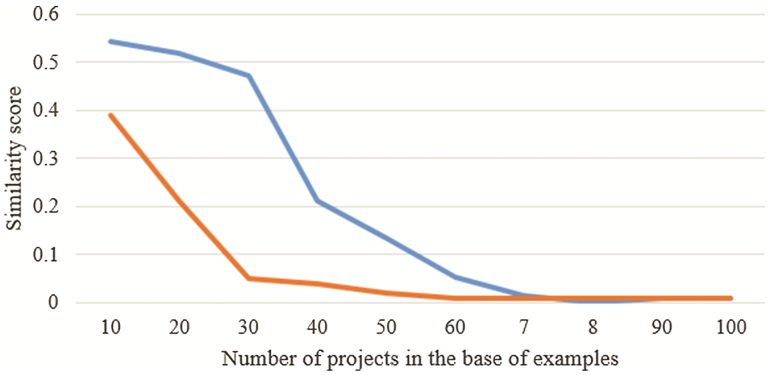
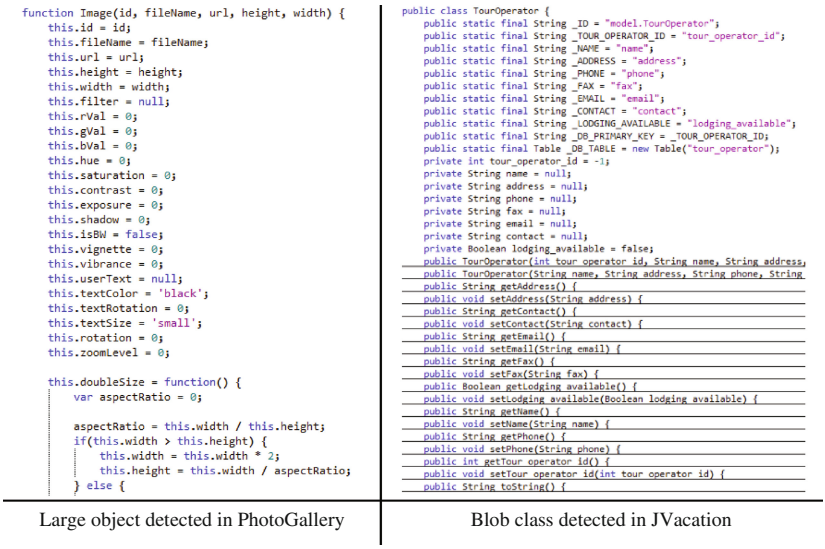


Fig. 3. Impact of the number of projects on the similarity between JS entities and code elements.

To illustrate the similarity between JS smells and the ones in the base of examples we extracted some JS smell instances and an equivalent selected code element from the base of examples with a smell instance.

As shown in Fig. 4, an object to be instantiated can be exceedingly large and often the object has many properties and functions that distinguish it as a code smell. Notice how it is logical to group the properties that are defined by lines 2 to 6. They are fundamental to the image entity. Lines 7 through 18 are really about effects, and so it may make sense to encapsulate these properties into their own object, perhaps called *ImageEffects*, and then include the object as a dependency within *Image*. As the code stands in the figure, the Image object is concerned about the properties of a text that the user may add while editing. Good software engineering practices suggest to separate

concerns, and so it would also make sense to also encapsulate text properties into their own object. This smell is similar to the blob classes which are characterized by being data-driven highly cohesive and standalone objects (no inheritance and rare coupling).



```
function Image(id, fileName, url, height, width) {
    this.id = id;
    this.fileName = fileName;
    this.url = url;
    this.height = height;
    this.width = width;
    this.filter = null;
    this.rVal = 0;
    this.gVal = 0;
    this.bVal = 0;
    this.hue = 0;
    this.saturation = 0;
    this.contrast = 0;
    this.exposure = 0;
    this.shadow = 0;
    this.isBW = false;
    this.vignette = 0;
    this.vibrance = 0;
    this.userText = null;
    this.textColor = 'black';
    this.textRotation = 0;
    this.textSize = 'small';
    this.rotation = 0;
    this.zoomLevel = 0;

    this.doubleSize = function() {
        var aspectRatio = 0;

        aspectRatio = this.width / this.height;
        if(this.width > this.height) {
            this.width = this.width * 2;
            this.height = this.width / aspectRatio;
        } else {
```

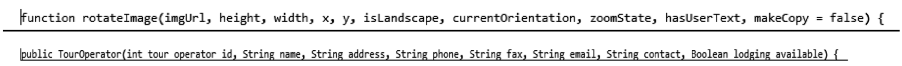
Large object detected in PhotoGallery

```
public class TourOperator {
    public static final String _ID = "model.TourOperator";
    public static final String _TOUR_OPERATOR_ID = "tour_operator_id";
    public static final String _NAME = "name";
    public static final String _ADDRESS = "address";
    public static final String _PHONE = "phone";
    public static final String _FAX = "fax";
    public static final String _EMAIL = "email";
    public static final String _CONTACT = "contact";
    public static final String _LODGING_AVAILABLE = "lodging_available";
    public static final String _DB_PRIMARY_KEY = _TOUR_OPERATOR_ID;
    public static final Table _DB_TABLE = new Table("tour_operator");
    private int tour_operator_id = -1;
    private String name = null;
    private String address = null;
    private String phone = null;
    private String fax = null;
    private String email = null;
    private String contact = null;
    private Boolean lodging_available = false;
    public TourOperator(int tour_operator_id, String name, String address,
    public TourOperator(String name, String address, String phone, String
    public String getAddress() {
    public void setAddress(String address) {
    public String getContact() {
    public void setContact(String contact) {
    public String getEmail() {
    public void setEmail(String email) {
    public String getFax() {
    public void setFax(String fax) {
    public Boolean getLodging_available() {
    public void setLodging_available(Boolean lodging_available) {
    public String getName() {
    public void setName(String name) {
    public String getPhone() {
    public void setPhone(String phone) {
    public int getTour_operator_id() {
    public void setTour_operator_id(int tour_operator_id) {
    public String toString() {
```

Blob class detected in JVacation

Fig. 4. JS Large object smell.

As shown in Fig. 5, a long parameter list is a straightforward to detect code smell, it is an important smell to fix because of two reasons. First, the long list of parameters hurts the understandability of the code and so developers will experience difficulty in capturing the method's behavior. Even if the name of the method implies that the image will be rotated, but why are there so many parameters? What does *makeCopy* have to do with rotating an image? Clearly, this adds some confusion if a developer is not familiar with the function. And it can result in confusion even to someone who wrote the method but needs to revisit it in order to modify the behavior. Secondly, having so many parameters could potentially indicate that the method has assumed more than one responsibility as it has evolved over time. It is often simpler, and easier for developers to quickly add another parameter, and then create a 'special case', or a typical branch of execution for the respective arguments in order to tailor the method to new requirements. This is where smells become introduced over the lifetime of an application in either JS or any other language. Having such a long list of parameters like in the constructor in Fig. 5 makes its invocation difficult as well.



```
function rotateImage(imgUrl, height, width, x, y, isLandscape, currentOrientation, zoomState, hasUserText, makeCopy = false) {
```



```
public TourOperator(int tour_operator_id, String name, String address, String phone, String fax, String email, String contact, Boolean lodging_available) {
```

Fig. 5. Long parameter list smell seen in JS function *rotateImage* and Java constructor *TourOperator*

Refused Bequest in an object-oriented language such as Java refers to any class that extends some parent/base class but does neither use or override a reasonable amount of the base class's behavior nor uses a reasonable amount of its properties. In JavaScript, the concept is slightly different but it can still be applied. JavaScript is class-free but it uses prototype-based inheritance. That is, object A can share properties from another object, B, the prototype. As shown in this figure, there is a JSON object created called *photo.* That object has some simple information about the image's owner, a date, as well as data about how the photo was developed. Another object is created that uses this *photo* object as its prototype, meaning it has access and shares the same properties like *stopBathTime*. However, the *Instagram* object adds a new member to itself, *filter*, and only uses one of the original properties from the photo object. The *Instagram* objects use of the photo object's properties does not justify its extension. Similarly, *DBConnector* class extends *DBFactory* but defines its own attribute and two functions without really using any of the properties of its parent class. Both situations can be detected through a low value of the base-object usage ratio metric (Fig. 6).

```
var photo = { date: "11.22.1999", owner: "Craig", exposureTime: 0.25, developerBathTime: 10, stopBathTime: 30, fixerBathTime: 1};

var instagramPhoto = Object.Create(photo);
instagramPhoto.filter = "1977";
var newText = instagramPhoto.owner + " used the " + instagramPhoto.filter + " filter.";
document.getElementById('newPhotoCaption12').innerHTML(newText);
```

```
public class DBConnector extends DBFactory {

    private Connection connection = null;
    // open db-connection
    public DBConnector() {
    // close db-connection
    public void close() {
```

Fig. 6. Refused bequest smell seen in JS function *Instagram* and Java class *DBConnector*

5 Related Work

The detection of code smells on software systems has been the subject of several studies over the past decade since their first introduction by Fowler and Beck [6]. They described 22 code smells as structural code flaws that may decrease the overall software quality and serve as indicators of software vulnerabilities. To cope with these smells, Fowler has introduced a set of 72 Refactoring operations to fix code smells and thus improving the system overall design. The detection process can either be manual, semi-automated or fully automated. Van Emden and Moonen developed one of the first automated code-smell detection tools for Java programs [25]. Mäntylä et al. [26] provided an initial formalization of the code smells, in terms of metrics, based on analyzing Fowler's smells description, they studied the correlation between the smells and provided a classification according to their similarity. Marinescu et al. [27] presented an automated framework for smells identification using detection strategies which are defined by metric-based rules. Moha et al. [28] presented a semi-automated technique called DECOR. This framework allows subjects to manually suggest their own defects through their description with domain specific language, then the framework automatically searches for smells and visually reports any finding. Most of the above-mentioned work focus mainly on smells specification in order to automate their detection without taking the developer's opinion in the detection process. To better incorporate the developer's preference,

Kessentini et al. [29] suggested a by-Example approach that uses a base of previously detected code smells as a base of examples to generate user-oriented detection rules. This work was extended [30] to reduce the effort of the detection and correction process. The main limitation of the by-Example approach is the requirement of a large base of examples (manually detected and validated code smells) for each code smell type, this work does not rely on the use of a manually validated database of JS code smells examples, instead, any project can be used as a training through the automated generation of its smells instances using any detection tool.

6 Threats to Validity

In our experiments, we raise multiple construct threats that are related to the random sampling of the projects that belong to the base of examples, we mitigated this threat by selecting projects from two languages that both support object oriented concepts and their size was proportional to the size range of the JS projects. Also, the types of smells that exist in the base of examples represents another threat. We have used two state of the art and very popular detection tools that are known for their accuracy and also they are being cited by the authors of JSNOSE. We take into account the internal threats to validity in the use of stochastic algorithms since our experimental study is performed based on 31 independent simulation runs for each problem instance and the obtained results are statistically analyzed by using the Wilcoxon rank sum test with a 95% confidence level ($\alpha = 5\%$). Another threat is raised through the lack of metrics that may better capture the structure and properties of elements. As part of our future work, we are investigating the use of dynamic analysis to capture the call graphs of objects and records their afferent and efferent communications to be able to approximate their coupling and cohesion scores.

7 Conclusion and Future Work

The paper introduces a novel detection of the JS code-smells using classic smelly examples that are extracted from existing C++ and Java projects. The purpose of this paper was to automate the tuning of JS detection rules in order to avoid human intervention and to also take achieve example-like detection that benefits from the maturity of existing studies. This tuning has required the extension of a specific similarity function to identify structurally similar entities between JS and the elements extracted from the base of examples. The tuning process was done by GP that has taken as input the JS detection rules that have been evolving to detect the expected smells in the extracted subset of elements. The evaluation of this work has shown promising results that have proven the capability of our approach to replicate the results of JSNose with 92% precision.

We are planning as future work to extend the base of code smells and identify popular method-level defects such as feature envy and shotgun surgery, to do so, it is necessary to extend the base of metrics used to define the rules as measures like coupling, cohesion and complexity are mandatory to accurately detect those smells. This will require further

investigation on how to measure these metrics on JS dynamic environment. We also plan on extending the evaluation to incorporate front end and back end JS projects.

References

1. Richards, G., Lebresne, S., Burg, B., Vitek, J.: An analysis of the dynamic behavior of JavaScript programs. In: Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, pp. 1–12 (2010)
2. Richards, G., Hammer, C., Burg, B., Vitek, J.: The eval that men do. In: Mezini, M. (ed.) ECOOP 2011. LNCS, vol. 6813, pp. 52–78. Springer, Heidelberg (2011). doi: 10.1007/978-3-642-22655-7_4
3. Ratanaworabhan, P., Livshits, B., Zorn, B.G.: JSMeter: comparing the behavior of JavaScript benchmarks with real web applications. In: WebApps 2010, vol. 10, p. 3 (2010)
4. Nikiforakis, N., Invernizzi, L., Kapravelos, A., Van Acker, S., Joosen, W., Kruegel, C., Piessens, F., Vigna, G.: You are what you include: large-scale evaluation of remote JavaScript inclusions. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security, CCS 2012, Raleigh, North Carolina, USA, pp. 736–747 (2012)
5. Guarnieri, S., Pistoia, M., Tripp, O., Dolby, J., Teilhet, S., Berg, R.: Saving the world wide web from vulnerable JavaScript. In: Proceedings of the 2011 International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, Ontario, Canada, pp. 177–187 (2011)
6. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: improving the design of existing programs. Addison-Wesley Longman Publishing Co., Inc., Boston (1999)
7. Madsen, M., Tip, F., Andreasen, E., Sen, K., Møller, A.: Feedback-directed instrumentation for deployed JavaScript applications. In: Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, Texas, pp. 899–910 (2016)
8. Crockford, D.: JSLint: The javascript code quality tool, 95 (2011). http://www.jslint.com
9. Kovalyov, A., Kluge, W., Perez, J.: JSHint, a JavaScript code quality tool (2010)
10. Pradel, M., Schuh, P., Sen, K.: TypeDevil: Dynamic type inconsistency analysis for JavaScript. In: Proceedings of the 37th International Conference on Software Engineering, ICSE 2015, Florence, Italy, pp. 314–324 (2015)
11. Raychev, V., Vechev, M., Sridharan, M.: Effective race detection for event-driven programs. In: Proceedings of the 2013 ACM SIGPLAN International Conference, OOPSLA 2013, Indianapolis, Indiana, USA, pp. 151–166 (2013)
12. Mesbah, A., Prasad, M.R.: Automated cross-browser compatibility testing. In: Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu, HI, USA, pp. 561–570 (2011)
13. Fard, A.M., Mesbah, A.: JSNOSE: detecting JavaScript code smells. In: Proceedings of the 13th IEEE International Conference on Source Code Analysis and Manipulation, SCAM 2013, pp. 116–125 (2013)
14. Marinescu, R., Ganea, G., Verebi, I.: inCode: continuous quality assessment and improvement. In: Proceedings of the 14th European Conference on Software Maintenance and Reengineering, pp. 274–275 (2010)
15. Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., De Lucia, A., Poshyvanyk, D.: Detecting bad smells in source code using change history information. In: 2013 IEEE/ACM 28th International Conference on Automated Software Engineering (2013)

16. Palomba, F.: Textual analysis for code smell detection. In: Proceedings of the 37th International Conference on Software Engineering, ICSE 2015, Florence, Italy, pp. 769–771 (2015)
17. Garrett, J.J.: Ajax: a new approach to web applications (2005)
18. Cornelissen, B., Zaidman, A., Van Deursen, A., Moonen, L., Koschke, R.: A systematic survey of program comprehension through dynamic analysis. IEEE Transactions on Software Engineering **35**(5), 684–702
19. Silva, L.H., Ramos, M., Valente, M.T., Bergel, A., Anquetil, N.: Does JavaScript software embrace classes? In: Proceedings of the 22nd International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 73–82 (2015)
20. Mendes, T., Valente, M.T., Hora, A.: Identifying utility functions in Java and JavaScript. In: 2016 X Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS), 19–20 September 2016, pp. 121–130 (2016)
21. Humberto Silva, L., Hovadick, D., Tulio Valente, M., Bergel, A., Anquetil, N., Etien, A.: JSClassFinder: a tool to detect class-like structures in JavaScript. arXiv preprint arXiv: 1602.05891 (2016)
22. Merlo, E., Antoniol, G., Di Penta, M., Rollo, V.F.: Linear complexity object-oriented similarity for clone detection and software evolution analyses. In: Proceedings of IEEE 20th International Conference on Software Maintenance, ICSM 2004, pp. 412–416 (2004)
23. Van Belle, T., Ackley, D.H.: Uniform subtree mutation, pp. 152–161 (2002)
24. http://pmd.sourceforge.net/snapshot/. Accessed 17 July 2016
25. Van Emden, E., Moonen, L.: Java quality assurance by detecting code smells. In: Proceedings of the 9th Working Conference on Reverse Engineering (WCRE 2002), Washington, DC, USA, pp. 97–106 (2002)
26. Mäntylä, M., Vanhanen, J., Lassenius, C.: A taxonomy and an initial empirical study of bad smells in code. Proceedings of the International Conference on Software Maintenance, ICSM **2003**, 381–384 (2003)
27. Marinescu, R.: Detection strategies: metrics-based rules for detecting design flaws, pp. 350–359 (2004)
28. Moha, N., Gueheneuc, Y.-G., Duchien, L., Le Meur, A.-F.: DECOR: a method for the specification and detection of code and design smells. IEEE Trans. Software Eng. **36**(1), 20–36 (2010)
29. Kessentini, M., Kessentini, W., Sahraoui, H., Boukadoum, M., Ouni, A.: Design defects detection and correction by example. In: Proceedings of the 19th IEEE International Conference on Program Comprehension (ICPC 2011), 22–24 June 2011, pp. 81–90 (2011)
30. Ouni, A., Kessentini, M., Sahraoui, H., Boukadoum, M.: Maintainability defects detection and correction: a multi-objective approach. Automated Software Engineering **20**(1), 47–79 (2012)