JavaScript Code Quality Analysis

Master's Thesis

Joost-Wim Boekesteijn

JavaScript Code Quality Analysis

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Joost-Wim Boekesteijn
born in Hoogeveen, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
`www.ewi.tudelft.nl`

M-Industries BV
Rotterdamseweg 183c
Delft, the Netherlands
`www.m-industries.com`

JavaScript Code Quality Analysis

Author:	Joost-Wim Boekesteijn
Student id:	1174355
Email:	joost-wim@boekesteijn.nl

Abstract

Static analysis techniques provide a means to detect software errors early in the development process, without actually having to run the software that is being analyzed. These techniques are common for statically typed languages and have found their way into IDEs such as Eclipse and Visual Studio. However, applying the same techniques to dynamically typed languages is much less common. Tool support is less mature and the amount of published research is relatively small.

For this project, we design and build a static analyis tool for JavaScript code. We start by giving background information on relevant parts of the JavaScript language, followed by a survey of existing tools and research. In the design of our analysis tool, we achieved a clear separation of responsibilities between the different modules for parsing, analysis, rule definition and reporting. The level of detail in the default reporter makes our tool an ideal candidate for integration in a JavaScript IDE. On the other hand, our tool is also suited for batch analysis of large code collections.

To validate our tool, we set up an experiment in which we ran our analysis tool on two large collections of JavaScript code: one from a repository of open source JavaScript packages, mostly for server-side use; another one gathered from client-side code on a large number of popular websites. We present high-level global results as well as more detailed results for selected projects and websites.

Preface

During the last seven months, starting in December 2011, I worked on my graduation project for M-Industries BV in Delft. This thesis is both the end result of that project and the final report for my Master's degree in Computer Science from the Delft University of Technology.

Overall, doing the research and working on my project has been a very rewarding and positive experience. It provided enough interesting challenges to keep me motivated and it gave me lots of opportunities to learn new things, while it also allowed me to apply the knowledge and the skills that I acquired over the past several years.

Completing this project successfully would not have been possible without the help and support of a number of people. First of all, I would like to thank Andy, my university supervisor. He put me in touch with M-Industries and helped me with the definition of my project. He was always encouraging and constructive in his feedback and criticism on my ideas for research, the plans for my project and the draft versions of my thesis.

Next, I would like to thank the team at M-Industries. First of all Corno, who has been my supervisor within the company. I learned from his views on model-driven development and I enjoyed our discussions on the design and implementation of my analysis tool. Other than him, I would also like to thank Rick, Gerbert, Dan and Gerben, my colleagues at M-Industries, for showing interest in my work, giving feedback on my tool and providing an environment in which I could completely focus on my project.

Finally, I am grateful to my parents for their continuous support and encouragement throughout my studies.

Joost-Wim Boekesteijn
Delft, the Netherlands
July 2, 2012

Contents

List of Figures

Chapter 1

Introduction

For software engineers, JavaScript might not be an obvious choice as the language to use for their next project. JavaScript has objects, but no classes. It has functions as first-class citizens, meaning functions can appear in all places where one might normally pass around values or objects. It allows dynamic code evaluation. Several libraries and frameworks rely heavily on the usage of closures, which at first sight can look intimidating compared to programming styles found in synchronous, object-oriented systems. Furthermore, the language is dynamically typed, which means that you will never see a type specification on a variable declaration or a function parameter. Because of its origins, the language is usually thought to only be useful as a scripting language for web browsers. However, as we will see, several language properties and characteristics actually make the language a good fit for non-browser application development as well. In recent years, this has been popularized by frameworks such as Node.js[1], which mainly support an event-driven, asynchronous programming style outside the web browser environment.

Because the language is relatively young, the level of tooling surrounding the language is lacking in several areas, especially when compared to a language like Java. This applies to editor support (e.g. autocompletion, debugging) as well as other aspects such as code quality analysis, unit testing and code coverage tracking. This is also reflected in the amount of published research on topics specifically related to JavaScript.

This thesis describes a project that has been carried out at M-Industries BV[2] in Delft, a company that uses the JavaScript language for web application development, both server- and client-side. Before we will describe the problem statement and the research questions, we will give more details on the problem context.

1.1 Context

The practical part of this project is carried out at a company, but this project also has a scientific component that relates to existing research in the field of software engineering. We will first describe which problems M-Industries encountered that led to this project, then we will show how we can use existing software engineering

[1] `http://nodejs.org/`

[2] `http://www.m-industries.com/`

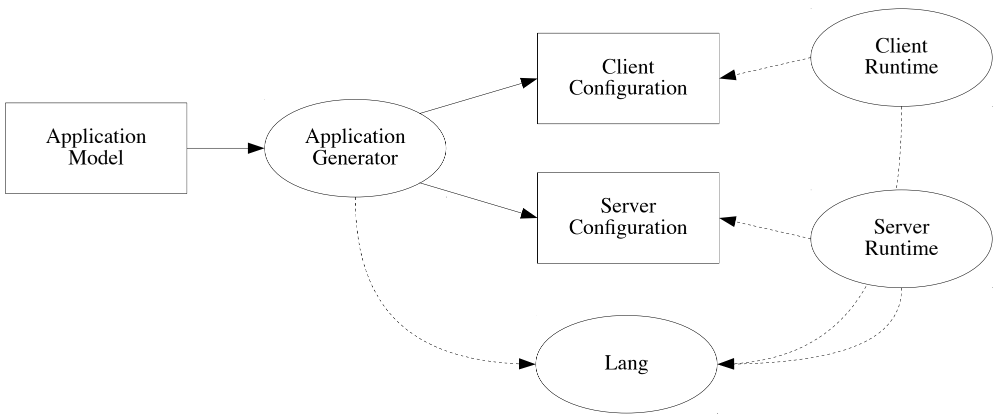


Figure 1.1: High-level M-Industries platform overview.

research to help us solve several types of problems that we will encounter during this project.

1.1.1 Company

M-Industries uses a model-driven approach to building software, where the focus is on accurately describing a client's business processes using their own data modeling language (the 'schema language', see appendix A for more details) and a supporting software framework. The combination of the schema language and the framework is known as the M-Industries platform. With this platform, the company currently targets industrial customers who need to create, extend or replace enterprise resource planning systems for production and supply chain management. The platform is being used to successfully model and control complex industrial production processes.

The actual application that is delivered to a customer is generated from a description of the relevant business processes in the schema language (known as the 'application model'), combined with the logic that operates on the model data, using the accompanying framework. The graphical interface is automatically generated from a transformed application model and a set of reusable widgets definitions, resulting in a web application using standard HTML, CSS and JavaScript. The end users only interact with the system using their web browser. The generated application provides a fine-grained CRUD-style[3] interface for the core entities defined in the application model. An important notion is that the models should not be seen as purely descriptive, but as 'executable models', as a way to actually specify the implementation of a system.

Figure 1.1 shows the transformation from application model to the separate client and server configuration models, which are executed by the different runtime modules. Rectangles represent data, ellipses show the code. Solid lines denote data-flow, dashed lines indicate dependencies. The application generator, the client runtime and the server runtime all depend on the core 'lang' module, which provides all essential operations to work with models and datasets.

[3]CRUD: Create, Read, Update and Delete.

From a purely technical viewpoint, the entire platform is ultimately used to generate a web application and to provide the runtime components necessary to execute the application. The server-side part of the framework was previously written in PHP, but has been migrated to JavaScript running in Node.js. This migration was done because the JavaScript language seemed more suitable for code analysis tools and because it allowed sharing of code between the client (i.e. the browser) and the server.

The use of the JavaScript language seems a bad fit for a statically typed schema language, as JavaScript has dynamic, weak typing and is very flexible in constructing objects and changing them at runtime. Compare this to the schema language, which relies on static typing and forces all runtime data to always conform to the schema definition. There are no native language constructs in JavaScript that support this, which means that all operations on datasets need to be done using the framework. The framework libraries guarantee that creation or modification of a dataset results in a new dataset that still conforms to the schema. Building the statically typed schema language on top of a language with a static type system would probably have resulted in a mismatch between the two type systems. Therefore, the choice was made to create a static type system on top of a dynamically typed language.

We observe that the schema language and the framework restrict the software development process in one way, but the JavaScript programming language still allows developers to circumvent these restrictions. The ultimate goal would be to replace JavaScript with a set of domain specific languages (DSLs) that force developers to use patterns that are in line with the design philosophy behind the platform. However, it is not feasible to create these languages from scratch, because the desired features, concepts and patterns are still very much a moving target. M-Industries decided to keep using the JavaScript language in a restricted form until they can migrate to their own DSLs. The JavaScript code will be restricted using a rule set with anti-patterns that should not occur in the application. These rules will have to be enforced while committing code to the version control system. M-Industries approached the Delft University of Technology with the question if it was feasible to create such a code analysis tool as part of a Master's project.

1.1.2 Research

In the field of software engineering research, there are several specific research topics that relate to code quality analysis: the development of code analysis mechanisms, the definition of code smells, code metrics and empirical research to software quality based on large collections of source code. This project aims to combine ideas from existing research and see if and how these ideas can be applied to a dynamically typed language like JavaScript. For the development of a code analysis tool, we look at research and existing tool implementations. To define a good set of metrics and code smells that are specific for the JavaScript language, we can learn from research into metrics for object-oriented systems and we can also inspect the small collection of existing tools for JavaScript code inspection. To evaluate the analysis tool, we will analyze a large collection of JavaScript code gathered from open source projects and from a collection of popular websites. There are several publications that describe methods that can be used to perform this kind of empirical research.

1.2 Problem Statement

M-Industries needs a flexible static code analysis tool that can be used to make sure their code base conforms to their own set of progressing rules. They currently use JSLint[4] to check JavaScript code quality, but the design of JSLint makes it nearly impossible to write new rules and extend the tool. We will give more details about the shortcomings of JSLint in chapter 3. These are the requirements to the analysis tool:

- The tool is written in JavaScript and runs in Node.js.
- The primary data structure is a model of the JavaScript language expressed in the M-Industries schema language.
- The analysis engine is separated from the rule definitions and the reporting component.
- It it possible to integrate the tool with the version control system to validate all commits.
- It has the ability to regenerate source code from the model data.
- The tool is be able to process comments in order to pick up special analysis directives.
- There is an option to run the tool locally from within a number of IDEs and editors on different operating systems.
- The tool is at least as powerful as JSLint for JavaScript code analysis. In time, M-Industries is able to completely replace JSLint with the new analysis tool.

The university does not require that a fully functional solution is delivered at the end of the project, a prototype is also sufficient. For the company, the project will be successful when they have an analysis tool that can replace JSLint and that supports writing custom rules.

1.2.1 Research Questions

Based on the problem statement, we can formulate the following research questions:

RQ1 What is the state of code quality analysis tools for JavaScript code?

RQ2 What is the effect of using a typed abstract syntax tree model on writing analysis code?

RQ3 Is there a significant difference in code quality between server-side and client-side JavaScript code?

These three questions and their answers will be repeated as part of the conclusion in chapter 6. To find the answers to these questions and to meet the requirements of M-Industries, we devised a project to build and evaluate a static analysis tool for JavaScript code.

[4] `http://jslint.com/`

1.3 Project Structure

The project consists of two parts, both providing several challenges in the areas of research and engineering:

1. The design and implementation of a static analysis tool for JavaScript code.
2. The evaluation of the static analysis tool on a large collection of open source JavaScript projects and JavaScript code gathered from websites.

The first part, the construction of the static analysis tool, is mostly an engineering effort. It provides direct value to the company where the project is carried out as they have the need for a flexible static analysis tool. In this tool, we aim to bring together the best parts of other static analysis tools, specifically tailored to the analysis of JavaScript code. The second part of the project illustrates how the analysis tool can be applied to run a large-scale empirical experiment. In published research from the area of software engineering and code analysis, this method is often used to verify the workings of a tool and to gather data from real-world software projects.

In the project, we will go through the standard phases of problem analysis, design, implementation and evaluation. It is encouraged to have a working tool prototype available early, so we can critically evaluate it, improve the design and adapt our implementation. The problem analysis and evaluation phases include exploring the field, finding similar or equivalent tools and analysis mechanisms and deciding which existing technologies can be reused or combined in order to solve our problem.

1.4 Thesis Structure

The remainder of this thesis is structured as follows:

- Chapter 2 provides details on the background for this project. This chapter contains a short introduction to relevant parts of the JavaScript language.
- Chapter 3 provides details on the background for this project. This chapter contains information on code smells and static analysis in the context of JavaScript. Relevant research and existing tool implementations are used to explain these ideas.
- Chapter 4 explains the design of the analysis tool. We will discuss its different components and the challenges we encountered during the implementation of the tool.
- Chapter 5 describes the setup, execution and results of the evaluation of the analysis tool.
- Chapter 6 concludes the thesis with a list of contributions of this project, answers to the research questions and possibilities for future research and analysis tool improvements.

Chapter 2

The JavaScript Language

This chapter is meant as a short introduction to the JavaScript programming language for developers with experience in languages such as Java and C#. Special attention will be given to language features that are important in the area of code quality analysis.

JavaScript is an object-oriented programming language. As a scripting language, it is designed to control and automate an existing system. This existing system is called the host environment. The host environment exposes objects with properties and functions that can be accessed from JavaScript. The JavaScript language does not provide any built-in mechanism for generic input or output, but it relies on objects provided by the host environment to communicate with the host or with external systems [6].

2.1 History

The JavaScript language was created in 1995 by Brendan Eich, who was hired by Netscape to create a scripting language in order to enable web pages to be more interactive. As seen in figure 2.1, the language design was influenced by Self, an object-oriented language without classes, C and Java for their syntax and Scheme for its functional features [6, 5].

In 1996, Netscape 2.0 was released with support for JavaScript, making it the first publicly available web browser to do so. Later that year, Microsoft released version 3.0 of Internet Explorer. Microsoft's browser also included a scripting engine, called 'JScript' which was in fact reverse engineered from Netscape's JavaScript engine. In November 1996, after Microsoft made their version of JavaScript in the form of JScript, Netscape made the step to standardize the JavaScript language in order to prevent fragmentation among different JavaScript engines. The first version of the standard was published in June 1997 by Ecma International. Because of trademark issues, the standardized version of the language is called ECMAScript. In this report, the names ECMAScript and JavaScript will be used interchangeably. Since 1997, a second, third and fifth edition of the standard have been published. The Ecma never reached agreement on the fourth edition, which was abandoned. The fifth edition [6] has been updated to 5.1, which was published in June 2011.

JavaScript actually started out as 'Mocha', was renamed to 'LiveScript' but was finally named 'JavaScript' because of an agreement between Sun Microsystems and Netscape [20]. Nowadays, the name 'JavaScript' is a trademark of Oracle, which ac-

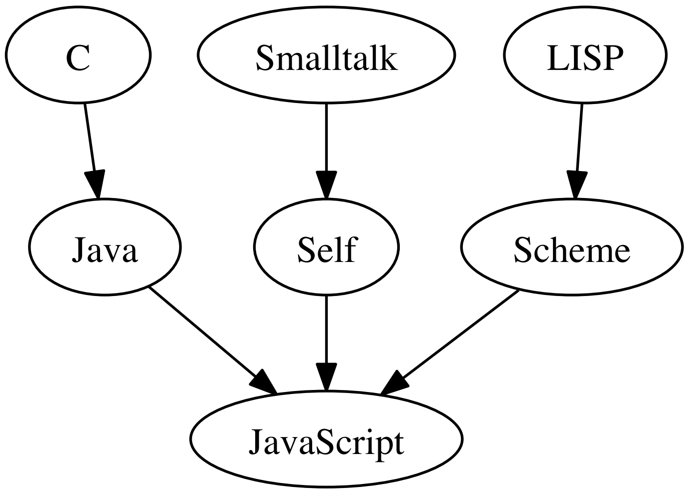


Figure 2.1: Key influences to the JavaScript language.

quired Sun Microsystems in 2010. The name of the language sometimes leads to people confusing it with Java, which provides another method to enable interactivity on web pages. With Java, developers can create 'applets', programs that can be embedded in a web page and that can expose a graphical interface inside the web page. Upon loading a page with an embedded applet, the visitor's browser will download the applet byte code and launch a Java virtual machine to execute the applet's code. Java applets run in a sandboxed environment. Direct access to the visitor's machine's resources is limited or non-existent at all. This method of integration is also used by Adobe's Flash and Microsoft's Silverlight. However, contrary to Java, JavaScript has more direct integration in the browser environment. The browser exposes parts of its environment through APIs like the Document Object Model[1] (DOM) to the JavaScript engine, allowing JavaScript code complete control over the web page's contents. JavaScript code is distributed in source form and is executed by the browser in the context of the currently opened web page.

2.2 Host Environments & Engines

Because of its origins, JavaScript is almost always automatically associated with client-side programming in the environment of a web browser. However, the language itself is not limited to web browsers only. As early as 1996, Netscape released an application server called 'Enterprise Server 2.0' which contained support for CGI-style server-side JavaScript programming. A more recent example is Node.js, a combination of Google's V8 JavaScript engine with a standard library and a programming model that has been optimized for asynchronous network programming, e.g. for writing web applications directly on top of the built-in HTTP library.

[1]`http://www.w3.org/DOM/DOMTR`

2.2.1 The Browser Host Environment

The environment offered by web browsers consists of several built-in objects and functions exposed as properties in the global object, `window`. They are exposed to allow JavaScript code to interact with web pages and several browser interfaces. Some examples of these properties are:

- **window**, the global object that contains all exposed properties.
- **alert**, **prompt** and **confirm** are functions that allow for simple interaction with users in the form of modal dialogs with a text box or ok/cancel buttons. These dialogs are shown by the browser using the native UI toolkit of the platform the browser runs on.
- **document** is an object conforming to the DOM HTMLDocument interface as defined by the World Wide Web Consortium (W3C). It allows interaction with the current web page's content.
- **XMLHttpRequest** is an object conforming the the XMLHttpRequest[2] interface as defined by the W3C. It allows JavaScript code to perform HTTP requests. First available in 1999 for Microsoft's Internet Explorer only, it now has been standardized and is implemented by all major browsers. It forms the basis of a programming style called Asynchronous JavaScript and XML [16] or 'AJAX' where JavaScript code is used to asynchronously send and receive data after the initial HTTP requests of fetching a page and its resources have completed.
- **console** has been introduced in recent browser versions to provide developers with a logging interface that is more advanced than using `alert` to show a dialog. The `console` object provides functions such as `log`, `info` and `error`. Browsers usually offer a panel or window with developer tools in which the logged strings are shown.

Not all interfaces of these objects and functions have been standardized. In fact, one reason why libraries such as jQuery exist is to provide a uniform interface and an abstraction layer over the implementation differences in built-in objects from various web browsers.

2.2.2 Engines

Because JavaScript started as a scripting language for a web browser and because web browsers still remain the most common host environment for JavaScript, significant development on JavaScript engines has been done by browser makers. Major web browsers come with their own JavaScript engine:

- **Microsoft Internet Explorer** contains the Chakra JScript engine. The source code is not available and the engine only comes bundled with the browser, it is not available separately.

[2] `http://www.w3.org/TR/XMLHttpRequest/`

- **Mozilla Firefox** uses Mozilla's open source SpiderMonkey engine. A standalone/embeddable version is also available.
- **Google Chrome** uses Google's open source V8 engine, also available standalone.
- **Apple Safari** contains JavaScriptCore, which is open source and being maintained by the WebKit project.

Just as with the 'browser wars' of the nineties, during which the competition between Microsoft and Netscape led to most of the features that ended up in the current version of the HTML standard, the current battle seems to be about JavaScript performance [23, 9, 25, 22, 24]. In each new version of their product, browser makers are trying to improve upon the JavaScript performance of the previous version. In the first JavaScript engine, parsed JavaScript code was interpreted directly, but current engines are using more advanced techniques such as byte code interpretation, just-in-time compilation and compilation to native machine code. An overview of and comparison between the currently available JavaScript engines would probably be interesting enough for a separate research assignment. Because this current assignment is focused on determining and measuring JavaScript code quality, it is not feasible to give an in-depth overview of the different engines, instead we can focus on the actual language itself.

2.3 Language Features

In section 2.1, it was mentioned that JavaScript is influenced by Java, Self and Scheme. The influence of Java can be seen directly in the syntax of JavaScript, which should look familiar to Java programmers. A simple example of JavaScript code is given in listing 1. The listing shows the syntax of several JavaScript language features: variables, operators, function definition, function calls, conditional statements, control flow statements, object creation, string concatenation and exception handling. Java programmers will also notice a big difference with the Java language: variables seem to be 'type-less', we do not see any type information when variables are declared. In the remainder of this section, a selection of language features will be discussed. It is far beyond the scope of this document to provide a detailed overview of the complete JavaScript language, so the selection of the features is limited to those features that make JavaScript significantly different from languages like Java.

2.3.1 Types

In JavaScript, variables do not have an associated type, but values do. A variable can always hold any type of value. The ECMAScript specification defines six language types, types that can be directly manipulated by the programmer:

- **Undefined** is a type with only one value: `undefined`. It is the default value of all variables that have been declared but have not had a value assigned yet.
- **Null** is another type with only one value: `null`.

```
function seqsum(start, end) {
  if (end >= start) {
    var result = 0;

    for (var x = start; x <= end; x++) {
      result += x;
    }

    return result;
  } else {
    throw new Error("end ("+end+") < start ("+start+")");
  }
}

try {
  var x = seqsum(2, 5);
  var y = seqsum(5, 2);
} catch (e) {
  // handle exception here by inspection of e
}
```

Listing 1: Definition and use of a function to add a sequence of numbers.

- **Boolean** has two values: `true` and `false`.

- **String** is a sequence of zero or more 16-bit unsigned integer values. ECMAScript has no separate character type such as `char` in C. Each stored value is considered to be a UTF-16 encoded character. Matching single and double quotes can be used interchangeably to notate string values.

- **Number** is the only numeric type. It represents a 64-bit floating point value as defined in the IEEE 754 [13] standard. This corresponds to Java's `double` type.

- **Object** is the base of all other types except for Undefined and Null. Every Object is a dynamic collection of properties. A property is a named collection of attributes. ECMAScript defines a number of attributes, the most important one being the `value` attribute which stores the actual value of the property. Because of this, objects can be used as a key-value map (a dictionary).

The ECMAScript standard defines a number of built-in objects (e.g. Function, Array, Date, RegExp and Error), but no other types. The type of a value can be determined by using the `typeof` operator, which will be discussed in section 2.3.4. A language where variables have no associated type and can take any kind of value is said to use dynamic typing. Source code analysis of languages with dynamic typing is harder than analysis of languages with static typing, because the type of a variable's value can not always be determined reliably before actually running the program.

2.3.2 The Global Object

One of the built-in objects in the ECMAScript standard is the global object. This object contains several built-in values (e.g. `NaN`, `Infinity`) and functions (e.g. `eval`, `parseInt`). Implementers are free to add their own properties to the global object. The example in listing 2 shows a function `test` which, when called, will add (or overwrite) a property `x` with the value 5 in the global object. The global object is important because, in web browsers, it represents the global scope. Every variable or function that is defined at the top level (not inside a function) will be stored as a property in the global object. In the example, this means the function `test` itself is also placed in the global object.

```
function test() {
  x = 5; // changes the global object
  var y = 5; // creates a local variable
}
```

Listing 2: The global scope.

Authors of JavaScript libraries that are meant to be used together with other code should especially be cautious not to 'pollute' the global object with large numbers of functions and variables, as this might cause collisions with other code's functions and variables. Techniques to prevent this from happening (e.g. simulating namespaces, using anonymous functions) will be discussed in more detail in the next chapter, in section 3.3.3 on coding guidelines for correct 'Library Behavior' of JavaScript code.

2.3.3 Classes, Objects and Prototypes

Let's first look at how object-oriented languages with classes work, simplified:

- A class is a collection of methods and properties with specific attributes. A class is like a blueprint: it only defines and describes the properties and behavior of objects created from the class.
- An object is a particular instance of a class that conforms to the definition –the blueprint– of that class. It contains the properties from the class definition as actual values. It does not 'contain' the methods of the class, as these are shared by all objects of the same class.

JavaScript is an object-oriented language, but it has no notion of classes like in languages such as Java and C#. Instead, the language uses the concept of *prototypes*. This idea comes from the Self language, an object-oriented language without classes. In the paper describing the Self language [32], the authors explain how Self implements inheritance: every objects contains a number of slots (properties) which may store state (values) or behavior (functions). Every object has a pointer to its parent object (its prototype). If a slot does not exist in an object, it passes the message (read, write, function call) on to its parent until there is no more parent object. This is known as the prototypal inheritance chain. In the same way that methods in a Java class are shared by all instances of that class, every property in the prototype of a Javascript object is shared with all other objects having the same prototype. Objects are created by cloning an existing object and by then making the desired changes to the cloned object. Both objects and prototypes can be changed at runtime: properties can be added, changed and deleted. This poses a challenge for source code analysis that tries to do any kind of type checking on JavaScript objects.

Object Creation with Constructor Functions

To understand inheritance, we must first understand how objects can be created in a way that allows us to define an inheritance chain. In JavaScript, this can be accomplished by calling functions in a special way that causes the function to be treated as a 'constructor function'. The code in listing 3 will be used to explain constructor functions.

```
1  function Employee(name, dept) {
2    this.name = name;
3    this.dept = dept;
4  }
5
6  var bob1 =     Employee("Bob", "Engineering");
7  var bob2 = new Employee("Bob", "Engineering");
```

Listing 3: The constructor function.

On the first line, we see the definition of a function `Employee` that takes two arguments. We can call this function directly, like on line 6. The result of that function call is that the two properties '`name`' and '`dept`' are set on whatever object '`this`' points to at the time of the call. Normally, this would be the global object, as discussed in section 2.3.2. In a web browser, this would mean that the properties '`name`' and '`dept`' are written to the `window` object. The function does not return anything, so the value of the variable `bob1` is `undefined` (the implicit default return value of a JavaScript function). On line 7, we see the same function call, but now prefixed with the `new` operator. Because of this operator, some extra things happen:

1. A new empty object is created.
2. The hidden `prototype` property of the new object is set to the `prototype` property of the constructor function. In the example, this is the `Employee` prototype object (the implicit default value).
3. The constructor function is called in the context of the new object. In the example, that means `this` is bound to the newly created object. The two properties '`name`' and '`dept`' are written to this new object.
4. Depending on the result of the constructor function, one of these things happens:
    - If the result is an Object, this result will be returned. This would be the case when the function contains an explicit return statement with a return value which is an object other than `this`.

- Otherwise, the newly created object is returned. This is what happens in the example.

This means that, after line 7 has been executed, the variable `bob2` contains an object with its hidden prototype property set to `Employee`. The object contains the two properties '`name`' and '`dept`'.

Prototypal Inheritance

In the previous section, we have seen how the hidden `prototype` property of an object points to its prototypal object. To change the value of this hidden property and implement inheritance, we can change the `prototype` property of a function, which can then be called as a constructor function using the `new` operator. This means that we explicitly change the prototypal inheritance chain.

```
1  function Employee(name, dept) {
2    this.name = name;
3    this.dept = dept;
4  }
5
6  Employee.prototype.toString = function() {
7    return "name: "+this.name+", dept: "+this.dept;
8  };
9
10 function Engineer(name, spec) {
11   Employee.call(this, name, "engineering");
12   this.spec = spec;
13 }
14
15 Engineer.prototype = new Employee;
16
17 function Marketeer(name) {
18   Employee.call(this, name, "marketing");
19 }
20
21 Marketeer.prototype = new Employee;
22
23 var bob = new Engineer("Bob", "JavaScript");
24 bob.toString(); // name: Bob, dept: engineering
25
26 var alice = new Marketeer("Alice");
27 alice.toString(); // name: Alice, dept: marketing
```

Listing 4: Prototypal inheritance.

The code in listing 4 will be used to explain inheritance. The listing contains a typical example of inheritance, partly inspired by 'The employee example' from Mozilla's page on the details of the JavaScript object model[3]. On the first line, an

[3]`https://developer.mozilla.org/en/JavaScript/Guide/Details_of_the_Object_Model`

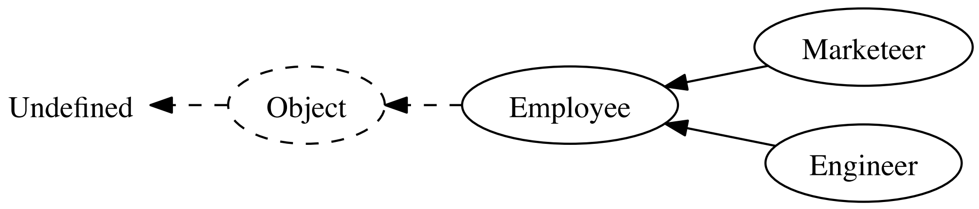


Figure 2.2: The explicit and implicit prototypal inheritance chain from listing 4.

`Employee` constructor function is defined. On lines 10 and 17, constructor functions for `Engineer` and `Marketeer` are defined. To set up the inheritance, the `prototype` property of both constructor functions is set to a new instance of an `Employee` object. At runtime, this results in the inheritance chain as depicted in figure 2.2. The variable `bob` has its hidden prototype set to `Employee`. For `alice`, this is `Marketeer`. The solid lines then indicate the explicit prototypes, pointing to `Employee`. After that, the dashed lines indicate the implicit prototypes: from `Employee` to the default `Object` and at that point, the prototypal inheritance chain stops.

Note the way in which the base constructor function is called from the constructor functions of `Engineer` (line 11) and `Marketeer` (line 18), using the `call` function on a `Function` object. This is somewhat similar to reflection in languages like Java or C#. The `call` functions takes an Object as its first argument, allowing the caller to explicitly indicate the context in which the function will be evaluated. The effect is that, when the `Employee` function is called as a normal function from one of the constructor functions, `this` points to the newly created `Engineer` and `Marketeer` objects.

Overriding & Extending built-ins

Because ECMAScript has no real primitive types, that is, even Boolean, String and Number are based on Object, this means that it is possible to extend the prototypes of these objects with new properties. Other than extending prototypes, it is also possible to *override* built-in properties. This means, for example, that we can do things like replacing the `toString` function on all boolean values with a function that always returns `"false"`, as shown in listing 5.

```
true.toString(); // default, returns "true"
Boolean.prototype.toString = function() { return "false"; }
true.toString(); // now returns "false"
```

Listing 5: Overriding built-in functions.

After this, all code that relies on the string representation of `false` to be `"false"` would be broken. This makes the language very flexible, but overriding the behavior of built-in objects should be done with extreme caution, since other code might depend on specific behavior of these objects. However, the ability to extend built-in objects has made it easier for JavaScript library developers to bridge differences between various browsers and JavaScript language versions. Let's say, for example, that a new version of JavaScript defines a built-in function `round` on the Number object prototype. In engines with an older version of JavaScript, a library could then check for the existence of this `round` function and, if not available, add it to the Number object prototype itself.

2.3.4 Operators

In this section, several operators and their specific behavior will be discussed. A detailed overview of all JavaScript operators can be found in section 11 of the ECMAScript standard [6].

new

As shown in the examples explaining object construction and inheritance, the `new` operator is a unary operator that can be prefixed before anything that evaluates to a `Function` object, causing JavaScript to create a new object and apply the specified function call to that object. This is explained in more detail in section 2.3.3.

typeof

The `typeof` operator is a unary operator that can be used to determine the type of a value. It returns a string corresponding to the lowercase type name of the value. For example, for a number, it returns `"number"` and for a string, it returns `"string"`. For all objects, including custom objects created with constructor functions, it returns `"object"`. For arrays and for the value `null`, it also returns `"object"`, which is a bit confusing and might cause errors if not handled properly.

instanceof

The `instanceof` operator is a relational operator that takes two arguments. It checks if the right hand argument occurs somewhere up the prototypal inheritance chain of the left hand argument. To clarify this: if, in the example from listing 4, we would have evaluated the expressions `bob instanceof Engineer` and `alice instanceof Engineer` that would have resulted in `true` and `false`, respectively.

delete

Using the `delete` operator, properties can be removed from objects at runtime. In a way, this is similar to the way dictionaries or hashmaps can be used in other languages, but because JavaScript makes no difference between an object and a dictionary, `delete` can also be used to delete functions from objects at runtime. This can be an extra challenge for source code analysis, because properties can not be assumed to be present forever after their definition.

== and ===

The equality operators `==` and `!=` expose the type coercion behavior of JavaScript where values will be converted from one type to another following a set of conversion rules. This can cause surprises and it also means that the transitivity law does not necessarily hold, as shown in the example in listing 6. Luckily, JavaScript also provides the `===` and `!==` operators which perform true comparison of values without type coercion. This is the preferred way to compare values.

```
''  ==   0  // true
 0  == '0'  // true
''  == '0'  // false
```

Listing 6: An illustration of type coercion with equality operators.

2.3.5 Function Argument Processing

When a JavaScript function is called, the actual arguments are not checked against the formal arguments in the function definition. The actual arguments are always made available as a local variable `arguments` inside the function body. This means that the example in listing 7 is perfectly valid code. This can be used for variable argument lists and to implement function overloading as it is found in Java and C#. Traditional source code analysis would give an error when there is a mismatch between the formal and actual arguments, but for JavaScript code, this should be tolerated.

```
function test1(a, b) {}
function test2() {}

test1();
test2(1, 2);
```

Listing 7: An illustration of function argument processing.

2.3.6 Eval

In JavaScript, the global object contains a function `eval` that can be used to execute arbitrary code, passed in as a string. The JavaScript engine will parse and evaluate this code in the scope in which the `eval` function is called. There are other ways to reach the same result, for example constructing a new `Function` object of which the function body can also be passed in as a string. Usage of these constructs makes it very hard to do reliable source code analysis, because when a string passed to the `eval` function comes from an external source (e.g. user input, network), there is no way to determine up front what effect this can have on the program.

2.3.7 Scope, Hoisting & Closures

Although the syntax of JavaScript is largely inspired by C-like languages such as Java, there are differences in the scoping rules. Where C has block scope, allowing developers to 'hide' variables from an enclosing outer scope by declaring a local inner variable with the same name, JavaScript only has function scope. This means that all variables declared using `var` inside a function belong to that function. When a variable is declared outside a function, it belongs to the enclosing scope, which usually is the global scope.

An illustration of the effects can be seen in listing 8. On line 5, in the if-block inside the function `test1`, a local variable `x` is declared and initialized. However, the function already contained a local variable `x`. Because there is no block scope, the value of `x`, which was 1, gets overwritten by the new value 2.

The function `test2` shows a related phenomenon, better known as the 'hoisting' of variable declarations. Internally, JavaScript moves all variables declarations to the top of the enclosing function. This means that the function `test2` is executed as if lines 12 & 13 were swapped. Otherwise, the variable `a` would end up in the global scope.

```
1  function test1() {
2    var x = 1;
3
4    if (true) {
5      var x = 2;
6    }
7
8    return x;
9  }
10
11 function test2() {
12   a = 5;
13   var a;
14   return a;
15 }
16
17 test1(); // returns 2
18 test2(); // returns 5
```

Listing 8: Function scope & variable hoisting.

With function scope, variables are only accessible to the function they were declared in. They are invisible to the outside. However, something interesting happens when we return a function from another function. Note that this is possible because functions are first-class citizens within JavaScipt, as inherited from the Scheme language [14]. This means that a function is treated as a value just like other values such as a number or a string. Function values can be stored in a variable, can be passed as an argument to another function and can also be returned from another function. An example of this last case is given in listing 9. The function `makeAdder` puts the value of the argument `x` in a local variable `left` and then returns an anonymous

```
1  function makeAdder(x) {
2    var left = x;
3
4    return function (right) {
5      return left + right;
6    }
7  }
8
9  (makeAdder(5))(6); // returns 11
```

Listing 9: A demonstration of function closure.

function that takes one argument, `right`, and adds its value to the value of the `left` variable in its outer scope. On line 9, the function `makeAdder` is called first and the resulting anonymous function is directly called after that. Because JavaScript has support for closures, the anonymous function still has access to the outer function's `left` variable, even after the outer function already terminated.

2.4 Future

Ecma International has several 'Technical Committees' or 'TCs' that maintain and develop their standards. TC39 is the committee that works on the ECMAScript specification. People from Adobe, Yahoo, Google, Microsoft, Mozilla, Opera and Apple are member of this committee, so all major browser vendors are represented. The next edition of the ECMAScript specification is targeted for release in December 2013 [17], after which it will probably take a number of years before a significant percentage of end users actually use a browser with full support for the new version. An example of a new feature is support for globalization/internationalization, which means that several of the language's built-in objects will be made aware of locales, time zones and currencies. Microsoft recently released [26, 2] a preview version of their reference implementation of this new standard as a way to gather community feedback.

Most of the committee's work can be followed on a publicly accessible wiki[4].

[4] `http://wiki.ecmascript.org/`

2.5 Summary

In this chapter, we briefly discussed the history of the JavaScript language, followed by a short description of the most common host environment for JavaScript code: the web browser. The chapter closed with a brief description of the committee working on the next JavaScript language versions. Most attention, however, was given to language features that set JavaScript apart from languages like Java and C#: the lack of typed variables and classes, the prototypal inheritance chain and functions as first-class citizens. Although the syntax of JavaScript might look familiar to C and Java programmers, these language features make it a completely different language, especially from the perspective of using existing static analysis techniques on JavaScript code. The next chapter on code quality will assume some knowledge about the language features discussed here as those language features will be used to explain how existing ideas on code quality apply to JavaScript code.

Chapter 3

Software Quality

In this chapter, several different aspects of software quality will be discussed. The term ‘software quality’ is a container for a number of different ideas. Relevant research topics in the area of software quality are the identification of ‘code smells’, certain anti-patterns that indicate low code quality and ‘code clones’, identical or similar fragments of code that occur in multiple locations within the same project. The presence of code smells and code clones often indicates that refactoring the code or redesigning the program will benefit the overall quality of the software. Somewhat related, there also is a large amount of work in determining and defining ‘code metrics’ as a means to quantify different aspects of source code.

Static analysis techniques can be used to detect code smells, find code clones and calculate code metrics while writing software, helping developers to prevent making commonly made errors. These techniques become particularly useful when developers are provided with tools to automatically analyze their code and assist them while writing software, integrated in their IDE, or as part of a continuous integration system.

In practice, maintaining a high level of software quality can also be done by providing a set of guidelines or rules that software developers should adhere to when writing software. This does not guarantee high quality code, but it prevents common mistakes and encourages a consistent writing style within a project, which helps when multiple people work together on the same code base.

This chapter deals with several of these ideas and approaches that ultimately aim to improve overall software quality. We assume some knowledge about the language features discussed in the previous chapter as those language features will be used to explain how existing ideas on code quality apply to JavaScript code.

3.1 Code Smells

The third chapter in Martin Fowler’s book on refactoring [8] –jointly written with Kent Beck– explains code smells, “certain structures in the code that suggest the possibility of refactoring”, as a way to give software engineers an indication *when* to refactor code (instead of *how* to perform the refactoring itself). They proceed to list a number of different code smells. This list is partially reproduced in section 3.1.1. The topic of code smells is linked to all other sections in this chapter: coding guidelines (3.3), code clones (3.2) and code metrics (3.6). Their relationships can be explained like this:

- **Coding guidelines** usually contain rules/recommendations that prevent code smells from being introduced in a project.
- **Code clones** are an example of a code smell ('Duplicated Code'). They deserve their own section (3.2) because there is a large area of research in code clone identification and detection.
- **Code metrics** can be used to detect the presence of code smells, although Beck and Fowler explicitly state that "no set of metrics rivals informed human intuition". Their definitions of code smells therefore do not contain any absolute values on what might be considered 'too much' or 'too little' of anything.

3.1.1 Code Smells Applied to JavaScript

The original definitions of Beck and Fowler's code smells are written for object-oriented languages with classes such as Java. We will inspect a selection of these code smells and argue if and how they apply to JavaScript code. The separate sections on code clones (3.2) and code metrics (3.6) contain some more examples of how Beck and Fowler's code smell definitions apply to JavaScript code.

Divergent Change

When a class is often changed in different places for different reasons (e.g. supporting a new storage format, supporting a new type of domain object), it is recommended to extract new classes such that every new class handles exactly ony 'variation' of a specific type that occurs in the system. This applies equally as well to JavaScript objects and their prototypes.

Shotgun Surgery

This is the opposite of the 'Divergent Change'. Shotgun surgery means that, when making a specific kind of change in one place, a large number of other classes also have to be updated. The risk is that you forget to update one of those other classes, resulting in a broken system. With languages like Java, you still have the compiler that warns you if you rename a field in one class and forget to update the reference to that field in another class. This is detected by the static type system. JavaScript lacks static types, which means that errors due to shotgun surgery will manifest themselves as runtime errors when properties are inconsistently renamed. For this reason, this code smell is more dangerous when it occurs in JavaScript code, but on the other hand, also very hard to detect accurately because of the lack of static types in JavaScript.

Data Clumps

When a group of data items often appears together, say as fields in a number of classes and as parameters in various methods, it is recommended to group them together in a new class. JavaScript makes this process very easy because of the support for object literals, where 'anonymous' objects can be constructed inline to group a set of properties together.

Primitive Obsession

This code smell describes the phenomenon that occurs when people who are new to programming with classes and objects prefer a loose collection of primitive values over a class. The reason usually is to avoid the overhead of wrapping primitives together in a class, or that they are "awkward to create when you want them for only one or two things". This does not really apply to JavaScript because JavaScript makes it easy to create objects, even when they are destined to have a short life span. Also, almost everything in JavaScript already *is* an object, as the language has very little real primitive types.

Inappropriate Intimacy

When methods from one class access a large number of private members from another class, this is described as inappropriate intimacy. This usually occurs with inheritance, where subclasses have access to their parent's private variables and methods. JavaScript has no support for access modifiers such as Java's `public`, `private` and `protected`[1]. This means that, by default, there is no way to differentiate between 'appropriate' and 'inappropriate' intimacy as all objects can always access all other objects' properties.

Incomplete Library Class

Beck and Fowler describe that most code relies on library classes instead of constantly re-inventing the wheel. Because library authors can not accurately design for all specific use cases of their library, users sometimes feel the need to make small modifications to the library. They write that this is "usually impossible". Not so for JavaScript. Due to the prototypal nature of the language, any object's behavior can be altered or extended. This can cause problems when altering existing behavior in combination with other code running in the same context that relies on the default unaltered behavior of the library. This, in fact, might be considered a new and JavaScript-specific code smell.

3.1.2 Overview

To conclude this section on code smells and as a summary of the previous explanations, all code smells from Beck and Fowler are listed in table 3.1 along with an indication of their relevancy to JavaScript code. In the table, the names for the code smells "Parallel Inheritance Hierarchies" and "Alternative Classes with Different Interfaces" are shortened for layout purposes. They are marked with an asterisk. For full explanations of the code smells, please refer to the third chapter of Fowler's book on refactoring [8].

3.2 Code Clones

Software engineers speak of code clones when they encounter multiple fragments of code (e.g. files, modules, functions, blocks) that are completely or partially identical. Code clones are created whenever a developer copies a specific piece of code and

[1]There is, however, a method to simulate private variables by using closures.

Code Smell Name	Relevant? (with optional notes)	
Duplicated Code	✓	
Long Method	✓	
Large Class	✓	
Long Parameter List	✓	Formal and actual arguments may be different in JS. Check at call site!
Divergent Change	✓	
Shotgun Surgery	✓	Higher risk of errors because JS lacks static types.
Feature Envy	✓	
Data Clumps	✓	Very easy to fix in JS.
Primitive Obsession	×	Less likely to occur, reasoning does not fully apply to JS.
Switch Statements	✓	Use identical property names on objects to get polymorphism.
Parallel Inheritance*	✓	
Lazy Class	✓	
Speculative Generality	✓	
Temporary Field	✓	
Message Chains	✓	
Middle Man	✓	
Inappropriate Intimacy	×	The classical notion of 'private' variables does not apply.
Alternative Classes*	✓	
Incomplete Library Class	×	Library behavior can be modified by changing the prototypes.
Data Class	×	JS objects are more likely to be used as 'dumb data holders'.
Refused Bequest	✓	
Comments	✓	

Table 3.1: Relevancy of code smells for JavaScript.

pastes it somewhere else, possibly making some small modifications afterwards. Code clones can also ‘appear’ when, for example, two people independently write a highly similar class or method in a different file. Sometimes their existence is legitimate, but in many cases, these clones are unwanted and –with some refactoring and redesign of the code– should be carefully merged back into one code unit. Beck and Fowler also state this in their definition of the ‘Duplicated Code’ code smell. The reason for doing so is that this makes it easier to perform software maintenance, e.g. fixing bugs and adding new functionality. For example, say a specific function has been copied and pasted into three other files within a piece of software. The original engineer who did this is aware of the clones, but whenever someone else starts making changes to the original function, what is the chance that this person will also update the cloned functions? This question has influenced an area of research in identifying and preventing code clones, which is an interesting problem in itself. Having a low number of code clones is generally believed to be an indicator of higher software quality.

A recent publication from Roy et al. [30] aims to provide an overview of the currently available code clone detection techniques and tools. The article gives a good general description of the steps involved in the clone detection process and also provides a classification of different clone detection techniques with an increasing level of transformation from the raw source text:

- **Textual** approaches perform little to no transformation on the source text of the program, so raw source code is used to detect clones. These techniques are mostly language-independent, so existing tools should be able to work for JavaScript code without modifications, in the most optimistic scenario.

- **Lexical** methods do not directly work with raw source text, but they first tokenize the input. This results in a stream of tokens which can be scanned for duplicate token sequences. When a JavaScript-specific tokenizer is used in the first step, this could work perfectly for JavaScript code.

- **Syntactic** clone detectors again take it one step further, by also parsing the source code and performing the match detection on ASTs. If the match detection is generic enough, this should also work for JavaScript code, when parsed into the desired AST format.

- **Semantic** analysis handles programs at an even higher level, for example by representing a program as a graph with data and control dependencies. Techniques for finding similar subgraphs are then used to find clones. This might prove to be a bit harder for JavaScript, since the dynamic nature of the language often makes it hard to perform full-program static analysis.

This classification is not strict, the authors have also identified *hybrid* approaches where analysis is performed at two levels, for example a combination of syntactic and semantic analysis.

3.2.1 JavaScript Code Clones

There is no reason why the different techniques for code clone detection would not work for JavaScript code. In practice, however, there seems to be little research that explicitly targets code clones in JavaScript code.

Synytskyy et al. [31] describe an approach to finding code clones in web pages which focuses on HTML fragments but which also deals with both inline and external JavaScript code (via `<script>` tags without or with a `src` attribute). They admit to using a very simple clone detection algorithm as clone detection is not the primary focus of their research. They seem to treat included files and inline code blocks as single code fragments between which clones are detected.

Calefato et al. [3] published an article on function clone detection in web applications. They treat JavaScript functions as code fragments, the smallest unit of comparison. Code clone candidates are detected by finding duplicate function names in a codebase. They argue that JavaScript code clones are created by developers copying and pasting functions from one file into another, possibly changing the function's body but keeping the function name intact. These function clone candidates are then manually compared and classified.

More recent research from Roy and Cordy, in an article written for the 2010 International Workshop on Software Clones, focused on comparing the occurrence of code clones in open source Python projects to open source code written in C, Java and C#. The purpose was to compare code clones in scripting languages to 'traditional' languages. The article mentions PHP, Perl and Ruby, but JavaScript was not mentioned. No other articles from the same workshop neither from the 2011 edition seem to deal with code clones in JavaScript code.

The open source PMD[2] software contains a Copy/Paste Detector abbreviated as CPD. It uses a string matching algorithm comparing string hash values and works on the lexical level. The included CPD in version 4.3 of PMD contains support for JavaScript (with the `ecmascript` language option), but this is not exposed in all parts of the interface.

CPD also allows users to add support for new languages via plugins, in the form of a language definition and a tokenizer. We found two open source projects using this method to analyze JavaScript code: one standalone JavaScript module[3] and another module[4] as part of the Sonar open source quality management platform.

The commercial ECMAScript CloneDR[5] tool uses syntactic methods to detect clones using ASTs. Ira Baxter, who wrote the paper 'Clone Detection Using Abstract Syntax Trees' [1], is CTO of the company that offers the ECMAScript CloneDR tool.

The web itself is an interesting source of research data, since all client-side JavaScript code is always distributed in source form. This would be a nice opportunity to perform a large scale search for code clones, for example to detect 'stolen' code or plagiarism, although the resources to perform this kind of research are usually only available to large companies such as Google. See for example their 'Web Authoring Statistics'[6]

[2] `http://pmd.sourceforge.net/`
[3] `http://code.google.com/p/pmd-cpd-javascript/`
[4] `http://docs.codehaus.org/display/SONAR/JavaScript+Plugin`
[5] `http://www.semdesigns.com/products/clone/ECMAScriptCloneDR.html`
[6] `https://developers.google.com/webmasters/state-of-the-web/`

publication from 2005 in which the HTTP response headers and HTML source of over a billion web pages were examined.

3.3 Coding Guidelines

In any place where people collaboratively design and build software –e.g. software development companies and large open source projects– guidelines and standards for writing and designing software are usually documented and made available internally or publicly. Examples of this can be found in Java guidelines from Oracle[7] and the Eclipse foundation[8], GNU coding standards from the Free Software Foundation[9] and documentation from Microsoft regarding design guidelines for libraries[10], coding conventions for the C# language[11] and security guidelines[12] for developers. Because a specific piece of code is often written once (or at least by one person) and after that, possibly read many more times by other developers, it makes sense to have a set of guidelines enforcing a consistent writing style. This makes it easier for other people to read and understand the code, effectively improving the maintainability of the software.

Due to the increased usage of JavaScript code and the increasing size of JavaScript codebases, several people and framework authors also started documenting their own JavaScript-specific coding standards: Microsoft's ASP.NET AJAX framework standards[13], Google's[14] and Mozilla's[15] global JavaScript guidelines, the guidelines for the jQuery library[16] and Douglas Crockford's personal code conventions[17].

As opposed to the formal and descriptive syntax and grammar of a language, coding standards and guidelines are prescriptive and more informal: certain writing styles and conventions are considered to be 'correct' and 'good' while others, although valid from a formal point of view, should be avoided at all costs. An example in JavaScript is the '`eval`' function, which is part of the ECMAScript specification. Usage of this function (and its equivalents) is usually heavily discouraged by JavaScript coding standards. This is not completely subjective, because from a security standpoint, there are good reasons not to use `eval` when it is not strictly necessary and in many cases, use of `eval` can be prevented by rewriting –and redesigning– small parts of a program.

In this section, several common themes in the area of coding guidelines will be discussed in the light of their relevance to JavaScript code.

[7]http://www.oracle.com/technetwork/java/codeconvtoc-136057.html
[8]http://wiki.eclipse.org/index.php/Development_Conventions_and_Guidelines
[9]http://www.gnu.org/prep/standards/standards.html
[10]http://msdn.microsoft.com/en-us/library/ms229042.aspx
[11]http://msdn.microsoft.com/en-us/library/ff926074.aspx
[12]http://msdn.microsoft.com/en-us/library/8a3x2b7f.aspx
[13]http://www.asp.net/ajaxlibrary/act_contribute_codingStandards.ashx
[14]http://google-styleguide.googlecode.com/svn/trunk/javascriptguide.xml
[15]https://developer.mozilla.org/En/Mozilla_Coding_Style_Guide#JavaScript_practices
[16]http://docs.jquery.com/JQuery_Core_Style_Guidelines
[17]http://javascript.crockford.com/code.html

3.3.1 Indentation & Formatting

Most of the aforementioned coding standards and guidelines contain a section about the desired formatting of code e.g. whether to use tabs or spaces, how to indent block statements, where to place brackets (on the same line as the function definition, or on the next one) and where to place comments. The goal of these rules is to improve readability of source code and to prevent mistakes and errors caused by constructs that can easily be misinterpreted (by humans).

Code formatting rules can be expressed formally as operations on (parts of) an Abstract Syntax Tree (AST). The process to turn an AST back into readable source code is also known as 'pretty-printing' or 'code formatting'. For other languages, this has been implemented in source code formatters such as GNU's 'indent' tool or the built-in source code formatting of Eclipse and Visual Studio. Because JavaScript is really not much different with its C and Java-like syntax, such tools also exist for JavaScript code, both in IDEs and standalone, for example in Closure Compiler[18], JS Beautifier[19] and UglifyJS[20].

Optional Semicolons

JavaScript has an interesting feature, which is Automatic Semicolon Insertion (ASI), as defined in section 7.9 of the ECMAScript standard [6]. Developers can choose to omit the semicolon in their source code. The JavaScript parser will then automatically insert it for them, based on several rules. The standard gives the example as shown in listing 10. We see that an intended multiline return statement gets converted to an empty return statement followed by an unreachable expression (expressions are valid statements in JavaScript). Such behavior might not automatically be obvious to all readers, so this makes a good case for requiring developers to always include the semicolon as part of a JavaScript coding guideline.

```
return
1 + 2
// when parsed, the preceding two lines get rewritten to:
return;
1 + 2;
```

Listing 10: Automatic Semicolon Insertion.

Obfuscation

The fact that JavaScript code can only be distributed in source form –i.e. there is no standardized intermediate byte code or executable format– means that all client side JavaScript code in web applications is readable to anyone with access to the web application. Authors do not always feel comfortable with this and thus 'obfuscate' their source code, removing all unnecessary whitespace and shortening identifiers in

[18]`http://code.google.com/closure/compiler/`
[19]`http://jsbeautifier.org/`
[20]`https://github.com/mishoo/UglifyJS/#readme`

the process. This can also be done automatically, similar to the process of pretty-printing. The purpose is to make the source code harder to understand for humans. A side effect is that the download size (in bytes) of the program usually decreases, which is why the process is also referred to as minimization. An example of this can be seen in listing 11, where Google's Closure Compiler Service was used to minimize the `seqsum` function from listing 1.

```
function d(a,b){if(b>=a){for(var e=0,c=a;c<=b;c++)e+=c;
return e}else throw Error("end ("+b+") < start ("+a+")");}
```

Listing 11: A minimized version of the `seqsum` function.

3.3.2 Naming Conventions

Naming conventions for JavaScript identifiers (variables names, function names) usually follow the conventions used in the ECMAScript standard, more specifically: the way all the built-in objects and functions are named. There are no real classes in the language, but all constructor functions (e.g. `Date`, `Error`) start with an uppercase character, while 'member' functions and non-constructor functions in the global object start with a lowercase character and use camel case (e.g. `eval`, `setInterval`, `addEventListener`).

3.3.3 Library Behavior

In recent years, together with the rise of the AJAX programming style and the overall improvements in JavaScript engine performance, a number of JavaScript libraries were created to abstract over browser differences and to provide developers with helper functions, UI toolkits and collections of reusable components for recurring tasks such as event handling, creating animations, making calls to server-side code, performing input validation and building client-side user interfaces using widgets not available as native HTML controls.

To use these libraries from client-side code, developers need to include a reference to the library source code in the HTML source of a web page. The library then exposes one or more functions or objects in the global namespace which can be used in all other JavaScript code on the same page. Library code and user code[21] run in the same context and have access to the same global variables. Combined with the fact that the JavaScript language has no built-in support for modules, namespaces or any kind of mechanism to logically separate collections of code from each other, this means that library authors should avoid storing too much properties in the global object as this can cause collisions with other code, as already mentioned in section 2.3.2. There are three ways to avoid this from happening: defining and instantly calling an anonymous function, extending prototypes of existing objects and simulating namespaces with a nested structure of objects and/or functions.

[21]Note: there is no formal difference between 'library code' and 'user code'.

One-Time Use of an Anonymous Function

JavaScript allows us to define a function and immediately call it without putting the function in the global object. An example of this technique is given in listing 12. The syntax might look a bit confusing at first sight, but it is easy to explain: from the opening bracket to the penultimate closing bracket, we write an expression that evaluates to a function. This is possible because functions are first-class citizens and JavaScript allows anonymous functions to be defined. After that, we call the function directly by writing a pair of braces, which is the standard syntax for a function call. Because of function scope, all variables defined inside the function stay inside the function. The function itself (the function *value*) is not assigned to a variable and hence is not stored in the global object. This technique is recommended when a piece of code only has to run a single time during script execution.

```
(function(){
  var x = 10;
  // perform some tasks here
})();
```

Listing 12: Calling an anonymous function.

Extending Existing Objects

The ECMAScript standard describes several built-in objects such as String, Array, Function, Date and RegExp. The prototypes of these objects can be extended with new properties such as functions that provide new behavior. For example, a library could extend the `String` object's prototype with a `reverse` function that reverses a string as demonstrated in listing 13.

```
String.prototype.reverse = function(){ /* reverse... */ };
"monkey".reverse(); // returns "yeknom"
```

Listing 13: Extending the String object prototype.

Only extending existing and built-in objects does not pollute the global object, but when it is done on large scale and in combination with other libraries that also do this, there still is a risk of collisions, for example when two libraries both define a `String.reverse` function. Therefore, it is a safer option to use simulated namespaces, as explained in the next paragraph.

Simulating Namespaces

Library authors need to expose *at least one* object or function in the global object in order to make their code accessible to other code. In the ideal case, they expose *exactly one* object or function with a carefully chosen 'unique' name, e.g. a name that is not likely to be used by other code. This is the best way to minimize the risk of collisions with other libraries or with user code. The idea is that this single property (object or function) in the global object provides access to more objects and functions, enabling the creation of a nested structure of named objects and functions, in effect simulating namespaces as they are found in other languages like Java and C#. Libraries should check whether a part of the namespace path already exists before defining it, so instead of overwriting an object, they can extend it with their own properties. This is to prevent collisions when multiple libraries are used together sharing the same namespace (i.e. having a common 'prefix').

CommonJS Modules

Although JavaScript has no built-in support for modules or namespaces, there has been a community effort by the CommonJS organization to define a standard[22] for a module system in JavaScript, with several implementations available, also in the Node.js framework. The specification uses the existing ECMAScript syntax to provide a system that allows developers to organize code in logical modules that export a public API. The specification describes two reserved identifiers:

- **require** is a function that must be called with a single argument, a string containing the name of a module. It returns an object with the exported properties of the module.

- **exports** is an object that is available within a module. It must be used to declare the public API of that module, by assigning properties to it.

Listings 14 and 15 show how this works in practice by defining a `math` module that exports a function `add` in one file and using it in another file.

```
exports.add = function(a, b) {
  return a + b;
};
```

Listing 14: The `math` module in `math.js`.

```
var mathlib = require('math');

var x = mathlib.add(3, 4);
```

Listing 15: Including the `math` module in another file.

In the Node.js implementation of modules, each module must be defined in its own file and each module is executed in its own module-level scope. This means that variables defined at module level do not end up in the global scope of the code that uses a module.

[22] `http://wiki.commonjs.org/wiki/Modules/1.0`

3.3.4 Portability & Feature Detection

Ever since Netscape and Microsoft provided the first publicly available JavaScript implementations in their browsers, web developers have been facing the task to write portable code, i.e. code that works correctly in different browsers. For the C language, 'portable' means that code can be compiled by a large collection of different C compilers on different platforms, while still exhibiting the same behavior. This can be achieved by *a*) only using language constructs that are supported by all targeted compilers and *b*) by using macros and conditional compilation for platform-specific parts. This makes for a good analogy to client-side JavaScript code.

First, let's look at only using language features from a well-supported and widespread language version, the basic idea behind writing portable code. Section 3.4 of the GNU Coding Standards recommends that only C89 features are to be used instead of the newer C99 standard or any kind of vendor-specific extensions. The same goes for JavaScript. Version 3 of the ECMAScript specification is supported by most major browser versions currently in use, while some version 5 features are not available in Internet Explorer 8.0 or earlier[23] and other browsers. As an example of vendor-specific extensions, JavaScript 1.7[24] –a Mozilla version of ECMAScript– supports the `yield` and `let` keywords for generator expressions and block scope. These keywords are not available in other browsers' engines, so it is not recommended to use them for anything other than code that only runs on Mozilla JavaScript engines.

The second idea in writing portable C code is the use of macros to compile platform-specific code, for example for differences between the UNIX and Windows platforms. This is necessary in low-level networking code when dealing with sockets, for which there are different APIs on UNIX and Windows platforms. JavaScript has no macros, but developers have to handle differences in browser-supported APIs, for example when using the Canvas[25] element, WebSockets[26] or WebGL[27]. Some browsers support these APIs and expose the necessary objects in their JavaScript environment while others do not. This has to be detected and handled appropriately at runtime. There are two ways to approach this problem: user-agent detection and feature detection.

User-agent detection

User-agent detection, sometimes also referred to as 'user-agent sniffing' is the practice of inspecting a browser's identification string to determine which features are supported. The user-agent string –which is available client-side through JavaScript– usually contains the name of the browser maker and the version number and name of the layout engine in use. If a developer knows that a specific browser version does or does not support a requested feature, the user-agent string can be inspected to ascertain whether that feature will be available. This works in practice, but it is not very reliable and prone to errors, especially when new browser versions are released. In the past,

[23] `http://msdn.microsoft.com/en-us/library/windows/apps/s4esdbwz(v=vs.85).aspx`

[24] `https://developer.mozilla.org/en/JavaScript/New_in_JavaScript/1.7`

[25] `http://www.w3.org/TR/html5/the-canvas-element.html`

[26] `http://dev.w3.org/html5/websockets/`

[27] `https://www.khronos.org/registry/webgl/specs/1.0/`

this technique was mostly used to write browser-specific code for both Netscape and Internet Explorer.

Feature detection

Feature detection, on the other hand, provides a much more robust way to check for the existence of specific features. The idea is that developers inspect the global object by testing for the existence of (constructor) functions or properties that they would like to use. If features are not available, they can fall back to other code or disable certain program features. The advantage of this method is that it requires no knowledge about what browser version supports which specific feature, thus providing far better compatibility with new and unknown browser versions. For an example of this technique, see listing 16 with a snippet from the jQuery 1.7 source code where the global `window` object is tested for the existence of a `JSON` object containing a `parse` function. If it exists, this function is called. Otherwise (not shown in the snippet), jQuery uses the `eval` function to parse the data. In the future, when more browsers will probably have native support for the `JSON` object and its `parse` method, jQuery will automatically use the native `JSON.parse` function.

```
// Attempt to parse using the native JSON parser first
if ( window.JSON && window.JSON.parse ) {
  return window.JSON.parse( data );
}
```

Listing 16: Feature detection in the jQuery library.

3.3.5 Documentation Comments

In their original list of code smells, Beck and Fowler list ‘Comments’ as a code smell, but then quickly explain that comments are a ‘sweet smell’. They often found comments to be superfluous, so they rightfully state that comments should primarily be used to explain *why* a specific piece of code was written instead of explaining *what* code does. However, for library authors and developers using modern IDEs, comments have a second purpose where describing *what* a method does might not be considered so bad after all.

Programming languages like Java and C# allow developers to write comments in a special format that can be used to automatically generate documentation on classes and their methods. The information embedded in these comments is often used by the IDE (at least both Eclipse and Visual Studio do this) to show more detailed information about a class or method while editing or navigating code. The general idea here is that properly documented libraries are easier to use and maintain, thus contributing positively to the overall software quality, even though the quality of the documentation says nothing about the quality of the code itself.

In the case of Java and C#, the specification of the comment format and the tools to generate the documentation are provided by Oracle and Microsoft. The ECMAScript standard provides no such specification nor any such tools. However, there are two different styles of code comments that seem to be popular, partly based on IDE support for these styles: JSDoc for Eclipse and VSDoc for Visual Studio. Apple also offers their HeaderDoc[28] system with partial support for JavaScript.

JSDoc

JSDoc[29] is inspired by the Javadoc-syntax used to write code comments for the Java language. It is supported by the JavaScript Development Tools[30] (JSDT) in Eclipse, there is an open source documentation generator[31] available and it is recommended in Google's JavaScript Style Guide. An example of this documentation comment style can be seen in listing 17. When using this style of comments, Eclipse with the JSDT installed will show additional information in its autocomplete dialogs. JetBrains products with JavaScript editing support such as IntelliJ IDEA[32] and WebStorm also support JSDoc-comments. AppCellerator's Titanium Studio and Aptana Studio (the former is based on the latter) provide a similar system with their ScriptDoc[33] documentation comments.

```
/**
 * Detailed information about an employee.
 * @param {String} name Employee name.
 * @param {String} dept Employee department name.
 * @constructor
 */
function Employee(name, dept) {
  this.name = name;
  this.dept = dept;
}

/**
 * Returns this Employee's properties as a string.
 * @returns {String}
 */
Employee.prototype.toString = function() {
  return "name: "+this.name+", dept: "+this.dept;
};
```

Listing 17: JSDoc-annotated version of the Employee example.

VSDoc

VSDoc is inspired by the syntax of C# code comments, which means XML data on comment lines starting with three forward slashes. There seems to be little or no support for this style of code comments outside Microsoft's Visual Studio. However,

[28]http://developer.apple.com/opensource/tools/headerdoc.html
[29]http://usejsdoc.org/
[30]http://www.eclipse.org/webtools/jsdt/
[31]https://github.com/micmath/jsdoc/#readme
[32]http://www.jetbrains.com/editors/javascript_editor.jsp?ide=idea
[33]http://wiki.appcelerator.org/display/tis/ScriptDoc+(SDOC)+2.0+Specification

because of its adoption by the jQuery project for providing annotated versions of the popular jQuery library, it deserves to be mentioned here. An example of this documentation comment style can be seen in listing 18. Using this style, Visual Studio will show the information embedded in the comments while editing your code, for example during autocomplete of methods.

```
function Employee(name, dept) {
    /// <summary>
    ///   Detailed information about an employee.
    /// </summary>
    /// <param name="name" type="String">
    ///   Employee name.
    /// </param>
    /// <param name="dept" type="String">
    ///   Employee department name.
    /// </param>
    /// <returns type="Employee" />

    this.name = name;
    this.dept = dept;
}

Employee.prototype.toString = function () {
    /// <summary>
    ///   Returns this Employee's properties as a string.
    /// </summary>
    /// <returns type="String" />

    return "name: " + this.name + ", dept: " + this.dept;
};
```

Listing 18: VSDoc-annotated version of the Employee example.

3.4 Static Analysis Tools

Several rules from coding guidelines can be checked by means of static analysis. This means that program source code is parsed after which an analyzer uses a symbolic representation of a program (usually the AST) to detect stylistic violations or 'forbidden' constructs. The program is analyzed without executing it, as opposed to dynamic analysis, where program code is usually instrumented and data is gathered at runtime for later analysis. In this section, we discuss several popular static analysis tools for JavaScript.

3.4.1 JSLint

A popular tool that performs static analysis of JavaScript code is the open source JSLint tool by Douglas Crockford. The name JSLint is inspired by the well-known C source code analysis tool 'lint' [18], which dates back to 1978. The analyzer is written in JavaScript itself and contains a parser for JavaScript code. JSLint performs several checks. The reasoning behind these rules should be familiar to readers of this document. A selection of the rules and their motivation:

- After analysis, all **global variables** are explicitly listed. They should be inspected, because they can be the result of misspelled names or plain design errors. The same happens for all property names. Property names that were used only once are highlighted, because these might be the result of spelling mistakes.

- **Optional semicolons** are not allowed. The analyzer expects semicolons to be used at the end of statements to avoid ambiguity while reading source code.

- To avoid problems with the misunderstanding of **function scope** and **hoisting**, all variables are expected to be declared at the top of the function that contains them.

- **Variables** are expected to be declared before they are used, although JavaScript does not necessarily require this.

- For **equality comparisons**, JSLint expects developers to use the === form instead of the shorter == form which causes type coercion and might cause unexpected results. The same holds for inequality comparisons.

- **Constructor functions** are expected to start with an uppercase character. Calling these functions should always be done using the `new` operator (and vice versa), otherwise the global object might be filled with properties that do not belong there.

- The use of the **eval** function and its aliases is marked as an error.

Most of the checks can be enabled or disabled individually before performing analysis of JavaScript code. There is an online version and there are wrappers like jslint4java[34] which enable Java developers to programmatically control and evaluate the results of a JSLint analysis run.

JSHint[35] is a fork of JSLint, which was created to have a more configurable version of JSLint. It has several differences in the set of rules and options that are used to check a program.

JavaScript Lint[36] is another open source JavaScript code analysis tool, written in Python, partly inspired by JSLint.

Section 3.5 contains a more in-depth evaluation of the structure and design of JSLint itself, in the context of replacing it with an analysis tool of equivalent ‘strength’.

[34]`http://code.google.com/p/jslint4java/`
[35]`http://jshint.com/`
[36]`http://www.javascriptlint.com/`

3.4.2 Closure Linter

Closure Linter[37] is a JavaScript analysis tool written in Python provided by Google as a companion to their JavaScript Style Guide. The Closure Linter can detect (and fix) deviations from the rules in the style guide. It also recognizes JSDoc-style comments and uses these in its checks. Several examples of the checks that are performed:

- **Single-quoted strings** are preferred over double-quoted strings.
- Appropriate **JSDoc** comments should be present. For example, a function that does not have a `return` statement should not have a superfluous `@return` JSDoc tag. This also works the other way: functions that return a value should have documentation on the return value. All function parameters should also be documented.
- Lines should not exceed 80 characters in length. Text in documentation comments is excluded.
- Top-level function definitions in an rvalue (i.e. a function value being assigned to a variable) should end with a semicolon.
- Multiple values in parameter lists should be separated by a comma followed by a space.

3.4.3 Dojo Checkstyle

The Dojo Checkstyle utility[38] is distributed as a part of the Dojo Toolkit. It is primarily targeted at Dojo developers, enabling them to check their code against the the Dojo style guide. The utility is written in JavaScript. String-based checks are used to verify conformance to the rules, thereby making it a very primitive form of static analysis. Some examples of rules in the Dojo Checkstyle utility:

- An `else` statement must not be followed by a space, except if it is an `else if` statement.
- **Trailing commas** are not allowed at the end of an object expression.
- **Opening curly braces** should not be placed at the start of a line.
- **Tabs** should be used instead of **spaces**. Every sequence of two or more consecutive spaces triggers this warning.
- **Operators** –more precisely all equality and boolean operators– should be surrounded by a single space both before and after the operator.

[37] `http://code.google.com/p/closure-linter/`
[38] `http://dojotoolkit.org/reference-guide/util/checkstyle.html`

3.4.4 Doctor JS

Doctor JS[39] also performs analysis of JavaScript code. It is based on CFA2 [33], a mechanism for context-free control flow analysis. From the analysis tools mentioned in this section, this is probably using the most sophisticated technique of JavaScript code analysis, at least on a high level. It can be used for type analysis of JavaScript code, inferring the types of function arguments and return values. Doctor JS is open source and written in JavaScript. When verified on June 26, 2012, the online version of the tool did not seem to work.

3.4.5 Inspection-JS

Inspection-JS[40] is the engine that powers the JavaScript inspections in the WebStorm IDE from JetBrains. It is discussed in more detail in section 3.7.2.

3.4.6 WALA

The T.J. Watson Libraries for Analysis[41] (WALA) contain a generic program analysis framework, primarily targeted at analysis of Java code, but it also has a front end for JavaScript code. It uses a number of different techniques like type system analysis, dataflow analysis, pointer analysis and call graph construction. A more in-depth evaluation of the tool is necessary to determine which analysis mechanisms are actually supported for JavaScript code. Also note that this is an analysis framework instead of a ready-to-use analysis tool.

3.5 JSLint Evaluation

In this project, we specifically set out to design a static code analysis tool for JavaScript that is as powerful as JSLint. Because of this requirement, we will first have a look at the internals of JSLint[42].

3.5.1 Parser

To start, we should note that JSLint does not exclusively check JavaScript code. It also validates HTML, CSS and JSON data. It tries to auto-detect the file type by looking at the first characters from the input data. Because we are only interested in the JavaScript analysis component, we would like to discard the other analysis code. Unfortunately, JSLint is structured like one big parser that can switch between different parse modes like 'html', 'style', 'styleproperty', 'script' and 'scriptstring'. Having these different parse modes makes sense, because both JavaScript and CSS can be embedded in HTML, so there should be some kind of mechanism to deal with embedded languages and special escape sequences to return to the host language when a CSS or

[39] `http://doctorjs.org/`
[40] `http://sixthandredriver.com/inspection-js.html`
[41] `http://wala.sourceforge.net/`
[42] JSLint version 2011-12-09 has been inspected for this evaluation.

JavaScript snippet is embedded in an HTML file. In the case of JSLint, there is no easily separated part that exclusively deals with JavaScript code, as some of the parsing logic is shared between the different parse modes.

For the parts of the parser that deal with JavaScript code, we see that all logic for analysis and option handling is contained in the parsing code. This has the advantage that it is relatively easy to generate very accurate errors based on the current parser state at the moment of error detection, especially for low-level syntactic errors. Errors can even be detected when the file contains one or more syntax errors. On the other hand, because the code for the parser is mixed with the code for the analysis logic, it is not easy to extend JSLint with custom error checks. This design choice also makes it hard to use JSLint for checks that span multiple files, or any checks that need to keep track of some global state within one file, as this should all become part of the JSLint parser state.

3.5.2 Error Messages

Because inspection of the parser code did not result in a good overview of the JavaScript-specific analysis capabilities, we decided to do a bottom-up analysis, starting with the error messages that JSLint can generate and tracing them back to the place where they are generated. JSLint contains a dictionary with error strings. Inspection of this dictionary gives a good idea on the amount of different rules that are being checked by JSLint. The dictionary contains 178 unique error strings with placeholders for error-instance specific information. Some examples of those strings:

- A constructor name '{a}' should start with an uppercase letter.
- '{a}' was used before it was defined.
- Unexpected parameter '{a}' in get {b} function.

These error messages are referenced using a number of different error reporting functions like `quit`, `warn` and `stop`, indicating the severity of the error. The first error from the previous list is known as `constructor_name_a` and is referenced like shown in listing 19.

```
warn('constructor_name_a', token);
```

Listing 19: JSLint error reporting.

Using several regular expressions, we made a list of all error reporting function calls which directly referenced an error message string by its unique name. By manual inspection of the JSLint source code, these errors were categorized according to the categories listed in table 3.2. The ADsafe[43] category of errors is designed to only allow a subset of the JavaScript language that has limited access to any other scripts or global variables running in the same context. In the end, we are only interested in the categories of syntactic and semantic JavaScript errors. To be clear, we consider

[43] ADsafe, `http://www.adsafe.org/`

Category	Description	Occurrences	Tests
CSS	Style sheet errors	50	0
HTML	Markup errors	42	0
JSON	JSON data errors	9	0
ADsafe	ADsafe JavaScript subset violations	49	0
Syntactic	Syntactic JavaScript errors	34	22
Semantic	Non-syntactic JavaScript errors	90	89
Alias	Alias for another error	2	0
JSLint	Errors in JSLint comment directives	4	0
Other	*Not yet categorized*	*93*	0

Table 3.2: JSLint error message categories.

an error syntactic if it needs information on token level to decide whether there is a violation. To get a good understanding of these errors, we wrote test cases that each tried to trigger exactly one unique JSLint error, focusing mostly on the semantic errors. The number of tests written is also shown in table 3.2.

In total, we can identify 373 places in the JSLint source code where one of the 178 predefined errors is being triggered. We observe several errors that are being triggered from different places in JSLint, making it more difficult to understand how the check for that rule is implemented. Out of the 373 occurrences, 83 are halting errors, meaning JSLint is not able to continue parsing. All other 290 occurrences are normal warnings. This analysis is not complete, as 93 error reporting calls have not been inspected due to the fact that classifying these errors by reverse engineering the JSLint parser code is a laborious process. We also decided that, having identified 124 different syntactic and semantic checks with unit tests for 111 of those, we had collected enough material to test our own analysis tool.

3.5.3 Findings

JSLint provides a relatively large set of JavaScript-specific rules that promote good engineering practices as observed by its author, Douglas Crockford. The collection of rules is of most value to us, as we are less impressed with the general design of the tool itself. Error checking and option handling code is mixed with the parser code and it is hard to write custom checks. JSLint also combines code to validate CSS, HTML, JSON and JavaScript in a way that makes it hard to completely separate these parts from each other. As the same error can sometimes be triggered from multiple places, the actual definition of some rules is spread out over the parser code. In chapter 4, we will design an analysis tool that aims to improve on these shortcomings, while still being as powerful as JSLint with regards to the amount and type of errors that can be detected.

3.6 Code Metrics

In their list of code smells, Beck and Fowler define three smells that contain a quantifier in their name: ‘Long Method’, ‘Large Class’ and ‘Long Parameter List’. This means we have three aspects that can be measured in a code base: method length, class size and parameter list length. They do not give any absolute values on what should be considered ‘large’ or ‘long’ but instead rely on the human intuition of the software engineer working with the code to judge when something is getting too large or too long. We now entered the domain of code metrics, an area of research that goes back to 1976 when Thomas McCabe introduced his complexity measure better known as the ‘McCabe complexity’. Since then, there has been a substantial amount of research [21, 19, 7] on calculating and using code metrics. In general, metrics are calculated based on the textual, tokenized or parsed representation of one or more source files. The output of a metric is usually represented as a single number. The meaning of this number, based on the definition of the metric, is subject to personal interpretation, based on differences in coding style and available language constructs.

Tools that can calculate metrics are commonly available for a number of different IDEs. Some examples are the ‘Code Analysis Tools’[44] in Visual Studio, and Java analysis tools such as Checkstyle[45], PMD[46] and CodePro Analytix[47], all available as an Eclipse plugin. Because of the integration of metrics and code analysis in modern IDEs, it is easy to find extremes and outliers that might be reason for closer inspection by the developer. This way, long methods can sometimes be split in a set of distinct methods, large classes can be divided in several related, smaller classes and long parameter lists can be removed in favor of passing objects as arguments. This is how the use of code metrics can contribute to software quality in general.

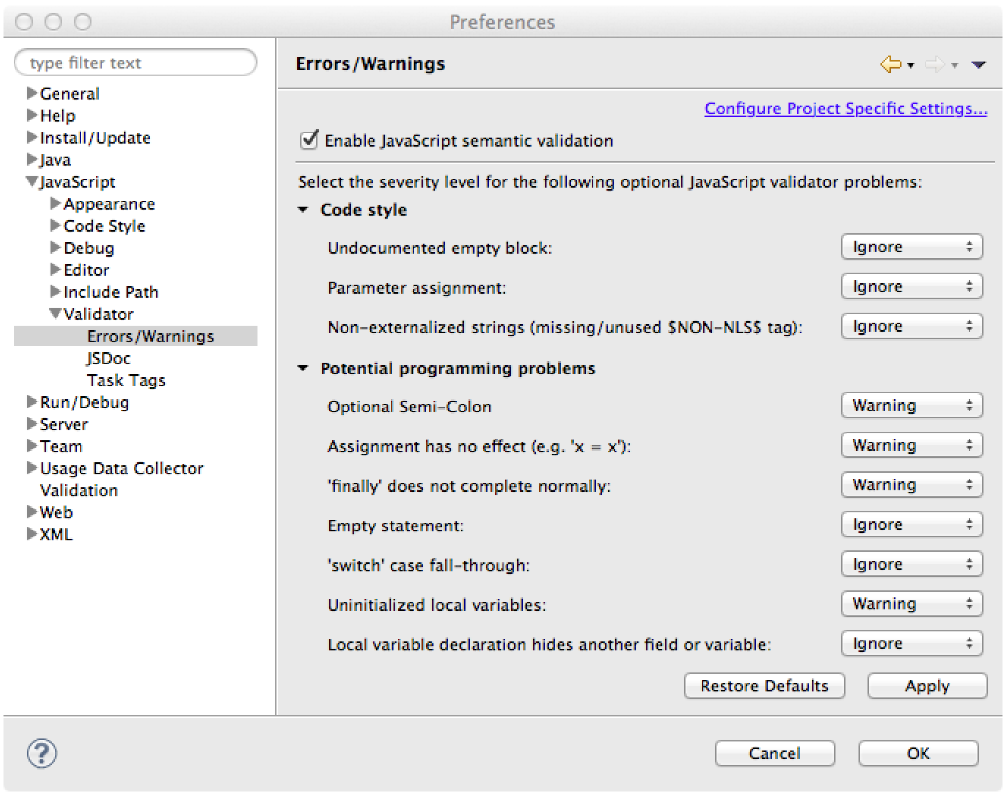
Because JavaScript lacks classes and inheritance like found in class-based object-oriented languages, several standard metrics make no sense for JavaScript or are hard to map to the prototypal inheritance mechanism. A literature search using Google Scholar, ACM Digital Library and IEEE Xplore did not result in any relevant research regarding code metrics for JavaScript code.

Calculation of metrics for JavaScript does not seem to be a very popular feature of actual code analysis frameworks. A web search for implementations of metrics calculation for JavaScript code only resulted in one project: jsmeter[48], an open source project (written in JavaScript) that calculates seven metrics including the cyclomatic complexity for JavaScript code. Apart from that, the WebStorm IDE contains several ‘function metrics’[49] as part of its code inspection feature.

3.7 IDE Support

Discussed tools from the previous sections were standalone tools. Modern IDEs targeted at web developers can support writing JavaScript code by offering integration for

[44] `http://msdn.microsoft.com/en-us/library/dd264897.aspx`
[45] `http://checkstyle.sourceforge.net/`
[46] `http://pmd.sourceforge.net/`
[47] `http://code.google.com/javadevtools/codepro/doc/`
[48] jsmeter, `http://code.google.com/p/jsmeter/`
[49] See section 3.7.2 for more details.


Preferences

Errors/Warnings

Configure Project Specific Settings...

☑ Enable JavaScript semantic validation

Select the severity level for the following optional JavaScript validator problems:

Code style

Undocumented empty block:	Ignore
Parameter assignment:	Ignore
Non-externalized strings (missing/unused $NON-NLS$ tag):	Ignore

Potential programming problems

Optional Semi-Colon	Warning
Assignment has no effect (e.g. 'x = x'):	Warning
'finally' does not complete normally:	Warning
Empty statement:	Ignore
'switch' case fall-through:	Ignore
Uninitialized local variables:	Warning
Local variable declaration hides another field or variable:	Ignore

Figure 3.1: Eclipse JSDT JavaScript semantic validation.

these standalone tools or by providing their own plugins for code analysis and refactoring. In this section, we look at two IDEs and we compare their features in the area of JavaScript code quality.

3.7.1 Eclipse JavaScript Development Tools

The JavaScript Development Tools project from the Eclipse IDE is open source and contains a number of features to provide support for JavaScript editing, code formatting, code analysis, refactoring and debugging in the Eclipse IDE. To get an idea of what is supported in the most recent version[50], we give a brief summary of the code quality analysis and refactoring features.

Code Quality Analysis

The plugin ships with a set of rules that can be found in the 'JavaScript semantic validation' feature. There are three categories of rules: 'Code style', 'Potential programming problems' and 'Unnecessary code'. Rules can be enabled individually and the severity of the resulting message can be configured (choice between 'warning' and 'error'), as can be seen in figure 3.1. Some examples of these rules:

[50]Tested with Eclipse 3.7.1 and JavaScript Development Tools 1.3.1.

- A local variable declaration hides another field or variable. For example, when a function parameter name is identical to the name of a global variable.
- Statement without semicolon. Identical to the rule in JSLint. Every statement should be terminated with a semicolon.
- Uninitialized local variable. Declared, but not assigned to.
- Local variable is never read. Declared and assigned to, but not read from.
- Assignment has no effect. Triggered when the left and right hand side of an assignment statement are equal.

Refactoring

Similar to the refactoring features for the Java language, a selection of refactoring options is also offered for the JavaScript language:

- Rename.
- Move.
- Change Function Signature.
- Extract Function.
- Extract Local Variable.
- Inline.
- Introduce Parameter.

3.7.2 JetBrains WebStorm

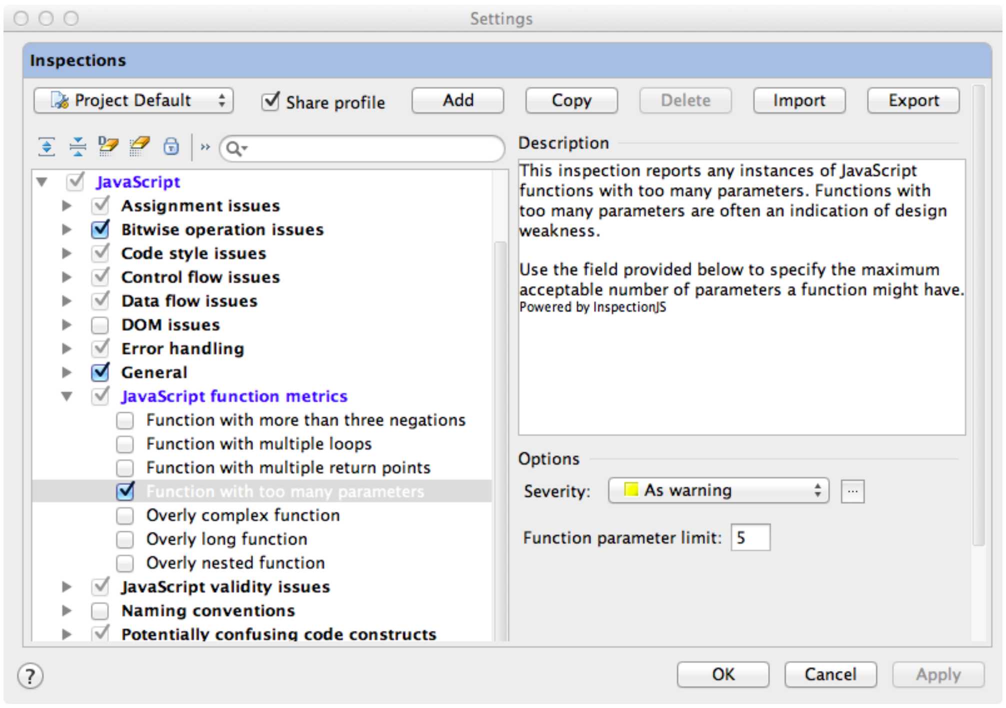
The WebStorm IDE from JetBrains is specifically targeted at web developers and contains support for JavaScript code editing, analysis and refactoring. Although there is an open source edition of the core IDE, sources of the JavaScript-specific plugins and extensions are not available. In this section, we present a short overview of the included features for code analysis and refactoring. This has been evaluated in the most recent version of WebStorm at the time of writing[51].

Code Quality Analysis

Code analysis features are available through the plugin Inspection-JS, which is bundled with WebStorm. There are more than 10 different categories of rules like 'Assignment issues', 'Code style issues', 'Error handling', 'JavaScript function metrics' and 'Naming conventions'. Rules can be enabled individually, the severity of the resulting message can be configured and, in the case of metrics, thresholds can be set. See figure 3.2 for the configuration of these rules. WebStorm 3.0, released in December 2011[52], integrates support for code analysis with JSLint and JSHint. Overall, code quality analysis support seems to be far more extensive than in Eclipse.

[51]Tested with WebStorm 4.0.2.

[52]`http://www.jetbrains.com/webstorm/whatsnew/whatsnew_30.html`



- ▼ ✓ JavaScript
  - ▶ ✓ Assignment issues
  - ▶ ✓ Bitwise operation issues
  - ▶ ✓ Code style issues
  - ▶ ✓ Control flow issues
  - ▶ ✓ Data flow issues
  - ▶ ☐ DOM issues
  - ▶ ✓ Error handling
  - ▶ ✓ General
  - ▼ ✓ JavaScript function metrics
    - ☐ Function with more than three negations
    - ☐ Function with multiple loops
    - ☐ Function with multiple return points
    - ✓ Function with too many parameters
    - ☐ Overly complex function
    - ☐ Overly long function
    - ☐ Overly nested function
  - ▶ ✓ JavaScript validity issues
  - ▶ ☐ Naming conventions
  - ▶ ✓ Potentially confusing code constructs

This inspection reports any instances of JavaScript functions with too many parameters. Functions with too many parameters are often an indication of design weakness.

Use the field provided below to specify the maximum acceptable number of parameters a function might have.
Powered by InspectionJS

Severity: As warning

Function parameter limit: 5

Figure 3.2: Metrics in the WebStorm JavaScript inspections.

Refactoring

Just like Eclipse, WebStorm also offers a selection of refactoring options for the JavaScript language:

- Rename.
- Change Signature.
- Move.
- Copy.
- Safe Delete.
- Extract Method.
- Introduce Variable.
- Introduce Parameter.
- Inline.

This is mostly similar to the list of refactoring options offered by the JSDT project.

3.8 Summary

This chapter dealt with a number of different views on what code quality means and how it can be determined and measured. Starting with the description of code smells by Beck and Fowler, we applied several of their ideas to JavaScript code. After that, the code smell ‘Duplicated Code’ was discussed in more detail in the separate section on code clones. Much of the ideas from existing research on code clones seem to apply to JavaScript as well, but JavaScript-specific research is lacking and implementations of code clone detectors for JavaScript are hard to find. The next section on coding guidelines treated several well-known guidelines and standards and their relevance to JavaScript code, in particular on library behavior and on portability, followed by a section discussing several static analysis tools. The penultimate section contains a short discussion on code metrics, on which there has been a good amount of research dating back to 1976. However, just as with code clones, relevant research and actual implementations are hard to find. In the final section, code quality-related features from two popular IDEs are compared. Table 3.3 gives an overview of JavaScript-specific tools and technologies discussed or mentioned in this chapter. Unless otherwise noted, source code of the tools is available (where applicable).

All in all, this means that there are good possibilities for future research and work in implementing code quality analysis tools for JavaScript, especially concerning code clones and metrics. Available code analysis tools tend to ‘stick’ to one set of rules or guidelines and do not seem to be very flexible or configurable.

Category	Name	Notes
Formatting	JS Beautifier	Configurable output styles.
	Closure Compiler	Also performs optimization and obfuscation.
	UglifyJS	Idem.
Code Clones	PMD CPD	Runs standalone or integrated in Eclipse.
	ECMAScript CloneDR	Closed source, Windows only. Not evaluated.
Documentation	JSDoc	Inspired by JavaDoc. Supported by Eclipse, JetBrains, Aptana.
	ScriptDoc	Similar to JSDoc. Identical syntax, different tags.
	VSDoc	Supported by Visual Studio.
Code Analysis	JSLint	Full parser & static code analysis.
	JSHint	Forked from JSLint, more flexible.
	JavaScript Lint	Inspired by JSLint.
	Closure Linter	Written in Python, including full parser.
	Dojo Checkstyle	Written in JavaScript. String-based matching.
	Doctor JS	CFA2 control flow analysis, type inference.
	WALA	Framework, supports various analysis techniques.
Metrics	jsmeter	Calculates seven metrics.
IDE	Eclipse w/ JSDT	Highlighting, autocompletion, JSDoc, analysis and refactoring.
	WebStorm	Closed source. Similar to Eclipse, but with more analysis features including metrics.

Table 3.3: Tools mentioned in chapter 3.

Chapter 4

Tool Structure and Implementation

In the evaluation of JSLint, we have noticed that it lacks a clear separation in distinct components or analysis stages. We would like to propose a much more structured approach, composing the static analysis tool of five different processes, each with its own responsibility. These processes do not have any overlap in functionality and they are connected by exchanging typed datasets. This is done to prevent tight coupling, which would make it hard to make changes in a component without having to propagate these changes to other components. It also allows us to insert a new component between any two existing components, as long as it conforms to the original data interface. These new components could perform useful transformations on the data before handing it over to the next stage.

We recognize several processes with their own input and output formats, as shown in table 4.1. The processes are connected as shown in figure 4.1. Note that we made an important decision that makes our approach fundamentally different from JSLint: we require that all input is valid JavaScript. Because the analysis code of JSLint is integrated in its parser, it can detect certain types of syntactic errors while being able to continue parsing. We are not able to do this, as the AST and typed AST models can only represent valid JavaScript. Any syntactic errors will therefore be reported by the parser, not by our analysis process. We can still intercept parser errors and report them directly, but we have to completely skip the analysis process, as we have no AST available when a parser error occurred.

All processes and relevant datasets will be discussed in-depth in the remaining sections of this chapter.

Process	Input	Output
Parsing	JavaScript code	AST
Model Transformation	AST	Typed AST
Analysis	Typed AST & Rules	Analysis results
Filtering	Analysis results & Filters	Filtered results
Reporting	Filtered results & Original source	Results report

Table 4.1: Analysis tool processes and datasets.

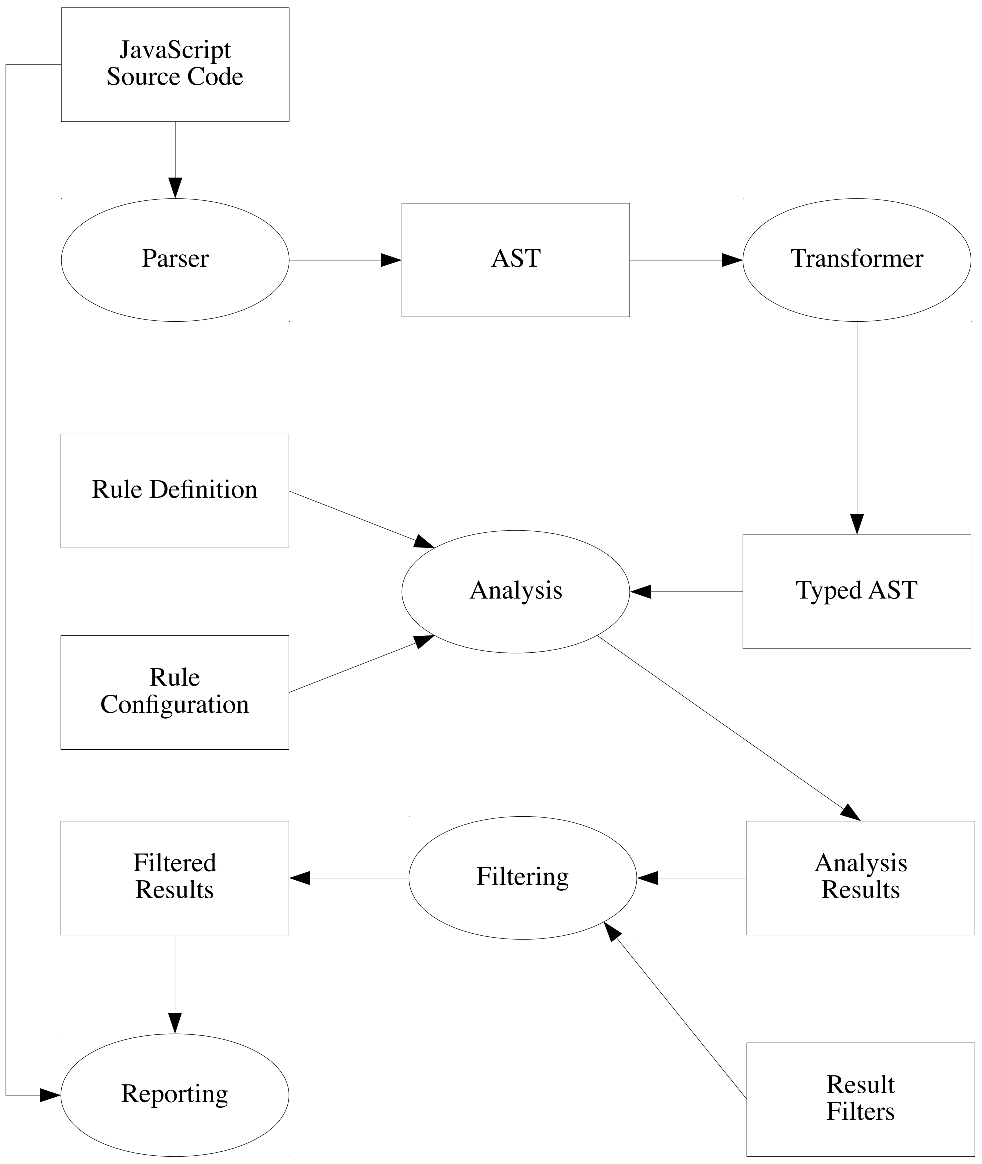


Figure 4.1: Analysis tool components and data-flow.

4.1 Parsing

Because the target of this project is to design and create a static analysis tool, we decided to use an existing JavaScript parser, as writing a parser from scratch would probably consume too much time. To find a usable parser, we compared and evaluated several publicly available JavaScript parsers, considering this list of requirements:

- It is written in JavaScript and it works in Node.js.
- It is open source, but licensed to allow commercial use.
- Its API and its output formats are standardized and documented.
- Detailed token location information (offset or line and column) is included in the AST.
- The AST has enough information to support regeneration of source code. It includes raw literals in the output.
- All line and block comments are returned, with location information or a connection to an AST node.

Table 4.2 shows several JavaScript parsers that we compared and evaluated. For the location information and comment requirements, we constructed several test input files. When we could not reliably determine the location of every single node in the resulting AST, we rated this as ‘partial’ or ‘no’. The presence of all comments (both line and block types) in the AST (or a separate comments list) was also tested.

The Esprima[1] parser meets all requirements and it has an easy-to-understand output format. We decided to use this parser to start building our analysis tool. Due to the fact that the Esprima project was relatively young, we assumed that the likelihood of getting our patches accepted into the main development branch was high. During this project, we made several contributions[2] that were successfully incorporated in the Esprima parser. As an example, raw literals were not available when we started this project, but we proposed a patch that added support for raw literals, arguing that this is an important feature for people using the parse tree for code generation. This allows people to inspect the type of string literal quotes (single, double) or the numerical notation (decimal, octal, hexadecimal) of a specific literal.

Now that we have a parser, we need to transform its output to a typed dataset conforming to a schema. This step will be explained in the next section.

4.2 Model Transformation

To generate a complete model of the JavaScript language that covers all parts of the ECMAScript 5.1 language specification, we decided to start with the output from the Esprima parser when subsequently parsing all source files of the M-Industries framework. Our goal was to be able to represent all these parsed files in a model. We created

[1] `http://esprima.org/`

[2] `https://github.com/ariya/esprima/commits?author=joostwim`

Name	Loc. Info?	Comments	License
Narcissus	yes	partial	MPL / GPL / LGPL
Uglify-JS	partial	partial	BSD
Esprima	yes	yes	BSD
ZeParser	yes	partial	MIT
PEG.js	no	no	MIT
Language.js	no	no	MIT
es-lab	yes	no	Apache 2.0

Table 4.2: Open source JavaScript parsers.

a schema for this model using a top-down process, gradually expanding the schema as we encountered new constructs (statements, expressions) that we could not represent during the previous attempt.

Using this approach, we started with two top-level schema types: a `program` type and a `statement` type, with the `program` type having exactly one property, `body`, containing a list of `statement` instances. However, the `statement` type itself was still empty, so translating the Esprima AST to a model immediately failed for the first statement we encountered, as we could not represent this statement in our model. Because the first statement we encountered was a `VariableDeclaration`, we added a state group `type` as a property to our `statement` type, with a specific state for the variable declaration. Listing 20 shows the schema for this first version of the model.

```
type "program"
  list "body"
    component "statement" -> "statement"

type "statement"
  stategroup "type"
    state "variable declaration"
      list "declarations"
        text "id"
        stategroup "initializer"
          state "value"
            component "init" -> "expression"
          state "empty"

type "expression"
  stategroup "type"
```

Listing 20: The first partial JavaScript language schema.

When repeatedly applying this process, we eventually produced a version of the model that was able to represent the ASTs of all source files in the M-Industries framework. This model contained the top-level types `program`, `statement`, `expression` and `literal`, with appropriate subtypes for the last three of those types.

One downside of this approach is that it is completely unidirectional: the modeled, typed version of the AST is not being tested nor being used, so while we can claim that we successfully created a model of the AST, there is no way to guarantee that this is a complete and correct model. We needed to improve this by setting up a method to test our model.

4.2.1 Parser & Model Tests

Now we have decided to test our model, it is important that we establish what we exactly want to test. We want to make sure that we have a complete mapping of all JavaScript language constructs to concepts in our model, and that we do not lose any semantic information while constructing our model. The best way to test this is to regenerate source code using our model data. To do this, we build a serializer that reads the model data and produces JavaScript source code. This enables us to compare the generated source code to the original source code. If we do that using a string comparison, we implicitly require that the syntax and comments are completely identical as well, which is a level of detail that we do not consider relevant at this moment. The focus is on having a method that guarantees complete semantic equivalence of the original input and the regenerated source code. We can do this if we perform the comparison on a semantic level, by parsing the regenerated source code again and comparing the parse trees. This is the roundtrip process to validate our model, shown in listing 21.

```
ast = parse(source);
regen = serialize(transform(ast));

is_equal = (parse(regen) == ast);
```

Listing 21: The roundtrip process in pseudocode.

The approach we described earlier has two disadvantages:

- The output of the transformation (the model) is not verified to be correct.
- The input data set does not use all language constructs (e.g. getters and setters in object literal expressions).

With the roundtrip process, we have a solution for the first point. We now have a method that ensures us no semantic information is lost during the translation from the AST to our model. That still leaves the second point. To ensure that we can model all language constructs, we need a way to have more certainty that all possible language constructs can be expressed in our model. We approach this from two different directions:

- Manually compare the language specification to our model schema.
- Run a large set of test code through our parsing and modeling chain, while also validating the results with the roundtrip process. We use the official Test262[3]

[3]http://test262.ecmascript.org/

test suite, from which we extract all parsable files[4] and we feed them to our parser and subject them to the roundtrip process.

After we completed these two tasks, we had enough certainty that we were able to express the complete JavaScript language in our model. It proved to be necessary to actually perform this validation, as we uncovered several places where our model was lacking some information. For example, because getters and setters in object literal expressions were not used in the M-Industries codebase, we forgot to include them in our model. This also happened for the `debugger` statement that acts as an explicit breakpoint when running code in a JavaScript debugger.

The end result is that we have a complete model of the JavaScript language that we can use for our analysis process. However, during the work on the model, we noticed several areas where we could improve it by taking advantage of specific features of the M-Industries modeling language.

4.2.2 Normalization

After we completed the first version of the schema for the model of the JavaScript language, we used several different approaches to normalize and improve the schema.

Textual Properties

In the initial schema we created, there are several textual properties that can be constrained to a limited set of options. As an example, listing 22 shows a 'unary expression' which is the parsed form of expressions like '`typeof x`'. These expressions have an operator and an argument. With a textual operator property, all analysis code that wants to handle special cases (e.g. disallow 'typeof' expressions) has to use string comparisons in `if` or `switch` statements on the value of the operator property. We would like to make these choices explicit in the schema itself, using state groups to represent all possible values for the operator. Listing 23 shows how this was done in the normalized version of the schema.

```
type "expression"
  stategroup "type"
    state "unary"
      component "argument" -> "expression"
      text "operator"
```

Listing 22: Schema fragment with textual operator property.

Shared Properties

In some cases, several related statements or expressions share a set of identical properties. Where possible, we try to put all the shared properties in a containing type fragment and add a state group for all the different variants. We only apply this when the shared properties have more in common than just their syntax. As an example, both

[4] We found 11,147 parsable files in revision 335 from April 16, 2012.

```
type "expression"
  stategroup "type"
    state "unary"
      component "argument" -> "expression"
      stategroup "operator"
        state "delete"
        state "void"
        state "typeof"
        state "+"
        state "-"
        state "~"
        state "!"
```

Listing 23: Schema fragment with state group operator property.

‘binary expression’ and ‘assignment expression’ have one ‘operator’ property and two ‘expression’ properties for the left and right hand sides, but they are only similar on a syntactic level. It would not be logical to group these two together, as the ECMAScript specification defines different behavior for these two types of expressions.

```
x + y;
x = y;
```

Listing 24: Syntactic similarity of binary and assignment expression.

Listing 25 shows how the different types of iteration statements are grouped together, all sharing a ‘body’ property in the containing type fragment. To keep the example small, we omitted the details of the different iteration statements.

```
type "statement"
  stategroup "type"
    state "iteration"
      stategroup "type"
        state "while"
        state "do while"
        state "for"
        state "for in"
      component "body" -> "statement"
```

Listing 25: Grouping all iteration statements.

References

Because the schema language supports making references to instances of specific type fragments, we searched for places in the schema where we could enhance the model by using references. As it turns out, there are several language constructs that can benefit from having some of their properties modeled as a reference.

The first example is the `break` statement (and, similarly, the `continue` statement). The ECMAScript specification [6] section 12.8 states that a program is syntactically incorrect if the following condition is true:

> The program contains a **break** statement with the optional *Identifier*, where *Identifier* does not appear in the label set of an enclosing (but not crossing function boundaries) *Statement*.

This means that, whenever we encounter a labeled `break` statement, we should already have parsed the label that it points to. We extended our AST transformer to keep track of the current label set while transforming the input AST to our AST model, following the mechanism outlined in the language specification. Using this label set, we were able to resolve references to labels thus enabling us to make these references explicit in the schema, like demonstrated in listing 26. In this listing, we see that a labeled `break` statement does not contain a textual property with the name of the label, instead it only has a reference to a labeled statement. The labeled statement, which is a specific type of statement in our list of statement types, does contain a textual property with the label name. Modeling this with references allows for more powerful program analysis. For example, to find all labeled statements that do not have any `break` or `continue` statements pointing to them, we can use built-in dataset traversal methods from the framework.

```
type "statement"
  stategroup "type"
    state "labeled"
      text "label"
      component "body" -> "statement"
    state "break"
      stategroup "type"
        state "labeled"
          reference "statement" -> "statement.type*labeled"
        state "unlabeled"
          reference "statement" -> "breakable statement"
```

Listing 26: Usage of references to model a `break` statement.

The second example illustrating the benefits of using references in our model comes from JavaScript identifiers. We decided to enhance the model by indicating the difference between a place where a variable is declared and a place where it is being used. In JavaScript, one can declare a variable using a number of different language constructs:

- As a parameter in a function parameter list.
- As parameter of a `catch` clause on a `try` statement.
- As function name in a function declaration statement.
- As function name in a function expression.

- As variable declaration in a variable declaration list.

In the end, all these methods accomplish the same: they introduce a new identifier in their scope. In the schema, we added a special top-level 'declaration' type to model these declarations of identifiers. Listing 27 shows how this is expressed in the schema. Before this change, the 'id' property of a function declaration was a textual property. The listing also shows that an identifier expression has a reference to a declaration, which is in fact the only place in the schema where we make such a reference.

```
type "declaration"
  text "id"

type "statement"
  stategroup "type"
    state "function declaration"
      component "id" -> "declaration"
      component "function" -> "function"

type "expression"
  stategroup "type"
    state "identifier"
      reference "declaration" -> "declaration"
```

Listing 27: Definition of a function declaration.

Now we have all our identifier expressions modeled as references to a declaration, we observe that there are two cases that we can not express in our model, when the reference has to point to one of these items:

- The built-in `arguments`[5] identifier that is available in every function that does not already declare a parameter or local variable with the name 'arguments'.
- A global variable. This can be classified as one of:
  - Language built-in globals (e.g. `eval`, `Function`, `Date`).
  - Host environment globals such as `console` in Node.js and `window` in the browser.
  - User-defined globals.

To process all identifier expressions correctly, we adapted our AST transformer to keep track of the scope in which a variable is declared. When we start transforming a file, we initialize the top scope object of that file. For every function we enter, we create a new scope object that is linked to its parent scope. The special `arguments` identifier is handled at function level.

[5]From the ECMAScript specification [6], section 10.6: "When control enters an execution context for function code, an arguments object is created unless (as specified in 10.5) the identifier **arguments** occurs as an *Identifier* in the functions *FormalParameterList* or occurs as the *Identifier* of a *VariableDeclaration* or *FunctionDeclaration* contained in the function code."

Actually resolving references to local variables is a two-pass process, because JavaScript allows the use of variables before their declaration. See section 2.3.7 for more information. Only after we processed the entire program, we can successfully resolve all references to local variables. When those references have been resolved, the remaining unresolved references are classified as global variables.

Our improved AST transformer can now classify all identifier expressions as references to locals, function-level 'arguments' or global variables, but we still have a problem with two language constructs:

- The **eval** function.
- The **with** statement.

In listing 28, we see an example of the `eval` function being used to dynamically execute a JavaScript code fragment, which declares a local variable `x`. On line 3, we have an identifier expression referring to this local variable, but it is impossible to correctly resolve this reference. Theoretically, the string argument supplied to the `eval` function can come from anywhere, the exact contents is only known at runtime, at the time the function is called.

```
1 function test() {
2   eval("var x = 42;");
3   console.log(x);
4 }
5
6 test();
```

Listing 28: Dynamically introducing a new variable using `eval`.

For the `with` statement, we have an example in listing 29. The `with` statement introduces all properties from a JavaScript object in the local scope. This means that, in the example, we can correctly resolve the reference to the variable `z`, but there is no way to determine where `x` and `y` come from. Just as with the `eval` function, we can not deal with variables that are only added to the local scope at runtime.

This is a limitation of static analysis of JavaScript code. We can find earlier research in which the authors describe the same problems. Guha et al. [12] give a formal specification of the core JavaScript language that lacks `eval`, `with` and `this`. Chugh et al. [4] propose an approach for incremental analysis of JavaScript code in which they inline all dynamically evaluated code blocks. Guarnieri et al. [11] describe ACTARUS, a static taint analysis system for JavaScript code. The system does not support the use of 'reflective calls' e.g. dynamic code evaluation using `eval`. Jang and Choe [15] have been working on a mechanism to do points-to-analysis for JavaScript code, which is necessary to enable further research and technology for code analysis and code optimization. Their analysis works for a subset of JavaScript –called 'SimpleScript'– that lacks dynamic code evaluation, prototypes, property deletion and a specific form of type coercion.

In the M-Industries codebase, we do not allow the usage of `eval` or `with`. JSLint takes a more radical approach: its parser does not even recognize the `with` statement.

```
var test = { x: 1, y: 4 };
var z = 9;

with (test) {
  console.log(x + y + z);
}
```

Listing 29: Dynamically introducing new variables using `with`.

This concludes the section on schema normalization. We ended up with a schema that has most textual properties replaced with state groups. Shared properties have been moved to containing type fragments and references are used to model labels and variables. There are some limitations to what kind of variable references we can resolve, but we know exactly what the problem is and we can prevent it by not using `eval` and `with`.

4.3 Analysis

In the whole chain, the responsibility of the analysis process is to transform a typed AST into an analysis result set. To perform the analysis, the process expects a typed AST and a rule set.

A rule set contains a list of rules. Each rule is composed of three parts: an instrument, a set of user options and a result metric filter function. The instrument specifies how to calculate a certain metric value for a certain type of input. An example would be an instrument that calculates function body length (as statement count). When you use this instrument in a rule, you have to specify how to interpret the resulting metric values. This is done with the metric filter function of the rule. A rule that reports lengthy functions would use the same instrument as a rule that reports very short functions, but with a different metric filter.

After our evaluation of the ruleset in JSLint in section 3.5, we identified two rule types: syntactic rules and semantic rules. We used these different rule types in the design of our own analysis tool. The current analysis engine works on single files.

Listing 30 gives an explanation of the analysis process. For every rule, we first check if there are user options for that specific rule. After that, we get a list of all nodes from the AST that match a specific filter pattern, using the `query` function on the dataset. Querying with filter patterns is a built-in feature of the M-Industries framework. The query language is somewhat similar to XPath[6] queries, but with a *very* limited syntax. Some examples of filter patterns that are being used in the rules:

- `expression.type⋆unary.operator⋆delete`
- `expression.type⋆update.type⋆postfix`
- `statement.type⋆return`

[6] `http://www.w3.org/TR/xpath/`

```
function analyse(typed_ast, ruleset, user_options) {
  var metrics = [];

  for (var rule in ruleset) {
    rule.config = user_options[rule.name] || rule.config;

    for (var element in typed_ast.query(rule.pattern)) {
      switch(rule.type) {
        case "semantic":
          metrics.append(semantic_analysis(rule, element));
          break;
        case "syntactic":
          metrics.append(syntactic_analysis(rule, element));
          break;
      }
    }
  }

  return metrics;
}
```

Listing 30: Partial analysis process pseudocode.

These patterns allow you to globally select all nodes in the dataset that are created from a certain top-level schema type[7]. The patterns include the possibility to filter on specific state group property values. The pattern

```
expression.type*unary
```

will return all unary expressions whereas the pattern

```
expression.type*unary.operator*delete
```

is more specific and only returns the unary `delete` expressions.

The result of the `query` function is a flat list of nodes anywhere in the AST. For every node in the list, we apply the rule, which can be using either a semantic or a syntactic instrument. ‘Applying’ a rule means providing several helper functions and then calling one or more functions that contains the actual instrument definition. The instrument definition is explained in the next section. Finally, the results from all rules are appended to and returned in the `metrics` array, which is later turned into a typed list (see listing 31). Thus, the analysis process can be described with this mapping:

$$(typed\_ast, ruleset, user\_options) \rightarrow metrics$$

4.3.1 Instrument Definition

Instruments are defined as JavaScript objects. They must be defined in a file named ‘rule.js’. The directory name is used as the identifying name of the instrument. The

[7]Like the XPath ‘descendant’ axis.

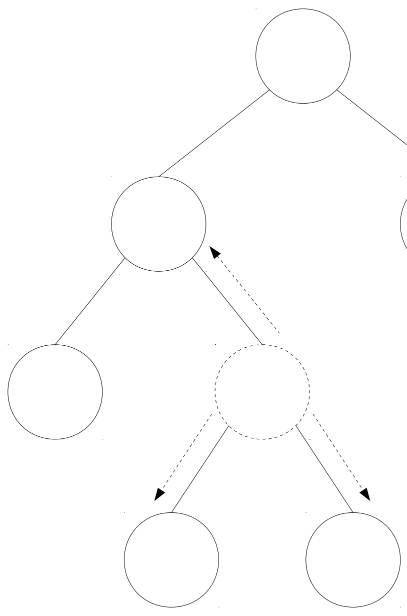


Figure 4.2: Tree navigation for semantic instruments.

directory containing the 'rule.js' file should also contain a file named 'test.js' which contains several lines of JavaScript code and a special directive in a block comment that indicates the expected results of applying the instrument to those lines of code.

Semantic Instrument

The semantic instruments are used for any checks that do not need token level information to calculate the metric values. The filter function of a semantic instrument is called for each element from the list of AST query results. The element object has methods for simple tree traversal (up, down), as illustrated in figure 4.2. The instrument definition has four properties:

- **type**. *String*. Must be 'semantic'.
- **message**. *String*. Description, used for detailed reporting.
- **pattern**. *String*. Query pattern for the nodes that will be checked.
- **filter**. *Function*. Actual implementation of the instrument. Maps individual AST nodes to a metric value and a list of optional supporting nodes:

$$element \rightarrow (supporting\_nodes, metric\_value)$$

Syntactic Instrument

We separated the syntactic instruments from the semantic instruments. An instrument is classified as syntactic when it needs information on token level to calculate its metric value. Even then, the process to write a syntactic instrument is still split in two steps:

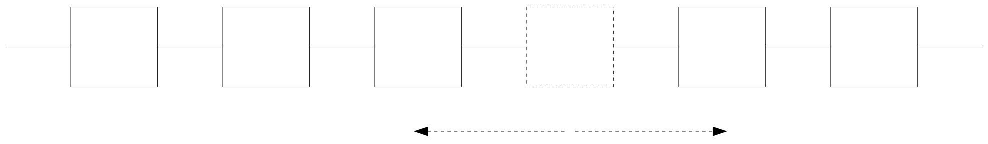


Figure 4.3: Token list navigation for syntactic instruments.

first, starting from a selected element, you traverse the tree to determine if the element should be checked on token level. For each element from this filtered list, the first and last token are determined and passed on to the second step, which has to calculate the metric value based on the tokens. Tokens are part of a token list, which can be walked in both directions, as illustrated in figure 4.3. The instrument definition has five properties:

- **type**. *String*. Must be 'syntactic'.
- **message**. *String*. Description, used for detailed reporting.
- **pattern**. *String*. Query pattern for the nodes that will be checked.
- **filter**. *Function*. First part of the instrument. Filters the list of nodes:

$$element \rightarrow boolean$$

- **inspect**. *Function*. Final part of the instrument. Receives token level information on filtered nodes, maps this to a metric value and a source range:

$$(start\_token, end\_token) \rightarrow ((range\_start, range\_end), metric\_value)$$

The analysis process is flexible enough to support new analysis & instrument styles. Appendix B contains an overview of all instruments that were written during this project. After all metrics are calculated for a single file, the results are returned in a dataset conforming to the result schema.

4.3.2 Results

The result schema is shown in listing 31. For every metric in the list, we set a number of different properties. 'Context' points to the origin of the metric in the source file. For example, if we have an instrument that calculates function body length, the 'context' will be the complete function for which the length was calculated. Semantic instruments can optionally include a list of 'support' nodes that give more detailed information about a metric. For example, an instrument that detects variable hiding[8] will set the context to the hiding variable and will add the hidden variable to the list of support nodes.

Before the actual results are displayed using one of the reporters, they are first handed over to the filtering process.

[8]Variable hiding occurs when a declaration `X` hides an identifier `X` defined in a containing scope.

```
type "location"
  number "start"
  number "end"

type "metric"
  text "rule"
  text "message"
  number "value"
  component "context" -> "location"
  list "support"
    component "node" -> "location"

type "result"
  list "metrics"
    component "metric" -> "metric"
```

Listing 31: The analysis result schema.

4.4 Filtering

There is one optional step that will be executed between analysis and reporting: filtering the result set. The resulting metric values can be filtered separately per rule. For example, when there is an instrument that calculates function body length or cyclomatic complexity, users are usually only interested in values exceeding a certain threshold value, for a specific rule. Filter are expressed as JavaScript functions that perform this mapping:

$$metric\_value \rightarrow boolean$$

4.5 Reporting

As the final process in our tool, we report the filtered analysis results back to the user. The input for the reporting process is an instance of the 'result' type from the analysis result schema in listing 31. The responsibility of the reporter is to transform this data to any kind of desired representation. Let us repeat two requirements from section 1.2:

- It should be possible to integrate the tool with the version control system to validate all commits.
- There should be an option to run the tool locally from within a number of IDEs and editors on different operating systems.

The version control system in use at M-Industries is Git[9]. Git has the ability to specify a 'pre-receive hook'[10] that can run arbitrary code when receiving a number of commits. Using this hook, we have the ability to conditionally accept a collection of commits sent to a specific Git repository, depending on the exit code of an external

[9]`http://git-scm.com/`
[10]`http://git-scm.com/book/en/Customizing-Git-Git-Hooks#Server-Side-Hooks`

process. If the exit code indicates that an error has occurred, the text written on the standard output stream will be relayed to the client that is sending the commits. With a simple command line wrapper that invokes the analysis process, we integrated this in the main M-Industries Git repository.

The other requirement was that the analysis tool could be run locally, from inside the editor or IDE. The developers from M-Industries use a number of different editors[11] on different operating systems[12]. We decided it would not be worth the effort to create a specific plugin for one or more of these editors, but instead tried to find an approach that would work for all editors and operating systems. The easiest method, supported by all editors, was to have the analysis tool run as an external process while providing its output on the standard output stream. Listing 32 shows an example of this output. Lines formatted according to the pattern `FILE_NAME:LINE:COLUMN` are recognized by most editors and are usually transformed into links that open the file in the editor and scroll to the indicated line and column when clicked.

```
rules/syntactic/statement_semicolon/test.jscript:1:6
log()
     ^ - Missing semicolon.

rules/syntactic/statement_semicolon/test.jscript:2:14
var x = 3 + 4
             ^ - Missing semicolon.
```

Listing 32: Example of detailed textual output.

4.5.1 Reporters

To support different output scenarios, we created a number of different reporters:

- **Simple textual**. Tab-delimited output with one line per metric, indicating file path, location, metric name and metric value. Can easily be processed in Excel to create pivot tables that give an overview of all violations in a set of files.

- **Detailed textual**. List of results with indications of occurrence in the source code (as shown in listing 32). Produces a summary overview after the last file has been validated. This reporter is also used for the Git hook.

- **Syntax-highlighted, as HTML**. To simulate how our tool should work when integrated in an IDE. Shows syntax-highlighted source code with marked regions for all analysis results.

- **SQL**. Custom reporter to directly insert the results in a database table. Useful when you want to collect and analyze metrics from a large collection of files. This was used for the validation of our analysis tool, discussed in the next chapter.

[11]WebStorm, PhpStorm, Eclipse, VIM and TextPad.
[12]Windows, Linux and Mac OS X.

To reproduce snippets of source code in the reporter, the original JavaScript source string must be provided as input to the reporter. This is also used to translate start and end offsets to line and column numbers.

4.6 Summary

In this chapter, we gave a detailed overview of the different processes and datasets involved in the complete analysis process. We motivated our choice to use an existing parser for the first step, explained how we arrived at an enriched AST by using several features of the M-Industries schema language and explained how we treat the two different types of instruments currently supported by the analysis process. After that, we shortly discussed the filtering step before we finished with some details on the reporting process.

During the project, our implementation of the tool became more than just a proof of concept. It is currently being used in production, as a pre-receive hook in the version control system of M-Industries. We do not yet have the same amount of instruments (‘rules’) as JSLint, but we will certainly be able to replace JSLint with our static analysis tool in the near future.

Chapter 5

Evaluation

A large number of studies use empirical data to gather statistics or to verify their methods. This is possible due to the fact that a huge amount of JavaScript code is publicly available on the web, as it is one of the building blocks of modern web applications.

Yue and Wang [34] automatically visited 6,805 homepages of popular websites with an instrumented version of the Firefox browser. The goal was to find out how much 'insecure' code was executed just by visiting the homepage of these sites. Guarnieri and Livshits [10] created GATEKEEPER, a tool for static analysis of a subset of the JavaScript language. They evaluated it by analyzing over 8,000 JavaScript widgets. To find security vulnerabilities using incremental analysis, Chugh et al. [4] collected JavaScript code from 100 popular websites, inlining all dynamically evaluated code using a custom Firefox extension. Guarnieri et al. [11] tested their system ACTARUS on 9,726 web pages, obtained by crawling more than 50 popular websites. Vulnerabilities were found in 11 of those sites. Richards et al. [28] performed an empirical study to find out whether the dynamic code evaluation features are actually used in real-world JavaScript code by gathering data from 100 popular websites. To do this, they added instrumentation code to the WebKit JavaScript engine. A year later, Richards et al. [29] updated the modified WebKit JavaScript engine from their previous research to also trace all strings that were dynamically evaluated at runtime. In total, 10,000 popular sites were tested. To record as much realistic website behavior as possible, a subset of their test sites was visited manually with tracing enabled, while the user performed common tasks such as logging in and clicking on buttons.

To evaluate our own analysis tool, we set up a similar experiment. The goal of this experiment was to run a specific set of analysis rules on a large collection of JavaScript code, gathered from different sources and including a good representation of both client-side and server-side code. Next to JavaScript code collected from public websites, we will also consider JavaScript code that is written to run in the Node.js runtime. Because Node.js is based on Google's V8 JavaScript engine, developers targeting Node.js can take advantage of modern JavaScript engine features, such as getters and setters in object literals and an increased number of library functions on built-in objects.

In this chapter, we will explain the process of gathering and analyzing the data and we will discuss the results of the experiment.

5.1 Gathering

To collect a large body of JavaScript code specifically written for the Node.js runtime, we decided to download the entire contents of the default package repository for Node.js modules: the central repository of the npm[1] package manager. This process is rather straightforward, as the whole system is open and set up to make it easy to distribute packages to end users. Gathering JavaScript code from websites takes more effort, but it can be done with publicly available tools, without having to resort to making modifications to JavaScript engines or browsers.

In this section, we will explain how we collected JavaScript code from these two sources, starting with the Node.js packages.

5.1.1 NPM

As already explained in section 3.3.3, Node.js has an implementation of the CommonJS module system that allows developers to design their system so that it can be composed of several modules, each with its own explicit public API. This module system is also being used to include third-party modules.

```json
{
  "name": "esprima",
  "description": "ECMAScript parsing infrastructure",
  "homepage": "http://esprima.org",
  "main": "esprima.js",
  "bin": {
    "esparse": "./bin/esparse.js"
  },
  "version": "0.9.9"
}
```

Listing 33: Simplified package metadata of the Esprima parser.

Both Node.js and npm understand the `package.json`[2] metadata format. Listing 33 gives an example of the `package.json` file of the Esprima parser package. The `main` property points to the `esprima.js` file, which in its turn exports a `parse` function as its public API.

In addition to understanding the metadata format, npm also provides a mechanism to install[3] a package and its dependencies from a remote source. The default remote source is the central online npm repository.

The central package list can be browsed through the website. It is possible to download a list of all packages in the repository, which is what we did on April 6, 2012. At that moment, the repository contained 8,594 packages. The list only contains package metadata. Using the metadata, we determined the url of the last published version of a package. We downloaded all packages, extracted the archives and then tried to determine the set of JavaScript files that would be loaded at runtime when

[1] `http://npmjs.org/`
[2] `http://npmjs.org/doc/json.html`
[3] `http://npmjs.org/doc/install.html`

Package state	Successes	Failures
In repository	8,594	-
Containing an archive URL	8,551	43
Successfully downloaded	8,525	26
Has .js file as entry point	7,302	1,223
Has $\geq$ 1 parsable JavaScript file	7,262	40

Table 5.1: Gathering & processing npm packages.

using that package. This excludes unit tests and other support files that are sometimes included in a package.

The process of obtaining these files was divided in a number of steps. Table 5.1 shows the amount of packages remaining after each successive step, including the number of packages that we had to skip during that step. Several packages did not include a URL, so we could not download them. We also encountered '404 Not Found' errors while downloading some of the packages. For every downloaded package, we determined the filename of the JavaScript file that would be loaded when evaluating a `require('pkgname')` expression. For some packages, we could not find a JavaScript file as an entry point[4]. This explains the 1,223 packages that were ignored in the penultimate step.

Using the algorithm described in listing 34, we then made a list of all dependencies inside a package. We could not determine this list for 40 packages, because we encountered an error while parsing the main JavaScript file of those packages.

In the end, we found 19,669 parsable JavaScript files in 7,262 unique npm packages, resulting in a mean number of 2.71 files per package. The total set of parsable files, amounting to 137 MB of JavaScript code, will be used for the analysis process as described in section 5.2.

5.1.2 Web

The web is an excellent place to gather JavaScript source code, mostly due to the vast amount of websites that use JavaScript. In Google's 'Web Authoring Statistics' from December 2005, they examined slightly over a billion web pages and found at least one `<script>` tag on roughly half of those pages[5]. However, the process to collect these scripts is less straightforward than the retrieval of packages from the npm package repository.

Listing 35 shows the two methods that can be used to embed and execute JavaScript code on a web page. On line 5, we see a `<script>` tag with a reference to an external file. On line 10, the `<script>` tag contains inline JavaScript code. References to external script files are usually used when including frameworks, libraries or other code that is shared between a large number of pages. Because the browser caches external resources, every subsequent request to a page that contains a reference to the

[4]Some npm packages are written in other languages or only run standalone.

[5]`https://developers.google.com/webmasters/state-of-the-web/2005/scripting`

```
function find_dependencies(file_or_directory) {
  dependencies = [];

  function internal_find(file_or_directory) {
    file_path = require.resolve(file_or_directory);

    if (!dependencies.contains(file_path)) {
      dependencies.push(file_path);
      ast = parse(file.load(file_path));

      for (require_call in find_require_calls(ast)) {
        if (is_pkg_file(require_call.argument)) {
          internal_find(require_call.argument);
        }
      }
    }
  }

  internal_find(file_or_directory);

  return dependencies;
}
```

Listing 34: Pseudocode of the package dependency resolution algorithm.

same external script can be served from the cache. Inline scripts, on the other hand, are usually smaller and specific for one single page.

```
<html>
  <head>
    <title>Test</title>
    <script type="text/javascript" src="/external.js"></script>
  </head>
  <body>
    <script type="text/javascript">
      window.alert('Hello World.');
    </script>
  </body>
</html>
```

Listing 35: External and inline JavaScript in HTML.

The first challenge was to determine a set of websites that we could visit to collect JavaScript code from the `<script>` tags. Based on earlier research [28, 34, 11, 4] that we mentioned in the introduction of this chapter, we decided to use the Alexa top sites list[6]. This list is updated daily and contains one million website URLs, ranked by popularity using a combination of average daily visitors and pageviews over the past month[7]. We downloaded this top sites list on April 4, 2012.

[6] `http://www.alexa.com/topsites`
[7] `http://www.alexa.com/help/traffic-learn-more`

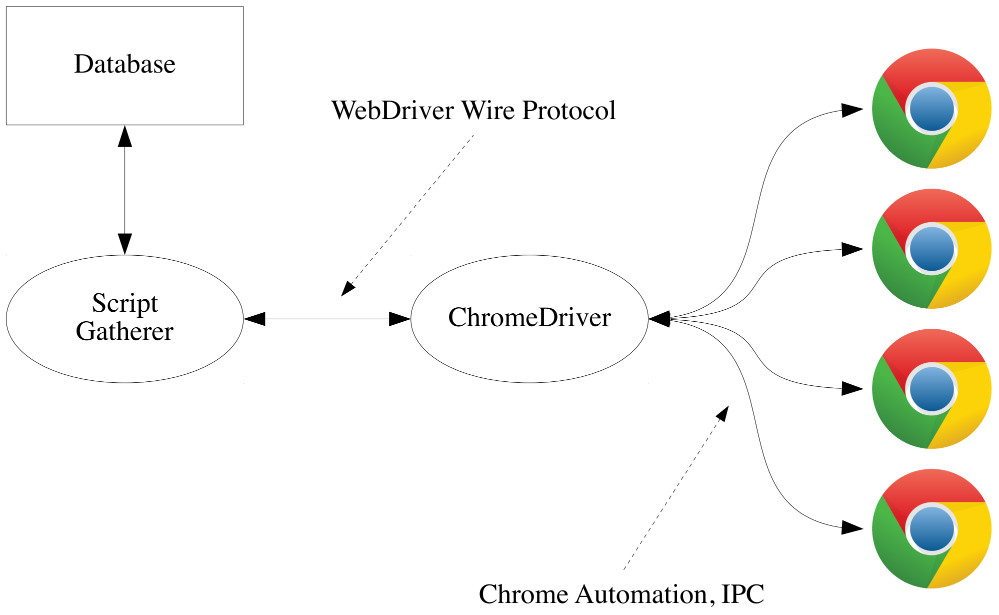


Figure 5.1: Script gathering using ChromeDriver and Chrome.

Having a list of website URLs, we needed a mechanism to automatically visit a large number of websites and collect all JavaScript code used on those sites. We started looking for a scriptable browser that we could instruct to visit a site and collect all code embedded in or referenced from `<script>` tags. We examined three options:

- **PhantomJS**[8]. The first option we tried was PhantomJS, a headless WebKit browser that can be controlled using JavaScript. Unfortunately, version 1.5.0 of PhantomJS has problems with web pages and scripts in encodings other than UTF-8. This was a problem for a large number of Chinese sites in our list.

- **Firefox through Selenium**[9]. Our second option was using the Selenium browser automation system in combination with the Firefox browser. This combination could handle non-UTF-8 encoded pages correctly, but had issues processing the contents of script blocks containing strings with new line characters.

- **ChromeDriver**[10]. Our third and final option was to directly use the ChromeDriver, originally created to be used through Selenium. ChromeDriver uses the Chrome automation API to control a locally running Chrome browser. It can be controlled using the WebDriver Wire protocol[11]. The combination of ChromeDriver and Chrome has no issues with non-UTF-8 encoded pages nor with script blocks containing strings with new line characters.

[8] `http://phantomjs.org/`
[9] `http://seleniumhq.org/`
[10] `http://code.google.com/p/chromedriver/`
[11] `http://code.google.com/p/selenium/wiki/JsonWireProtocol`

Figure 5.1 shows the final setup[12] used to retrieve scripts from websites. Our gathering process was set up to visit 4 websites in parallel, which is why the figure shows four separate Chrome processes.

To keep the script gathering simple, we limited ourselves to only visiting the homepages of the sites, similar to what Yue and Wang [34] did in their research. If the focus had been on the runtime behavior of JavaScript code, we would probably have chosen another approach like simulating user interaction and visiting multiple pages from the same site.

```
sites = db.load("sites")

for (site in sites) {
  chrome.load(site.url);

  scripts = chrome.execute(function () {
    return document.scripts;
  });

  for (script in scripts) {
    if (script.src) {
      result = http.get(script.src);
      db.save(script.attrs, result);
    } else {
      db.save(script.attrs, script.innerText);
    }
  }
}
```

Listing 36: Simplified pseudocode of the gathering algorithm without parallelization.

Listing 36 describes the process used to gather the script data for every site. We saved all tag attributes, regardless of script type. External scripts were downloaded outside the Chrome process[13]. For every external script, we saved the HTTP response headers and the HTTP status code.

Our target was to have at least 10,000 websites on which we could successfully collect all scripts from the homepage. We started the gathering process with a list of 12,000 websites, because earlier tests on a small subset revealed that the gathering process sometimes failed due to timeouts or HTTP errors.

[12]We used ChromeDriver 19.0.1068 and Chrome 18.0.1025 on Windows 2008 R2 SP1.

[13]Requests were made with the correct User-Agent and Referer headers, as if they originated from the browser process.

Script type	Successes	Failures
All inline scripts	126,692	-
Filtered on ‘type’ attribute	125,593	1,099
Parsable	125,376	217

Table 5.2: Gathering & processing inline script blocks.

Script type	Successes	Failures
All external scripts	125,909	-
HTTP OK & proper Content-Type	123,732	2,177
Parsable	123,319	413

Table 5.3: Gathering & processing external script files.

On April 24, 2012, we collected scripts spread over 125,909 external script files and 126,692 inline script blocks on 10,505 websites. The gathering process took almost 10 hours to complete on an Amazon EC2 Medium instance[14]. After downloading, we filtered the scripts according to these criteria:

- Inline scripts must match:
  - The tag does not include a `type` attribute **OR**
  - The `type` attribute of the tag is set to `text/javascript`.
- External scripts must match:
  - The HTTP status code is 200 OK **AND**
  - The Content-Type HTTP response header is one of:
    - application/x-javascript
    - text/javascript
    - application/javascript
    - text/html
    - text/plain

Ideally, the last two Content-Types should not have been included, but we found a number of cases[15] in which JavaScript files were served as either ‘text/html’ or ‘text/plain’.

We then normalized the text by converting the source text of both inline and external scripts to the UTF-8 encoding. After that, we stripped all HTML comments from the source text. The resulting source text was parsed and if this did not result in any syntax errors, we marked the script as parsable and included it in the final set, ready for the analysis step. After these post-processing and filtering steps, we were left with 3,554 MB of JavaScript code gathered from the web. Table 5.2 shows the filtering results for inline scripts, table 5.3 shows the filtering results for external scripts.

[14]`http://aws.amazon.com/ec2/#instance`

[15]In the final set, 4,075 scripts were served as ‘text/html’, 490 scripts as ‘text/plain’.

	Source		
Category	NPM	External	Inline
Set A	19,669	123,319	125,376
Set B	19,536	5,815	8,444

Table 5.4: File analysis sets.

5.2 Analysis

For the analysis process, we used the SQL reporter as described in section 4.5.1. To produce our results, we used a combination of rules and custom queries on the ASTs of all files that were part of the analysis set. All simple metrics (computed without our static analysis tool) were calculated for the full set of files, which we will refer to as 'Set A' from now on. For the results obtained using our static analysis tool, we used a smaller set of files ('Set B'), due to time constraints & performance reasons. Table 5.4 shows the amount of files per source type in each set.

The analysis process for one file is single-threaded and CPU bound. Because Node.js has no built-in support for threading, we split the analysis program in two parts:

1. A slave process that performs the actual analysis. Input is a string of JavaScript source code and a ruleset, output is a collection of metric values.
2. A master process that reads all script filenames from the database, puts them in a dispatch queue with a concurrency of four tasks and starts slave processes accordingly. Upon completion of a slave process, analysis results are saved in the database.

With this setup, we reached an average CPU utilization rate of 92% for the duration of the analysis. To give an indication of the performance: the analysis process for the 'unused declarations' rule took 5 hours and 45 minutes of CPU time[16] to complete for 33,795 files on an Intel Core i7-2677M processor, which offers four virtual cores[17]. For every file in the analysis set, there are some fixed costs in parsing the source code and transforming it to our enhanced AST. Checking more rules does not increase the amount of used CPU time linearly, until the rule set gets very large. In scenarios where a large set of files must be analyzed independently, it is preferable to run our analysis tool on a multicore system. If it turns out that there is more demand for single-file, multi-rule analysis, we can consider parallelization of the analysis engine itself, so that each rule is checked in its own process.

[16]Total of 'user' and 'sys' times as reported by 'time', divided by the number of virtual cores.

[17]The processor has two physical cores, but Intel's Hyper-Threading Technology enables multiple threads to run simultaneously on a single core.

5.2.1 Ruleset

We used two different analysis mechanisms. The first one used six custom queries that were directly executed on the AST from the parser without transforming it to our enhanced AST. This was done mostly for performance reasons. We obtained these metrics:

- Total token length (everything except whitespace and comments).
- Total comment text length.
- Number of block comments containing a JSLint directive.
- Number of block comments containing a JSHint directive.
- Number of block comments containing JSDoc-style documentation.
- Number of line comments containing VSDoc-style documentation.

The rest of the analysis results were obtained using our static analysis tool, with this ruleset:

- Identifier name length, with three separate types:
  - Function (both declaration and expression).
  - Variable (as part of a VariableDeclaration).
  - Function parameter.
- Function body length (statement count).
- Function parameter list length.
- CallExpression argument list length.
- Number of boolean literal arguments in CallExpressions.
- Number of unreachable statements.
- Number of unused labels.
- Number of unused declarations (variables, functions and function parameters).
- '`with`' statement usage.
- '`eval`' usage.
- Cyclomatic complexity (per function).

```
testConditional1(condition,
  function isTrue() {
    // handle true case
  }
);

testConditional2(condition,
  function isTrue() {
    // handle true case
  }, function isFalse() {
    // handle false case
  }
);
```

Listing 37: Use of function expressions in JavaScript.

Cyclomatic Complexity

The cyclomatic complexity deserves a special mention, as it was a rule where we had to make a fundamental choice in our method of measuring it. Consider the fragment of JavaScript code shown in listing 37.

Different branch paths increase the cyclomatic complexity of a function, but it is not always clear if a JavaScript function expression (which becomes a closure when evaluated at runtime) means that there is a branch point. Ideally, you would want to know if the function executes always, sometimes or never. In the example from the listing, we can gather from the context that the testConditional1 block will increase the cyclomatic complexity by one, because the isTrue callback will be executed conditionally. However, the testConditional2 block will *also* increase the complexity by one, because we expect only one callback function to execute.

Kent Beck also wondered how to measure the cyclomatic complexity in languages like JavaScript, as demonstrated by his tweet

> "anyone know of a good reference for measuring cyclomatic complexity in a language with closures?"

which he posted on November 10, 2009[18]. We decided to go for the worst-case scenario by counting all function expressions as complexity-increasing branch points. An improved analysis algorithm might use knowledge about the underlying library or framework to give a more accurate number.

Dynamic Code Evaluation

Using another rule, we measured the usage of the '`eval`' function[19] with a string literal as an argument. Our point here is that there should be no reason to use *dynamic* code evaluation if you supply a *constant string* to be evaluated. These instances can probably easily be rewritten so that they do not rely on '`eval`' anymore.

[18]`https://twitter.com/kentbeck/status/5590593549`

[19]eval 'synonyms' such as 'setTimeout', 'setInterval', 'execScript' and 'new Function' were also included.

We should note that this is a very primitive method of detecting dynamic code evaluation. More advanced flow-sensitive algorithms that use some sort of type inference can achieve much better results here. If this would have been a main goal in our research, we would probably have used dynamic analysis techniques to be sure that we detected all dynamic code evaluation calls.

Unused Declarations

For the unused declaration detection, we did not include unused global variables and the parameter of the catch-clause in a try-catch statement. Global variables might be used for communication between script blocks and the catch parameter is always required to be present, so it makes less sense to include these in the unused declaration metric.

5.3 Results

This section presents the analysis results on a global level. We try to find high-level differences in order to characterize the code from our three script source types. The included graphs were produced using R[20].

5.3.1 Direct Metrics

The first set of results were obtained from rules that directly measure comparable properties of files, statements or expressions. We will inspect the distributions of those values among the three different script type sources, in order to determine if there are significant differences between them.

Code size

Figure 5.2 shows the distribution of total token length in the three different sources. As expected, inline script blocks and external scripts are on opposite sides of the spectrum. Inline scripts are often generated dynamically or are only used to invoke functions from a library defined in an external file. External scripts are used for reusable libraries or are built by concatenating multiple script files in order to minimize the number of HTTP requests while loading a webpage. The code size in the npm packages is somewhere in between those two types of scripts.

Comment/token ratio

In figure 5.3, we see the distribution of the ratio:

$$total\_comment\_text\_length/total\_token\_length$$

A ratio of 2.0 means that there is twice as much comment text as there are tokens. When we compare the three different sources, there is a clear difference between the npm modules and the scripts we found on the web (both inline and external). The

[20]`http://www.r-project.org/`

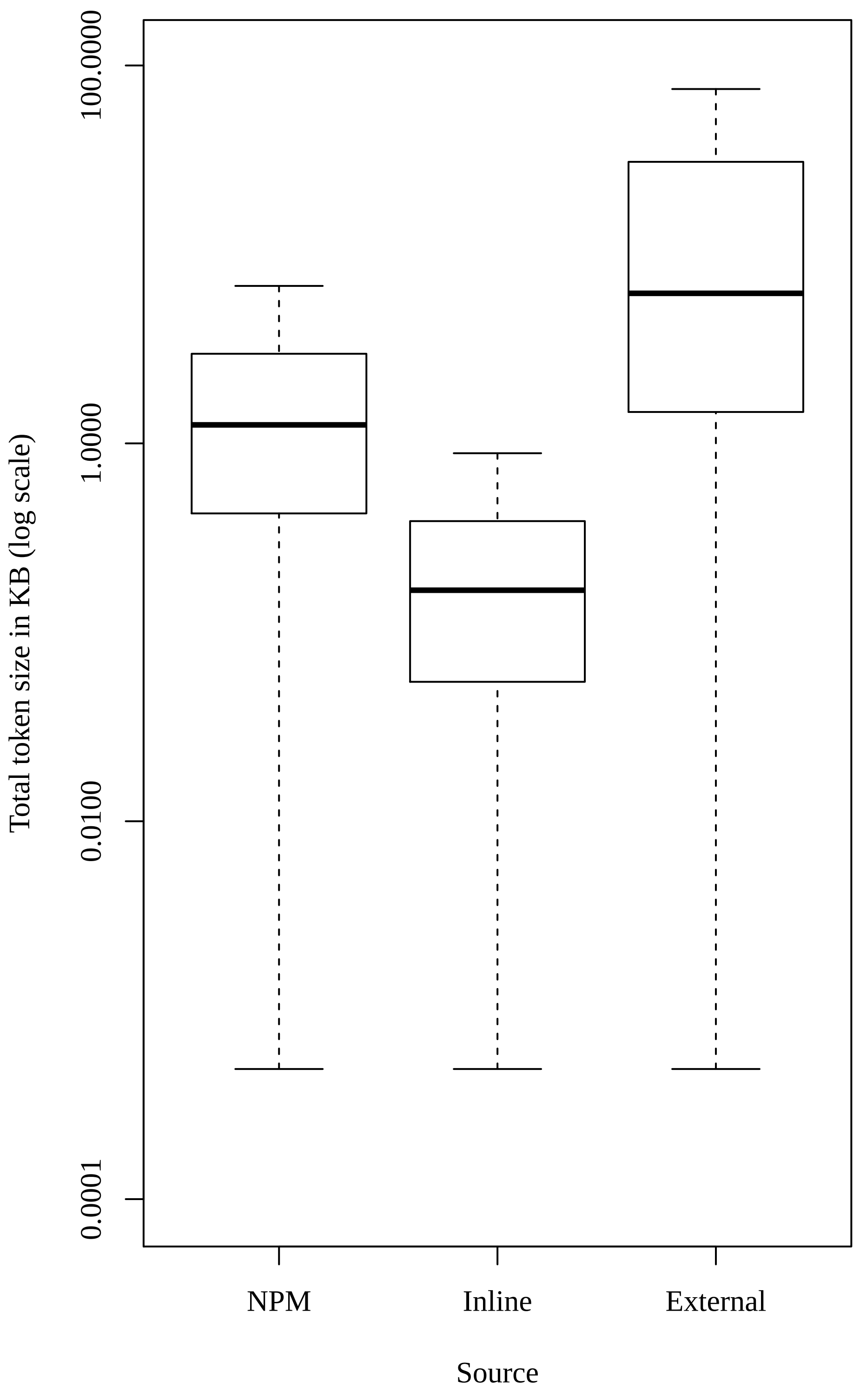


Figure 5.2: Total token length (set A).

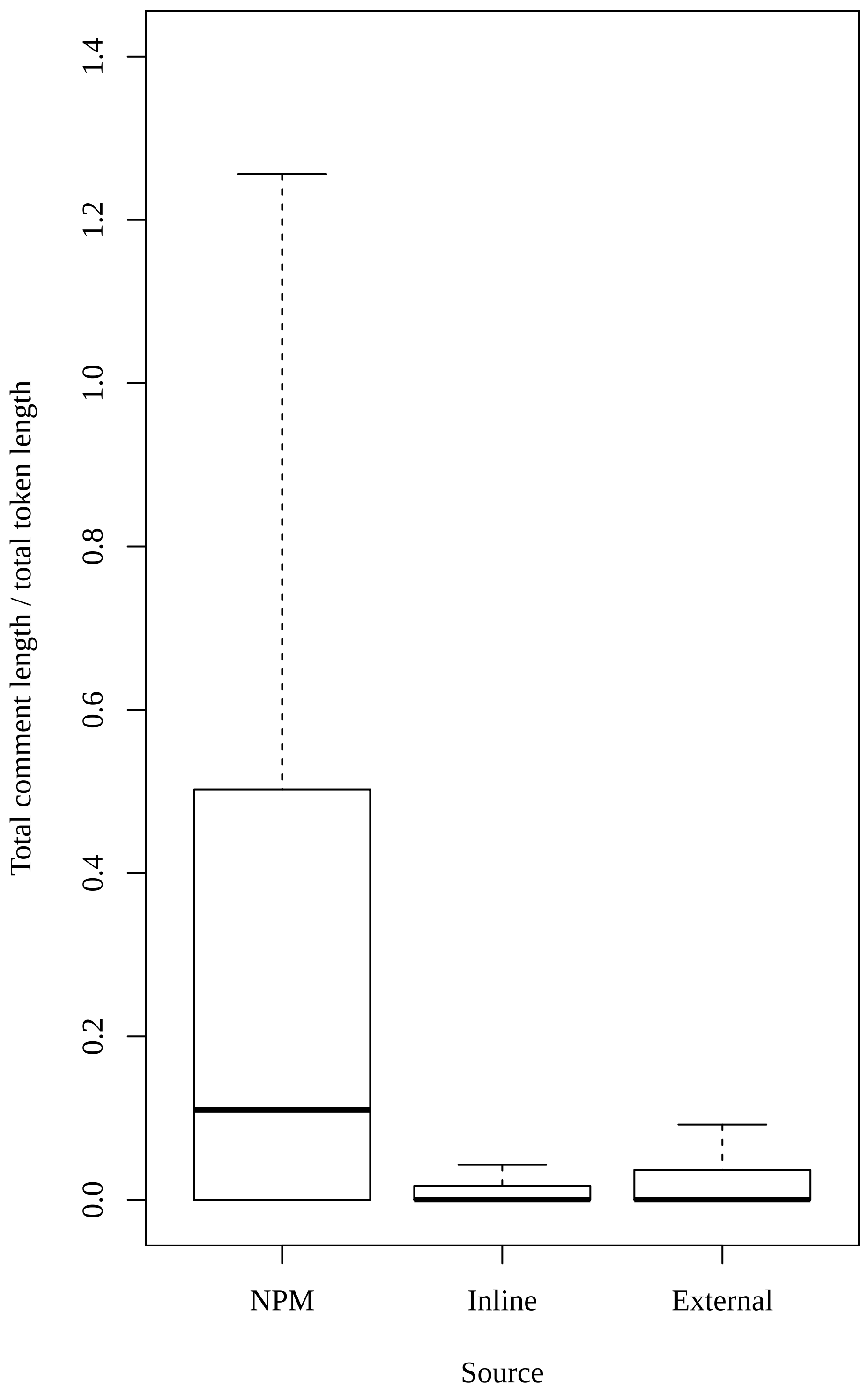


Figure 5.3: Comment / token ratio (set A).

npm modules usually have a reasonable amount of documentation and 11.5% of all the files even have more comment text than token text. This does, of course, not give any insight in the *quality* of the comments itself. However, we will regard the presence of comments as having a positive effect on code quality.

Identifier Length

We measured the identifier lengths from three different types of identifiers:

- Variable.
- Function declaration & function expression.
- Function parameter.

Figure 5.4 shows the distribution of these different types of identifiers among the three script source types. The npm identifiers show that variables and function parameters follow roughly the same distribution, while function names are substantially longer. These same patterns show up in the external and inline web scripts, although we have to note that the differences are much more extreme: parameter names are very short and variable names are a little bit longer, but still much shorter than those in the npm packages. However, function names are overall much longer on the web, certainly in the inline script blocks.

Function Body Length & Cyclomatic Complexity

The function body length metric turned out to be not very useful, as larger JavaScript programs are commonly organized by wrapping all code in one large function that is being invoked immediately, as a mechanism to prevent pollution of the global scope. This also happens on a lower level, to organize a program in different logical sections. Manual inspection of several files with functions having large function body lengths confirmed these assumptions. Section 5.4 contains examples of outliers.

Our definition of the cyclomatic complexity metric recursively visits all statements and expressions contained in a function. Because JavaScript programs are often composed of a structure of nested function definitions, we ended up with huge cyclomatic complexity values, which makes it hard to see the *real* complex functions. We did not anticipate this, but we suggest to adapt the metric so that it does not recursively visit all contained functions. Our idea to make function expressions increase the complexity is still valid and not affected by this change. We did not have the time to include the revised measurements in our report.

Function Parameters And Arguments

Figure 5.5 shows the distribution of the number of arguments in a function call expression. Because JavaScript allows variable argument function calls, this does not necessarily mean that all these arguments must be accessed through a named function parameter. The built-in '`arguments`' identifier provides all function arguments in an array to be able to deal with large numbers of arguments. This seems to match the values seen in figure 5.6, although there are some interesting outliers with around 30 or 50 function parameters.

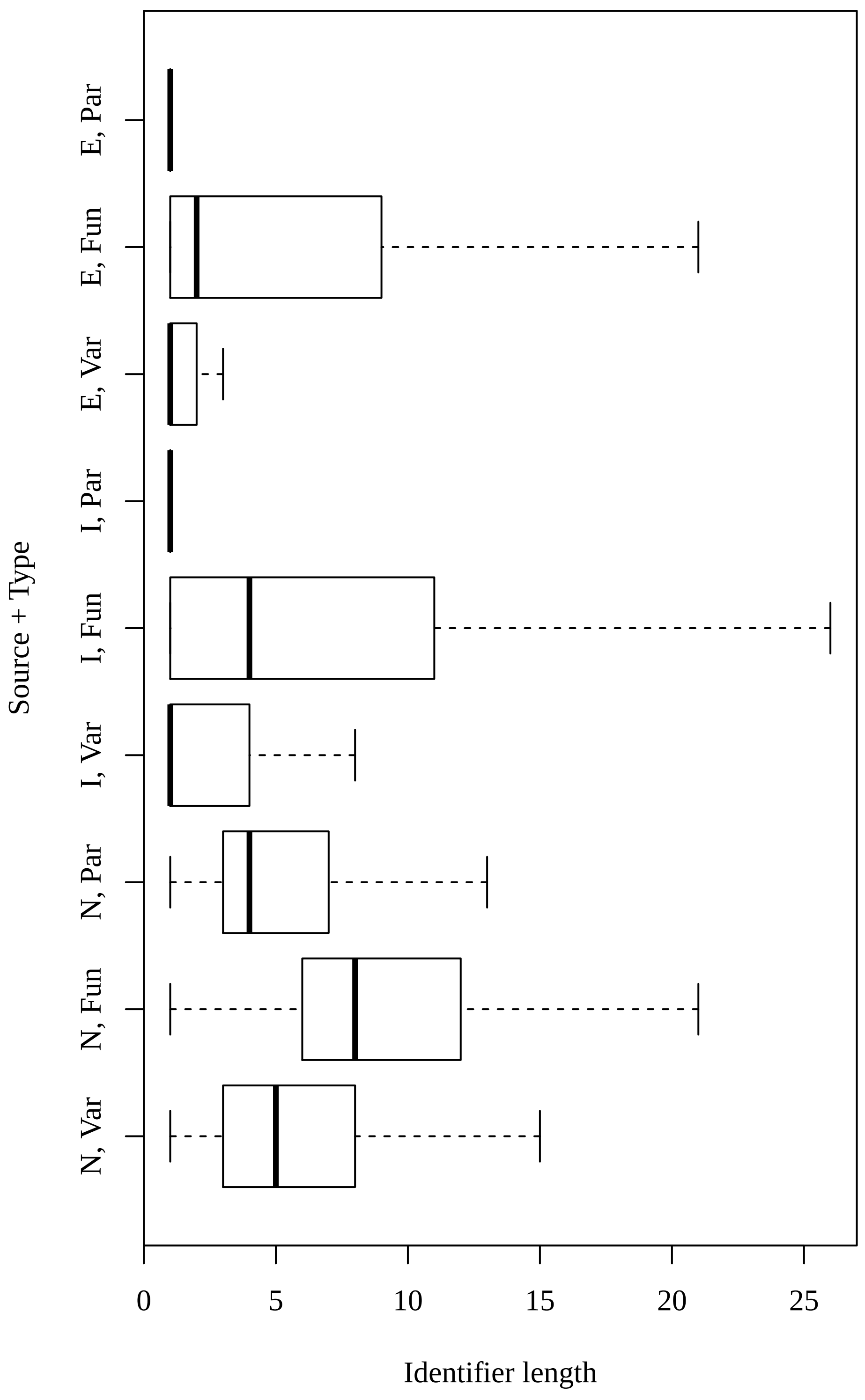


Figure 5.4: Identifier length (set B).

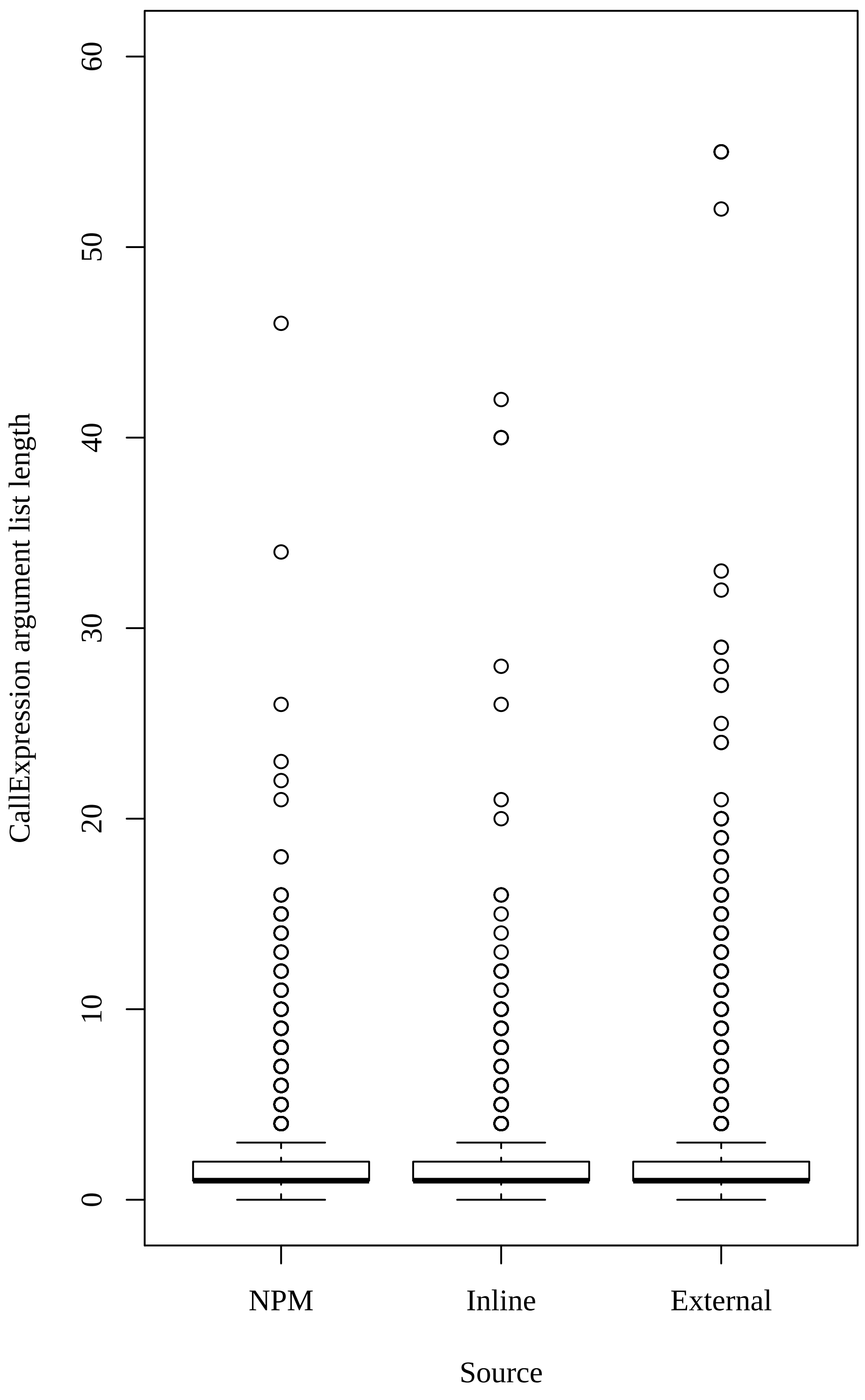


Figure 5.5: Call argument list length (set B).

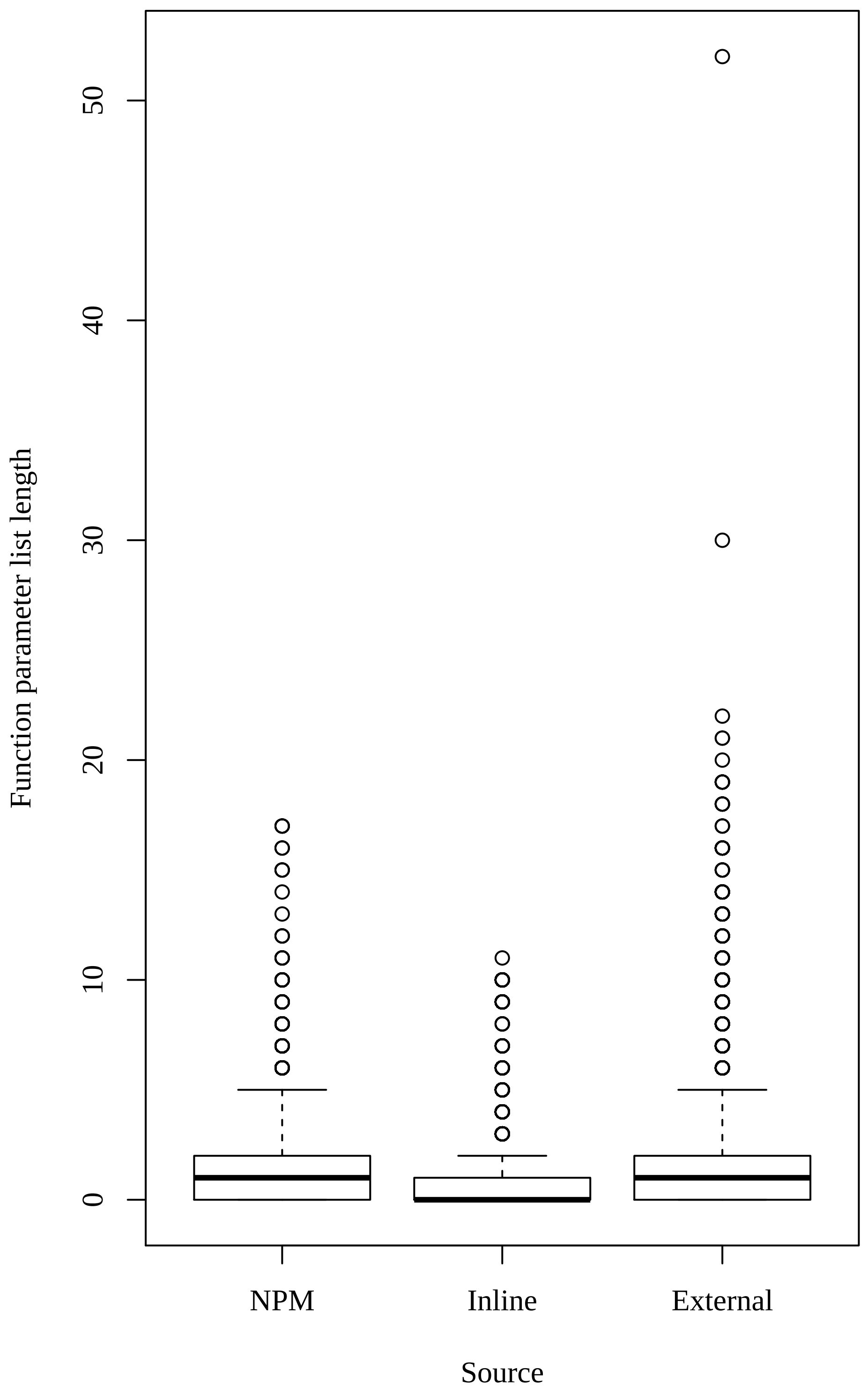


Figure 5.6: Function parameter list length (set B).

	Source		
Metric	NPM	External	Inline
Unused labels	32	6	1
Unreachable statements	986	343	38
Unused declarations	39,805	41,932	2,160
‘with’ statements	59	187	14
‘eval’ usage	10	740	145
‘new Function’ usage	103	2058	2

Table 5.5: Frequency metrics result overview (set B).

5.3.2 Frequency Count Metrics

Table 5.5 lists all results from metrics that count instances of certain phenomena in the different script source types. We present these results in a table to show the exact amount of occurrences, and because showing the value distributions makes no sense for these kind of metrics. Contrary to the advice from most coding guidelines, the dynamic code evaluation features that impact static analysis tools are still frequently used. It is hard to extract any general patterns from the other metric values, but we can see that our analysis mechanism can be used as a good starting point for code cleanup or refactoring.

The other results in frequency count metrics are discussed separately, in the remainder of this subsection.

Special Comments

In table 5.6, we see the relative and absolute number of files that contain at least one comment that can be classified as:

- A special directive for JSLint and/or JSHint.
- A documentation comment, as discussed in section 3.3.5.

The idea is that these special comment types indicate that the developer was at least aware of code quality analysis tools or methods to provide structured API documentation. We observe that inline script blocks contain no significant amount of any of these types of comments whatsoever. Relatively, the npm files show the most awareness of code quality tools by including directives for JSLint or JSHint and by having a significant amount of files containing JSDoc-style documentation comments.

		Source		
Category	Metric	NPM	External	Inline
Directives	JSLint	1.0% (204)	0.3% (325)	0.0% (1)
	JSHint	0.4% (88)	0.0% (38)	0.0% (0)
Documentation	JSDoc	17.0% (3340)	4.7% (5840)	0.5% (594)
	VSDoc	0.0% (3)	0.2% (260)	0.0% (2)

Table 5.6: Comments: directives & documentation (set A).

	Source		
Booleans	NPM	External	Inline
2	320	476	140
3	29	64	7
4	2	4	1
5	0	0	0
6	4	16	1

Table 5.7: Number of boolean literal arguments (set B).

Boolean Literals as Function Arguments

Calls to functions that take a high number of boolean parameters tend to get unreadable rather quick. After analysis of set B, we found several cases of function calls with 6 boolean literals. Table 5.7 shows the results. We will inspect some of those cases in section 5.4.

5.4 Outliers: Closer Inspection

In this section, we show examples of code that we found using our analysis tool.

5.4.1 Long Function

Listing 38 shows part of a function with over 250 assignment statements. This can better be written as a single assignment statement with an object literal at the right hand side. Object literals provide concrete syntax for simple dictionaries like the one constructed here.

5.4.2 Boolean Literals as Function Arguments

Listing 39 gives two examples (from different files) of function calls with a high number of boolean literal values, which makes for less readable code. If you have no IDE support, it is easy to make a mistake here. This example also demonstrates that it is not always possible for developers to ‘fix’ these kind of issues. The `initMouseEvent` function, which takes six boolean parameters, some objects and some numbers, is part

```
Country["Andorra"] = "AD";
Country["United Arab Emirates"] = "AE";
Country["Afghanistan"] = "AF";
Country["Antigua and Barbuda"] = "AG";
Country["Anguilla"] = "AI";
Country["Albania"] = "AL";
// ...
```

Listing 38: Long function.

of the DOM standard. The `eatExpressions` function however, is a user-created function. The readability of the code might be improved if this function is split in a set of differently named functions for some frequently used combinations of the boolean arguments. Another option is the use of object literals to pass a set of explicitly *named* options as arguments, which improves the understandability of the code.

```
a.initMouseEvent("click",true,true,_,0,0,0,0,0,
                 false,false,false,false,0,this);

match = this.eatExpressions(false, match, stack,
                            true, false, true);
```

Listing 39: Boolean literals in function call.

5.4.3 Function Parameter List Length

In listing 40, we see two randomly picked examples of function definitions with unusual long parameter lists. These might be refactored by using object literals to simulate passing named parameters, or by putting the functions inside an object that shares some state between different function calls, removing the need to pass all arguments explicitly every time a function call is made.

```
function cmCreatePageviewTag(_PID,_TN,_SSTR,
  _CID,_H,_ER,_ERM,_T,_F,_ADL,_APPI,_ADB,
  _SRES,_SESSID,_APNM,_APSNUM,_APSNM,_ATR,
  _SBCPM,_RURL,_DURL) { /* ... */ }

TropoWebAPI.prototype.record = function(attempts,
  bargein, beep, choices, format, maxSilence,
  maxTime, method, minConfidence, name, required,
  say, timeout, transcription, url, password,
  username) {  /* ... */ };
```

Listing 40: Function parameter list length.

5.4.4 Function Call Argument List Length

In listing 41, we see an example of a function call that contained 42 arguments. Obviously, the intent was to provide a function that looked like it supported named parameters. However, if that is the real reason behind this design, it would be better to have the function take an object literal with key-value pairs for all named options.

```
AC_FL_RunContent(
  'width', '980',
  'height', '260',
  'src', 'banner',
  'quality', 'high',
  'pluginspage', 'http://www.adobe.com/go/getflashplayer',
  'align', 'middle',
  // ...
);
```

Listing 41: Function call argument list length.

5.4.5 Unreachable Statements

A random selection of found unreachable statements gave results like in listing 42. The `break` after a `return` is unreachable and can be removed.

```
switch(unit) {
  case 'm':    // Meter
    return geolib.round(distance, round);
    break;
  case 'km':    // Kilometer
    return geolib.round(distance / 1000, round);
    break;
  // ...
```

Listing 42: Unreachable statements.

5.4.6 Unused Declarations

We chose an example from a file that had only one unused declaration, because in this case, it seemed more likely that the author did know how to write compact & correct code, but forgot to remove a declaration, for example while refactoring the code. In listing 43, the `dt` function parameter is never used inside the function. This means it can probably be removed safely, unless it is part of a public API that needs to remain stable.

```
Animation.prototype.work_queue =
  function(started, dt, executiontime) {
    var t, _base, _results;
    t = now();
    _results = [];
    while (this.queue.length && t - started < executiontime) {
      if (typeof (_base = this.queue.shift()) === "function")
        _base();
      _results.push(t = now());
    }
    return _results;
  };
```

Listing 43: Unused declaration.

5.5 Threats to Validity

There are several reasons why our analysis results and statistics could have given the wrong impression about JavaScript code quality on the web and in npm packages:

- Code on the web is often optimized during deployment, which means: short identifier names and no comments or whitespace. This heavily influences the metrics that deal with comment/source ratio or identifier length, making code from the web look worse than code from npm packages.
- Duplicate code cause extra skewness in the distribution of metric values. In our collection of web scripts, we found 12,654 duplicate files. Our collection of npm files contains 522 duplicate files.
- Files larger than 100 KB were excluded from the analysis, due to performance reasons and time constraints. Because of their size, these files are potentially interesting for further analysis. They might, for example, show patterns that were not present in smaller files.
- Not all JavaScript code is written by humans. Some of it is machine-generated. Languages and systems like CoffeeScript, Google Web Toolkit and Google Dart compile to JavaScript and do not have readability of the generated source as a design goal. This code should not be judged based on code readability metrics.
- For all websites, we only visited the homepage and we did not interact with the browser. This means that a lot of potential JavaScript source code was never loaded nor executed.

This is the reason we have not made any bold statements about differences in code quality between npm packages and JavaScript code gathered from the web. Instead, we showed how our analysis tool can be used to find extreme values of certain metrics, which, in their turn, can point to candidates for refactoring.

5.6 Summary

Based on the evaluation methods used in comparable research, we set up our own experiment in which we gathered code from the web and from a repository of open source JavaScript packages. We gave a detailed description of our own process, including the gathering and post-processing of JavaScript code from both sources. After that, we calculated a number of different metrics for all the collected files, in order to get more insights in the code quality of publicly available JavaScript code.

After evaluation of the analysis results, we found it very hard to make any kind of general statements regarding differences in code quality between the code from the open source packages and code we gathered from the web. For our project, it turns out that the value of performing the analysis on larger codebases lies primarily in detecting outliers and potential errors. Based on the analysis results, the locations with extreme metric values can be subjected to further manual inspection.

Chapter 6

Conclusions and Future Work

In this chapter, we list our contributions, we give answers to the original research questions and finally, we look ahead by providing concrete examples of possibilities for future research and enhancements to our analysis tool.

6.1 Contributions

If we look at our background research, the design and implementation of our static analysis tool and our empirical research on real-world JavaScript code quality, we can make this list of contributions and achievements:

- For our background research, we provided a detailed overview of the challenges in static analysis of JavaScript code.
- Also as a part of our background research, we evaluated and compared several existing analysis tools for JavaScript code.
- There is no complete, official overview of all rules in the popular JSLint tool and unit tests are also lacking. We made an inventarisation of the rules and rule types in JSLint, including a large number of unit tests that each trigger a specific JSLint rule.
- We created a model of the JavaScript language including an enhanced AST that supports and simplifies building analysis tools for JavaScript code. This model was validated against the ECMAScript language specification and it is backed up by extensive tests on synthetic and real-world JavaScript code.
- We designed and implemented a modular, extensible static analysis tool. All individual components transform one immutable, typed dataset to another immutable, typed dataset. The complete internal chain can be described as a series of transformations on typed datasets, resulting in low coupling and high cohesion.
- Before the end of the project, M-Industries successfully started using our analysis tool to continuously validate their code against their own ruleset. We provided editor integration for our tool as well as a Git pre-receive hook for the version control system.

- For the evaluation of our analysis tool, we gave a detailed description of the process and our results, with data on real-world JavaScript code quality for both client- and server-side JavaScript code.
- We showed how our analysis tool can be used successfully to analyze a large codebase and find potential bugs, dead code, inefficient code and code that can benefit from refactoring.

6.2 Conclusions

To formulate our conclusions, we will repeat the research questions from section 1.2.1:

RQ1 What is the state of code quality analysis tools for JavaScript code?

RQ2 What is the effect of using a typed abstract syntax tree model on writing analysis code?

RQ3 Is there a significant difference in code quality between server-side and client-side JavaScript code?

We answered **RQ1** in chapters 2 and 3, where we gave a description of the JavaScript language and where we evaluated existing research and tools in the area of code quality analysis. There is relatively little research in this area, and the amount of tools is small compared to analysis tools for Java code. The existing tools are lacking in sophistication. However, if the JavaScript language is restricted by removing some of the dynamic features, it becomes feasible to write advanced static analysis tools.

In chapter 4, where we discussed the implications of our JavaScript language model for static analysis tools, we answered **RQ2**. By explicitly forcing the analysis code to deal with all possible states for elements that have been modeled as a state group, we decreased the chance of making errors in the analysis code at the cost of a slightly more verbose rule instrument syntax. We also greatly simplified the way in which analysis tools can work with declarations and references to declarations, at the cost of one extra transformation from the 'plain' AST to our enhanced AST. Our model enables people to write transformations on the AST while providing the guarantee that the output will be valid JavaScript.

The evaluation of real-world JavaScript code as discussed in chapter 5 provided an answer to **RQ3**, although it turned out that there is no simple yes-or-no answer to this question. The inherent differences in code that has been published to the npm package repository and code that is being used on the web causes several metrics and characteristics to show unsurprising values and results. It could be that our analysis methods were not exhaustive enough to get good insighs in code quality differences, or that these code quality differences might not be so extreme as we initially assumed –or maybe hoped– them to be. However, in doing the work necessary to answer this last research question, we showed that our analysis tool with its current ruleset can be used to perform code analysis on large codebases, uncovering several types of defects and code smells.

6.3 Future Work

There are enough possibilities for more research, improvements to the analysis tool and new applications that can be built on top of our language model and our analysis engine:

AST Transformations. Our typed AST model and our analysis engine can be used to guarantee correctness and prevent errors while making changes to the AST. We can think of several useful applications of source-to-source transformations:

- **Refactoring Tools**. Based on our language model, we can provide tooling that is similar to the idea of 'Semantics-Based Code Search' as described by Reiss [27] and as implemented in IntelliJ IDEA[1] and Semantic Designs' Source Code Search Engine[2].
- **Instrumentation**. With AST transformations and serialization to JavaScript code, we have the ability to generate instrumented JavaScript code. As an example, it is relatively simple to construct a tool that analyzes function coverage by counting runtime function entrances.
- **Minification & Optimization**. The current version of our tool discards all comments and whitespace while serializing the AST back to JavaScript code. Similar to existing tools for code obfuscation (name mangling) and optimization (dead code elimination, function inlining), it is also possible to write a number of optimization rules in the form of AST transformations. Renaming declarations (e.g. variables, functions) is very cheap because they have been modeled as references.

IDE Integration. Because our tool is completely written in JavaScript, it is an ideal candidate for integration in a browser-based development environment, such as Eclipse Orion[3], Cloud9[4] or Adobe's Brackets[5].

Severity Levels. For the reporting, we currently take the metric output value of an instrument and filter it according to a certain threshold. This could be extended to support categorizing the results in a number of predefined severity levels like 'warning', 'error', 'critical' etc.

Indentation Checks. JSLint can process a file while it keeps track of the actual and expected indentation levels, with a configurable number of spaces for indentation. Our current semantic and syntactic rules are not suited to perform such checks.

Multi-file Analysis. The current version of the analysis engine processes one file at a time. We would like to investigate if we can implement multi-file analysis on top of

[1] `http://www.jetbrains.com/idea/documentation/ssr.html`
[2] `http://www.semdesigns.com/Products/SearchEngine/`
[3] `http://wiki.eclipse.org/Orion`
[4] `http://c9.io/`
[5] `http://brackets.io/`

our current engine. The CommonJS Module specification and its implementation in Node.js makes it possible to get more information on dependencies between different modules in a project. For this to work, we first need to reduce the memory usage of the datasets in the analysis engine, as this currently causes trouble when we work with models for over 200 KB of code.

Language Support. We already mentioned that, in time, M-Industries wants to replace JavaScript with several custom DSLs for parts of the application development environment. Ideally, it should be possible to run the static analysis tool on DSL code as well. This requires support of multiple language models and possibly also other traversal and analysis mechanisms. The modular and extensible design of our tool should be able to support this.

Tree Traversal. How do our tree traversal methods compare to other mechanisms, such as in *strategic programming* systems like Rascal[6] and Stratego/XT[7]? Is it feasible to have a fully declarative language for the definition of our rule instruments?

Historical Analysis. Of the 8,594 npm packages we found in the central npm repository, at least 6,616[8] included the URL of a publicly available Git repository. When used in combination with static analysis tools, this can be used to do research in the area of software evolution.

Web Application Architecture. Use static analysis to determine which parts of an application must run server-side and which parts can run client-side. Use this to automatically distribute application code between client and server.

[6]`http://www.rascal-mpl.org/`

[7]`http://strategoxt.org/`

[8]Found by looking for ‘github’, ‘bitbucket’ or ‘gitorious’ in the repository URL.

Bibliography

[1] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. "Clone Detection Using Abstract Syntax Trees". In: *Proceedings of the International Conference on Software Maintenance*. ICSM '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 368–. ISBN: 0-8186-8779-7.

[2] Claudio Caldato and Adalberto Foresti. *Prototypes of JavaScript Globalization & Math, String, and Number extensions*. `http://blogs.msdn.com/b/interoperability/archive/2011/11/21/10239281.aspx`.

[3] F. Calefato, F. Lanubile, and T. Mallardo. "Function clone detection in web applications: A semiautomated approach". In: *Journal of Web Engineering* 3 (2004), pp. 3–21.

[4] Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. "Staged information flow for javascript". In: *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*. PLDI '09. Dublin, Ireland: ACM, 2009, pp. 50–62. ISBN: 978-1-60558-392-1. DOI: `http://doi.acm.org/10.1145/1542476.1542483`.

[5] Douglas Crockford. *Crockford on JavaScript – Volume 1: The Early Years*. `http://yuilibrary.com/theater/douglas-crockford/crockonjs-1/`. 2010.

[6] *ECMAScript Language Specification*. `http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf`. June 2011.

[7] Norman Fenton and Shari Lawrence Pfleeger. *Software metrics (2nd ed.): a rigorous and practical approach*. Boston, MA, USA: PWS Publishing Co., 1997. ISBN: 0-534-95600-9.

[8] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA, 1999. ISBN: 0-201-48567-2.

[9] Geoffrey Garen. *Announcing SquirrelFish*. `http://www.webkit.org/blog/189/announcing-squirrelfish/`. 2008.

[10] Salvatore Guarnieri and Benjamin Livshits. "GATEKEEPER: mostly static enforcement of security and reliability policies for javascript code". In: *Proceedings of the 18th conference on USENIX security symposium*. SSYM'09. Montreal, Canada: USENIX Association, 2009, pp. 151–168.

[11] Salvatore Guarnieri, Marco Pistoia, Omer Tripp, Julian Dolby, Stephen Teilhet, and Ryan Berg. “Saving the world wide web from vulnerable JavaScript”. In: *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ISSTA ’11. Toronto, Ontario, Canada: ACM, 2011, pp. 177–187. ISBN: 978-1-4503-0562-4. DOI: http://doi.acm.org/10.1145/2001420.2001442.

[12] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. “The essence of javascript”. In: *Proceedings of the 24th European conference on Object-oriented programming*. ECOOP’10. Maribor, Slovenia: Springer-Verlag, 2010, pp. 126–150. ISBN: 3-642-14106-4, 978-3-642-14106-5.

[13] “IEEE Standard for Floating-Point Arithmetic”. In: *IEEE Std 754-2008* (2008), pp. 1 –58. DOI: 10.1109/IEEESTD.2008.4610935.

[14] “IEEE Standard for the Scheme Programming Language”. In: *IEEE Std 1178-1990* (1991), p. 1. DOI: 10.1109/IEEESTD.1991.101032.

[15] Dongseok Jang and Kwang-Moo Choe. “Points-to analysis for JavaScript”. In: *Proceedings of the 2009 ACM symposium on Applied Computing*. SAC ’09. Honolulu, Hawaii: ACM, 2009, pp. 1930–1937. ISBN: 978-1-60558-166-8. DOI: http://doi.acm.org/10.1145/1529282.1529711.

[16] Jesse James Garrett. *Ajax: A New Approach to Web Applications*. http://adaptivepath.com/ideas/ajax-new-approach-web-applications.

[17] John Neumann. *ECMAScript*. http://www.ecma-international.org/activities/Golden%20jubilee/cc-2011-014.ppt. 2011.

[18] S. C. Johnson. “Lint, a C Program Checker”. In: *COMP. SCI. TECH. REP*. 1978, pp. 78–1273.

[19] Stephen H. Kan. *Metrics and Models in Software Quality Engineering*. 2nd. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN: 0201729156.

[20] Paul Krill. *JavaScript creator ponders past, future*. http://www.infoworld.com/print/39704. June 2008.

[21] Michele Lanza, Radu Marinescu, and Stéphane Ducasse. *Object-Oriented Metrics in Practice*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005. ISBN: 3540244298.

[22] Jens Lindström. *Carakan Revisited*. http://my.opera.com/core/blog/2009/12/22/carakan-revisited. 2009.

[23] David Mandelin. *SquirrelFish*. http://blog.mozilla.com/dmandelin/2008/06/03/squirrelfish/. 2008.

[24] Kevin Millikin and Florian Schneider. *A New Crankshaft for V8*. http://blog.chromium.org/2010/12/new-crankshaft-for-v8.html. 2010.

[25] Shanku Niyogi. *The New JavaScript Engine in Internet Explorer 9*. http://blogs.msdn.com/b/ie/archive/2010/03/18/9981410.aspx. 2010.

[26] Shanku Niyogi, Amanda Silver, John Montgomery, Luke Hoban, and Steve Lucco. *Evolving ECMAScript*. `http://blogs.msdn.com/b/ie/archive/2011/11/22/evolving-ecmascript.aspx`.

[27] S.P. Reiss. "Semantics-based code search". In: *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*. IEEE. 2009, pp. 243–253.

[28] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. "An analysis of the dynamic behavior of JavaScript programs". In: *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*. PLDI '10. Toronto, Ontario, Canada: ACM, 2010, pp. 1–12. ISBN: 978-1-4503-0019-3. DOI: `http://doi.acm.org/10.1145/1806596.1806598`.

[29] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. "The eval that men do: A large-scale study of the use of eval in javascript applications". In: *Proceedings of the 25th European conference on Object-oriented programming*. ECOOP'11. Lancaster, UK: Springer-Verlag, 2011, pp. 52–78. ISBN: 978-3-642-22654-0.

[30] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach". In: *Sci. Comput. Program.* 74 (7 May 2009), pp. 470–495. ISSN: 0167-6423. DOI: `10.1016/j.scico.2009.02.007`.

[31] N. Synytskyy, J.R. Cordy, and T. Dean. "Resolution of static clones in dynamic web pages". In: *Web Site Evolution, 2003. Theme: Architecture. Proceedings. Fifth IEEE International Workshop on*. IEEE. 2003, pp. 49–56.

[32] David Ungar and Randall B. Smith. "Self: The power of simplicity". In: *Conference proceedings on Object-oriented programming systems, languages and applications*. OOPSLA '87. Orlando, Florida, United States: ACM, 1987, pp. 227–242. ISBN: 0-89791-247-0. DOI: `http://doi.acm.org/10.1145/38765.38828`.

[33] Dimitrios Vardoulakis and Olin Shivers. "CFA2: a Context-Free Approach to Control-Flow Analysis". In: *Logical Methods in Computer Science* 7.2 (2011).

[34] Chuan Yue and Haining Wang. "Characterizing insecure javascript practices on the web". In: *Proceedings of the 18th international conference on World wide web*. WWW '09. Madrid, Spain: ACM, 2009, pp. 961–970. ISBN: 978-1-60558-487-4. DOI: `http://doi.acm.org/10.1145/1526709.1526838`.

Appendix A

M-Industries Schema Language

In this appendix, we give a description of a subset of the M-Industries schema language. This subset covers all parts of the schema language that have been used in the examples contained in this thesis.

The schema language has similarities to XML Schema[1], with the important addition of tagged unions[2] (as 'state groups') to describe relations in which a set of child properties only exists for a specific state of their parent property.

Listing 44 specifies the schema syntax used in this document. Some details have been omitted for the sake of brevity. Because the listing only describes the concrete syntax, the semantics of most of these concepts are explained in the remainder of this appendix.

```
schema = { "type", identifier, type fragment } ;

type fragment = { propdef } ;

propdef = "text", propname
        | "number", propname
        | "list", propname, type fragment
        | "dictionary", propname, type fragment
        | "group", propname, type fragment
        | "stategroup", propname, { identifier, type fragment }
        | "component", propname, identifier
        | "reference", propname, identifier
        ;

propname = identifier ;
```

Listing 44: Schema syntax specification in EBNF.

[1] http://www.w3.org/TR/xmlschema-0/
[2] http://en.wikipedia.org/wiki/Tagged_union

A.1 Schema

A schema is a collection of uniquely named type fragments. We will refer to these top-level named type fragments as ‘schema types’.

A.2 Type Fragments

A type fragment is a collection of uniquely named properties. Because the schema language uses a static type system, every property must have a specific built-in type, which can be a primitive or composite type. A type fragment can not exist on its own, it is always part of either a schema type or a composite type. Properties are always required to have a value, there is no support for optional properties other than explicitly modeling this using a state group.

A.3 Primitive Value Types

The schema language has two primitive types for text (strings) and numbers:

- Text.
- Number.

Software engineers will notice the lack of a ‘boolean’ primitive, as well as several number variants. Boolean properties can always be expressed using the more powerful state group type, so there was no reason to include booleans as a primitive type. One of the design goals of the schema language was to create a set of orthogonal modeling constructs. People do not need to make arbitrary choices while creating a model if there is no overlap between the building blocks of the schema language.

It is possible to assign a ‘numerical type’ to a number in order to differentiate between several types of numbers, e.g. currencies, dates, times or distances. Implementation-wise, the text and number types are directly mapped to the primitive JavaScript types String and Number. Listing 45 shows the use of these primitive types in the definition of a ‘person’ type.

```
type "person"
  text "name"
  number "age"
```

Listing 45: Primitive types for text and number.

A.4 Composite & Reference Types

To support modeling complex datasets, the schema language defines types for several composite & reference types:

- **List**. An ordered collection of type fragments.
- **Dictionary**. A map from strings to type fragments.
- **Group**. A named type fragment, used to make a logical group of related properties.
- **State Group**. A dictionary of named states with associated type fragments.
- **Component**. An contained instance of a schema type (parent instance owns child instance). To be used for composition.
- **Reference**. A reference to a type fragment using the instance path of the type fragment. To be used for association.

In listing 46, we see an example of how a state group is used to explicitly model optional fields. We see types for a function statement and a function expression. The only difference between the two types is that the identifier is optional for a function expression, while the identifier is always required for a function declaration. The state group in the function expression type illustrates how we can make the optionality of a property explicit in the model, by describing the two different states (empty, named) and the properties (id) that depend on those states.

```
type "function"
  list "body"
    component "statement" -> "statement"

type "function declaration"
  text "id"
  component "function" -> "function"

type "function expression"
  stategroup "id"
    state "empty"
    state "named"
      text "id"
  component "function" -> "function"
```

Listing 46: Sharing model fragments using components.

The ‘component’ type should be used for classical object composition where the child object has no independent existence outside its containing parent object. The child object is part of the parent object. Listing 46 shows how the types ‘function declaration’ and ‘function expression’ both share the common definition of a JavaScript function (a list of statements), modeled as a schema type. This is a clear case of composition. A ‘function’ type instance can not exist on its own. It has to be part of a function declaration or function expression.

When there is no direct ownership or containment in the relation and when you want to reference any existing type fragment (not just instances of schema types), use a ‘reference’ type. This expresses an associative relation between two objects. The relation implies no ownership and the associated object exists on its own, parallel to the associating object. Any type fragment can have multiple references pointing to it.

```
outer: while (true) {
  while (true) {
    if (x) break outer;
  }
  break outer;
}
```

Listing 47: Code example for break statements.

```
type "break statement"
  stategroup "type"
    state "labeled"
      reference "statement" -> "statement.type*labeled"
    state "unlabeled"
      reference "statement" -> "breakable statement"
```

Listing 48: Using references in the schema.

Listing 48 gives an example of the use of references. JavaScript allows the use of both labeled and unlabeled break statements. A labeled break statement refers to a labeled statement, an unlabeled break statement automatically refers to the first containing breakable statement (switch, for, while or do). The code example in listing 47 gives an example of two break statements that refer to the same labeled statement. This could not be modeled with components: the labeled statement is no part of the break statement and the label can be referenced from multiple locations.

Appendix B

Overview of Instruments

This appendix contains an overview of the instruments written during this project. The tables B.1, B.2 and B.3 show all those instruments, organized in three different packages. The package structure is only used to group related instruments together. In every table, the column ‘Type’ indicates the instrument type which can be one of ‘syn’ for syntactic instruments and ‘sem’ for semantic instruments.

Package	Name	Type
metric	call_arg_list_length	sem
	call_boolean_args	sem
	cyclomatic_complexity	sem
	function_body_length	sem
	function_name_length	syn
	function_par_list_length	sem
	function_param_length	syn
	single_use_function	sem
	string_literal_length	syn
	variable_name_length	syn
	write_to_function_parameter	sem

Table B.1: Analysis engine instruments, metrics package.

Package	Name	Type
m-industries	assignment_expression	sem
	assignment_expression_confusion	sem
	assignment_of_function	sem
	assignment_of_this	sem
	binary_expression_and	sem
	binary_expression_implicit_tostring	sem
	binary_expression_instanceof	sem
	binary_expression_or	sem
	binary_expression_shift_left	sem
	binary_expression_shift_right	sem
	binary_expression_shift_right_unsigned	sem
	binary_expression_xor	sem
	builtin_function	sem
	call_function_expression	sem
	conditional_expression	sem
	deprecated_function_call	sem
	forbidden_expression_new	sem
	forbidden_statement_break	sem
	forbidden_statement_debugger	sem
	forbidden_statement_if	sem
	forbidden_statement_iteration	sem
	forbidden_statement_switch	sem
	forbidden_statement_throw	sem
	forbidden_statement_try	sem
	forbidden_statement_with	sem
	if_without_else	sem
	member_expression	sem
	member_expression_computed	sem
	new_expression	sem
	numerical_literal	sem
	numerical_literal_soft	sem
	object_expression_keys	sem
	object_expression_keys_literal	sem
	return_function_expression	sem
	this_function_assignment	sem
	unary_delete	sem
	unary_not	sem
	unary_typeof	sem
	update_expression	sem

Table B.2: Analysis engine instruments, M-Industries package.

Package	Name	Type
generic	call_confusion	syn
	call_spacing	syn
	constructor_confusion	syn
	function_brace	syn
	function_name_lowercase_first	syn
	function_name_uppercase_first	syn
	implicit_global	sem
	labeled_statement_in_switch	sem
	object_expression_spaces	syn
	one_variable_declaration_per_function	sem
	param_list_spacing	syn
	regex	sem
	statement_semicolon	syn
	statement_space	syn
	string_quotes	syn
	superfluous_predefined_global	sem
	superfluous_writable_global	sem
	trailing_decimal	syn
	unreachable_statement	sem
	unused_label	sem
	unused_variable	sem
	update_expression_confusion	sem
	use_before_define	sem
	use_block	sem
	variable_hiding	sem
	write_to_readonly_global	sem

Table B.3: Analysis engine instruments, generic package.