Normalizing Object-oriented Class Styles in JavaScript

Widd Gama, Manar H. Alalfi, James R. Cordy, Thomas R. Dean

School of Computing, Queen's University, Kingston, Canada
{gama, alalfi, cordy, dean}@cs.queensu.ca

***Abstract*—JavaScript is a dynamic, weakly typed, multi-paradigm programming language that supports object-oriented, imperative, and functional programming styles. While web developers appreciate this flexibility when implementing complex and interactive web applications, this wide range of possible styles can hinder program comprehension and make maintenance difficult, especially in large projects involving many different programmers. A particular problem is the several different ways in which object-oriented classes can be expressed in JavaScript. In this work we aim at enhancing the maintainability of object-oriented JavaScript applications by automatically normalizing the representation of classes to a single model.**

I. Introduction

JavaScript is the most widely used client-side scripting language, is increasingly popular as a crucial component of the AJAX technology, and is one of the core components of the upcoming HTML5 standard [18]. When used in the object-oriented style, JavaScript is a prototype-based language, with no explicit class declaration to provide a definitive description of the fields and methods of a given object. In JavaScript, the developer simply defines a constructor function that builds object instances, and the application can later add or remove properties from the instance. The flexibility of the language and the lack of any clear mapping to conventional class-based object-oriented concepts has led to a plethora of different class representation techniques when developing applications. Sometimes, multiple techniques are used in the same program, resulting in code that is inconsistent and difficult to maintain.

Figure 1 shows three of the many styles of encapsulation of data and methods possible in JavaScript. All three are almost identical in semantics. The first style (Figure 1(a)) is closest to the conventional syntactic class-based style. The function *Class* acts as both the class definition and the constructor. When the *new* operator is applied (line 6), an object is created and the function is invoked on it, adding two properties. The first property, *x*, is a scalar data property with the value 4, and the second property *m* is a method (since its value is a function). Both the value and the method are accessed using the name of the property (e.g., *c.x* and *c.m ()* respectively).

The second variation (Figure 1(b)) uses the function's prototype object. In this case the constructor function is an anonymous(lambda) function that is assigned to the variable *Class*. The initialization of the object, instead of being inside the constructor function, has been moved to the code following the creation of the object. This variation is semantically equivalent to the first, but reduces the level of encapsulation.

The third variation (Fig. 1(c)) uses the same *Class* variable and anonymous function, but leaves the assignment of the value to the *x* property inside the constructor function. The method *m*, instead of being assigned as a property of the object, is assigned to the prototype of the object. As a result, all objects created using the *Class* constructor function will share this same method, using the *__proto__* chain. While not identical to the first two in semantics, the differences are not important for most JavaScript code in practical use today.

(a) Object class variant 1

```
function Class () {
   this.x = 4;
   this.m= function m() {
       }
}
c = new Class();
```

(b) Object class variant 2

```
Class = function () {
}
c = new Class ();
c.x = 4;
c.m = function m() {
   }
```

(c) Object class variant 3

```
Class = function () {
   this.x = 4;
}
Class.prototype.m = function m() {
   }
c = new Class();
```

Fig. 1. Data and Method Encapsulation in JavaScript

This paper describes an automated approach to normalizing JavaScript code to a single consistent object-oriented style. In particular, we address the issue of the consistent encapsulation of data and functions as object classes (inheritance is left for future work). We begin with a survey of the class representation patterns that occur in a set of real world JavaScript web applications, and then show how these patterns can be automatically recognized and normalized to a single consistent class representation pattern.

II. Survey and Catalog of JavaScript Encapsulation Styles

In this paper we focus on the definition and encapsulation of methods in JavaScript. We began with a survey of web tutorials [3, 17], several JavaScript resource books [16, 6, 10], and a large set of over 70 production JavaScript applications. While the styles described in the books and tutorials were interesting and elegant, we found that most production applications used styles different from those described in the books. The five styles that we found to be used in practice in the collection of web applications we surveyed can be categorized as:

1) *Inner lambda*. In the inner lambda style, anonymous (lambda) functions are assigned directly to properties of the object *inside* the constructor function to create methods. The advantage of this style is that the functions

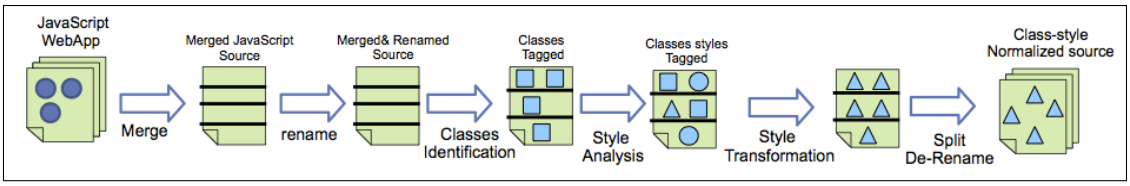

Fig. 2. Class Style Normalization Architecture

are defined at the same time as the class, and are explicitly encapsulated inside the class function. The disadvantage is that the function objects (methods) are rebuilt every time the constructor is called.

2) *Inner method*. In the inner method style, the method definitions also appear inside of the class function, but as standalone functions. In some ways, this is closest to the Java and C++ inline function class syntax for declaring methods. The name of the function appears immediately before the parenthesis, and a couple of housekeeping lines are inserted where the field properties are actually added. This similarity is however a bit deceiving, since in JavaScript the name of the property of the object is the important name, and the name of the function definition itself has no effect on the execution, and need not be the same as the name of the object's method property.
3) *Outer method*. In this style, the method definitions appear outside of the constructor (class) function as standalone functions, but are added as properties as part of the constructor. This style resembles the C++ class syntax for non-inlined functions.
4) *Outer lambda*. The outer lambda style is very similar to the outer method style, except that the method functions are created as lambda functions assigned as properties of the class constructor function, which are then copied to each created object as part of the constructor function. An advantage of this style is that the class name (constructor function name) appears with the declaration of the method functions, making the relationship explicit.
5) *Prototype lambda*. This style also defines the functions as lambda functions outside of the constructor function. But instead of storing the method functions as properties of the constructor, the functions are stored in the prototype object. This makes them available to all instances created by the constructor function through the *__proto__* chain.

III. Approach

To help alleviate the maintenance challenges posed by the inconsistency of these different styles, our plan is to recognize each of these style patterns in JavaScript applications, analyze their components, and automatically normalize them to a single preferred style. The choice of preferred style is admittedly arbitrary, but our feeling is that any consistent style is better than none. Section V proposes and justifies a particular choice.

The architecture of our normalization system is shown in Figure 2. Our system is implemented in the TXL source transformation language [5], using the existing TXL JavaScript grammar to parse and pattern match JavaScript source. Because the representation of the various styles may cross JavaScript source file boundaries, our first step is to merge the entire source of the JavaScript application into one file that we can process as a unit ("Merge" on the left of Figure 2). To assist in the recreation of original files, comments are added to encode the locations of the original file boundaries so that, when our process is complete, we can recreate the original source files after style normalization ("Split De-rename" on the right of Figure 2).

As in many reverse engineering tasks, unique renaming is applied so that function and variable names are globally unique before processing ("Rename" in Figure 2). In this way there is no confusion between local and global variables of the same name in different scopes or files. We rename all functions (including class constructor functions), parameters and variables to reflect their scope. Outer level functions are given a prefix to indicate that they are the top level of the application, and inner functions, parameters, variables and anonymous functions are renamed with that prefix combined with the outer function name as well as the name of the inner function itself. Since function expressions (anonymous functions) do not have names, we give them the standard name "Lambda" along with a number that indicates the function's order in the class so it will be uniquely named.

As a running example, we use a small class function from the *Wiso* web application [13] (Figure 3). Wiso is an AJAX-based open source web application and framework written in HTML, JavaScript and CSS. It can be a standalone application, or part of a 2.5D (isometry) game engine for web games. Wiso consists of 9 large JavaScript files, which after running the merging script are combined into one. This file is then parsed using the TXL JavaScript grammar and processed by our JavaScript unique naming transformation. The output for our small example constructor function is shown in Figure 4. Formatting is standardized based on the parse, and functions, parameters and variables are renamed to reflect the scope to which they belong. For example, the anonymous function assigned to *this.readyStateChange()* has become *_qcam_WAjaxRequest_Lambda_1()*.

The core of our process is the three central steps - "Class Identification", "Style Analysis" and "Style Transformation", described in detail in the following sections.

IV. Class Identification and Style Analysis

After merging the source files of the application and globally renaming functions, parameters and variables as described in the previous section, we then mark up the merged source

```
function WAjaxRequest (url, obj) {
 var xhr = WAjaxRequest.getComponent();

 this.readyStateChange = function() {
    if (xhr.readyState == 1) {
       if (obj.onLoading != null)
       obj.onLoading();
    }
    else if (xhr.readyState == 4) {
       if(xhr.status >= 200 && xhr.status < 300) {
       ...
    }
 }

 if (obj.method != null) {
    xhr.onreadystatechange = this.readyStateChange;

    if (obj.asynchronous == null)
       obj.asynchronous = true;

    xhr.open(obj.method, url, obj.asynchronous);
    ...
 }
}

WAjaxRequest.getComponent = function() {
  ...
}
```

Fig. 3. An example JavaScript class function from the Wiso web application

```
function _qcsm_WAjaxRequest (_qcsm_WAjaxRequest_url, _qcsm_WAjaxRequest_obj)
{
   var _qcsm_WAjaxRequest_xhr = WAjaxRequest.getComponent ();
   this.readyStateChange = function _qcsm_WAjaxRequest_Lambda1 ()
   {
      if (_qcsm_WAjaxRequest_xhr.readyState == 1) {
         if (_qcsm_WAjaxRequest_obj.onLoading != null)
            _qcsm_WAjaxRequest_obj.onLoading ();
      }
      else if (_qcsm_WAjaxRequest_xhr.readyState == 4) {
         if (_qcsm_WAjaxRequest_xhr.status >= 200 && _qcsm_WAjaxRequest_xhr.
               status < 300) {
         ...
      }
   }
   if (_qcsm_WAjaxRequest_obj.method != null) {
      _qcsm_WAjaxRequest_xhr.onreadystatechange = this.readyStateChange;
      if (_qcsm_WAjaxRequest_obj.asynchronous == null)
         _qcsm_WAjaxRequest_obj.asynchronous = true;
      _qcsm_WAjaxRequest_xhr.open (_qcsm_WAjaxRequest_obj.method,
         _qcsm_WAjaxRequest_url, _qcsm_WAjaxRequest_obj.asynchronous);
      ...
   }
}
_qcsm_WAjaxRequest.getComponent = function _qcsm_Lambda1 ()
{
  ...
}
```

Fig. 4. The Wiso example class function after parsing and unique renaming

to tag each function with its role and attributes. The input to this stage is the merged and renamed source of the original application, shown for our small Wiso example in Figure 4.

JavaScript does not distinguish between class constructors and functions. A function may be an object constructor, an object method, or simply an independent or routine function. We will refer to a function that has properties and methods inside of it as a *class function*, and a function that is either inside or a property of that function as a *method function*. The main goal of this stage is to identify class and method functions, and to recognize the encapsulation patterns by classifying and tagging the method implementation style in the recognized class functions.

Somewhat surprisingly, in practice we have found that one class function may use a number of different method styles, and so may be tagged with more than one style pattern. For example, a single class function may use both the inner lambda style (where anonymous method function objects are assigned as properties inside the class function) and the prototype lambda style (where anonymous method function objects are assigned as properties outside the class function) in the same constructor, as is the case in the Wiso example

```
function _qcsm_WAjaxRequest (_qcsm_WAjaxRequest_url, _qcsm_WAjaxRequest_obj)
{
   var _qcsm_WAjaxRequest_xhr = WAjaxRequest.getComponent ();
   this.readyStateChange = function
         <<<< _qcsm_WAjaxRequest readyStateChange InnerLambda >>>>
         _qcsm_WAjaxRequest_Lambda1 ()
   {
      if (_qcsm_WAjaxRequest_xhr.readyState == 1) {
         if (_qcsm_WAjaxRequest_obj.onLoading != null)
            _qcsm_WAjaxRequest_obj.onLoading ();
      }
      else if (_qcsm_WAjaxRequest_xhr.readyState == 4) {
         if (_qcsm_WAjaxRequest_xhr.status >= 200 && _qcsm_WAjaxRequest_xhr.
               status < 300) {
         ...
      }
   }
   if (_qcsm_WAjaxRequest_obj.method != null) {
      _qcsm_WAjaxRequest_xhr.onreadystatechange = this.readyStateChange;
      if (_qcsm_WAjaxRequest_obj.asynchronous == null)
         _qcsm_WAjaxRequest_obj.asynchronous = true;
      _qcsm_WAjaxRequest_xhr.open (_qcsm_WAjaxRequest_obj.method,
         _qcsm_WAjaxRequest_url, _qcsm_WAjaxRequest_obj.asynchronous);
      ...
   }
}
_qcsm_WAjaxRequest.getComponent = function
      <<<< _qcsm_WAjaxRequest.getComponent getComponent OuterLambda >>>>
      _qcsm_Lambda1 ()
{
  ...
}
```

Fig. 5. The Wiso example after class analysis and style identification

(Figure 3). In the collection of open source web applications we have analyzed (Section VI), we found that applications often use multiple style patterns in one class function, which strengthens our motivation since having multiple class styles in one application can significantly complicate understanding and maintenance.

We use TXL rules to tag instances of each style in the parsed and renamed JavaScript source with a special markup, using TXL patterns and constraints to implement recognition of each of the five style patterns enumerated in Section II. Because JavaScript may be mixed with XML, we chose a markup syntax based on <<<< and >>>> brackets, which are not legal syntax in JavaScript or XML. The markup encodes a relationship of our model, relating a class function name, a method function name, and a style pattern. For example:

<<<< ClassName MethodName patternStyle >>>>.

The TXL rules search for functions that contain other functions and function object properties to identify and distinguish class functions. Each class function is then matched against each of the style patterns described in Section II, and each method function of the class is marked with its style using the markup above. For example, the TXL rules look for a method function defined inside the scope of a class function using an anonymous function expression in order to mark up the *inner lambda* style. The attributes of the method functions in the corresponding meta-model help in characterizing the patterns we are looking for.

The marked-up result of running our class pattern identification on the parsed and renamed Wiso example of Figure 4 is shown in Figure 5. In this example, for the class function *WAjaxRequest*, the property *readyStateChange()* has been tagged as an inner lambda method, and the property *getComponent()* has been tagged as an outer lambda method.

V. Style Normalization

Once we have identified all of the class and method functions in the JavaScript application and marked up each method

```
function _qcsm_WAjaxRequest (_qcsm_WAjaxRequest_url, _qcsm_WAjaxRequest_obj)
{
   this._qcsm_WAjaxRequest_xhr = WAjaxRequest.getComponent ();
   if (_qcsm_WAjaxRequest_obj.method != null) {
      this._qcsm_WAjaxRequest_xhr.onreadystatechange = this.readyStateChange;
      if (_qcsm_WAjaxRequest_obj.asynchronous == null)
         _qcsm_WAjaxRequest_obj.asynchronous = true;
      this._qcsm_WAjaxRequest_xhr.open (_qcsm_WAjaxRequest_obj.method,
         _qcsm_WAjaxRequest_url, _qcsm_WAjaxRequest_obj.asynchronous);
      ...
   }
}
_qcsm_WAjaxRequest.prototype.readyStateChange = function
      _qcsm_WAjaxRequest_Lambda1 ()
{
   if (this._qcsm_WAjaxRequest_xhr.readyState == 1) {
      if (_qcsm_WAjaxRequest_obj.onLoading != null)
         _qcsm_WAjaxRequest_obj.onLoading ();
   }
   else if (this._qcsm_WAjaxRequest_xhr.readyState == 4) {
      if (this._qcsm_WAjaxRequest_xhr.status >= 200 && this.
            _qcsm_WAjaxRequest_xhr.status < 300) {
      ...
   }
}

_qcsm_WAjaxRequest.prototype.getComponent = function _qcsm_Lambda1 ()
{
   ...
}
```

Fig. 6. The Wiso example JavaScript class function after normalization to prototype lambda style

function with its style pattern, we are ready to transform the program to normalize to a single style. The question arises as to which style we should choose? As observed in Section II, the style most similar to Java and C++ is probably the *inner method* style. However, in JavaScript both the *inner method* and the *inner lambda* styles have the drawback that every time a new object is constructed, the method functions must be recreated in the new instance. While conceptually elegant, this is very inefficient.

Another possibility is the *outer method* style, which is very similar to the C++ style for non-inlined functions. However, the drawback of both the *outer method* and the *outer lambda* style in JavaScript is that they are implemented in the global namespace, which means that the program will increase in static size and, as programs grow, the chances of a naming conflict will rise rapidly.

All of these drawbacks can be resolved by delegating the method to the prototype of the class function, as in the *prototype lambda* style pattern. JavaScript is by nature a prototype-based language, and thus the *prototype lambda* style is also the most consistent with its original design philosophy. In addition, prototypes are used in inheritance, so all method functions that are properties of the prototype of a class function are public and can be accessed and modified by child class functions. Thus, while we could transform to normalize to any of the styles, we have selected the *prototype lambda* style to be the target of our normalizing transformation.

After all method functions are tagged with their style, the normalizing transformation is carried our by another TXL program that looks for instances of each style markup, and transforms the marked-up method function to the *prototype lambda* style. Since the prototype lambda style pattern defines method functions outside the scope of the class function, inner method and inner lambda pattern methods are first extracted and relocated outside the class function, effectively transforming them to outer method and outer lambda form. As part of this step, variables bound as closures in the functions

```
function WAjaxRequest (url, obj)
{
   this.xhr = WAjaxRequest.getComponent ();
   if (obj.method != null) {
      this.xhr.onreadystatechange = this.readyStateChange;
      if (obj.asynchronous == null)
         obj.asynchronous = true;
      this.xhr.open (obj.method, url, obj.asynchronous);
      ...
   }
}

WAjaxRequest.prototype.readyStateChange = function ()
{
   if (this.xhr.readyState == 1) {
      if (obj.onLoading != null)
         obj.onLoading ();
   }
   else if (this.xhr.readyState == 4) {
      if (this.xhr.status >= 200 && this.xhr.status < 300) {
      ...
   }
}
WAjaxRequest.prototype.getComponent = function ()
{
 ...
}
```

Fig. 7. The final normalized Wiso example JavaScript class function after derenaming and splitting

Application Name (LOC)	AdaptCSM (62152)	JSCook (1827)	Chinese TiddlyWiki (27963)	Freecivnet (23141)	QuickConnect (15894)	Wiso (829)
Prototype Lambda	14	14	548	62	5	56
Inner Lambda	10	0	36	28	156	12
Outer Lambda	58	0	27	2	0	3
Outer Method	0	0	0	22	0	0
Inner Method	0	0	0	0	0	0
Total Transformed Pattern	68	0	63	52	156	15

TABLE I
SUMMARY OF JAVASCRIPT APPLICATIONS TESTED

are transformed into properties. A second step then transforms the outer method forms to outer lambda, and finally a third step transforms the outer lambda forms to prototype lambda form. Each step is implemented as a TXL source transformation matching the parsed, renamed and tagged source output by our style identification phase.

Figure 6 shows the result of running the prototype lambda normalizing transformation on the tagged Wiso example of Figure 5. As an example, the method function assigned to property *readyStateChange*, identified and tagged as an inner lambda, has been extracted and transformed to prototype lambda style in the result.

The final step of our process is the de-renaming of the source to restore original names, and the splitting of the merged source into the original JavaScript file structure, in the case of Wiso the nine separate source files. Figure 7 shows the final result for the small example Wiso class function.

VI. EVALUATION

In order to test our approach, we gathered several open source web applications of varying application domains and sizes, ranging from about 500 to about 62,000 lines of code. Table I summarizes the application name, number of lines of code, number of instances of each class style pattern we identified in the application, and finally the number of patterns we were able to transform to prototype lambda form. In every case we were able to identify and normalize all identified patterns in the applications.

Strangely, although both the web tutorials and the reference books recommend the inner method style, as we can see from the styles listed in Table I, to our surprise none of the test applications that we gathered actually used this style. We have since found other examples using this style that we will be processing in future work.

The performance of our system varied little from small to large JavaScript applications, consistently processing programs in less than a minute, and our framework scales well to realistic JavaScript programs. The test applications also had a mix of class patterns, showing that our method can deal with mixed styles. In some cases programs were already using the target pattern (prototype lambda), in which case the transformation did not make any changes after style identification.

For example, the application *Freecivnet* has four patterns: prototype lambda, inner lambda, outer lambda and outer method. Inner lambda appears 28 times in the application, outer lambda appears 2 times, outer method appears 22 times, and prototype lambda appears 62 times. All inner lambda, outer lambda and outer method patterns were transformed successfully to prototype lambda and obviously the three prototype lambda instances were left in their original form.

VII. Related work

While the potential maintenance problems associated with type and style consistency in JavaScript have been studied by several other researchers [1, 14], there seems to be remarkably little other work on JavaScript style improvement. Perhaps the closest other work to ours has been in migration from procedural to object-oriented language paradigms, such as that by Zou et al. [19], and Sneed et al. [15]. As pointed out by Ciupke [4], the biggest challenge in this kind of reengineering is the identification of appropriate classes and their representation. By contrast, in our work we are able to use a lightweight pattern-based identification and automated transformation, because even though the patterns we are looking for differ widely in structure and appearance, they already represent conceptually similar object-oriented class models.

Many authors have discussed the normalization of Cobol dialects [7, 8], which is similar to our problem and has the same goal: easier long-term maintenance. One dialect translation effort that bears strong similarity to our work is that of Ekman and Hedin [9], who automatically recognize patterns in Java code that can be better expressed using newer Java features, in a similar way to our recognition of JavaScript class patterns that can be expressed using a different style.

Malton [11] describes a general transformation-based mechanism for dialect conversion, API migration and language migration that is the abstract model for our technique. The work of Ceccato et al. [2] on GOTO elimination in legacy code bears some similarity to our work, since they use TXL in a similar way to identify and transform GOTO patterns. ReAJAX is a tool aimed at improving maintainability and comprehension of complex Ajax applications [12] in a way similar to our work.

VIII. Conclusion

This paper presents a class normalization framework for JavaScript designed to increase stylistic consistency and thus improve web application comprehension and maintainability. Our automated approach identifies class patterns in JavaScript applications and automatically transforms them to a chosen standard, such as the prototype lambda style, yielding a consistent and easier to maintain result.

We are presently working on processing a large number of additional web applications, and in particular those using the inner and outer method styles for class representation. While class representation is the biggest problem, JavaScript also allows for a number of different expressions of the inheritance concept, and in future we hope to attack normalization of inheritance styles as well.

Acknowledgments

This work is funded in part by the Natural Sciences and Engineering Research Council of Canada, and by the Ministry of Higher Education, Kingdom of Saudi Arabia.

References

[1] C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for javascript. In *ECOOP*, pages 428–452, 2005.

[2] M. Ceccato, P. Tonella, and C. Matteotti. Goto elimination strategies in the migration of legacy code to Java. In *CSMR*, pages 53–62, 2008.

[3] Chriswa. Writing object-oriented JavaScript, Part 1, 2003. http://www.codeproject.com/Articles/5608/Writing-Object-Oriented-JavaScript-Part-1 (last access 25 May 2012).

[4] O. Ciupke. Automatic detection of design problems in object-oriented reengineering. In *TOOLS (30)*, pages 18–32, 1999.

[5] J. R. Cordy. The TXL source transformation language. *Sci. Comput. Program.*, 61(3):190–210, 2006.

[6] D. Crockford. *JavaScript - the good parts: unearthing the excellence in JavaScript.* O'Reilly, 2008.

[7] T. R. Dean, J. R. Cordy, K. A. Schneider, and A. J. Malton. Using design recovery techniques to transform legacy systems. In *ICSM*, pages 622–631, 2001.

[8] A. V. Deursen, P. Klint, and C. Verhoef. Research issues in the renovation of legacy systems. In *FASE*, pages 1–21, 1999.

[9] T. Ekman and G. Hedin. Automatic renovation of Java programs using ReRAGs - examples and ideas. In *ECOOP Workshop on Object-Oriented Refactoring*, 2004.

[10] D. Goodman and M. Morrison. *The JavaScript Bible (6th ed.).* Wiley, 2007.

[11] A. J. Malton. The software migration barbell. In *ASERC Workshop on Software Architecture*, 2001.

[12] A Marchetto, P. Tonella, and F. Ricca. ReAjax: a reverse engineering tool for Ajax web applications. *IET Software*, 6(1):33–49, 2012.

[13] Nikogj. Wiso, http://sourceforge.net/projects/wiso-project/ (last access 25 May 2012).

[14] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of JavaScript programs. In *PLDI*, pages 1–12, 2010.

[15] H. M. Sneed. Object-oriented Cobol recycling. In *WCRE*, pages 169–178, 1996.

[16] S. Stefanov. *JavaScript Patterns - Build Better Applications with Coding and Design Patterns.* O'Reilly, 2010.

[17] S. Stefanov. Three ways to define a JavaScript class, http://www.phpied.com/3-ways-to-define-a-javascript-class/ (last access 25 May 2012).

[18] W3C. HTML5: Differences from HTML4, http://www.w3.org/TR/2012/WD-html5-diff-20120329 (last access 25 May 2012).

[19] Y. Zou and K. Kontogiannis. Quality driven transformation compositions for object oriented migration. In *APSEC*, pages 346–355, 2002.