ABSTRACT

With the increase in open source and large scale projects in the software industry, good documentation has become increasingly rare despite its ability to directly improve software projects. In particular, JavaScript web applications serve as prime examples of unwieldy systems greatly affected by the lack of documentation due to the volatile environment of web development and its rapidly changing technologies. This paper proposes a tool that can automatically generate useful documentation with minimal developer input through the recognition of design patterns. We consider this semantic code analysis because analysis of design patterns is the best approximation we have of intent, or semantic meaning, of code. Supporting common design patterns as its base, the system hosts an extensible plugin system for domain-specific design patterns. In doing so, existing codebases and new ventures can be equipped with some base documentation that can be extended as needed.

1. INTRODUCTION

Well-written internal documentation can significantly enhance both the internal quality and external quality of complex software systems. Internally, documentation improves readability, maintainability, reusability, reliability. Externally, documentation indirectly improves correctness, extendability, and reusability.

In turn, documentation produces several benefits:

1. Software is easier to extend and maintain. Documentation enables developers to communicate independent of space and time, which in turn enables organizations to scale likewise. This is essential in open source software.

2. Software is easier to test and reuse, since expectations of behavior are clearly communicated.

3. On-boarding is less expensive, since new hires may consult the code base rather than other developers.

Nevertheless, outside of open-source, internal documentation often does not exist. This is for a few reasons:

1. To some degree, a good language implementation will remove much of the need for internal documentation if it favors clarity and structure [3].

2. Documentation conflicts with timeliness and economy since it takes developers time to initially write and maintain it. In time-sensitive projects, the costs may not be worth the benefits.

3. The initial investment of writing documentation takes some time to recover. For certain volatile projects, this initial investment may not be repaid due to documentation being simply thrown away.

4. The cost of maintaining documentation can be expensive. This includes not only updating documentation as it changes, but the costs that are incurred whenever a contributor gets confused by out-of-date documentation.

These reasons against documentation are especially relevant to JavaScript web applications, which exist in conditions where requirements are extremely volatile and technological obsolescence is rapid.

However, the software industry stands much to gain if these reasons can be undermined. JavaScript is becoming increasingly popular, and is the top used language on GitHub. It has a vibrant community, extensive open-source offerings, and due to its wide applicability, it is increasingly being promoted as a first programming language for beginners. Furthermore, with the advent of node.js, a platform which enables JavaScript to run on web servers (instead of only web browsers), we expect the popularity to only increase.

We propose to design and implement a tool that can automatically generate useful documentation for JavaScript application code with minimal developer input that captures high-level code intent. If this is possible, the major reasons for not writing documentation will become invalid, and organizations will get all of the benefits of documentation at very little cost.

2. BACKGROUND

To do this, we take advantage of the fact that JavaScript exhibits design patterns – formal solutions to design problems in a particular domain – which communicate high-level expectations about a codebase.

Design patterns necessarily express intent since they necessarily solve a particular problem. For example, in C there is no iterator construct, so to iterate over an array, a programmer has to declare a temporary variable (e.g. i) representing the index, point it at the start of the array, and increment it after processing an element in the array, checking
each time if the index has exceeded bounds. This pattern of declaring a temporary variable, processing an element, incrementing the index, and checking for an exit condition always expresses the desire to iterate.

Since design patterns necessarily occur wherever a programmer cannot otherwise express intention clearly by a built-in language feature or abstraction, they represent the only parts of code that are necessarily unclear (there may be others, if for instance, an amateur developer attempts to express himself). It seems to follow that if we can construct a tool that reliably finds and explains them, we can document a codebase in such a way that no part is unclear.

Through the research and development we have accomplished so far, we believe this kind of tool should employ a code-matching algorithm at the abstract syntax tree (AST) level that detects known design patterns in JavaScript applications. Specifically, the design patterns we ultimately decided to focus on were the following:

1. **MVC (Model-View-Controller):** The MVC pattern is common in most non-trivial Javascript applications, as it allows developers to better organize their UI code and enforce separation of concerns. Common examples of MVC libraries are Backbone.js, Ember.js, and Angular.js. For simplicity, we focused on identifying specific instances of various MVC libraries.

2. **Singleton:** The Singleton pattern is, similarly, a classic design pattern implemented across many JavaScript applications. Used as a proof of concept, we want to show the versatility of our tool through the recognition of various aspects of the singleton, including public versus private properties and methods, instantiation methods, and instance variables.

3. **Classes:** There is no built-in support for classes in Javascript – instead, the language supports a prototypical inheritance scheme. As such, developers have adopted the function based implementation of inheritance in various ways to closely represent classes found in other languages. Because of this, we felt the recognition of multiple different class implementations would be an extremely important design pattern to include in our analysis.

4. **Decorators:** Decorators are common in functional applications where function composition is widely used as well as in applications with cross-cutting concerns, e.g., logging. Decorators are also simple to identify, both manually and automatically, which made them ideal for prototyping and testing our system.

5. **Module Systems / Dependency Injection:** Large scale JavaScript applications also commonly employ some sort of module system. Again, since this isn’t built into the language, this pattern is vital to organizing and streamlining development. They allow exporting and importing of developer defined dependencies that can be loaded at will.

In the rest of this paper, we will briefly describe some existing research work related to semantic code analysis and code matching, the details of our own system design & implementation, and our results in successfully identifying design patterns.

---

### 3. RELATED WORK

#### 3.1 Contract Inference

Some prior work exists in the domain of contract inference, i.e., determining formal specifications from source code. Current research has yielded many important results, but for most communities contract inference remains impractical or unpopular, as they often produce incomplete, unsound, or irrelevant (redundant or trivial) inferences [5].

Existing approaches for contract inference can be divided into static approaches and dynamic approaches.

**Static approaches** infer contracts through analysis of source code. Static contract inference generates sound contracts, but sound inference of interesting properties is in general undecidable, and inferred contracts tend to be conservative and incomplete.

Some possible inferences of static approaches include: interval constraints, affine relationships among variables, extended typing information, sequence diagrams describing the flow of information among objects, and invariants of loops.

**Dynamic approaches** infer contracts through observation of program state over multiple executions at predefined points (such as the entry and exit of a routine). Dynamic contract inference is useful because it is flexible, scalable, comparatively simple to implement, and tends to work well in practice. However, the contracts generated from dynamic inference may be unsound, incomplete (tend to be trivial, irrelevant, and redundant), and the effectiveness is heavily dependent on the quality of test cases used to execute the program (must execute program paths extensively and with varied values, and are often not present for legacy code).

Generally, dynamic approaches perform the following steps:

1. Generate a test suite and profile execution
2. Perform analysis on the profile results to generate a change profile
3. Perform analysis on the change profile to infer a contract

In step 3, analysis on the change profile can take a variety of (complementary) methods, for instance, templated-based invariant detection, predicate mining, and decision tree learning [2].

#### 3.2 Abstract Syntax Tree Comparison

Comparing source code using the abstract syntax tree (AST) is also a field with some activity, as its applications are useful in understanding source-code changes and comparing code for other purposes, like plagiarism detection.

There exists research on algorithms for comparing the ASTs of C programs, which can track changes in source code to variables in scope, types, and functions [4]. These changes can in turn be used to answer higher level code questions about the ongoing development and refactoring of a codebase.

Other work has focused on attempting to fingerprint code at the AST level for the purposes of plagiarism detection. Researchers have used these fingerprints to index AST representations in a database, expanding positive matches out to pinpoint blocks of duplicated (if modified) code [1].
4. SYSTEM MODEL

We are tackling the problem at hand with a system that automatically detects design patterns in application source code. Our system works in two stages, as shown in Figure 1. In the first stage, which occurs only once, we set up the code analysis system by inputting a catalog of design patterns and their representative ASTs. In the second stage, the user inputs a codebase, which the system analyzes and documents.

Our job in the first stage consists of cataloging common design patterns found in various JavaScript applications. Each catalog entry consists of an AST structure, design pattern name, a description of the pattern’s intention. In the case where multiple ASTs are required to capture a design pattern, we catalog all of them and for each catalog entry make a note in the associated design pattern description about which part of the pattern it pertains to. For example, if three ASTs are required to describe a Model-View-Controller pattern, all three would be cataloged with MVC as the associated pattern and some indication of which part of the pattern the AST represents: Model, View, or Controller. These ASTs will use a canonical, simplified subset of the source language, which will allow for better code matching (see System Implementation for why this is the case). Finally, this catalog will be saved to a database with a serialized representation of each AST (a syntax tree “fingerprint”). These fingerprints will be crucial in the next stage, where detection of design patterns occurs.

The documentation generation application (the second stage in the diagram) has access to the database of design patterns we generated in the first stage. With this application, a user may input programs (which may consist of multiple files) to the system for analysis. The system analyzes the target program at the AST level by identifying specific design patterns through two methods. First, syntax fingerprints of AST nodes are queried against fingerprints in the database, possibly yielding a matched design pattern. This process is repeated for all AST nodes of the target program. Second, we match it against our plugin system to determine possible matches with our pattern recognition algorithms for inspecting AST structures. Each file is run once with each plugin to identify the existence of any of the defined patterns, outputting a standard output object containing all identified patterns. Finally, the system aggregates any detected design patterns into a stylized report webpage for the user.

In order to make our tool extensible, we modularize our design pattern identifiers, and allow easy insertion or removal of a design pattern identifier. In order to implement recognition for a new design pattern, one need merely write a function that fulfills our function signature, taking in an AST and outputting a set of annotated pointers to the source code, with each pointer representing a component of the design pattern.

5. SYSTEM IMPLEMENTATION

In order to accomplish source code matching at the AST level, we employ various AST transformations and utilize syntax tree fingerprinting techniques, building on previous research [1].

5.1 AST Construction

First, we parse source code files and generate an abstract syntax tree for each unit of the program, creating nodes based on the Mozilla JavaScript Parser API. Specifically, we use a tool called Acorn to parse and manipulate JavaScript ASTs. Given an input JavaScript file, Acorn outputs a JavaScript object representation of the abstract syntax tree. Appendix A shows a representation of a JavaScript object (essentially “printf” for JS).

Due to the complexity and nuances associated with JavaScript, our project aims towards addressing a specific subset of the language, ignoring features that are commonly considered bad practice in industrial use. Language features such as `with`, `eval`, `void`, and operators like `==` should be taken out before entering the fingerprinting stage, introducing the need for a separate stage to deconstruct and simplify the AST into a more manageable subset of the language.

We constructed various utility functions to assist in the manipulation of our AST. Most prominent in our functions were a variety of tree traversal functions for finding nodes, parsing types according to the Parser API, getting child nodes, and just walking the tree through a variety of different tree-traversal algorithms.

For managing our output, we constructed a few helper classes for organizing the relevant parts of our recognized design patterns. Prior to outputting our HTML, we organized our output into a set of CodeCatalogs, consisting of CodePointers that consist of the location and name of the pointer to the portion of the code. A CodeCatalog consists of a collection of CodePointers and CodeCatalogs for more complex design patterns consisting of multiple sections. Each CodeCatalog contains standard setter and getter methods, used for handling various aspects of the code. CodeCatalogs are further extended for each design pattern to allow for pattern-specific helper methods.

5.2 AST Reduction

Given the initial AST produced by Acorn, we reconstruct a separate reduced AST containing only the minimum essential nodes representing the original JavaScript code to simplify the fingerprinting process. There are two main steps to this reconstruction.

First, nodes representing features of the language commonly considered bad practice (such as the `with`, `eval`, and `void` statements) are removed completely during the reduction. They are preserved in the original tree to allow for us to generate warnings in the final documentation to the developer as expressions in need of refactoring.

In the second step, we reduce the available nodes in the Mozilla Parser API to a more manageable subset to reduce fingerprinting complexity. Specifically, loop node structures (for loops, do-while loops) can all be converted into a generic while loop. Switch statements and ternary operators can be converted into a series of if, else if, and else statements. Variations in declarations can be generalized as variable declarations by simulating function and variable hoisting in the abstract syntax tree.
Figure 1: The block on the left represents the first stage, where we catalog design patterns and corresponding ASTs to the system database. The block on the right represents the second stage, the core application in which a user can analyze a codebase and generate a documentation report. The large black arrows symbolize the input and output of our system.

As an example, in the case of a `switch` statement, the source program can be reliably rewritten as a series of `if / else` statements, as shown in the sample code (see Appendix B). To maintain source maps, we preserve all positions in the AST associated with each switch case in their corresponding `if / else` block, ignoring the initial switch operator. We assume the cases to be represented as an equality comparison between the variable in question and the given value in the original AST to simulate the switch. We can then assign the consequent and alternate attributes in the `if / else` nodes as the code block under the currently examined switch case, and a recursive conversion of the next switch case respectively. By checking for the absence of a switch case, and instead for the presence of a ‘default’ block statement, we can simulate another else block, and append that as the alternate to the final `if / else` chain.

Once we have this simplified AST, we then traverse it to fingerprint each node.

### 5.3 Tree Fingerprinting Techniques

Given an AST, we want to create “fingerprints” of its nodes and store these associated values in an indexed database. For a subtree $t$, its fingerprint contains a tuple of its weight (size of the subtree) $w(t)$, a hash value $H(t)$, and a pointer to its parent node in the AST (similar to Chilowicz et al. [1]). Knowing the weight of a subtree will allow us to discard certain categories of design patterns in the detection stage (for example, only look for low-level patterns when looking at subtrees of low weight below some threshold). The more interesting part of this fingerprint is the value generated by a hash function $H$ we develop to identify various types of subtrees.

In developing this hash function $H$, we must consider some properties it should uphold in order for the system to be robust, complete, and efficient. First of all, it should take into account the tree structure of its internal nodes. A good function will minimize the probability of hash value collisions for two structurally different subtrees. In addition, the hash function ought to be computed incrementally from child nodes – this makes fingerprinting a linear algorithmic process as we build up hash values on an AST bottom-up. Robustness is the most difficult property to ensure for this function – it should allow for near-matching against fingerprints of design patterns queried from the database (while also keeping hash collisions at a reasonable level). For example, consider the attached code snippets that should all be successfully matched against a `for each` pattern (see Appendix C).

We do not hope to capture all the possible syntactic representations of a given design pattern into a single hash value, as this will likely result in many false-positives from the detection process. Rather, as referenced above, the cataloging process must take into account various syntactic representations of each design pattern and fingerprint these separately (number of representations is determined on a case-by-case basis). Overall, this part of the AST-matching algorithm requires calibration in order to achieve optimal system performance.

### 5.4 Design Pattern Matching

At the core of our project, we introduce various pluggable matching algorithms for identifying the individual design patterns. Out of the box, we support MVC, Singletons, Decorators, Dependency Injection/Module Systems, and Classes. The system itself contains support for custom-built pattern detection algorithm plugins into the system.
to support design pattern complexity and pattern variability. For example, modern Model-View-Controller libraries are highly variable largely due to differences in functionality and offered features. Differences in available functionality results in differing levels of complexity, as well as different ways to structure the various components of the libraries. For example, just in comparing Backbone.js and Ember.js (two of the most popular MVC frameworks), we notice that common functionality such as message passing, event propagation, and URL routing are implemented very differently. Dependency injection, a feature found natively in Ember, is absent in Backbone. Backbone’s MVC structure lacks an inherent “Controller”, instead operating around a Model-View-Collection model, with Collections being a construct and an inherent “Controller”, instead operating around a Model-View-Collection model, with Collections being a construct to maintain the relationship between models and views.

Given the constructed AST following fingerprinting, we run these through the set of base plugins, generating custom objects to output into our report generator. Our recognition algorithms operate as follows:

1. **MVC**: Because of the complexities involved, we decided to focus on the Ember.js and Backbone.js detection of the various Model, View, Collection/Controller, and Router detection. We do this through the detection of inheritance through `extend` methods offered by the libraries themselves (`Backbone.Model.extend, etc.`), and pull out the objects given in the AST. For Models, we are able to pull out the relevant data attributes. For Backbone, this is done through the detection of `get`/`set` method invocations, as well as the parsing of the `defaults` dictionary. In Ember, this is done through just parsing the attributes provided in the model definition. We are also able to parse the relevant URL endpoints offered by the routers through the detection of route definitions. In Ember, this is clearly defined through `this.route` keywords. In the Backbone router, we can similarly find such examples in the `routes` attribute under the Router definition.

2. **Singleton**: The singleton is a classic design pattern and can be found in most languages with a fairly similar base implementation. The design pattern itself is decomposable into multiple distinctive portions – private and public methods. In JavaScript, there are a few ways to declare singletons. In its simplest form, singletons can be implemented through a simple object. To add scope complexity to the singleton, we can adjust it to include the standard instantiation and retrieval methods through the usage of an Immediately Invoked Function Expression (IIFE), hiding private variables and exposing public variables in the returned object. To determine the existence of the Singleton design pattern, we detect the presence of an “existence check” in a function attribute on the returned object of an IIFE (this adds the shared behavior without changing the name or interface of the decorated function) they can be discovered by searching for declarations where the identifier on the left hand side is identical to the identifier of some argument to some function, `g`, on the right hand side, where both this identifier and `g` are known to be functions.

3. **Decorators**: In JavaScript, decorators are used when multiple functions share essential behavior, e.g. logging or memoization. Since decorators typically take the form `f = g(f)` (this adds the shared behavior without changing the name or interface of the decorated function) they can be discovered by searching for declarations where the identifier on the left hand side is identical to the identifier of some argument to some function, `g`, on the right hand side, where both this identifier and `g` are known to be functions.

4. **Modules**: Client JavaScript lacks any native module or import system, creating a problem for developers trying to organize large code bases. Although Node introduced a module system based on the CommonJS specification, and there are plans for first-class modules, imports, and exports in ECMA Script 6, the majority of client JavaScript code today has to make do with a number of conventions and design patterns, collectively known as the “module pattern.” In this pattern, developers wrap function declarations in an immediately-invoking function expression (IIFE), which returns an object containing all the exported functions. This allows developers to keep methods “private” by not adding them to the returned object (the IIFE creates a new scope which closes over the functions when immediately invoked). The result of the IIFE can then be stored in a variable, creating the “module.” To verify a module pattern, we identify IIFEs that are exported to named variables. We then traverse the body of the IIFE to determine which methods it exports. Exported methods are marked public, and the remaining methods are considered private. If we find that the IIFE doesn’t return a module-like object, we invalidate the module match.

5. **Classes**: Like the module pattern, JavaScript also has limited first-class support for class-based object-oriented programming. Although JavaScript has a `new` keyword which instantiates classes based on a prototype, this is a far cry from full support for inheritance, public and private methods and properties, and many of the other trappings one would expect from a classically object-oriented language.

Developers emulate object-oriented patterns by assigning methods to an object prototype, and achieve inheritance through using custom `extend` methods (or, since ECMA Script 5, tools like `Object.create()`), that read off the methods and properties of a prototype and create a new one based on it.

To identify class patterns, we first look for constructor definitions, which are typified by assignments to `this` and are seen called with the `new` keyword. Granted...
this, we look for methods defined on that constructor’s prototype (typically in the same file, or module as it were) and tag those methods. Such methods will also typically reference the this keyword. We also look for some of the common ways of expressing inheritance, and then map the inheritance structures for our identified classes.

5.5 Report generation

The final part of the system is a report generation step that takes a number of catalogs (which represent design pattern matches in the source code) created in the previous stages and generates a usable interface for browsing the new documentation. Each pattern matching module adheres to an interface that allows our report generator to recursively print the contents of a CodeCatalog to some HTML markup. Once all the source code in the target directory has been run through our documentation tool, we generate a static web page which includes styling (for ease of navigation). A sample screenshot is shown in Figure 2.

6. SYSTEM PERFORMANCE

6.1 Quality of Identified Patterns

Success of our resulting documentation ultimately relies upon its usefulness to a target developer for the application source code being analyzed. However, there is also the intermediate step of evaluating the correctness of design-pattern matches.

Initially, we will use our knowledge of design patterns to evaluate the quality of matches, which is in general straightforward. There will be some gray area around evaluating imprecise implementations of design patterns – i.e., when some code is attempting to follow a particular pattern but does so incorrectly. Ultimately, we are focusing on the straightforward cases before expanding to the ambiguous ones.

Given our initial satisfaction with the design pattern matches, we will then expand to use existing documentation as feedback, comparing our analysis with the extant documentation on well-documented libraries and applications. It’s worth noting, though, that we expect existing documentation may vary considerably in its depth and focus, so it is of limited use in determining the effectiveness of the patterns we seek to identify in our code.

The final stage of performance evaluation will come in the form of direct feedback on documentation quality from other developers, who will poll to provide their own analysis of code patterns and compare them with the patterns that our tool produces.

6.2 Efficiency of Analysis

Another important component of the success of our tool will be its efficiency and/or resource cost of running our analysis. Ultimately, since our analysis is one-time and static, we believe that we can make reasonable trade-offs in speed efficiency for the sake of correctness and utility. However, analysis still cannot be prohibitively expensive, and success in this domain will mean a tool that runs within reasonable time constraints.

Furthermore, since we will likely have to use some kind of stateful storage for the fingerprinting component, we will also have to consider costs like the number of database queries. We expect these costs can be reasonably mitigated through caching mechanisms and software design.

7. RESULTS

7.1 Pattern Extraction

Despite only being a prototype, we successfully hit all of our major goals with our system.

First, our system is able to reliably discover and extract each of the patterns we decided to focus on (i.e., Model-View-Controller, Singleton, Class, Decorator, and Module). Our system works even when the writer of the codebase is somewhat malicious, as in the case of minifiers, since it is purely syntactic and does not depend on developer input in any way. It is worth noting that our system likely generates a number of false positives and false negatives as we encountered many during development, but since the simplest patterns are also the most common, we expect that the value lost from these defects is marginal.

Second, our system runs in an acceptable amount of time. Of the ten most popular open-source JavaScript applications on GitHub, none took more than a few minutes for our system to process. As we expect our tool to be used infrequently, we judge a few minutes to be an acceptable time to wait.

Finally, our system is also highly extensible. New additions to the pattern catalog involve simply writing a single function capable of detecting instances of a given pattern.

7.2 Report Generation

The effectiveness of our tool ultimately depends on the goal of the end user. Our tool does not replace the need for developers to document the low-level intentions of their code; for instance, a codebase that uses no design patterns would still need to be documented normally.

However, for large applications that make extensive use of design patterns, especially in web applications guided by a framework, our tool is able to generate auxiliary documentation that gives a useful overview of the target system’s architecture. For developers being onboarded to a new system, we suspect our tool could be of great use.

Finally, it is important to note that our system is still a prototype and is intended to gain most of its value from users configuring it to their needs. In particular, we can imagine users benefiting from extending our system to detect antipatterns so that they can be rewritten more succinctly, or extending it to be a general linter.

8. FUTURE WORK

With our current implementation of our documentation generation tool, we’ve built the infrastructure for a powerful tool that has the potential to change the way developers approach existing code bases. Old, monolithic systems now become more readily accessible to new developers, which could potentially reignite interest in large open source communities. It could also potentially redefine the way through which developers approach refactoring, as this provides them with a conclusive report on areas heavily influenced by conflicting design patterns, as well as areas lacking good design at all.

Feature-wise, the first thing that can be done is to increase the scope of design patterns recognized. Other common patterns found in modern Javascript applications include the
Observer Pattern, as well as a variety of library-specific design patterns used. Because our system also supports developer plugins, domain specific design patterns can be generated by developers. Because plugins need to be created manually through our plugin system at the moment, it would be useful to create a language for creating such plugins, abstracted specifically to determine design constructs through high-level descriptions of function and attribute identification.

Work also needs to be done in improving the user interface of our final code report. Documentation generation needs to be streamlined, and the potential for a large online repository of design patterns of existing open source libraries is an attractive potential byproduct of our tool. Further user-testing can be done to determine which attributes of the final output could be considered more or less important.

Code consumption itself has the potential to be drastically changed as well. With our current implementation, we provide the infrastructure and services to consume code and produce readable documentation. By combining this with the rapid rise in popularity of large open source repository services such as Github and Bitbucket, our project has the potential to reinvent developer’s primary method of code consumption. Instead of the typical code browsing through your standard directory structure, we could easily combine this with browser extensions to provide an integrated interface for perusing open source repositories (which, as most good developers will tell you, is essential to becoming a better developer). Our tool would provide immediate feedback on existing open-source repositories, what design patterns they use, the files in which specific design patterns can be found, and the specific components of each design patterns, combined to give developers an overarching summary of the repository.

Finally, our tool can be extended to support other languages. Though work would need to be done for AST reconstruction and fingerprinting, design patterns can similarly be identified through a similar process. Given that Javascript is the most popular language currently, the potential behind extending this to other, older, languages lies in the old and deprecated software systems written in less modern languages. This would be particularly useful in reviving interest in old but important services and applications – a prime example here is OpenSSH.

9. ETHICAL CONCERNS
As our tool is focused on consuming source code, we expect developers to focus their usage on source code legally available to them. As consuming open source software has no legal and ethical ramifications, we don’t envision problems occurring in this regard. Potential problems involved with our product are only potential problems when consuming any closed source code bases surrounding intellectual property.

10. CONCLUSIONS
Source code analysis is a field ripe with potential, as evidenced by a proliferation of existing tools around JavaScript code health (JSHint, JSLint, Google Closure compiler, etc.). In Javascript in particular, the amount of open source code has exploded through platforms like Bitbucket and GitHub.
Consequently, the need to understand JavaScript codebases has never been greater.

At the same time, JavaScript is still haunted by some of the fundamental design mistakes made in its infancy (the language was, after all, designed in only 10 days!). A lack of first-class module and import systems, poor support for classical object-oriented programming, and loosely-typed values with no encapsulation all have perpetuated conventions that are enforced through culture, not code, and can make understanding large codebases very difficult for newcomers. Even for seasoned JavaScript developers, the process of ingesting and understanding a new codebase can be slow. We hope that we can continue to build out the system described in this paper (and that others contribute) so that the JavaScript community might experience great improvements in programmer productivity.

11. REFERENCES


APPENDIX

A. AST representation of “console.log(‘default’);”

```json
{
  "type": "ExpressionStatement",
  "start": 166,
  "end": 189,
  "expression": {
    "type": "CallExpression",
    "start": 166,
    "end": 188,
    "callee": {
      "type": "MemberExpression",
      "start": 166,
      "end": 177,
      "object": {
        "type": "Identifier",
        "start": 166,
        "end": 173,
        "name": "console"
      },
      "property": {
        "type": "Identifier",
        "start": 174,
        "end": 177,
        "name": "log"
      },
      "computed": false
    },
    "arguments": [
      {
        "type": "Literal",
        "start": 178,
        "end": 187,
        "value": "default",
        "raw": "'default'"
      }
    ]
  }
}
```

B. Code differences:

Switch

```javascript
switch(num) {
  case 1:
    console.log('one');
    break;
  case 2:
    console.log('two');
    break;
  default:
    console.log('default');
}
```

If/else:

```javascript
if(num === 1) {
  console.log('one');
}
else if (num === 2) {
```
```javascript
console.log('two');

else {
    console.log('default');
}

Nodes:
Switch:

{ "type": "SwitchStatement",
  "start": 0,
  "end": 191,
  "discriminant": {
      // Describe 'num'
    },
  "cases": [
    { "type": "SwitchCase",
      "start": 16,
      "end": 58,
      "consequent": [
        { // 'console.log('one')' statement
        },
        { // 'break' statement
        }
      ],
      "test": {
        // case 1:
      },
      { // case 2: SwitchCase
      },
      { // default SwitchCase
      }
  ],
  "test": {
    // num === 1 (converted into checking equality)
  },
  "consequent": {
    // console.log('one')
  },
  "alternate": {
    // Otherwise (else if...)
    "type": "IfStatement",
    "test": {
      // num === 2
    },
    "consequent": {
      "type": "BlockStatement",
      // console.log('two')
    },
    "alternate": {
      // Otherwise (else - default case)
      "type": "BlockStatement",
      "test": {
        // console.log('default')
      }
    }
  }
}

C.
For-each pattern syntax representations

for (var i = 0; i < 10; i++) {

    // loop body
    ...
}

for (it = new Iterator(target); it.hasNext(); it.next()) {

    // loop body
    ...
}

D.
AST representations of sample code
```
Figure 3: Acorn AST representation of a basic switch statement.
Figure 4: Acorn AST representation of a basic if/else block.