NORMALIZING OBJECT-ORIENTED CLASS STYLES
IN JAVASCRIPT

by

WIDD GAMA

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Master of Science

Queen’s University
Kingston, Ontario, Canada
January 2013

Copyright © Widd Gama, 2013
Abstract

JavaScript is the most widely used client-side scripting language, and has become increasingly popular as a crucial component of the AJAX technology. JavaScript is a dynamic, weakly typed, multi-paradigm programming language that supports object-oriented, imperative, and functional programming styles. Many different programmers 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. We begin by analyzing the different ways that JavaScript programmers have represented the class concept, identifying and cataloguing the different class patterns used in the language. We choose one of these, and show how it is possible to automatically migrate JavaScript applications from any mix of class styles to the chosen one, making it easier to understand and maintain object-oriented JavaScript programs.
Acknowledgments

I would like to thank my professors, Jim Cordy and Tom Dean for their continuous guidance and unlimited help that they have offered me. This work would not have been completed without your enlightenment. I would like to thank Professor Manar Alalfi for her valuable suggestions and mentoring my thesis work. And I would also like to extend my appreciation to all of my lab mates for their support.

A special thank you to the Ministry of Higher Education and their representatives in Canada, the Saudi Arabian Cultural Bureau, for providing me with a scholarship to pursue my graduate study and achieve a long-life dream.

Finally, my deepest gratitude goes to my family, my mother Fareeda, my father Adel and all of my brothers and sisters for their limitless encouragement and endless prayers that motivated and helped me achieving what I have reached so far. My husband Turki and my daughter Waard, I could not have done this without you being beside me, your great efforts to provide me with all the help you can so I can manage continuing my degree mean the world to me.

Thank you all.
Co-Authorship

A preliminary version of this work has been published in the proceedings of the 14th IEEE international symposium on Web Systems Evolution (WSE 2012), co-authored with Professors Manar Alalfi, James Cordy and Thomas Dean [41]. In this paper I am the primary author.
Table of Contents

Abstract i
Acknowledgments ii
Co-Authorship iii
Table of Contents iv
List of Tables vii
List of Figures viii

1 Introduction 1
  1.1 Motivation .............................................. 2
  1.2 Contribution ......................................... 2
  1.3 Thesis Organization .................................. 3

Chapter 2:
  Background .............................................. 4
  2.1 JavaScript Overview .................................... 4
  2.2 Software Transformation ............................... 7
  2.3 Related Work ......................................... 11
  2.4 Summary .............................................. 13

Chapter 3:
  Survey of JavaScript Class Styles .......................... 14
  3.1 JavaScript Fundamentals ............................... 14
  3.2 Catalogue of JavaScript Class Styles .................... 17
  3.3 Mixed and Uncommon Class Styles ....................... 24
  3.4 Summary .............................................. 28
Chapter 4:

Approach ................................................. 29
4.1 Overview ............................................. 29
4.2 Merge .................................................. 30
4.3 Rename ................................................ 31
4.4 Class Identification ................................. 36
4.5 Style Analysis ....................................... 38
4.6 Style Transformation ............................... 39
4.7 De-Rename and Split: ............................ 40
4.8 Summary ............................................. 41

Chapter 5:

Class Identification and Style Analysis ............... 43
5.1 Class Identification ................................. 43
5.2 Style Analysis ....................................... 45
5.3 Summary ............................................. 53

Chapter 6:

Style Normalization ................................... 54
6.1 Pattern Transformation .......................... 54
6.2 De-rename and Split ............................... 58
6.3 Summary ............................................. 60

Chapter 7:

Evaluation ................................................ 61
7.1 Selection of Test Applications .................... 61
7.2 Testing Procedure ................................ 62
7.3 Chinese Tiddly Wiki ............................... 63
7.4 T4d Tigra Menu ..................................... 67
7.5 Sudoku ............................................... 71
7.6 Summary ............................................. 74

Chapter 8:

Conclusions ............................................. 75
8.1 Summary ............................................. 75
8.2 Limitations .......................................... 76
8.3 Future Work ......................................... 76

Bibliography ........................................... 78
Appendix A:
  Unique Naming Code .............................................. 84

Appendix B:
  Class Identification Code ........................................... 90

Appendix C:
  Style Analysis Code ................................................ 92

Appendix D:
  Style Normalization Code ........................................... 100
  D.1 Class Untagging ..................................................... 100
  D.2 Style Normalization ............................................... 101
List of Tables

<table>
<thead>
<tr>
<th>Table Number</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Summary of Surveyed Web Applications that Contain Class Definitions</td>
<td>18</td>
</tr>
<tr>
<td>3.2</td>
<td>Class patterns in the surveyed web applications (part 1)</td>
<td>25</td>
</tr>
<tr>
<td>3.3</td>
<td>Class patterns in the surveyed web applications (part 2)</td>
<td>26</td>
</tr>
<tr>
<td>7.1</td>
<td>Summary of applications’ info</td>
<td>62</td>
</tr>
<tr>
<td>7.2</td>
<td>Applications’ class styles</td>
<td>62</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Prototype chain of an object in JavaScript [21]</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>TXL transformation phases [20]</td>
<td>8</td>
</tr>
<tr>
<td>4.1</td>
<td>Class Style Normalization Architecture</td>
<td>30</td>
</tr>
<tr>
<td>5.1</td>
<td>The class style adapted DMM model</td>
<td>47</td>
</tr>
<tr>
<td>7.1</td>
<td>A snapshot of the Chinese Tiddly Wiki application.</td>
<td>66</td>
</tr>
<tr>
<td>7.2</td>
<td>A snapshot of the Tigra Menu application.</td>
<td>70</td>
</tr>
<tr>
<td>7.3</td>
<td>A snapshot of the Sudoku application (1).</td>
<td>73</td>
</tr>
<tr>
<td>7.4</td>
<td>A snapshot of the Sudoku application (2).</td>
<td>73</td>
</tr>
<tr>
<td>7.5</td>
<td>A snapshot of the Sudoku application (3).</td>
<td>74</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

JavaScript [17] is a dynamic, weakly typed scripting language, used primarily in web browsers but also on the web server side. Originally developed at Netscape in 1995, it is the most commonly used mechanism to provide dynamic interaction in web applications, as well as asynchronous interaction with the server (e.g. AJAX [18]). The language supports imperative, functional and object-based styles, and is one of the core components of the upcoming HTML5 standard [42].

When used in the object-oriented style, JavaScript is a prototype-based language, sharing many concepts with the language Self [39]. In the Self approach, objects are a collection of slots (or properties), which map names to values. Values can be scalar values or object references. As in Self, there is no distinction between values and methods. Functions are first class objects in JavaScript and a method is simply a property that maps a name to a function object. A characteristic of prototype languages is that there is no explicit class declaration that provides 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. The
application can later dynamically add or remove properties from the instance.

1.1 Motivation

The flexibility of JavaScript, 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. We found that sometimes multiple techniques are used in the same program, resulting in code that is inconsistent, difficult to maintain and comprehend. In this thesis we aim to improve the maintainability and code comprehension of JavaScript applications by normalizing the different styles of classes. A consistent style makes it easier for programmers to understand and maintain, and tools to analyze.

1.2 Contribution

This work describes an approach to normalize JavaScript code to a single consistent style of object-oriented programming. In particular, we address the issue of the consistent encapsulation of data and functions as object classes. A second contribution is a survey of the class representation styles that occur in a set of real world JavaScript web applications. We have also created models that map the actual representation of each class style found in the JavaScript applications to help us characterize each style.
CHAPTER 1. INTRODUCTION

1.3 Thesis Organization

The thesis is organized as follows: Chapter 2 reviews the related work and introduces some of the JavaScript language concepts. Chapter 3 describes some of the fundamentals of JavaScript, the object-oriented code patterns that result from these concepts and presents a survey of object class patterns we have observed in a large set of JavaScript programs. Chapter 4 outlines the overall architecture of our approach, Chapter 5 describes how each class definition and class styles can be identified. Chapter 6 describes our transformations for normalizing any mix of the identified patterns in a web application to a single pattern. Chapter 7 presents an evaluation demonstrating the proposed approach. Finally, Chapter 8 concludes the thesis and discusses our plans for future work.
Chapter 2

Background

This chapter gives a basic overview of the JavaScript language, concentrating on the features related to objects and prototyping. It briefly introduces the TXL source transformation language used in our research, and reviews and contrasts with other work in JavaScript style improvement.

2.1 JavaScript Overview

JavaScript is an object-based language that uses prototypes as an abstraction mechanism. It lacks the usual explicit representation of classes in other object-oriented languages, but supports the notion using prototypes and properties. This class-free paradigm defines functions as objects and methods as properties [11]. Methods are attached to functions and defined as properties of these functions, which can be implemented inside or outside the scope of the function. Functions are objects in JavaScript, and these objects have properties [26]. Properties can be one of three kinds: data types, methods or internal properties. For example, Listing 2.1 shows
the `personName` property has a scalar value which is `name`, whereas the `showName` property is an example of a function defined as a method since `showName` refers to an anonymous function, and `Person` refers to a function defined as a class since there is an object `p` instantiated from that function.

A third kind of properties, internal properties, are not directly accessible but they are managed by the interpreter. An example is the `__proto__` property that references the prototype of an object, and can alternatively be accessed via the `Object.getPrototypeOf()` method.

The `__proto__` property is used to implement the inheritance of methods from one object to another.

**Listing 2.1 A class function and method in JavaScript.**

```javascript
function Person (name)
{
    this.personName = name;
    this.showName = function ()
    {
        return "Hello, " + this.personName;
    }
}

var p = new Person ("Mike");
var y = p.toString();
var x = p.showName();
alert (x);
```

Objects are created using the `new` keyword as shown in Listing 2.1. Each function object has a prototype object which is identified using the prototype property of the function object. When the `new` keyword is used, a new object is created, and the implicit prototype field `__proto__` is set to the prototype object. Any properties or methods added to the prototype object are thus inherited by any objects created.
using the function. Any values or methods properties added directly to the object will override any of the value or method properties of the prototype object. Delegating methods to the prototype property of an object is very powerful especially when it comes to inheritance and reusing existing objects, properties and methods.

For example, when JavaScript evaluates an expression to retrieve or write a property of an object, first, JavaScript looks in the object itself to see whether the property has a local value assigned to the current object; if not it searches the prototype chain. It starts by looking in prototype property of the object’s constructor to see if the property is defined there and goes up in the chain until the property is found or it reaches the root prototype (the native "Object" object). Any created object is associated with the constructor function’s prototype.

For example, Listing 2.1 shows the relationship between the Person function, the
Person prototype and the person object $p$. In this case, the `showName` function has been added as a property of the object $p$ inside the function `Person`. Thus the reference to `showName` in the second to last line of the listing refers to this property. The method `toString` is not defined as a property of $p$, nor was it defined as a property of the `Person.prototype` object, so it is inherited from `Object.prototype`.

### 2.2 Software Transformation

There are many software transformation languages such as: ASF+SDF [19], Stratego [35], TXL [8] and Maude [7]. We use the TXL language to write the code transformation programs in because it is one of the simplest languages to use but any transformation language could have been used. TXL is a transformation language that transforms code from one language to another, or in our case the transformation is done to the same language, JavaScript, but to different pattern.

TXL transformation has three phases as shown in Figure 2.2 [20]: Parse, transform and unparse. First, the parser takes the input and generates tokens from that input and then parses it based on the specified user-defined grammar of the source language, TXL grammar for JavaScript in our case. Next, the transform phase which takes the parsed input generated from the first phase and transforms it into a new tree in conformance to the desired output. Finally the last step, unparse which takes the generated tree by the transform phase and unparses it to generate the output text.

Any TXL program consists of two parts: the user-defined grammar and a set of TXL transformation rules or functions, Listing 2.2 shows a snippet of the used grammar in our work. The JavaScript grammar starts with a definition of the non-terminal `program` that is defined by another non-terminal `[js program]`. The
The basic unit of the TXL grammar is the `define` statement, each statement consist of one or more non-terminal type as shown in Listing 2.2. The grammar consists of non-terminal and terminal types, non-terminals like `program`, `js declaration or statement`. TXL also has a number of predefined non-terminals for example: `id` that defines identifiers, `stringlit` that defines a string enclosed between quotes. On the other hand, terminals are like the keywords, brackets, etc..
Listing 2.2 TXL grammar for JavaScript.

% Comments are /* */ and // to end of line
comments
  //
  /* */
end comments

% Compound symbols of JavaScript
compounds
  % can’t make /= a compound since it conflicts with js_regexp
  == += -= *= ++ -- ’%= !|= &| |< |<=
  <<= >>= >>>= &= ^= |= === !==
end compounds

% Keywords of JavaScript
keys
  abstract
  break byte case catch
  continue default do double else extends final finally
  for ’function goto if implements ’import in instanceof
  interface long
  new package private protected public
  return
  switch synchronized throw throws transient
  try var void while with
end keys

% JavaScript Programs
define program
  [js_program]
end define

define js_program
  [repeat js_declaration_or_statement]
end define

define js_subscope
  [repeat js_declaration_or_statement]
end define

define js_declaration_or_statement
  [js_declaration] [NL]
  | [opt js_label] [js_statement] [NL]
#ifdef COMMENTS
  | [comment] [NL]
#endif
end define
...

CHAPTER 2. BACKGROUND

9
Listing 2.3 Example of a TXL rule that tags prototype lambda class style.

```txl
class prototypeLambda {
  prototype Lambda
}
```

The second part of a TXL program is the transformation rules or functions, these
CHAPTER 2. BACKGROUND

rules or functions define what should be modified from the input and the way it is suppose to be transformed into. The difference between rules and functions is that rules are applied recursively until no matches are found whereas functions find the first match and stop. An example of a TXL rule is shown in Listing 2.3. Rules and functions have the same structure, they differ in the keyword rule and function. Both have a replacement and a pattern and they can take parameters. When the specified pattern is matched the replacement part is done and the transformation is completed. For example, in Listing 2.3 the goal of the rule [tag_ js_ prototype] is to produce a tree of the type [js _ assignment _ expn] , the rule searches for the specified pattern (that is specified after the replace statement) and replaces it by the replacement part that is specified after the by clause, the rule ends by the statement end rule.

2.3 Related Work

While the potential maintenance problems associated with type and style consistency in JavaScript have been studied and measured by several other researchers [1, 27], there seems to be remarkably little other work on JavaScript style improvement.

On the face of it, 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. [43], and Sneed et al. [30]. In both these cases, however, the objective was to extract and document the detailed design of the existing code, identify opportunities for object-oriented redesign, and then redeploy the new design in an object-oriented implementation. As pointed out by Ciupke [6], 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.

In many ways the JavaScript object-oriented style normalization problem is similar to dialect reduction, in which programs written in local or specialized dialects of a language are reduced to the core language. This was the original application of the TXL transformation system we have used in our work [10]. Many authors have discussed the normalization of Cobol dialects [12, 40], which is very 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 [15], 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 better expressed using a different style.

Malton [23] describes a general transformation-based mechanism for dialect conversion, API migration and language migration that is the abstract model for our technique. Malton’s migrations are done in three phases: source normalization, blind translation and target source optimization. Similarly, our work normalizes by parsing, renaming and identifying each class pattern, transforms the marked-up normalized code, and then derenames and splits the target result.

The work of Ceccato et al. [4] on Goto elimination in legacy code bears some similarity to our work. In their case they began by surveying four different previously introduced strategies to eliminate Goto statements in legacy systems, and then reviewed a large part of the legacy code to manually identify the Goto patterns that
existed in the system. Similarly, we began by surveying the literature on class styles in JavaScript and then studied a number of JavaScript applications in order to enumerate the class patterns that are actually used by developers. And like us, they used TXL to implement their identification and transformation of Goto patterns.

ReAjax is a tool used to do a design recovery in Ajax applications [24]. The purpose of making such a tool is to improve maintainability and comprehension of complex Ajax applications. They begin by doing a dynamic analysis to generate a Finite State Machine, and then explore under- and over-approximations of the FSM that can be controlled by manual refinement to yield an understandable model of the application.

2.4 Summary

This chapter gives an overview about JavaScript and discusses some of the characteristics of the language. It also reviews the related work to this work and highlights the differences between our approach and the mentioned researches. Finally, we give an overview of the software transformation language TXL. The next chapter talks about some of the fundamentals of JavaScript and provides a survey of different resources of the JavaScript class styles.
Chapter 3

Survey of JavaScript Class Styles

The previous chapter introduces JavaScript object-based and prototype concepts of the language, gives an overview of TXL the software transformation language that we use. Finally, it reviews the related work and compares our approach and motivation to other researches. This chapter discusses the JavaScript encapsulation fundamentals and surveys some real life applications to categorize the most common used class styles.

3.1 JavaScript Fundamentals

In JavaScript functions are objects that can be assigned and manipulated. They have properties such as length. They can be assigned to variables and passed as arguments. Any variable with a reference to a function can be turned into a function call simply by appending parenthesis with arguments (e.g. if $f$ is such a variable, then $f(p)$ calls the function with argument $p$).

When resolving object property references such as $obj.prop$ to values, the JavaScript
CHAPTER 3. SURVEY OF JAVASCRIPT CLASS STYLES

interpreter first looks inside the object for the property. If it is not there, it checks for the 
_proto_ property. This property points to another object that is considered to be the prototype of the object. The interpreter checks inside the prototype object for a property of the name. If it is not found, it continues to follow the _proto_ chain to the prototype of the prototype, and so on until it reaches the end. If the name has not been resolved to a value by that point, then an error is generated.

Objects in JavaScript are created using the new operator on a function reference (e.g., var myObj = new MyFun();) The new operator creates a new object, points the _proto_ property at the same object as the prototype property of the function, and calls the function to initialize the new object (assigning the new object’s properties to the this local variable of the function).

In general, this pattern is similar to most object-oriented languages. The fundamental difference is in how methods are associated with objects. In most object-oriented languages, such as Java [2] and C++ [36], the assignment of methods to objects is part of the syntax, in which methods are nested inside a class and all instances have the same methods. In JavaScript the functions that will act as methods must be explicitly assigned to either a property of the object or a property of one of the objects in the _proto_ property chain. Any function can be assigned to an object, and in fact can be assigned to objects of completely different classes. When the function is invoked as a method, the local variable this of the function is bound to the object.

Listing 3.1, 3.2 and 3.3 show examples of three of the many styles of encapsulation of data and methods possible in JavaScript. All three are almost identical in semantics. The first style (Listing 3.1) is the 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).

**Listing 3.1** Data and Method Encapsulation in JavaScript: Object class variant 1

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

**Listing 3.2** Data and Method Encapsulation in JavaScript: Object class variant 2

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

**Listing 3.3** Data and Method Encapsulation in JavaScript: Object class variant 3

```javascript
function Class () {
  this.x = 4;
}
Class.prototype.m = function m() {
  
}
c = new Class();
```
The second variant (Listing 3.2) uses the function’s prototype object. 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 example (Listing 3.3) 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 \( \text{Class} \) constructor function will share this same method, using the \texttt{._proto_.} chain. While not identical to the first two in semantics, the differences are not important for most JavaScript code in practical use today.

### 3.2 Catalogue of JavaScript Class Styles

We have surveyed several resources such as JavaScript books [33, 11, 17], web tutorials [5, 34, 29, 14, 32, 37, 21, 28, 16, 25] and a set of 40 JavaScript real-life applications that range in size from 800 to 85,000 lines of code. 14 of the 40 applications have class definitions, Table 3.1 summarizes the applications that have classes in them.

For example, the application AdaptCSM has 62,321 lines of code and a total of 8 class definitions. When we analyzed these applications, we observed that several of the theoretical styles outlined in the JavaScript reference books are not actually used in the practical applications, nor do they appear in the practical web tutorials. In addition, some of the most popular styles in the web tutorials are also not used in the real applications. The styles that we found that are used in the web applications and web tutorials can be categorized as shown in Table 3.2 and 3.3.

These common patterns are: inner method, outer method, inner lambda, outer
Table 3.1: Summary of Surveyed Web Applications that Contain Class Definitions

<table>
<thead>
<tr>
<th>Application Name</th>
<th>No. of classes</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adapt CSM</td>
<td>8</td>
<td>62321</td>
</tr>
<tr>
<td>Chinese Tiddly Wiki</td>
<td>63</td>
<td>30407</td>
</tr>
<tr>
<td>DScripted Toolkit for JavaScript</td>
<td>1</td>
<td>972</td>
</tr>
<tr>
<td>Edit Area</td>
<td>3</td>
<td>7502</td>
</tr>
<tr>
<td>FCKEditor</td>
<td>218</td>
<td>31582</td>
</tr>
<tr>
<td>Freeivnet freeivweb</td>
<td>12</td>
<td>14907</td>
</tr>
<tr>
<td>FROG</td>
<td>2</td>
<td>28665</td>
</tr>
<tr>
<td>GARC eLearning System</td>
<td>2</td>
<td>61664</td>
</tr>
<tr>
<td>GLUE Script</td>
<td>6</td>
<td>2696</td>
</tr>
<tr>
<td>JSCook menu</td>
<td>3</td>
<td>2049</td>
</tr>
<tr>
<td>Porcupine</td>
<td>1</td>
<td>25098</td>
</tr>
<tr>
<td>Quick Connect</td>
<td>20</td>
<td>17293</td>
</tr>
<tr>
<td>T4d Tigra Menu</td>
<td>10</td>
<td>6678</td>
</tr>
<tr>
<td>Tiny MCE</td>
<td>1</td>
<td>85202</td>
</tr>
<tr>
<td>Wiso</td>
<td>6</td>
<td>829</td>
</tr>
<tr>
<td>Sudoku</td>
<td>1</td>
<td>1136</td>
</tr>
</tbody>
</table>
lambda and finally the prototype lambda.

### 3.2.1 Inner Method

In this style the definition of the method appears inside the scope of the class and the method itself takes the form of a standalone function. Listing 3.4 shows the inner method style. Function `getArea()` and function `getCircumference()` appear inside function `circle()`. The assignment of the function object to the property ( `this.getArea = getArea;` ) is separated from the definition of the function ( `function getArea ()` ). In some ways, this is the closest to the Java method 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 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.
Listing 3.4 Inner method class style

```javascript
function circle (radius)
{
    this.radius = radius;
    this.getArea = getArea;
    this.getCircumference = getCircumference;
    function getArea ()
    {
        return (this.radius * this.radius * 3.14);
    }
    function getCircumference ()
    {
        var diameter = this.radius * 2;
        var circumference = diameter * 3.14;
        return circumference;
    }
}
var bigCircle = new circle (100);
var smallCircle = new circle (2);
alert (bigCircle.getArea ());
alert (smallCircle.getCircumference ());
```

3.2.2 Outer Method

In the outer method style the definition of the method appears outside the scope of the class and takes the standalone function form the same form as the inner method form and the method should be added to the property of the class as well. Listing 3.5 shows an example of this style. Like the inner method style, the constructor function contains the `this.method = method;` assignments to add the function objects as method properties. This style resembles the C++ class syntax for non-inlined functions. Function `getArea()` and function `getCircumference()` appear outside function `circle()`.
CHAPTER 3. SURVEY OF JAVASCRIPT CLASS STYLES

Listing 3.5 Outer method class style

```javascript
function circle (radius)
{
    this.radius = radius;
    this.getArea = getArea;
    this.getCircumference = getCircumference;
}
function getArea ()
{
    return (this.radius * this.radius * 3.14);
}
function getCircumference ()
{
    var diameter = this.radius * 2;
    var circumference = diameter * 3.14;
    return circumference;
}
var bigCircle = new circle (100);
var smallCircle = new circle (2);
alert (bigCircle.getArea ());
alert (smallCircle.getCircumference ());
```

3.2.3 Inner Lambda

In this style the method appears inside the scope of the class and takes the form of anonymous function (lambda) and it is assigned directly to the a property of the class. Listing 3.6 shows an example of this style. The function `circle` defines the class, creating a `radius` property which stores the radius of the circle and `getArea` and `getCircumference` properties which provide two methods.
CHAPTER 3.  SURVEY OF JAVASCRIPT CLASS STYLES

Listing 3.6 Inner lambda class style

function circle (radius)
{
    this.radius = radius;
    this.getArea = function ()
    {
        return (this.radius * this.radius * 3.14);
    }
    this.getCircumference = function ()
    {
        var diameter = this.radius * 2;
        var circumference = diameter * 3.14;
        return circumference;
    }
}
var bigCircle = new circle (100);
var smallCircle = new circle (2);
alert (bigCircle.getArea ());
alert (smallCircle.getCircumference ());

3.2.4 Outer Lambda

The outer lambda style (Listing 3.7) 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 inside the constructor function. One advantage of this style is that the class name (constructor function name) appears with the declaration of the method functions, making the relationship explicit.
Listing 3.7 Outer lambda class style

```javascript
function circle (radius) {
    this.radius = radius;
    this.getArea = circle.getArea;
    this.getCircumference = circle.getCircumference;
}
circle.getArea = function () {
    return (this.radius * this.radius * 3.14);
}
circle.getCircumference = function () {
    var diameter = this.radius * 2;
    var circumference = diameter * 3.14;
    return circumference;
}
var bigCircle = new circle (100);
var smallCircle = new circle (2);
alert (bigCircle.getArea ());
alert (smallCircle.getCircumference ());
```

### 3.2.5 Prototype Lambda

The prototype lambda style defines the method outside the scope of the class as an unnamed method, and the method is assigned to the prototype object of the class instead of assigning the method to a property of the class. This makes this method style available to all instances created by the constructor function through the `__proto__` chain. Listing 3.8 shows an example of this programming style.
Listing 3.8 Prototype lambda class style

```javascript
function circle (radius) {
  this.radius = radius;
}
circle.prototype.getArea = function () {
  return (this.radius * this.radius * 3.14);
}
circle.prototype.getCircumference = function () {
  var diameter = this.radius * 2;
  var circumference = diameter * 3.14;
  return circumference;
}
var bigCircle = new circle (100);
var smallCircle = new circle (2);
alert (bigCircle.getArea ());
alert (smallCircle.getCircumference ());
```

### 3.3 Mixed and Uncommon Class Styles

Some of the surveyed applications use more than one class style. For example: the application AdaptCSM in Table 3.2 has 174 instances of the prototype lambda pattern, 10 instances of the inner lambda pattern and 58 instances of the outer lambda pattern.

Some of the applications have different styles that are uncommon and not used in other applications, for example Listing 3.9 shows an example of the application JavaScript Jabber Messenger that uses the `extend` built-in function that copies/extends the prototype of a subclass from the base class, and Listing 3.10 shows an example of the application Logical Doc that uses the `create` built-in function to create a class.
### Table 3.2: Class patterns in the surveyed web applications (part 1)

<table>
<thead>
<tr>
<th>AppName</th>
<th>No of Patterns</th>
<th>Pattern Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>AdaptCSM</td>
<td>174</td>
<td>Prototype Lambda</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>Inner Lambda</td>
</tr>
<tr>
<td></td>
<td>58</td>
<td>Outer Lambda</td>
</tr>
<tr>
<td>Chinese Tiddly Wiki</td>
<td>548</td>
<td>Prototype Lambda</td>
</tr>
<tr>
<td></td>
<td>36</td>
<td>Inner Lambda</td>
</tr>
<tr>
<td></td>
<td>27</td>
<td>Outer Lambda</td>
</tr>
<tr>
<td>Dscripted</td>
<td>11</td>
<td>Prototype Lambda</td>
</tr>
<tr>
<td>Edit Area</td>
<td>95</td>
<td>Prototype Lambda</td>
</tr>
<tr>
<td>FCKeditor</td>
<td>532</td>
<td>Prototype Lambda</td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>Outer Lambda</td>
</tr>
<tr>
<td>Freeeivnet</td>
<td>62</td>
<td>Prototype Lambda</td>
</tr>
<tr>
<td></td>
<td>28</td>
<td>Inner Lambda</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Outer Lambda</td>
</tr>
<tr>
<td></td>
<td>22</td>
<td>Outer Method</td>
</tr>
<tr>
<td>FROG</td>
<td>26</td>
<td>Prototype Lambda</td>
</tr>
<tr>
<td>GARC</td>
<td>16</td>
<td>Prototype Lambda</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>Inner Lambda</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Outer Lambda</td>
</tr>
<tr>
<td>GLUE script</td>
<td>2</td>
<td>Prototype Lambda</td>
</tr>
<tr>
<td>JSCook</td>
<td>14</td>
<td>Prototype Lambda</td>
</tr>
<tr>
<td>porcupine</td>
<td>15</td>
<td>Prototype Lambda</td>
</tr>
<tr>
<td></td>
<td>96</td>
<td>Outer Lambda</td>
</tr>
<tr>
<td>Quick Connect</td>
<td>5</td>
<td>Prototype Lambda</td>
</tr>
</tbody>
</table>
Table 3.3: Class patterns in the surveyed web applications (part 2)

<table>
<thead>
<tr>
<th>AppName</th>
<th>No of Patterns</th>
<th>Pattern Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quick Connect</td>
<td>165</td>
<td>Inner Lambda</td>
</tr>
<tr>
<td>T4d Tigrar Menu</td>
<td>35</td>
<td>Outer Method</td>
</tr>
<tr>
<td>Tiny MCE</td>
<td>13</td>
<td>Prototype Lambda</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>Outer Lambda</td>
</tr>
<tr>
<td>Wiso</td>
<td>56</td>
<td>Prototype Lambda</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>Outer Lambda</td>
</tr>
<tr>
<td>Sudoku</td>
<td>5</td>
<td>Prototype Lambda</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>Inner Lambda</td>
</tr>
</tbody>
</table>

Listing 3.9 An example of uncommon class style

Ext.state.Provider = function ()
{
    this.addEvents ("statechange");
    this.state = {};
    Ext.state.Provider.superclass.constructor.call (this)
};
Ext.extend (Ext.state.Provider, Ext.util.Observable,
{
    get : function (B, A)
    {
        return typeof this.state [B] == "undefined" ? A : this.state [B]
    },
    clear : function (A)
    {
        delete this.state [A];
        this.fireEvent ("statechange", this, A, null)
    },
    set : function (A, B)
    {
        this.state [A] = B;
        this.fireEvent ("statechange", this, A, B)
    }
},...

CHAPTER 3. SURVEY OF JAVASCRIPT CLASS STYLES

Listing 3.10 Another example of uncommon class style

```javascript
Ajax.Base = Class.create({
  initialize : function (options)
  {
    this.options = {method : "post", asynchronous : true, contentType :
    "application/x-www-form-urlencoded", encoding : "UTF-8",
    parameters : ",", evalJSON : true, evalJS : true};
    Object.extend (this.options, options || {});
    this.options.method = this.options.method.toLowerCase ();
    if (Object.isString (this.options.parameters))
    {
      this.options.parameters = this.options.parameters.toQueryParams ()
    }
    else
    {
      if (Object.isHash (this.options.parameters))
      {
        this.options.parameters = this.options.parameters.toObject ()
      }
    }
  }
}
...
```

3.3.1 Style Comparisons

JavaScript programmers vary widely in which of these styles they prefer, and there is no clear winner in the systems we studied. The style used by one programmer may be unfamiliar or confusing to another. In practice, many JavaScript applications use a mixture of these styles, which makes understanding and maintaining these applications even more challenging.

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.
The advantage of this style is that the functions are defined at the same time as the variable, and are explicitly encapsulated inside the class function.

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 styles 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 (Listing 3.8). Since it is based on Self, 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.

### 3.4 Summary

This chapter discusses some of the fundamentals of JavaScript and encapsulation styles based on a survey of tutorials, books and real life applications. Some of the styles were found in many applications and some uncommon ones were limited to one application. The next chapter describes the approach that is proposed to solve the maintenance problem and complexity of code comprehension that are result of having multiple method patterns in JavaScript applications.
Chapter 4

Approach

The previous chapter outlined the problems with inconsistency of class styles in JavaScript programs and enumerated a number of commonly used different class patterns. In this chapter, we outline our approach to help alleviate the maintenance challenges that were discussed in Chapter 3.

4.1 Overview

We first identify class functions in JavaScript. Once identified, we analyze the class functions to recognize each of the method style patterns and automatically normalize them into a single preferred style. The choice of preferred style is admittedly arbitrary, but any consistent style is better than none. In Chapter 6 we will justify the style we chose. The architecture of our normalization system is shown in Figure 4.1.

The process begins by merging all of the JavaScript source of an application into a single source file to simplify the resolution of external references. Functions and properties in the merged source are then uniquely renamed so that they can be
easily moved between scopes when transforming between styles. A class identification program then identifies and tags all class functions in the merged source, and style analysis tags each method of each class with its style. Finally, a style normalization transformation transforms all classes in the merged, tagged source to a single selected style, the transformed merged application source is de-renamed and split back to its original source file structure.

In the following sections we explain each part of this process, demonstrating the intermediate results by example.

### 4.2 Merge

Our system is implemented in the TXL source transformation language [9], using the existing JavaScript grammar to parse and pattern match JavaScript source. Because the representation of various JavaScript method styles may cross JavaScript source file
boundaries, our first step is to merge the entire JavaScript source of the application into one file that we can process as unit using a shell script (Merge on the left of Figure 4.1). Markers 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 (Split De-rename on the right of Figure 4.1). An example of preserving the files boundaries is shown in Listing 4.1.

Listing 4.1 Example of the a JavaScript application after merging.

```javascript
/**"Users/widd/Desktop/applications/Wiso/lib/wajax.js"
   // ... Contents of wajax.js file ...
/**"/Users/widd/Desktop/applications/Wiso/lib/wconsole.js"
   // ... Contents of wconsole.js file ...
/**"/Users/widd/Desktop/applications/Wiso/lib/winstance.js"
   // ... Contents of winstance.js file ...
...
```

4.3 Rename

As in many reverse engineering tasks, unique naming is applied so that functions and variable names are globally unique before processing (Rename in Figure 4.1). 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, all variables, methods and parameters to reflect their scope. Outer level is given a prefix `@qcsm` to indicate that they are top level of the application. The
prefix @qcsm was chosen because it is unlikely to occur in a JavaScript application but it can be any prefix that does not occur in the system being transformed. Inner functions, parameters, variables and anonymous functions are renamed with a prefix that combines the outer function name as well as the name of the inner function. Since function expressions (anonymous functions) do not have names, we give them a standard name Lambda along with a number that indicates the functions’ order in the class so it will be uniquely named. Listing 4.2 shows the original JavaScript of a simple application that will be used throughout this chapter. Listing 4.3 shows the same application after the unique naming is finished.
CHAPTER 4. APPROACH

Listing 4.2 An example of a JavaScript application.

```javascript
function circle (radius)
{
    this.radius = radius;
    this.getCircum = gc;
    this.getSize = circle.size;
    this.getInfo = info;
    this.getArea = function ()
    {
        return (this.radius * this.radius * 3.14)
    }

    function gc ()
    {
        var diameter = this.radius * 2;
        var circum = diameter * 3.14;
        return circum;
    }
}

function info ()
{
    return "this is a circle";
}

circle.size = function ()
{
    return "this is a big circle";
}

var bigCircle = new circle (100);
alert ("The area is: " + bigCircle.getArea ());
alert ("The circumference is : " + bigCircle.getCircum ());
alert ("This is a : " + bigCircle.getInfo ());
alert (bigCircle.getSize ());
```

4.3.1 Renaming Classes:

Unique naming, renames all global functions, adding a prefix to their name, For example, the function circle has been renamed @qcsm@circle in Listing 4.3.
4.3.2 Renaming Variables

After each global function is renamed, the sub scope of each function is processed, renaming all variables declared inside that function. Variables can be declared in two different ways in JavaScript. The variable can be declared locally using the keyword: `var`, or the value is assigned directly to new variable without the keyword `var`. Both cases are renamed accordingly, and everywhere they are used inside the function. For example, the variable `diameter` has been defined inside the method `gc` which is method of the class `circle` in Listing 4.3. One can see that the variable `diameter` has been renamed to `@qcsm@circle@gc@diameter`. 
LISTING 4.3 The output of the Unique Naming program.

```javascript
function @qcsm@circle (@qcsm@circle@radius)
{
    this.@qcsm@circle@radius = @qcsm@circle@radius;
    this.@qcsm@circle@getCircum = @qcsm@circle@gc;
    this.@qcsm@circle@getSize = @qcsm@circle.size;
    this.@qcsm@circle@getInfo = @qcsm@info;
    this.@qcsm@circle@getArea = function @qcsm@circle@Lambda1 ()
    {
        return (this.@qcsm@circle@radius * this.@qcsm@circle@radius * 3.14)
    }
}

function @qcsm@circle@gc ()
{
    var @qcsm@circle@gc@diameter = this.@qcsm@circle@radius * 2;
    var @qcsm@circle@circle@gc@circum = @qcsm@circle@gc@diameter * 3.14;
    return @qcsm@circle@gc@circum;
}

function @qcsm@info ()
{
    return "this is a circle";
}

@qcsm@circle.size = function @qcsm@Lambda1 ()
{
    return "this is a big circle";
}

var @qcsm@bigCircle = new @qcsm@circle (100);
alert ("The area is: " + @qcsm@bigCircle.getArea ());
alert ("The circumference is: " + @qcsm@bigCircle.getCircum ());
alert ("This is a: " + @qcsm@bigCircle.getInfo ());
alert (@qcsm@bigCircle.getSize ());
```

4.3.3 Renaming Parameters

Since parameters of a function are local to the function, they are renamed with a prefix @qcsm concatenated to the function name combined with the original parameter name. All places where the parameter is used are also renamed accordingly.
CHAPTER 4. APPROACH

@qcsm@circle@radius in Listing 4.3 is an example.

4.3.4 Renaming Properties:

Properties of each class are renamed to reflect their scope, and of course all references to that property are renamed as well. For example: the property `getCircum` in Listing 4.3 is renamed by `@qcsm@circle@getCircum` because it is a property of the class `@qcsm@circle`.

4.3.5 Renaming Methods

Nested methods are renamed as well, in all method and lambda function patterns. Each method is then further analyzed so variables and parameters defined inside of these methods are renamed properly. `@qcsm@circle@gc` and `@qcsm@circle@Lambda1` are examples of methods that have been renamed in Listing 4.3.

After all classes, methods, variables and parameters are renamed the application is ready to be processed by Class Identification.

4.4 Class Identification

After the unique naming is done, the JavaScript file is ready for class identification ("Class Identification" in Figure 4.1). To distinguish classes from simple functions, the Class identification program tags each JavaScript class by simply checking to see if there is an object that is instantiated from that class using the keyword: `new`. Listing 4.4 shows the example JavaScript application after it has been processed by the Class identification program. We use the sequence `<<<< @qcsm@circle FunctionClass`
Classes in JavaScript can be one of three style: Function Class, Lambda Class and Variable Lambda Class. The process of identifying these class patterns is discussed in more detail in Chapter 5.

Listing 4.4 The output of the Class Identification program.

```javascript
function @qcsm@circle FunctionClass ClassDef >>>> @qcsm@circle (@qcsm@circle@radius)
{
    this.@qcsm@circle@radius = @qcsm@circle@radius;
    this.@qcsm@circle@getCircum = @qcsm@circle@gc;
    this.@qcsm@circle@getSize = @qcsm@circle.size;
    this.@qcsm@circle@getInfo = @qcsm@info;
    this.@qcsm@circle@getArea = function @qcsm@circle@Lambda1 ()
    {
        return (this.@qcsm@circle@radius * this.@qcsm@circle@radius * 3.14)
    }
    function @qcsm@circle@gc ()
    {
        var @qcsm@circle@gc@diameter = this.@qcsm@circle@radius * 2;
        var @qcsm@circle@gc@circum = @qcsm@circle@gc@diameter * 3.14;
        return @qcsm@circle@gc@circum;
    }
}
function @qcsm@info ()
{
    return "this is a circle";
}
@qcsm@circle.size = function @qcsm@Lambda1 ()
{
    return "this is a big circle";
}
var @qcsm@bigCircle = new @qcsm@circle (100);
alert ("The area is: " + @qcsm@bigCircle.getArea ());
alert ("The circumference is : " + @qcsm@bigCircle.getCircum ());
alert ("This is a : " + @qcsm@bigCircle.getInfo ());
alert (@qcsm@bigCircle.getSize ());
```
4.5 Style Analysis

Style analysis uses the class functions identified in the previous step as a basis to identify methods of the class in the Style analysis step. The markup consists of three elements: the unique name of the class, the name of the method definition and the style of the method. Listing 4.5 shows our example JavaScript application after style analysis has been performed. In the program all four styles of methods are shown. The markup <<< @qcs(circle @qcs(circle@qcs(circle@qcs(circle:getArea innerLambda >>> is an example of the inner lambda style, <<< @qcs(circle @qcs(circle@qcs(circle@qcs(circle:getCircum innerMethod >>> shows the inner method style, an outer method is tagged as the following <<< @qcs(circle @qcs(circle@qcs(circle@qcs(circle:getInfo OuterMethod >>>>, whereas the <<< @qcs(circle @qcs(circle@qcs(circle@qcs(circle:getSize OuterLambda >>>> is an example of the outer lambda method style.

The mechanism by which we identify the styles of the methods is described in more detail in Chapter 5.
Listing 4.5 The output of Style Analysis program.

```javascript
function @qcsm@circle FunctionClass ClassDef >>> (circle) {
    @qcsm@circle@radius = @qcsm@circle@radius;
    this.@qcsm@circle@getArea = function <<< @qcsm@circle circle@getArea innerLambda >>>> (circle) {
        return (this.@qcsm@circle@radius * this.@qcsm@circle@radius * 3.14)
    }
    function <<< @qcsm@circle circle@getCircum innerMethod >>>> (circle) {
        var @qcsm@circle@circum = @qcsm@circle@diameter * 3.14;
        return @qcsm@circle@circum;
    }
    function <<< circle circle@getInfo OuterMethod >>>> (circle) {
        return "this is a circle";
    }
    @qcsm@circle.size = function <<< circle circle@getSize OuterLambda >>>> (circle) {
        return "this is a big circle";
    }
    var @qcsm@bigCircle = new @qcsm@circle (100);
    alert ("The area is: " + @qcsm@bigCircle.getArea ());
    alert ("The circumference is: " + @qcsm@bigCircle.getCircum ());
    alert ("This is a : " + @qcsm@bigCircle.getInfo ());
    alert (@qcsm@bigCircle.getSize ());
}
```

4.6 Style Transformation

After each method style is tagged by the Style analysis program, then each style is transformed into the preferred style pattern, Prototype Lambda style ("Style Transformation" in Figure 4.1). Listing 4.7 presents an example of the style transformation program’s output. The method @qcsm@circle.prototype.@qcsm@circle@getArea is an
example of the transformed methods. Chapter 6 discusses this phase in details.

**Listing 4.6** The output of Style Normalization program.

```javascript
function @qcsm@circle (@qcsm@circle@radius)
{
    this.@qcsm@circle@radius = @qcsm@circle@radius;
    this.@qcsm@circle@radius = this.@qcsm@circle@radius;
}
@qcsm@circle.prototype.@qcsm@circle@getArea = function @qcsm@circle@Lambda1 ()
{
    return (this.@qcsm@circle@radius * this.@qcsm@circle@radius * 3.14)
}
@qcsm@circle.prototype.@qcsm@circle@getCircum = function ()
{
    var @qcsm@circle@gc@diameter = this.@qcsm@circle@radius * 2;
    var @qcsm@circle@gc@circum = @qcsm@circle@gc@diameter * 3.14;
    return @qcsm@circle@gc@circum;
}
@qcsm@circle.prototype.@qcsm@circle@getInfo = function ()
{
    return "this is a circle";
}
@qcsm@circle.prototype.@qcsm@circle@getSize = function @qcsm@Lambda1 ()
{
    return "this is a big circle";
}
var @qcsm@bigCircle = new @qcsm@circle (100);
alert ("The area is: " + @qcsm@bigCircle.getArea ());
alert ("The circumference is : " + @qcsm@bigCircle.getCircum ());
alert ("This is a : " + @qcsm@bigCircle.getInfo ());
alert (@qcsm@bigCircle.getSize ());
```

### 4.7 De-Rename and Split:

Finally, we must recreate the original files by de-renaming and splitting the merged files ("Split De-Rename" in Figure 4.1 ). Both de-renaming and splitting are performed by a script, Listing 4.7 is an example after running the de-renaming script on the output of the style transformation program.
Listing 4.7 The output after the de-renaming script processes the transformed JavaScript application.

```javascript
function circle (radius)
{
    this.radius = radius;
    this.radius = this.radius;
}
circle.prototype.getArea = function ()
{
    return (this.radius * this.radius * 3.14)
}
circle.prototype.getCircum = function ()
{
    var diameter = this.radius * 2;
    var circum = diameter * 3.14;
    return circum;
}
circle.prototype.getInfo = function ()
{
    return "this is a circle";
}
circle.prototype.getSize = function ()
{
    return "this is a big circle";
}
var bigCircle = new circle (100);
alert ("The area is: " + bigCircle.getArea ());
alert ("The circumference is : " + bigCircle.getCircum ());
alert ("This is a : " + bigCircle.getInfo ());
alert (bigCircle.getSize ());
```

4.8 Summary

In this chapter we demonstrate the approach that we propose in order to solve the maintenance issues that arise due to having multiple class styles in many JavaScript applications. Our approach has several phases, the very first input to the first phase, Merge, Figure 4.1 is the original JavaScript application that consists of multiple JavaScript files. Next we run a unique naming step on the merged JavaScript file to
rename all functions, parameters and variables in the application so they are scope independent, Rename in Figure 4.1. Then we run the class identification step that tags all class patterns in the application, Classes Identification in Figure 4.1. After that, in the Style Analysis phase in Figure 4.1 each method of the previously identified classes is marked up with the corresponding method style pattern. Next, we run the style transformation step that transforms all the previously marked up methods of the class to one style, Style Transformation in Figure 4.1. Finally, we de-rename the output of the Style Transformation so we have the original JavaScript application ready for evaluation, De-rename and Split in Figure 4.1. The next chapter discusses the Class Identification phase in further details and explains what is the criteria to tag a class, and what are the class patterns in JavaScript. Moreover, the Style Analysis phase is also explained.
Chapter 5

Class Identification and Style Analysis

We have previously discussed our approach to normalize different JavaScript method patterns to one single style. We have a sequence of transformation programs each of which targets a specific phase of the approach as discussed in Chapter 4. In this chapter we are going to discuss the Class Identification and Style Analysis phases, the different style of classes and the process of tagging these classes and methods.

5.1 Class Identification

Our class tagging program identifies each function that is a class. Because JavaScript may be mixed with XML, we chose a distinctive markup syntax using the character sequence "<<<" and ">>>" to bracket our annotations. Since these brackets are not legal syntax in JavaScript or XML so there are no ambiguities. Classes are distinguished from functions simply if there is an instance created from that class. So our
class identification searches for all classes that have new objects created from them. The markup we use indicates the class name, class type and the tag \textit{ClassDef}. This step is performed for technical reasons so when we run our Style Analysis program, it will take into consideration all different definitions of the class.

We identify three class definition styles: Function Class, Lambda Class and Variable Lambda Class, here is an example of the Function Class style:

\textbf{Function Class} \quad Listing 5.1 shows an example of the function class style that takes the form of a standalone function. The tag \texttt{<<<< sack FunctionClass ClassDef >>>>} indicates that this is a \textit{Function Class}, \textit{sack} is the name of that class and \textit{ClassDef} is to indicate that this is a class.
Listing 5.1 Example of a class definition style

function <<< sack FunctionClass ClassDef >>>> sack (file) {
  this.xmlhttp = null;
  this.resetData = function () {
    this.method = "POST";
    this.queryStringSeparator = "?";
    this.argumentSeparator = "&";
    this.URLString = "";
    this.encodeURIString = true;
    this.execute = false;
    this.element = null;
    this.elementObj = null;
    this.requestFile = file;
    this.vars = new Object ();
    this.responseStatus = new Array (2);
  }
;

  ...
}

ajaxObjectArray [ajaxObjectArray.length] = new sack ();
..
and to distinguish among a class constructor, routine function and method function. A class constructor has properties unlike a standalone function (Routine function). But all three functions have formal parameters and variables. The key element that indicates a class constructor is that there must be a new object instantiated from that constructor.

Prototype Lambda Model Figure 5.1 shows the adapted DMM model. A method function has three attributes: isOuter, isLambda and isPrototype. These attributes distinguish each class style from the other styles and will have different attribute values for each style. These attributes are attached to the method function and not the Class Function because a class function can have mixed styles of methods as discussed in Chapter 3. In the prototype lambda’s case, the isOuter attribute is true since this method is defined outside the scope of the class constructor function. The isLambda attribute is true, since the method is defined as function expression. Finally the isPrototype attribute is true because the method is delegated to the prototype property of a class constructor function.

Inner Lambda Model In the inner lambda style, the method function attributes have different values. isOuter is false since in this style the method is defined inside the scope of the class constructor. isLambda is true since the method is defined as a lambda function and isPrototype is set to false because the method is not assigned to the prototype object of the class.

Outer Lambda Model The isOuter attribute is true because the method is defined outside the scope of the class constructor function. isLambda is true because
Figure 5.1: The class style adapted DMM model
the method takes the form of a lambda function and \( \text{isPrototype} \) is \textit{false} because the method is not delegated to the prototype property of the class constructor.

**Inner Method Model** The \textit{method function} attributes of the inner method style have the following values: \textit{isOuter} attribute is false since the method is defined inside the scope of the class constructor method, \textit{isLambda} is false because the method is defined as a standalone function and \textit{isPrototype} is false because it is not a prototype method.

**Outer Method Model** Similarly to inner method style, \textit{isLambda} and \textit{isPrototype} attributes are \textit{false} because it is a standalone function that is not delegated to prototype of the class, but the \textit{isOuter} attribute is \textit{true} since this method style is defined outside the class constructor.

### 5.2.1 Tagging Method styles

Now the JavaScript file is ready to run the style analysis program. This program identifies each JavaScript method style and use another markup to tag instances of each method style pattern. The markup encodes a relationship of our model, relating a class function name, a method function name and the associated style pattern.

\[<<<< \text{ClassName MethodName PatternStyle} >>>>>\]

The style analysis program searches the previously marked up class definitions, as mentioned in the previous section, and looks inside them to see which method pattern is used. Each class is then matched against each of the method style patterns previously described in Chapter 3. and each method of the class is marked up with its style using the markup above. The attributes of the function styles in the
corresponding meta-model help in characterizing the pattern we are looking for.

**Listing 5.2** An example of the inner lambda style

```javascript
function TiddlyWiki() {
    var tiddlers = {};
    this.tiddlersUpdated = TiddlyWiki.prototype.false;
    this.namedNotifications = [];
    this.notificationLevel = 0;
    this.slices = {};
    this.clear = function TiddlyWiki clear innerLambda () {
        tiddlers = {};
        this.setDirty (false);
    };
    ...
}
```

**Inner method pattern**  We look for a method that is defined inside the class and as a standalone function. The method should be also assigned to a property inside the class.

**Inner lambda pattern**  Similarly, the inner lambda style, shown in Listing 5.2, is defined inside the scope of the class function but defined as anonymous function expression. Since the function is assigned directly to a property of the class, the property assignment is already implied. In Listing 5.2, the tag

```javascript
<<<< TiddlyWiki clear innerLambda >>>>
```

shows the class name *TiddlyWiki*, function name *clear* and *innerLambda* that indicates the class style.
Listing 5.3 An example of the outer lambda style

```javascript
function FileAdaptor ()
{
  ...
}
...

FileAdaptor.loadTiddlyWikiCallback = function
<<<< FileAdaptor loadTiddlyWikiCallback OuterLambda >>>>
(status, context, responseText, url, xhr)
{
  context.status = status;
  if (! status) {
    context.statusText = "Error reading file";
  }
  else {
    context.adaptor.store = new TiddlyWiki ();
    if (! context.adaptor.store.importTiddlyWiki (responseText)) {
      context.statusText = config.messages.invalidFileError.format ([url]);
      context.status = false;
    }
  }

  context.complete (context, context.userParams);
};
...
```

**Outer lambda pattern**  Outer lambda style is tagged as the method is defined outside the scope of a class and takes the form of a anonymous function expression, as shown in Listing 5.3. The method also should be assigned to a property of the class.  <<<< FileAdaptor loadTiddlyWikiCallback OuterLambda >>> encloses the class name, FileAdaptor, method name loadTiddlyWikiCallback and OuterLambda is the class style.
Outer method pattern  Outer method takes the standalone function form. It is defined outside the scope of the class function, and should be assigned to a property of the class.Listing 5.4 demonstrates outer method pattern. <<< menuitem getprop OuterMethod >>> encloses the class name menuitem, method name prop and the method style OuterMethod.
Listing 5.4 An example of the outer method style

```javascript
function menu_item(FunctionClass, ClassDef) {
    this.n_depth = o_parent.n_depth + 1;
    this.a_config = o_parent.a_config[n_order + (this.n_depth ? 3 : 0)];
    if (!this.a_config) return;

    this.o_root = o_parent.o_root;
    this.o_parent = o_parent;
    this.n_order = n_order;
    this.n_id = this.o_root.a_index.length;
    this.o_root.a_index[this.n_id] = this;
    o_parent.a_children[n_order] = this;

    var o_root = this.o_root, a_tpl = this.o_root.a_tpl;
    ...
}

function getprop(n_depth) {
    var s_value = null, a_level = this.o_root.a_tpl[this.n_depth];
    if (a_level) s_value = a_level[s_key];

    return (s_value == null ? this.o_parent.getprop(s_key) : s_value);
}

function getstyle(n_pos, n_state) {
    var a_css = this.getprop('css');
    var a_oclass = a_css[n_pos ? 'inner' : 'outer'];
    if (typeof(a_oclass) == 'string') return a_oclass;

    for (var n_currst = n_state; n_currst >= 0; n_currst --) if (a_oclass[n_currst])
        return a_oclass[n_currst];
}

function upstatus(b_clear) {
    window.setTimeout("window.status=unescape('" + (b_clear ? '' : (this.a_config[2] && this.a_config[2]['sb'] ? escape(this.a_config[2]['sb']) : escape(this.a_config[0])) + (this.a_config[1] ? ('' + escape(this.a_config[1]) + 'a_int')) + ''))", 10);
}
```
5.3 Summary

First, we discuss the Class Identification phase and show the different class definition styles in JavaScript. Then, we explain the process of marking up each class pattern with the tag that indicates the class name, class pattern and the tag ClassDef. After that, the JavaScript application is ready to be processed by the Style Analysis program, this program is created based on a model that differentiate each class style among other styles. The program will search in each previously marked up class to match each method to its equivalent pattern. Finally, the application is now ready to be processed by the style normalization program as the next chapter discusses the process of transforming all different patterns to the prototype lambda pattern.
In the previous chapter we talk about the Class Identification and Style Analysis phases of our approach. The class patterns and how to identify a class in JavaScript. We also talk about the method patterns, creating a model for each pattern to help us identify the characteristics of each pattern and how each pattern is tagged by the style analysis program.

6.1 Pattern Transformation

Once we have identified all of the class patterns in the JavaScript application and marked up each method with the corresponding class style as discussed in Chapter 5, we are ready to transform the applications different styles to a normalized single style. The question arises as to which style we should choose?
6.1.1 The Chosen Style

As observed in Chapter 3, the inner class style, both function and lambda, require the method to be defined inside the scope of the class constructor, which means that for every instantiated object of the class, these methods are recreated in the new instance and this is inefficient and can affect the performance of an application [3]. The outer class style, on the other hand; introduces another possible disadvantage which is polluting the global namespace since this style requires the methods to be implemented outside the scope of the class constructor. The prototype lambda style is the most natural style to the language since it is a prototypal language in the first place. By choosing this style we are adding all methods of a class function to the prototype property of that class, this way all methods are public and can be accessed and inherited by any object. In addition the object’s methods are defined only once and any instantiated objects will point to those methods, and by that we are enhancing the performance of any application [3].

6.1.2 Transformation Process

After all classes are tagged with their style, the normalizing transformation is carried out by another program that looks for instances of each style markup, and transform the the marked-up methods to prototype lambda style. Since the prototype lambda style pattern defines method functions outside the scope of a class, inner method and inner lambda pattern methods are first extracted and relocated outside the scope of the class function in a first step, effectively transforming them to outer method and outer lambda form. Any variables bound as closures in the functions are transformed into properties. A second step then transforms the outer
method and outer lambda to prototype lambda form. Listing 6.1 shows the same application that has been processed by the style analysis program in Listing 5.4 in Chapter 5. All three outer methods \textit{getprop}, \textit{getstyle} and \textit{upstatus} are transformed to prototype lambda \textit{menuitem.prototype.getprop}, \textit{menu item.prototype.getstyle} and \textit{menuitem.prototype.upstatus} in Listing 6.2.
Listing 6.1 The output of style normalization.

```javascript
function @qcsm@menu_item (@qcsm@menu_item@o_parent, @qcsm@menu_item@n_order)
{
  this.@qcsm@menu_item@n_depth = @qcsm@menu_item@o_parent.n_depth + 1;
  this.@qcsm@menu_item@a_config = @qcsm@menu_item@o_parent.a_config[ @qcsm@menu_item@n_order + (this.@qcsm@menu_item@n_depth ? 3 : 0) ];
  if (! this.@qcsm@menu_item@a_config) return;

  this.@qcsm@menu_item@o_root = @qcsm@menu_item@o_parent.o_root;
  this.@qcsm@menu_item@o_parent = @qcsm@menu_item@o_parent;
  this.@qcsm@menu_item@n_order = @qcsm@menu_item@o_order;
  this.@qcsm@menu_item@n_id = this.@qcsm@menu_item@o_root.a_index.length;
  this.@qcsm@menu_item@o_root.a_index [this.@qcsm@menu_item@n_id] = this;
  @qcsm@menu_item@o_root.a_children[ @qcsm@menu_item@n_order ] = this;

  @qcsm@menu_item@o_root = this.@qcsm@menu_item@o_root,
  @qcsm@menu_item@a_tpl = this.@qcsm@menu_item@o_root.@qcsm@menu_item@a_tpl;
...
}

@qcsm@menu_item.prototype.getprop = function @qcsm@Lambda1 (@qcsm@Lambda1@s_key)
{
  var @qcsm@Lambda1@s_value = null, @qcsm@Lambda1@a_level =
  this.o_root.a_tpl [this.n_depth];
  if (@qcsm@Lambda1@a_level) @qcsm@Lambda1@s_value =
    @qcsm@Lambda1@a_level[ @qcsm@Lambda1@s_key ];
  return (@qcsm@Lambda1@s_value == null ? this.o_parent.getprop (@qcsm@Lambda1@s_key) :
    @qcsm@Lambda1@s_value);
}

@qcsm@menu_item.prototype.getstyle = function @qcsm@Lambda2
(@qcsm@Lambda2@n_pos, @qcsm@Lambda2@n_state)
{
  var @qcsm@Lambda2@css = this.getprop ('css');
  var @qcsm@Lambda2@o_class = @qcsm@Lambda2@css[ @qcsm@Lambda2@n_pos ? 'inner' : 'outer' ];
  if (typeof (@qcsm@Lambda2@o_class) == 'string') return @qcsm@Lambda2@o_class;
...}
```
6.2 De-rename and Split

After the JavaScript is normalized to one method pattern, Listing 6.2 shows the same tigraMenu application after we de-rename the application from the unique naming and then split the application to recreate the original application in order to evaluate the new version of the application by running it as described in Chapter 4.
Listing 6.2 The normalized output after De-renaming is performed.

```javascript
function menu_item (o_parent, n_order)
{
    this.n_depth = o_parent.n_depth + 1;
    this.a_config = o_parent.a_config [n_order + (this.n_depth ? 3 : 0)];
    if (! this.a_config) return;

    this.o_root = o_parent.o_root;
    this.o_parent = o_parent;
    this.n_order = n_order;
    this.n_id = this.o_root.a_index.length;
    this.o_root.a_index [this.n_id] = this;
    o_parent.a_children [n_order] = this;
    var o_root = this.o_root, a_tpl = this.o_root.a_tpl;
    ...
}

menu_item.prototype.getprop = function (s_key)
{
    var s_value = null, a_level = this.o_root.a_tpl [this.n_depth];
    if (a_level) s_value = a_level [s_key];
    return (s_value == null ? this.o_parent.getprop (s_key) : s_value);
}

menu_item.prototype.getstyle = function (n_pos, n_state)
{
    var a_css = this.getprop ('css');
    var a_oclass = a_css [n_pos ? 'inner' : 'outer'];
    if (typeof (a_oclass) == 'string') return a_oclass;
    for (var n_currst = n_state; n_currst >= 0; n_currst --) if (a_oclass [n_currst])
        return a_oclass [n_currst];
}

menu_item.prototype.upstatus = function (b_clear)
{
    window.setTimeout ("window.status=unescape('' + (b_clear ? '' : (this.a_config [2] && this.a_config [2] ['sb'] ? escape (this.a_config [2] ['sb']) : escape (this.a_config [0]) + (this.a_config [1] ? '(' + escape (this.a_config [1]) + ')') : '')) + '}}")", 10);
```
6.3 Summary

This chapter describes how the transformation process is performed after the JavaScript application is processed by the class identification and style analysis programs discussed in Chapter 5. The next chapter discusses the evaluation phase of our approach, the acquired dataset of real-life applications and examples.
Chapter 7

Evaluation

The previous chapter discusses the final step in our approach, Style Normalization. This stage takes the previously method style marked up application and transforms each style to the prototype lambda style. We then run scripts to de-rename the transformed application and split the application to its original JavaScript file structure. This allows us to recreate the JavaScript application to test it. This chapter evaluates our technique on a number of JavaScript applications and shows the transformed output for each application.

7.1 Selection of Test Applications

In order to test our approach, we gathered a number of open source web applications of varying application domains and sizes and chose three representatives, ranging from about 1000 to about 30,000 lines of code. These applications cover all the class styles found in our survey. In addition two of the applications have a mix of class styles where the last one has only one class style. Table 7.1 summarizes the application


Table 7.1: Summary of applications’ info

<table>
<thead>
<tr>
<th>Application Name</th>
<th>No of Classes</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chinese Tiddly Wiki</td>
<td>63</td>
<td>30407</td>
</tr>
<tr>
<td>T4d Tigra Menu</td>
<td>10</td>
<td>6678</td>
</tr>
<tr>
<td>Sudoku</td>
<td>1</td>
<td>1136</td>
</tr>
</tbody>
</table>

Table 7.2: Applications’ class styles

<table>
<thead>
<tr>
<th>Application Name</th>
<th>No of Patterns</th>
<th>Method Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chinese Tiddly Wiki</td>
<td>548</td>
<td>Prototype Lambda</td>
</tr>
<tr>
<td></td>
<td>36</td>
<td>Inner Lambda</td>
</tr>
<tr>
<td></td>
<td>27</td>
<td>Outer Lambda</td>
</tr>
<tr>
<td>T4d Tigra Menu</td>
<td>35</td>
<td>Outer Method</td>
</tr>
<tr>
<td>Sudoku</td>
<td>5</td>
<td>Prototype Lambda</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>Inner Lambda</td>
</tr>
</tbody>
</table>

names, number of lines of code and number of class definitions of each of the tested applications.

7.2 Testing Procedure

Completeness of the transformation of each applications was manually verified to check that all identified instances of the class styles had been transformed to the prototype lambda style. In order to test the correctness of our transformations, we
ran each application before and after transformation, testing that all visible links and user interface functionalities provided by the applications exhibited the same behaviour before and after transformation.

7.3 Chinese Tiddly Wiki

The Chinese Tiddly Wiki is a reusable non-linear personal web notebook that keeps ideas and information organized [38]. This application is 1.7 MB in size and is composed of 30407 lines of code as shown in Table 7.1.

The application has 36 instances of the inner lambda and 27 instances of the outer lambda method patterns as described in Table 7.2.
Listing 7.1 The original Chinese Tiddly Wiki application.

    function TiddlyWiki ()
    {
        var tiddlers = {};
        this.tiddlersUpdated = false;
        this.namedNotifications = [];
        this.notificationLevel = 0;
        this.slices = {};
        this.clear = function ()
        {
            tiddlers = {};
            this.setDirty (false);
        };
        this.fetchTiddler = function (title)
        {
            var t = tiddlers [title];
            return t instanceof Tiddler ? t : null;
        };
        this.deleteTiddler = function (title)
        {
            delete this.slices [title];
            delete tiddlers [title];
        };
        this.addTiddler = function (tiddler)
        {
            delete this.slices [tiddler.title];
            tiddlers [tiddler.title] = tiddler;
        };
    }
    ...

Listing 7.1 shows a code snippet of the application before the transformation, whereas Listing 7.2 shows the application after all the transformations have been performed.
Listing 7.2 Chinese Tiddly Wiki after the transformation.

```javascript
function TiddlyWiki ()
{
    this.tiddlers = {};
    this.tiddlersUpdated = false;
    this.namedNotifications = [];
    this.notificationLevel = 0;
    this.slices = {};
}
TiddlyWiki.prototype.clear = function ()
{
    this.tiddlers = {};
    this.setDirty (false);
};
TiddlyWiki.prototype.fetchTiddler = function (title)
{
    var t = this.tiddlers [title];
    return t instanceof Tiddler ? t : null;
};
TiddlyWiki.prototype.deleteTiddler = function (title)
{
    delete this.slices [title];
    delete this.tiddlers [title];
};
TiddlyWiki.prototype.addTiddler = function (tiddler)
{
    delete this.slices [tiddler.title];
    this.tiddlers [tiddler.title] = tiddler;
};
...
```

The style normalization program transforms all the method style instances to the prototype style and the application runs without any changes in the original functionalities and behaviors. Figure 7.1 shows a snapshot of the application after transformation.
CHAPTER 7. EVALUATION

Figure 7.1: A snapshot of the Chinese Tiddly Wiki application.
CHAPTER 7. EVALUATION

7.4 T4d Tigra Menu

The Tigra Menu is free JavaScript menu navigation component for web sites, it has various menu styles such as drop down menu, pop-up navigation menu on mouse over and many other styles [31]. The application size is 1.5 MB and has 6678 line of code.

The application it has 35 instances of outer method style according to Table 7.2. All 35 instances of the outer method styles are transformed successfully and the applications has the same running output after the transformation as the original application. Listing 7.3 shows an example of the code before the transformation, where Listing 7.4 shows the same code of the application after the transformation has been performed. Figure 7.2 is a screenshot of the Tigra Menu application.
Listing 7.3 The Tigra Menu application before the transformation.

```javascript
function menu_item (o_parent, n_order)
{
    this.n_depth = o_parent.n_depth + 1;
    this.a_config = o_parent.a_config [n_order + (this.n_depth ? 3 : 0)];
    if (! this.a_config) return;

    this.o_root = o_parent.o_root;
    this.o_parent = o_parent;
    this.n_order = n_order;
    this.n_id = this.o_root.a_index.length;
    this.o_root.a_index [this.n_id] = this;
    o_parent.a_children [n_order] = this;

    var o_root = this.o_root, a_tpl = this.o_root.a_tpl;
    this.getprop = mitem_getprop;
    this.getstyle = mitem_getstyle;
    this.upstatus = mitem_upstatus;
    ...
}

function mitem_getprop (s_key)
{
    var s_value = null, a_level = this.o_root.a_tpl [this.n_depth];
    if (a_level) s_value = a_level [s_key];
    return (s_value == null ? this.o_parent.getprop (s_key) : s_value);
}

function mitem_getstyle (n_pos, n_state)
{
    var a_css = this.getprop ('css');
    var a_oclass = a_css [n_pos ? 'inner' : 'outer'];
    if (typeof (a_oclass) == 'string') return a_oclass;
    for (var n_currst = n_state; n_currst >= 0; n_currst --)
        if (a_oclass [n_currst]) return a_oclass [n_currst];
}

function mitem_upstatus (b_clear)
{
    window.setTimeout ("window.status=unescape('" + (b_clear ? '' : (this.a_config [2] && this.a_config [2] ['sb'] ? escape (this.a_config [2] ['sb']) : escape (this.a_config [0]) + (this.a_config [1] ? ('' + escape (this.a_config [1]) + '':'')) + '''))), 10);
}
```
Listing 7.4 Tigra Menu application after the transformation.

```javascript
function menu_item(o_parent, n_order)
{
    this.n_depth = o_parent.n_depth + 1;
    this.a_config = o_parent.a_config [n_order + (this.n_depth ? 3 : 0)];
    if (! this.a_config) return;

    this.o_root = o_parent.o_root;
    this.o_parent = o_parent;
    this.n_order = n_order;
    this.n_id = this.o_root.a_index.length;
    this.o_root.a_index [this.n_id] = this;
    o_parent.a_children [n_order] = this;
    var o_root = this.o_root, a_tpl = this.o_root.a_tpl;
    ...
}

menu_item.prototype.getprop = function (s_key)
{
    var s_value = null, a_level = this.o_root.a_tpl [this.n_depth];
    if (a_level) s_value = a_level [s_key];
    return (s_value == null ? this.o_parent.getprop (s_key) : s_value);
}

menu_item.prototype.getstyle = function (n_pos, n_state)
{
    var a_css = this.getprop ('css');
    var a_oclass = a_css [n_pos ? 'inner' : 'outer'];
    if (typeof (a_oclass) == 'string') return a_oclass;
    for (var n_currst = n_state; n_currst >= 0; n_currst --)
        if (a_oclass [n_currst]) return a_oclass [n_currst];
}

menu_item.prototype.upstatus = function (b_clear)
{
    window.setTimeout ("window.status=unescape('" + (b_clear ? '' : (this.a_config [2] && this.a_config [2] ['sb'] ? escape (this.a_config [2] ['sb']) : escape (this.a_config [0]) + (this.a_config [1] ? '(' + escape (this.a_config [1]) + ')': '') + '')"), 10);
}
```
Figure 7.2: A snapshot of the Tigra Menu application.
7.5 Sudoku

Sudoku is a logic game. The user has to place the numbers 1 to 9 in every row, column and square. But, and that is the challenge, no number can be used twice! Sudoku-Puzzle is a simple JavaScript which generates random puzzles [13]. Sudoku is the smallest application that we have tested as shown in Table 7.1 it is 90 KB and has 1136 line of code, it also has one instance function class style, 5 instances of prototype lambda pattern and 3 instances of inner lambda method patterns as seen in Table 7.2. The style normalization program transforms all of the instances of the inner lambda style to the prototype lambda style and the application performs the same after the transformation.

Listing 7.5 The Sudoku application before the transformation.

```javascript
function Token (number, row, col, maxlen)
{
    this.number = number > maxlen ? 1 : number;
    this.row = row;
    this.col = col;
    this.maxlen = maxlen;
    this.canceledValues = new Array ();
    this.addCanceledValues = function (val)
    {
        val = val > this.maxlen ? 1 : val;
        this.canceledValues [this.canceledValues.length] = val;
    };
    this.isCanceledValues = function (val)
    {
        return this.canceledValues.inArray (val);
    };
    this.isEqual = function (T)
    {
        if (this.number == T.number && this.row == T.row && this.col == T.col) return true;
        return false;
    };
}
```
CHAPTER 7. EVALUATION

Listing 7.6 The Sudoku application after the transformation.

function Token (number, row, col, maxlen)
{
    this.number = number > maxlen ? 1 : number;
    this.row = row;
    this.col = col;
    this.maxlen = maxlen;
    this.canceledValues = new Array ();
}

Token.prototype.addCanceledValues = function (val)
{
    val = val > this.maxlen ? 1 : val;
    this.canceledValues [this.canceledValues.length] = val;
}

Token.prototype.isCanceledValues = function (val)
{
    return this.canceledValues.inArray (val);
}

Token.prototype.isEqual = function (T)
{
    if (this.number == T.number && this.row == T.row && this.col == T.col) return true;
    return false;
}

Listing 7.5 is code snippet of the application before the transformation and Listing 7.6 is the same code after the transformation. Figures 7.3, 7.4 and 7.5 show screenshots of the Sudoku when initially loaded, the confirmation after clicking on the “show solution” link and finally the solved game.

The performance of our system changed little from small to large JavaScript applications, consistently processing programs in less than a minute, and our framework scales well to realistic JavaScript programs. These test applications also had a mix of class patterns, showing that our method can deal with mixed styles.
CHAPTER 7. EVALUATION

Figure 7.3: A snapshot of the Sudoku application (1).

Figure 7.4: A snapshot of the Sudoku application (2).
7.6 Summary

In this chapter we have described the evaluation of our approach on several JavaScript applications by comparing the output of the original application to the application after transformation. All of the examples yield the same output as the original application which means that the transformation was successfully performed and the behaviour of the application has been preserved. The next chapter concludes the thesis and discusses the future work.
Chapter 8

Conclusions

8.1 Summary

In this thesis we have presented a class normalization framework for JavaScript designed to increase stylistic consistency and thus improve web application comprehension and maintainability. While JavaScript has no native class concept, its prototype- and object-based programming paradigm allows for a range of representations for object classes. Classes can be implemented in several different ways varying on the way which methods are implemented inside them, and JavaScript web applications often mix these styles, making them difficult to understand and maintain. 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 and comprehend result.

The approach uses several TXL programs each of which has a certain role in preparing the JavaScript application for the transformation phase. Each program takes its input from the previous step’s output and process it. The final output from
CHAPTER 8. CONCLUSIONS

the style normalization program yields an application with consistent JavaScript class style.

8.2 Limitations

As Chapter 3 discusses, the styles we identify are the common, most used styles among various web applications that we have surveyed. But there are some applications that are implementing other not so common class styles, and our transformation programs do not identify them. In addition, some applications have different representations of the class definition than the ones described in Chapter 5. Another representation, called *Object Literal*, is also uncommon among web applications, and it has a different representation of properties and methods. Listing 8.1 shows an example of the Object Literal class definition style, which is not recognized by our class identification program.

**Listing 8.1 Object Literal Class definition example**

```javascript
var myObject = {
    prop1: 'some string value',
    prop2: 10,
    prop3: false,
    method1: function(){
        alert('I am ' + this.iAm);
    }
};
```

8.3 Future Work

We are presently working on processing a large number of additional web applications, and in particular those using the inner method class style. Also, normalizing class
definitions themselves to a standard format is planned. 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.
Bibliography


[29] Yehuda Shiran and Tomer Shiran. Object-oriented programming with JavaScript, part ii: Methods: Defining methods,


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


Appendix A

Unique Naming Code

1 % This unique naming TXL program renames all functions, lambda functions
2 % and their parameters, variables and methods.
3
4 include "Javascript.Gm"
5
6 function main
7 replace [program]
8     P [program]
9 by
10     P [nameFunctions '@qcsM]
11        [nameLambdaFunction '@qcsM]
12        [renameFunctionsUses]
13        [renameUndeclaredVariables '@qcsM]
14 end function
15
16 %Find each outer function
17 rule nameFunctions nFun [id]
18 skipping [js_subscope]
19 replace $ [repeat js_declaration_or_statement]
20     'function Name [id] (Parms [list js_formal_parameter] )
21         '{
22             Scope [repeat js_declaration_or_statement]
23         }
24    RestOfScope [repeat js_declaration_or_statement]
25
26 construct NamePrime [id]
27     nFun [+ '@'] [+ Name]
28
APPENDIX A. UNIQUE NAMING CODE

29 construct ParmIds [repeat id]
  - ['Parms']
31
32 construct parmsPrime [list js_formal_parameter]
  Parms [nameParameter NamePrime]
34
35 construct parmPrimeIds [repeat id]
  - ['parmsPrime']
37
38 % Renaming all methods, lambda functions, variables, parameters and
39 % properties inside this function.
40 construct NewFunction [js_declaration]
  'function NamePrime ( parmsPrime )
  {
   Scope [nameFunctions NamePrime]
   [nameLambdaFunction NamePrime]
   [nameVariable NamePrime]
   [$ each ParmIds parmPrimeIds]
   [renameUndeclaredVariables '@qsm]
   [renameFunctionsUses]
   [renameProperties NamePrime]
  }
49
51 by NewFunction
   RestOfScope
end rule

57 define id_pair
  [id] [id]
end define

61 % This is a special case where the property is assigned to a function name.
62 function renameFunctionsUses
  export listOfFunctions [repeat id_pair]
  _% empty
  replace * [repeat js_declaration_or_statement]
  X [repeat js_declaration_or_statement]
  by
  X [addEachFunction] % builds list of functions at current scope level
  [renameTheFunctionUses]
end function

73 rule addEachFunction
  skipping [js_subscope] % don't go into nested scopes
  replace $ [js_function_declaration]
  'function Name [id] ( Params [list js_formal_parameter] )
APPENDIX A. UNIQUE NAMING CODE

{'
    ss [js_subscope]
    }

construct NormalName [id]
    Name [removeUnique]
import listOfFunctions [repeat id_pair]

export listOfFunctions
NormalName Name
listOfFunctions
by
    'function Name ( Params )
    '{
        ss
    }
end rule

%This rule removes the unique naming in case the property is assigned
% to a function name.
rule removeUnique
replace [id]
    x [id]
    construct IndexOfCharAfterAt [number]
        _ [index x ']' [+ 1]
    deconstruct not IndexOfCharAfterAt
  1
by
    x [: IndexOfCharAfterAt 1000]
end rule

% Searching for all function references.
function renameTheFunctionUses
    import listOfFunctions [repeat id_pair]
    replace * [repeat js_declaration_or_statement]
    Stmts [repeat js_declaration_or_statement]
by
    Stmts [replaceIDPairs each listOfFunctions]
end function

% Removing unique naming from function references.
function replaceIDPairs anIdPair [id_pair]
deconstruct anIdPair
    OldName [id] NewName [id]
    replace * [repeat js_declaration_or_statement]
    Stmts [repeat js_declaration_or_statement]
by
APPENDIX A. UNIQUE NAMING CODE

```plaintext
stmts [$ OldName NewName]
end function

% Renaming variables declared without keyword "var".
rule renameUndeclaredVariables N [id]
  skipping [js_subscope]
  replace $ [repeat js_declaration_or_statement]
    VarName [id] = VarValue [js_conditional_or_expn] optSemi [opt ';']
    Scope [repeat js_declaration_or_statement]
  construct varNamePrimeList [repeat id]
    [buildID N VarName]
  deconstruct varNamePrimeList
    varNamePrime [id]
  by
    VarName [$ VarName varNamePrime] = VarValue optSemi
    Scope [$ VarName varNamePrime]
end rule

% Renaming all function parameters.
rule nameParameter F [id]
  replace $ [id]
  by
    F [+ '@'] [+ paramName]
end rule

% Renaming lambda function definition.
rule nameLambdaFunction nLFun [id]
  skipping [js_subscope]
  replace $ [js_function_expn]
    'function ( Parms [list js_formal_parameter] )
      '{
        Scope [repeat js_declaration_or_statement]
      '}
  construct LambdaNamePrime [id]
    nLFun [+ '@'] [+ 'Lambda'] [!]
  construct LParmIds [repeat id]
    . [^ Parms]
  construct LparmsPrime [list js_formal_parameter]
    Parms [nameParameter LambdaNamePrime]
  construct LparmPrimeIds [repeat id]
    . [^ LparmsPrime]
```
%Renaming all methods, lambda functions, variables, parameters
% and properties inside this function

construct NewLambdaFunction [js_function expn]
'function LambdaNamePrime ( LparmsPrime )
'{
  Scope [nameLambdaFunction LambdaNamePrime]
  [nameFunctions LambdaNamePrime]
  [nameVariable LambdaNamePrime]
  [renameUndeclaredVariables '@qcsm]
  [$ each LParmIds LparmPrimeIds]
  [renameProperties LambdaNamePrime]
  '}

by NewLambdaFunction
end rule

% renaming all declared variables.
rule nameVariable M [id]
replace $ [repeat js-declaration_or_statement]
'vear Vars [list js-variable_description] OptSemi [opt '];
Scope [repeat js-declaration_or_statement]

construct varName [repeat id]
[getVar each Vars]
construct varNamePrime [repeat id]
[buildID M each varName]
by
'vear Vars [$ each varName varNamePrime] OptSemi
Scope [$ each varName varNamePrime]
end rule

function getVar V [js-variable_description]
deconstruct V
N[id] - [opt js-variable-initialization]
replace [repeat id]
x [repeat id]
by
N x
end function

%replacing original variable name with uniquely named variable
function buildID M [id] varName[id]
where not varName [grep '@qcsm]
construct varNamePrime [id]
M [+ '@] [+ varName]
replace [repeat id]
  x [repeat id]
by
  x [. varNamePrime]
end function

% renaming function properties
rule renameProperties N [id]
  replace $ [repeat js_declaration_or_statement]
  'this. VarName [id] = VarValue [js_conditional_expn] optSemi [opt ';
  Scope [repeat js_declaration_or_statement]
  construct varNamePrimeList [repeat id]
    [. buildID N VarName]
  deconstruct varNamePrimeList
    varNamePrime [id]
by
  'this. VarName [$ VarName varNamePrime] = VarValue optSemi
  Scope [$ VarName varNamePrime]
end rule
Appendix B

Class Identification Code

1 % This code tags all three class definitions.
2 function main
3
4 replace [program]
5    P [program]
6 by
7    P [markClass P]
8    [markLambdaClass P]
9    [markVarLambdaClass P]
10 end function
11
12 % This rule tags the Function Class definition, it take the
13 % program as a parameter and searches entire application for
14 % any instances created from this class.
15
16 rule markClass P [program]
17 skipping [js_subscope]
18 replace $ [ js_declaration_or_statement]
19    'function FName [id] ( Params [list js_formal_parameter])
20      '{
21         SS [js_subscope]
22      } optSemi [opt ';
23     deconstruct * [js_lefthand_side_expn] P
24      'new FName .[js_arguments]
25
26 by
27    'function <<<< FName 'FunctionClass 'ClassDef >>>> FName (Params)
28      '{
29         SS
30      } optSemi
31 end rule
32
%% This rule tags the Lambda Class definition.
rule markLambdaClass P [program]
skipping [js_subscope]
replace $ [js_declaration_or_statement]
  LHS [js_member_expn] '\=' 'function FName [opt id]'
  '\{'
    SS [js_subscope]
  '\'} optSemi [opt '\;']
  deconstruct * [js_member_expn] LHS
  Name [id]
  deconstruct * [js_lefthand_side_expn] P
    'new Name _[js.arguments]' by
  LHS = 'function <<<Name 'LambdaClass 'ClassDef >>>> FName (Params)'
    '\{'
      SS
    '\'} optSemi
end rule

%% This rule tags the Variable Lambda Class definition
rule markVarLambdaClass P [program]
skipping [js_subscope]
replace $ [js_declaration_or_statement]
  'var Name [id] '\=' 'function FName [opt id]'
  '\{'
    SS [js_subscope]
  '\'} optSemi [opt '\;']
  deconstruct * [js_lefthand_side_expn] P
    'new Name _[js.arguments]' by
  'var Name = 'function
    <<<Name 'VarLambdaClass 'ClassDef >>>> FName (Params)
      '\{'
        SS
      '\'} optSemi
end rule
Appendix C

Style Analysis Code

% This txl program is to identify all class patterns: inner method,
% inner lambda, outer method, outer lambda and prototype lambda

#include "Javascript.Grm"
#include "factOverride.Grm"

function main
replace [program]
  P [program]
  export listOfAssignments [repeat class_method_value]
  - % empty
by
  P[tag_js_prototypeLambda]
  [markPossibleAssignmentsFunctionClass]
  [markPossibleAssignmentsLambdaClass]
  [markPossibleAssignmentsVarLambdaClass]
  [tag_js_innerMethod_innerLambda]
  [tag_js_outerMethod]
  [tag_js_outerLambda]
  [ClassFunctionSearch]
  [removeAssignmentTags]
end function

% This rule tags the prototype lambda style.
rule tag_js_prototypeLambda
replace [js_assignment_expn]
  LHS [js_member_expn] ' = ' function Name [opt id]
  ( Parms [list js_formal_parameter] )
  '{
    SS [js_subscope]
  }

deconstruct * [id] LHS
  'prototype
deconstruct * [repeat js_selector] LHS
  mName [id]
deconstruct LHS
  ClassName [id] 'prototype Rest [repeat js_selector]
by
LHS = 'function <<< ClassName mName 'PrototypeLambda >>>> Name (Parms)
  '{
    SS
  }
end rule

% Defining a new non terminal so we insert an intermediate tag for inner
% and outer method and outer lambda.
% This tags all assignments in the class and later checks whether this
% is an inner, outer method or outer lambda.

define c_m_v
  [repeat class_method_value]
end define

define class_method_value
  '<<<< [id] [id] [js_member_expn] >>>>
end define

redefine js_assignment_expn
  [js_conditional_expn] [repeat assign_js_conditional_expn]
  [opt class_method_value]
end redefine

% This rule looks inside the FunctionClass definition and calls the
% rule markEachPossibleAssignment that tags
% all possible assignment.

rule markPossibleAssignmentsFunctionClass
  skipping [js_subscope]
replace $ [js_function_declaration]
  'function <<< CName [id] 'FunctionClass 'ClassDef >>>> ClassName [id]
  (Params [list js_formal_parameter])
  '{
    SS [js_subscope]
  }
by
  'function <<< CName 'FunctionClass 'ClassDef >>>> ClassName (Parms)
APPENDIX C. STYLE ANALYSIS CODE

82 ' {
83     SS [innerLambda ClassName]
84     [markEachPossibleAssignment ClassName]
85     }
86 end rule
87
88 % This rule looks inside the Lambda Class definition and calls the rule
89 % markEachPossibleAssignment that tags all possible assignment.
90 rule markPossibleAssignmentsLambdaClass
91 skipping [js_subscope]
92 replace $ [js_declaration_or_statement]
93 LHS [js_member_expn] ' = ' function
94     <<< ClassName [id] 'ClassName' ClassDef >>>>
95     FName [opt id] '( Params [list js_formal_parameter] )'
96     '{
97     SS [js_subscope]
98     } optSemi [opt ' ; ']
99 by
100 LHS = 'function <<< ClassName 'ClassName' ClassDef >>>> FName (Params)
101     '{
102     SS [innerLambda ClassName]
103     [markEachPossibleAssignment ClassName]
104     }
105 end rule
106
107 % This rule looks inside the Variable Lambda Class definition and calls the rule
108 % markEachPossibleAssignment that tags all possible assignment.
109 rule markPossibleAssignmentsVarLambdaClass
110 skipping [js_subscope]
111 replace $ [js_declaration_or_statement]
112 ' var ClassName [id] ' = ' function
113     <<< Name [id] 'VarLambdaClass' ClassDef >>>>
114     FName [opt id] '( Params [list js_formal_parameter] )'
115     '{
116     SS [js_subscope]
117     } optSemi [opt ' ; ']
118 by
119 ' var ClassName = 'function
120     <<< Name 'VarLambdaClass' ClassDef >>>> FName (Params)
121     '{
122     SS [innerLambda ClassName]
123     [markEachPossibleAssignment ClassName]
124     }
125 end rule
126
127 % This rule does the actual assignment tagging.
rule markEachPossibleAssignment ClassName [id]  
replace [js_assignment.expn]  
  thisMethodName [id] = FunctionName [js_member.expn]  
by  
  this.MethodName = FunctionName <<< ClassName MethodName FunctionName >>>>  
end rule  

% Now we search for inner method. We create a global variable that consists  
% of a list of all the markups that are added to the assignments.  
function tag.js_innerMethod_innerLambda  
replace [program]  
P [program]  
  construct CandidateList [repeat class_method.value]  
  - ["P"]  
by  
P [tag.js_innerFunction CandidateList]  
  [tag.js_innerFunctionLambdaClass CandidateList]  
  [tag.js_innerFunctionVarLambdaClass CandidateList]  
end function  

% Searching for inner methods inside Function Class definitions.  
rule tag.js_innerFunction CandidateList [repeat class_method.value]  
replace $ [js_function_declaration]  
  'function <<< CName [id] 'FunctionClass 'ClassDef >>>> ClassName [id]  
  ( Params [list js_formal_parameter] )  
  '{  
    SS [js_subscope]  
  }  
by  
  'function <<< CName 'FunctionClass 'ClassDef >>>> ClassName ( Params)  
  '{  
    SS [innerMethod CandidateList]  
  }  
end rule  

% Searching for inner methods inside Lambda Class definitions.  
rule tag.js_innerFunctionLambdaClass CandidateList [repeat class_method.value]  
replace $ [js_declaration_or_statement]  
LHS [js_member.expn] = 'function  
<<< ClassName [id] 'LambdaClass 'ClassDef >>>>  
  FName [opt id] '( Params [list js_formal_parameter] ')  
  '{  
    SS [js_subscope]  
  } optSemi [opt ';']  
by  
LHS = 'function <<< ClassName 'LambdaClass 'ClassDef >>>> FName (Params)  
  '{
APPENDIX C. STYLE ANALYSIS CODE

178 SS [innerMethod CandidateList]
179 }optSemi
180 end rule
181
182 % Searching for inner methods inside Variable Lambda Class definitions.
183 rule tag_js_innerFunctionVarLambdaClass CandidateList [repeat class_method_value]
184 replace $ [js_declaration_or_statement]
185 'var Name [id] 'function
186 <<<<< ClassName [id] 'VarLambdaClass 'ClassDef >>>>>
187 FName [opt id] 'FName ( Params [list js_formal_parameter] )
188 '{
189 SS [js_subscope]
190 } optSemi [opt ;]
191 by
192 'var Name 'function
193 <<<<< ClassName 'VarLambdaClass 'ClassDef >>>>>
194 FName ( Params )
195 '{
196 SS [innerMethod CandidateList]
197 } optSemi
198 end rule
199
200 % Tagging the inner method, if only there is a tag of a previously marked up
201 % assignment that has the method name and the class name.
202 rule innerMethod CandidateList [repeat class_method_value]
203 replace [js_function_declaration]
204 'function MName [id] ( Params [list js_formal_parameter] )
205 '{
206 SS [js_subscope]
207 }
208
deconstruct * [class_method_value] CandidateList
209 <<<<< CName [id] MethodName [id] MName >>>>>
210 import listOfAssignments [repeat class_method_value]
211 export listOfAssignments
212 '<<<< CName MethodName MName >>>>>
213 listOfAssignments
214 by
215 'function '<<<< CName MethodName 'innerMethod >>>>> MName ( Params )
216 '{
217 SS
218 }
219 end rule
220
221 % Tagging inner lambda style
222 rule innerLambda CName [id]
APPENDIX C. STYLE ANALYSIS CODE

226 skipping [js_assignment_expn]
227 replace [js_assignment_expn]
228   LHS [js_member_expn] ' = ' 'function Name [opt id]
229     ( Params [list js_formal_parameter] ) '
230     '
231     '}
232     SS [js_subscope]
233     '}
234     deconstruct * [js_primary_expn] LHS
235     'this
236     deconstruct not * [repeat js_selector] LHS
238     deconstruct * [repeat js_selector] LHS
239     . mName [id]
240     by
241     LHS = 'function <<<< CName mName 'innerLambda >>>> Name ( Params )
242     '
243     '}
244     SS
245     '}
246 end rule
247
248 % This function is to identify the outer method
249 function tag_js_outerMethod
250 replace [program]
251     P [program]
252     construct CandidateList [repeat class_method_value]
253     . [^ P]
254 by
255     P [markEachOuterMethod CandidateList]
256 end function
257
258 rule markEachOuterMethod CandidateList [repeat class_method_value]
259 skipping [js_subscope]
260 replace [js_function_declaration]
261     'function FunctionNamed [id] ( Params [list js_formal_parameter] )
262     '
263     '}
264     SS [js_subscope]
265     '}
266     deconstruct * [class_method_value] CandidateList
267     <<<< cName [id] MName [id] FunctionNamed >>>>
268     import listOfAssignments [repeat class_method_value]
269     export listOfAssignments
270     '<<<< cName MName FunctionNamed '>>>>
271     listOfAssignments
272 by
273
APPENDIX C. STYLE ANALYSIS CODE

274     'function <<< cName MName 'OuterMethod >>>> FunctionNamed (Params)
275     '{
276         SS
277     '}
278 end rule
279
280 function tag js outerLambda
281 replace [program]
282     P [program]
283     construct CandidateList [repeat class method value]
284         - ['P]
285 by
286     P [markEachOuterLambda CandidateList]
287 end function
288
289 rule markEachOuterLambda CandidateList [repeat class method value]
290 skipping [js subscope]
291 replace [js assignment expn]
292     LHS [js member expn] = 'function Name [opt id]
293         '{ Parms [list js formal parameter] '}
294         '
295         SS [js subscope]
296         '}
297
deconstruct * [repeat js selector] LHS
298     . MethodName [id]
299
deconstruct LHS
300     ClassName [id] ' . MethodName Rest [repeat js selector]
301
deconstruct * [class method value] CandidateList
302     <<< ClassName mName [id] ClassName '. MethodName >>>>
303
304 import listOfAssignments [repeat class method value]
305 export listOfAssignments
306     '<<< ClassName mName ClassName '. MethodName >>>>
307 listOfAssignments
308 by
309     LHS = 'function <<< ClassName mName 'OuterLambda >>>> Name (Parms )
310         '
311         '{
312         SS
313         '}
314 end rule
315
316 function isLambdaClass Name [id]
317     match * [js assignment expn]
318     Name = _ [js function expn]
319 end function
APPENDIX C. STYLE ANALYSIS CODE

322 function isClassName [id]
323    match * [js_function_declaration]
324      'function Name ( _[list js_formal_parameter] )
325        '{
326          _[js_subscope]
327        '}
328 end function
329
330 function isVarLambdaClassName [id]
331    match * [js_variable_declaration]
332      'var Name = . [js_function_expn]
333 end function
334
335 % This function searches classes to remove all methods assignments to properties,
336 % except for inner lambda.
337 function ClassFunctionSearch
338    replace [program]
339      P [program]
340        import listOfAssignments [repeat class_method_value]
341 by
342      P [removePropAssignInClass each listOfAssignments]
343 end function
344
345 % This rule removes the assignment statement of the inner, outer methods
346 % and outer lambda.
347 rule removePropAssignInClass propAssign [class_method_value]
348    deconstruct propAssign
349      <<< ClassName [id] MethodName [id] FunctionName [js_member_expn] '>>>>
350    replace [repeat js_declaration_or_statement]
351      'this '. MethodName '=' FunctionName
352      <<< ClassName MethodName FunctionName >>>> optSemi [opt ';
353 by
354      Rest [repeat js_declaration_or_statement]
355
356    end rule
357
358 % This rule cleans up the code by removing the tagged assignments inside
359 % the classes.
360 rule removeAssignmentTags
361    replace [js_assignment_expn]
362      this .MethodName [id] = FunctionName [js_member_expn]
363      <<< .[id] .[id] .[js_member_expn] >>>>
364 by
365      this .MethodName = FunctionName
366 end rule
Appendix D

Style Normalization Code

D.1 Class Untagging

1 %This is a clean up code that removes all the class definition tags
2 % so the style normalization program looks for tagged methods only.
3 include "Javascript.Grm"
4 include "factOverride.Grm"
5
6 function main
7 replace [program]
8 P [program]
9 by
10 P [removeFunctionClassTag]
11 [removeLambdaClassTag]
12 [removeVarLambdaClassTag]
13 end function
14
15 % This rule removes the mark up from the Function Class definition.
16 rule removeFunctionClassTag
17 skipping [js_subscope]
18 replace $ [js_declaration_or_statement]
19 'function <<<< .[id] 'FunctionClass 'ClassDef >>>> FName [id]
20 ' (Params [ list js_formal_parameter ] ')
21 '{
22 SS [js_subscope]
23 } optSemi [opt ' ; ]
24 by
25 'function FName ( Params )
26 '{
27 SS
28 } optSemi
29 end rule
30
APPENDIX D. STYLE NORMALIZATION CODE

31 % This rule removes the mark up from the Lambda Class definition.
32 rule removeVarLambdaClassTag
33 skipping [js_subscope]
34 replace $ [js_declaration_or_statement]
35       'var Name [id]' 'function <<<< .[id] 'LambdaClass 'ClassDef >>>>
36           FName [opt id] '(Params [list js_formal_parameter] ')
37           '{
38               SS [js_subscope]
39               '} optSemi [opt ';']
40 by
41       'var Name' 'function FName ( Params)
42           '{
43               SS
44           '} optSemi
45 end rule
46
47 % This rule removes the mark up from the Variable Lambda Class definition.
48 rule removeLambdaClassTag
49 skipping [js_subscope]
50 replace $ [js_declaration_or_statement]
51       LHS [js_member_expn] 'function <<<< .[id] 'VarLambdaClass 'ClassDef
52           'function 'LambdaClass 'ClassDef >>>>
53           FName [opt id] '(Params [list js_formal_parameter] ')
54           '{
55               SS [js_subscope]
56           '} optSemi [opt ';']
57 by
58       LHS 'function FName ( Params)
59           '{
60               SS
61           '} optSemi
62 end rule

D.2 Style Normalization

1 % This is the Style Normalization program that transforms all four methods:
2 % inner and outer methods and %inner and outer lambda styles.
3
4 include "Javascript.Gm"
5 include "factOverride.Gm"
6
7 redefine js_compound_statement
8     '{
9     [repeat js_declaration_or_statement]
10     '}
11 end redefine
```java
rdefine js_case_clause
  [js_case_clause_value]
[repeat js_declaration_or_statement]
end redefine

function main
replace [program]
P [program]
by
  P[transform_prototype_lambda]
  [transform_outer_lambda]
  [transform_outer_method]
  [transformInnerMethods]
  [transformInnerMethodsInLambdaClass]
  [transformInnerMethodsInVarLambdaClass]
  [transformInnerLambda]
  [transformInnerLambdaInLambdaClass]
  [transformInnerLambdaInVarLambdaClass]
end function

% this rule removes the tag from the prototype lambda style.
rule transform_prototype_lambda
replace [js_assignment_exp]
  LHS [js_member_exp] = 'function
  <<<<< ClassName [id] mName [id] 'PrototypeLambda >>>>
  Name [opt id] ' ( Params [list js_formal_parameter] ' )
  '{
    SS [js_subscope]
  }
by
  LHS = 'function Name ' ( Params ' )
  '{
    SS
  }
end rule

% This rule transforms the outer lambda style.
rule transform_outer_lambda
replace [js_assignment_exp]
  LHS [js_member_exp] = 'function
  <<<<< ClassName [id] MethodName [id] 'OuterLambda >>>>
  fName [opt id] ' ( Params [list js_formal_parameter] ' )
  '{
    SS [js_subscope]
  }
by
  ClassName. 'prototype.MethodName ' = 'function fName' ( Params ' )
  '
```

APPENDIX D. STYLE NORMALIZATION CODE

rule transform_outer_method
  replace [js_declarator_or_statement]
    'function '<<<< className [id] MethodName [id] 'OuterMethod >>>>
      functionName [id] '( Params [list js_formal_parameter] ')
    '{
      SS [js_subscope]
    }
  by
    className.'prototype.MethodName '="function '(' Params ')
    '{
      SS
    }
end rule

rule transformInnerMethods
  replace [repeat js_declarator_or_statement]
    'function ClassName [id] '(' Params [list js_formal_parameter] ')
    '{
      ss [repeat js_declarator_or_statement]
      '}optSemi [opt ;]
    restOfScope [repeat js_declarator_or_statement]
  deconstruct * [js_declarator_or_statement] ss
    'function '<<<< _ [id] _ [id] 'innerMethod >>>>
      _ [id]
    '{
      _ [list js_formal_parameter] ')
    '{
      _ [repeat js_declarator_or_statement]
    } [opt ;]
  construct InnersUnchanged [repeat js_declarator_or_statement]
    _ [extractFun each ss]
  construct NewSS [repeat js_declarator_or_statement]
    ss [addThisVar InnersUnchanged]
    [addThisUndeclaredVar InnersUnchanged]
    [addThisParam InnersUnchanged Params]
  construct InnersChanged [repeat js_declarator_or_statement]
    _ [extractFun each NewSS]
  by
    'function ClassName '(' Params ')
    '{
      NewSS [removeInnerMethods]
    } optSemi
InnersChanged [ . restOfScope ]
end rule

rule transformInnerMethodsInLambdaClass
replace [repeat js_declaration_or_statement]
  LHS [js_member_expn] ' = ' function FName [opt id]
       ' ( Params [list js_formal_parameter] )'
       '{
         ss [repeat js_declaration_or_statement]
       } optSemi [opt ';']
restOfScope [repeat js_declaration_or_statement]
deconstruct * [js_declaration_or_statement] ss
  ' function '<<<. id >>. id' 'innerMethod '>>>>. id
  ' ( . [list js_formal_parameter] )
  '{
    ss [repeat js_declaration_or_statement]
  } . [opt ;]

construct InnersUnchanged [repeat js_declaration_or_statement]
  . [extractFun each ss]
construct NewSS [repeat js_declaration_or_statement]
  ss [addThisVar InnersUnchanged]
  [addThisUndeclaredVar InnersUnchanged]
  [addThisParam InnersUnchanged Params]
construct InnersChanged [repeat js_declaration_or_statement]
  . [extractFun each NewSS]
by
  LHS = ' function FName ' ( Params ' )
  '{
    NewSS [removeInnerMethods]
  } optSemi
InnersChanged [ . restOfScope ]
end rule

rule transformInnerMethodsInVarLambdaClass
replace [repeat js_declaration_or_statement]
  ' var Name [id] ' = ' function FName [opt id]
       ' ( Params [list js_formal_parameter] )'
       '{
         ss [repeat js_declaration_or_statement]
       } optSemi [opt ';']
restOfScope [repeat js_declaration_or_statement]
157 deconstruct * [js_declaration_or_statement] ss
158   'function '<<<< - [id] - [id] 'innerMethod '>>>> - [id]
159       '( - [list js_formal_parameter] ')
160       '{
161         - [repeat js_declaration_or_statement]
162       '} - [opt ;]
163
164 construct InnersUnchanged [repeat js_declaration_or_statement]
165       - [extractFun each ss]
166
167 construct NewSS [repeat js_declaration_or_statement]
168       ss [addThisVar InnersUnchanged]
169       [addThisUndeclaredVar InnersUnchanged]
170       [addThisParam InnersUnchanged Params]
171
172 construct InnersChanged [repeat js_declaration_or_statement]
173       - [extractFun each NewSS]
174   by
175   'var Name = 'function FName '( Params ')
176       '{
177         NewSS [removeInnerMethods]
178       }optSemi
179
180 InnersChanged [. restOfScope]
181 end rule
182
183 rule addThisVar Inners [repeat js_declaration_or_statement]
184   skipping [js_subscope]
185   replace $ [repeat js_declaration_or_statement]
186       'var Vars [list js_variable_description] optSemi[opt '];
187       Rest [repeat js_declaration_or_statement]
188
189 construct Assigns [repeat js_declaration_or_statement]
190       - [buildFieldAssignFromVarDescr Inners each Vars]
191
192 construct newVarStatement [repeat js_declaration_or_statement]
193       'var Vars [removeAnyBoundVariables each Assigns] optSemi
194
195 construct FixedRest [repeat js_declaration_or_statement]
196       Rest [fixBoundsVars each Assigns]
197
198 construct NewStatements [repeat js_declaration_or_statement]
199       newVarStatement [. Assigns]
200 by
201       NewStatements [removeVarStatementIfEmtpy]
202       [. FixedRest]
203 end rule
204
function fixBoundVars anAssign [js_declaration_or_statement]
    deconstruct anAssign
    'this . anAssignVar [id] - [repeat assign_js_conditional_expn]
        - [opt ';
    replace [repeat js_declaration_or_statement]
    x [repeat js_declaration_or_statement]
    by
    x [fixVarUses anAssignVar]
end function

function buildFieldAssignFromVarDescr Inners [repeat js_declaration_or_statement]
    aVarDescr [js_variable_description]
    replace [repeat js_declaration_or_statement]
    X [repeat js_declaration_or_statement]
    deconstruct aVarDescr
    VarName [id] Value [opt js_variable_initialization]
    deconstruct * [id] Inners
    VarName
    construct NewStatement1 [js_declaration_or_statement]
        'this . VarName = '0 ';
    construct NewStatement [js_declaration_or_statement]
        NewStatement1 [addInitIfGiven Value]
    by
    X [. NewStatement]
end function

function removeAnyBoundVariables anAssign [js_declaration_or_statement]
    deconstruct anAssign
    'this . VarName [id] - [assign_js_conditional_expn]
        optSemi [opt ';
    replace [list js_variable_description]
    X [list js_variable_description]
    by
    X [removeFirstVar VarName]
    [removeEndVar VarName]
    [removeSingleVar VarName]
end function

function removeFirstVar aVarName [id]
    replace * [list js_variable_description]
        aVarName - [opt js_variable_initialization] '
    by
        Rest [list js_variable_description]
end function

function removeFirstVar aVarName [id]
    replace * [list js_variable_description]
        aVarName - [opt js_variable_initialization] '
    by
        Rest [list js_variable_description]
end function

APPENDIX D. STYLE NORMALIZATION CODE 106
end function

function removeEndVar VarName [id]
replace * [list js_variable_description]
    Prev [js_variable_description],
    VarName . [opt js_variable_initialization]
by
    Prev
end function

function removeSingleVar VarName [id]
replace [list js_variable_description]
    VarName . [opt js_variable_initialization]
by
    %empty
end function

function removeVarStatementIfEmtpy
replace [repeat js_declaration_or_statement]
    'var optSemi [opt ';]
    Rest [repeat js_declaration_or_statement]
by
    Rest
end function

function addInitIfGiven Value [opt js_variable_initialization]
deconstruct Value
    '=' E [js_assignment_expn]
replace [js_declaration_or_statement]
    'this.Name [id] '=' 0 optSemi [opt ';]
by
    'this.Name '=' ( E ) optSemi
end function

function addThisParam Inners [repeat js_declaration_or_statement]
    Params [list js_formal_parameter]
replace [repeat js_declaration_or_statement]
    Stmts[repeat js_declaration_or_statement]
    construct NewStatements [repeat js_declaration_or_statement]
    . [addEachParam Inners each Params]
    deconstruct not NewStatements
    % empty
    construct RenamedStatements [repeat js_declaration_or_statement]
    Stmts [fixParamUses each Params]
by
APPENDIX D. STYLE NORMALIZATION CODE

301 NewStatements
302 [ . RenamedStatements]
303 end function
304
305 function addEachParam Inners [repeat js_declaraton_or_statement]
306     aParam [js_formal_parameter]
307     deconstruct aParam
308     ParamId [id]
309     deconstruct * [id] Inners
310     ParamId
311     replace [repeat js_declaraton_or_statement]
312     X [repeat js_declaraton_or_statement]
313     construct NewStatement [js_declaraton_or_statement]
314     'this.ParamId = ParamId ';
315 by
316     X [ . NewStatement]
317 end function
318
319 function fixParamUses aParam [js_formal_parameter]
320     deconstruct aParam
321     ParamId [id]
322     replace [repeat js_declaraton_or_statement]
323     x [repeat js_declaraton_or_statement]
324     by
325     x [fixVarUses ParamId]
326 end function
327
328 rule addThisUndeclaredVar Inners [repeat js_declaraton_or_statement]
329 skipping [js_subscope]
330     replace $ [repeat js_declaraton_or_statement]
331     VarName [id] = VarValue [js_conditional_expn] optSemi [opt ']
332     Rest [repeat js_declaraton_or_statement]
333     deconstruct * [id] Inners
334     VarName
335     construct ThisVar [js_declaraton_or_statement]
336     'this.VarName ' = VarValue optSemi
337 by
338     ThisVar
339     Rest [fixVarUses VarName]
340 end rule
341
342 rule fixVarUses OldName [id]
349 replace $ [js_member_expn]
350     OldName Sel [repeat js-selector]
351 by
352     this. OldName Sel
353 end rule
354
355 function extractFun functionDecl [js_declaration_or_statement]
356     deconstruct * functionDecl
357     'function '<<<< className [id] methodName [id] 'innerMethod '>>>>
358         fname [id] '('. params [list js_formal_parameter] ')'
359         '{
360             sub [js_subscope]
361             '} optSemi [opt ;]
362     end rule
363
364 replace [repeat js_declaration_or_statement]
365     ExtractedFuns [repeat js_declaration_or_statement]
366     construct innerMethod [js_declaration_or_statement]
367         className . 'prototype . methodName = ' function '(' params ')
368         '{
369             sub
370             '} optSemi
371 by
372     ExtractedFuns [. innerMethod]
373 end function
374
375 rule removeInnerMethods
376 replace [repeat js_declaration_or_statement]
377     'function '<<<< className [id] methodName [id] 'innerMethod '>>>>
378         fname [id] '('. params [list js_formal_parameter] ')'
379         '{
380             sub [js_subscope]
381             '} . [opt ;]
382     More [repeat js_declaration_or_statement]
383 by
384     More
385 end rule
386
387 rule transformInnerLambda
388 replace $ [repeat js_declaration_or_statement]
389     'function ClassName [id] '('. Params [list js_formal_parameter] ')
390     '{
391         ss [repeat js_declaration_or_statement]
392         '} optSemi [opt ;]
393     restOfScope [repeat js_declaration_or_statement]
394     deconstruct * [js_declaration_or_statement] ss
APPENDIX D. STYLE NORMALIZATION CODE

110

- [js_member_expn] = 'function
  '<<<< - [id] - [id] 'innerLambda >>>>
  - [opt id] '( - [list js_formal_parameter] )
  '{
    - [repeat js_declaration_or_statement]
  }
  - [opt ';

construct InnersUnchanged [repeat js_declaration_or_statement]
  - [extractLambda each ss]

construct NewSS [repeat js_declaration_or_statement]
  ss [addThisVar InnersUnchanged]
  [addThisUndeclaredVar InnersUnchanged]
  [addThisParam InnersUnchanged Params]

construct InnersChanged [repeat js_declaration_or_statement]
  - [extractLambda each NewSS]

by
  'function ClassName '( Params )
  '{
    NewSS [removeInnerLambda]
  }optSemi
InnersChanged [. restOfScope]
end rule

rule transformInnerLambdaInLambdaClass
  replace $ [repeat js_declaration_or_statement]

  LHS [js_member_expn] = 'function FName [opt id]
    '( Params [list js_formal_parameter] )
    '{
      ss [repeat js_declaration_or_statement]
      }optSemi [opt ';
    restOfScope [repeat js_declaration_or_statement]

  deconstruct * [js_declaration_or_statement] ss
    - [js_member_expn] = 'function
      '<<<< - [id] - [id] 'innerLambda >>>>
      - [opt id] '( - [list js_formal_parameter] )
      '{
        - [repeat js_declaration_or_statement]
      }
      - [opt ';

  construct InnersUnchanged [repeat js_declaration_or_statement]
  - [extractLambda each ss]

  construct NewSS [repeat js_declaration_or_statement]
  ss [addThisVar InnersUnchanged]
APPENDIX D. STYLE NORMALIZATION CODE

```
445     [addThisUndeclaredVar InnersUnchanged]
446     [addThisParam InnersUnchanged Params]
447
448     construct InnersChanged [repeat js.declaration_or_statement]
449       . [extractLambda each NewSS]
450     by
451     LHS = 'function FName '( Params ')
452       '{
453         NewSS [removeInnerLambda]
454       } optSemi
455     InnersChanged [. restOfScope]
456   end rule
457
458   rule transformInnerLambdaInVarLambdaClass
459   replace $ [repeat js.declaration_or_statement]
460     'var Name [id] '=' 'function FName [opt id]
461       '( Params [list js_formal_parameter] ')
462       '{
463         ss [repeat js.declaration_or_statement]
464       } optSemi [opt ';
465     restOfScope [repeat js.declaration_or_statement]
466
467     deconstruct * [js.declaration_or_statement] ss
468       . [js_member_expn] '=' 'function
469       '<<<< [id] [id] 'innerLambda >>>>
470         . [opt id] '( . [list js_formal_parameter] ')
471       '{
472         . [repeat js.declaration_or_statement]
473       } . [opt ';
474
475     construct InnersUnchanged [repeat js.declaration_or_statement]
476       . [extractLambda each ss]
477
478     construct NewSS [repeat js.declaration_or_statement]
479     ss [addThisVar InnersUnchanged]
480     [addThisUndeclaredVar InnersUnchanged]
481     [addThisParam InnersUnchanged Params]
482
483     construct InnersChanged [repeat js.declaration_or_statement]
484     by
485     'var Name = 'function FName '( Params ')
486       '{
487         NewSS [removeInnerLambda]
488       } optSemi
489     InnersChanged [. restOfScope]
490   end rule
491
```
function extractLambda functionDec [js_declaration_or_statement]
  deconstruct * functionDec
  LHS [js_member_expn] ' = 'function
  '<<<< className [id] methodName [id] 'innerLambda '>>>>
  fName [opt id] '(' (params [list js_formal_parameter]) ')
  '{
    sub [js_subscope]
  } OptSemi [opt ';']
replace [repeat js_declaration_or_statement]
  ExtractedFuns [repeat js_declaration_or_statement]
construct innerL [js_declaration_or_statement]
  className . 'prototype . methodName = 'function fName '(params')
  '{
    sub
  } OptSemi
by
  ExtractedFuns [. innerL]
end function

rule removeInnerLambda
replace [repeat js_declaration_or_statement]
  LHS [js_member_expn] ' = 'function
  '<<<< className [id] methodName [id] 'innerLambda '>>>>
  fName [opt id] '(' (params [list js_formal_parameter]) ')
  '{
    sub [js_subscope]
  } - [opt ';']
  More [repeat js_declaration_or_statement]
by
  More
end rule