OSS Vulnerabilities Through Islands of Knowledge

Masterarbeit

zur Erlangung des Grades eines Master of Science
im Studiengang Computer Science

vorgelegt von

Marco Brack

Erstgutachter:	Prof. Dr. Ralf Lämmel Institut für Informatik
Zweitgutachter:	Hakan Aksu Institut für Informatik

Koblenz, im März 2017

Erklärung

Hiermit bestätige ich, dass die vorliegende Arbeit von mir selbständig verfasst wurde und ich keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe und die Arbeitvon mir vorher nicht in einem anderen Prüfungsverfahren eingereicht wurde. Die eingereichte schriftliche Fassung entspricht der auf dem elektronischen Speichermedium (CD-ROM).

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☐	☐

.....

(Ort, Datum) (Marco Brack)

Zusammenfassung

Diese Arbeit schlägt die Benutzung von MSR (Mining Software Repositories) Techniken zum Identifizieren von Software Entwicklern mit exklusiver Fachkenntnis zu spezifischen APIs und Programmierfachgebieten in Software Repositories vor. Ein versuchsweises Tool zum finden solcher "Islands of Knowledge" in Node.js Projekten wird präsentiert und in einer Fallstudie auf 180 npm packages angewandt. Dabei zeigt sich, dass jedes package im Durchschnitt 2,3 Islands of Knowledge hat, was dadurch erklärbar sein könnte, dass npm packages dazu tendieren nur einen einzelnen Hauptcontributor zu haben. In einer Umfrage werden die Verantwortlichen von 50 packages kontaktiert und nach ihrer Meinung zu den Ergebnissen des Tools gefragt. Zusammen mit deren Antworten berichtet diese Arbeit von den Erfahrungen, die mit dem versuchsweisen Tool gemacht wurden, und wie zukünftige Weiterentwicklungen noch bessere Aussagen über die Verteilung von Programmierfachwissen in Entwicklerteams machen könnten.

Abstract

This thesis proposes the use of MSR (Mining Software Repositories) techniques to identify software developers with exclusive expertise about specific APIs and programming domains in software repositories. A pilot Tool for finding such "Islands of Knowledge" in Node.js projects is presented and applied in a case study to the 180 most popular npm packages. It is found that on average each package has 2.3 Islands of Knowledge, which is possibly explained by the finding that npm packages tend to have only one main contributor. In a survey, the maintainers of 50 packages are contacted and asked for opinions on the results produced by the Tool. Together with their responses, this thesis reports on experiences made with the pilot Tool and how future iterations could produce even more accurate statements about programming expertise distribution in developer teams.

Contents

Chapter 1

Introduction

Open Source Software (OSS) is exposed to a multitude of vulnerabilities. These include lack of code quality, insufficient maturity, low activity and insufficient user support [AAB+15].

Vulnerabilities become an important factor in the decision of whether an other project should depend on a piece of OSS once long term stability is a concern. If the development of a dependency project slows down or stops, this can incur huge costs for depending projects, which may have to find alternatives or continue the development of the dependency themselves.

One such vulnerability is expressed by the *Bus Factor*. The Bus Factor of a system is "the number of people on your team that have to be hit by a truck (or quit) before the project is in serious trouble" [WK02]. In other words, the Bus Factor is a measurement of how dependent a system is on key persons. Such dependencies can exist through, among other factors, isolated knowledge or expertise that is only available to a subset of persons in a system. If the last person with specific knowledge on a topic departs from a system, growth of the system might decline or come to an end. Thus, it can be crucial to identify such key persons and to determine the Bus Factor of a system.

Once a low Bus Factor is identified, counter-measures can be employed to prevent system failure[Awa07]. An obvious solution is the practice of *Cross-training*, which is "an instructional strategy in which each team member is trained in the duties of his or her teammates" [VCBSS96]. This practice necessarily improves the Bus Factor, as more persons are trained in the deficient areas. The other counter measures proposed by [Awa07] are "[Keeping] things simple", using "technologies that are familiar to more than one person on the team" and documentation.

The concept of a Bus Factor in a system is easily transferred to software development. Software developers in a team are usually experienced in differing areas, with some knowledge shared among them and some knowledge exclusive to few or just on developer.

In the context of software development, this thesis will from now on refer to these "areas" as *programming domains*.

Table 1.1 Numbers of commits in 3 popular projects on GitHub, retrieved 2017-03-09

Repository	Revision	C	D	$\varnothing C$	Rank 1	Rank 2	Rank 3
wp-calypso	`222af01`	16262	238	68.33	955	557	539
Ghost	`feaa25d`	7214	272	26.52	1118	395	267
sails	`ae6c00d`	6522	211	30.9	3700	1036	64

While the Bus Factor concept itself implies no specific mechanism through which a system is depending on a person, this thesis will examine software projects as systems and exclusive knowledge about programming domains as the dependency mechanism.

A rather superficial but quick metric for identifying a low Bus Factor in a software project is examining the number of commits by a developer in the project. If few developers stand out as the main contributors, it can be reasoned that the project is hugely dependent on them and that their departure might cause a significant crisis.

In table 1.1, for the projects **wp-calypso**[1], **Ghost**[2] and **sails**[3], the number of commits C, the number of developers D and the average number of commits per developer $\varnothing C$ are displayed together with number of commits by their top three contributors. Not surprisingly, the number of commits for the top contributors lies far above the average. The contributions by the developer with the most commits constitute 5.87%, 15.5% and 56.73% respectively. While 5.87% is a rather small fraction, 56.73% is enormous. The top contributor of sails has made more than half of all commits in the project. Clearly, the project must be heavily dependent on this developer. Additionally, even among the top 3 contributors, there exists some disparity. Compared to rank 1, the rank 2 developers, respectively, have 41.69%, 64.67% and 72% less commits.

Most appalling is the case of the rank 3 developer in sails: Compared to the developer with rank 1, they have 98.27% less commits. This number is even higher for all other developers in the project. Consequently, it can be reasoned that the Bus Factor for sails is 1 or 2. If the two developers with the most commits were to depart, the project would lose 72% of its "programming power" and most knowledge about the code generated by their 4736 commits would be lost. While these two consequences of this scenario might not necessarily mean the end of the project, they would surely significantly impair further development.

So it seems that the number of commits can be used as an indicator for the Bus Factor of a project. But this approach has two clear drawbacks:

- The number of commits does not necessarily correlate with the actual amount of code written.

[1] https://github.com/Automattic/wp-calypso
[2] https://github.com/TryGhost/Ghost
[3] https://github.com/balderdashy/sails

- The number of commits makes no statement about the nature of the changes.

These points can be demonstrated in a simple scenario: In a web application, ten developers consistently work with the connection to a database. One of them, *Developer A*, has the habit of meticulously committing every line he changes, but ultimately does not write any more code than his co-workers. *Developer A* soon ends up having a higher number of commits than his co-workers, without actually having done any more work and thereby making him any more expendable than the others. Using the number of commits metric in this case would falsely flag *Developer A* as a key person.

One of *Developer A*'s colleagues, *Developer B*, also contributes about the same amount of code as the others and has a normal committing rate. But his work is focused on the integration with web APIs instead of working with the database. No one else is doing work in the same area as him, he alone works on web API integration. Judging by the number of commits alone, *Developer B* will not stand out, because he has the same number as his colleagues, and would thus falsely not be identified by the number of commits metric.

Another approach for finding the Bus Factor of a software project is is presented in [AVH15] and [APHV16]. They use a heuristic for the *Degree of Authorship* (DOA) to determine the author of each source file in a repository. The DOA of a file for a developer depends on who first authored (created) the file, and the amounts of changes by the author in question and other authors. They then calculate the Bus Factor as follows:

> To calculate the [Bus] Factor, we use a greedy heuristic: we consecutively remove the author with [the most] authored files in a system, until more than 50% of the system's files are orphans (i.e., without author). Therefore, we are considering that a system is in trouble if more than 50% of its files are orphans.

In applying this heuristic to 133 repositories across six programming languages they found that 85 of the repositories (64%) had a Bus Factor of one or two. Only ten (7.5%) repositories had a Bus Factor greater than ten.

This approach solves the first of the problems outlined above: In keeping with the example scenario, *Developer A*, who commits every single line he changes, would have no effect on the DOA with this behaviour, since the amount of changes made by him does not differ from that of his colleagues. *Developer B*, on the other hand, who is the only one working with web APIs, would still not be identified as a key person. Assuming that he is considered as the author of ten (web API-related) files and the authorship of 90 other files (across multiple domains) is evenly distributed among his nine colleagues (since they all contribute the same amount of code), the heuristic described above would determine a Bus Factor of five even though the Bus Factor should be one.

So to determine the dependency of a software project on software developers, a more specialized version of the Bus Factor that takes the amount and the expertise

contained in the code written by the developers into account must be considered. The segregation of expertise can be encapsulated with the illustrative term **Islands of Knowledge** (sometimes just called "Island" or abbreviated "IoK" the following). In order to find the Bus Factor of a software project with regards to developer expertise, such Islands of Knowledge must be identified. To carry the metaphor of islands further, the more isolated and bigger an Island of Knowledge is, the more it signifies a low Bus Factor and the more the project in question is vulnerable to the departure of a key developer.

In order find such Islands of Knowledge and to be able to make a statement about the expertise contained in the contributions of a developer, code fragments must be assigned to the programming domains to which they belong. For example, if a developer writes a line of code that retrieves a document from a database, it can be inferred that this line of code belongs to a "Database" programming domain and that the author has some knowledge about the usage of databases. This technique has been used in the past by [Aks15] to identify personnel with expertise in desirable domains with the goal of aiding the process of hiring new programmers. The approach presented in this thesis is fundamentally the same as in [Aks15], but it differs in its technical implementation (the implementation is described in detail in chapter 3).

In the context of this thesis, a tool[4] for finding Islands of Knowledge in Node.js[5] (in the following it will be mostly referred to as "the Tool" or "the Island Finder") has been developed as a proof-of-concept. The Tool employs static code analysis and MSR-techniques (Mining Software Repositories) to extract statistics about the domain-usage by the programmers in git[6] repositories of Node.js software projects. In this thesis, the implementation of the Tool is described, so that it can be evaluated. Furthermore, the Tool is used in a case study to find Islands of Knowledge in multiple popular Node.js projects and the results of these experiments are reported. In summary, the research question of this thesis is:

> How well can a MSR tool find Islands of Knowledge in Node.js projects?

The evaluation of the Case Study results delivers no definitive answer for this question. The Tool, at the stage of writing this thesis, is able to identify concentrations of API uses, but lacks convincing metrics to classify them as dangerous Islands of Knowledge.

The rest of the thesis is structured like this: The following chapter 2 provides a deeper background on many of the concepts like programming domains and MSR that have been mentioned in this Introduction. Chapter 3 describes the design of the Island Finder. Chapter 4 shows how the tool has been applied in a Case Study

[4] https://github.com/turbopope/node-island-finder
[5] https://nodejs.org/en/
[6] https://git-scm.com/

to multiple popular Node.js projects and how these projects were chosen. Chapter 5 presents the results of the Case Study evaluates the Island Finder accordingly. 6 summarizes the thesis, lists threats to validity, and points out some lessons learned and potential future work.

Chapter 2

Background

This chapter serves as an introduction to various concepts used in the remainder of the thesis. It defines some otherwise ambiguous terms and provides the background to some of the used technologies.

2.1 Software Measurements

> [A] measurement is the assignment of numbers to properties of objects or events in the real world by means of an objective empirical operation, in such a way as to describe them[FL84].

In the context of software development, these objects may be artifacts like code and documentation and these events may be processes like the testing phase of a software project. Properties may be, for example, the complexity of the code, the length of the documentation or the duration of the testing phase[Fen94].

The Bus Factor is a property of software projects. Similarly, the existence or number of Islands of Knowledge is also a property of software projects, which can be used to derive the Bus Factor. In order to obtain these measurements, this work proposes specific metrics: One to measure the number of API uses, and one set of metrics to interpret the API use data to find Islands of Knowledge.

2.2 Software Repositories

In the following, software projects will be referred to as "Software Repositories". Although the term can be understood differently, this thesis will use it as a way to describe a version control system that contains both the code itself and the history of the code, including authorship-information and other metadata. This is in keeping with the language used by GitHub, which is centered around the concept of repositories, and the general use in the MSR (see below) community.

2.3 MSR

> The Mining Software Repositories (MSR) field analyzes the rich data available in software repositories to uncover interesting and actionable information about software systems and projects[20117b].

Because this thesis concerns itself with the analysis of software repositories, especially regarding authorship-information obtained from the version control system (VCS), it is firmly placed in the field of MSR. It employs typical MSR-techniques like static code analysis, mining the meta-data of commits and *Identity Management* (see below). It also faces numerous challenges common to MSR, like unparseable code, inconsistent meta-data and not being able to rely on developers following best practices or being consistent in their behaviour.

This work limits itself to examining **git**[1] repositories, because git is the VCS of virtually all software projects on the examined programming platform.

2.4 APIs

An *Application Programming Interface* (API) is a clearly defined set of software artifacts like function or class definitions. They are used by programmers to interface with a software framework or -library and use them to build software. In the following, these frameworks and libraries will be referred to as (external) *modules*. Although there is a distinction to be made between APIs and the modules to which they provide an interface, the two terms will be used interchangeably because the difference has no implications for this work.

Source code is usually littered with API usage[LPS11]. This is especially true for the Node.js ecosystem, where micro packages – with just a few lines of code and oftentimes containing only a single function – are becoming increasingly common[Hol17]. As the use of external modules increases, it also becomes more important to understand how developers use them and how knowledge about them is distributed among the members of a software repository.

2.5 API Domains

The concept of API domains is adopted from [LPS11]. They propose that "each API addresses some programming domain such as XML processing or GUI programming". By assigning an API to one or more programming domains, the APIs can be grouped into areas of expertise. This allows for a more abstract perspective on API usage. Ultimately, this thesis seeks to find Islands of Knowledge about these abstract areas of specialized knowledge.

[1] https://git-scm.com/

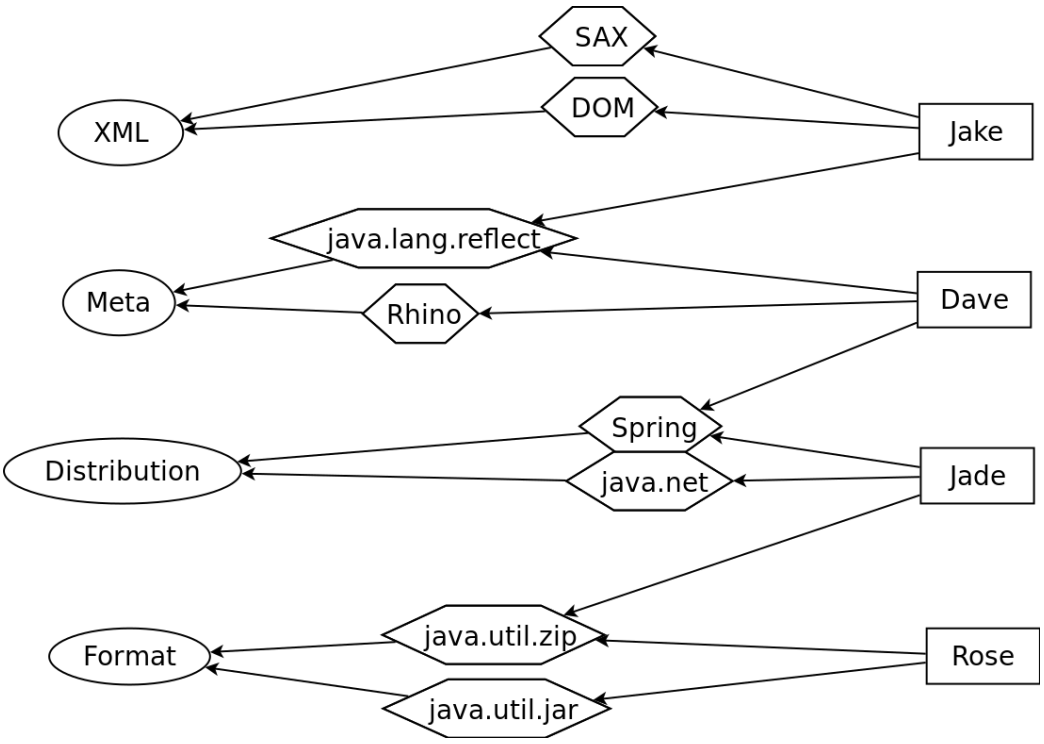

Figure 2.1 Example of an domain-API-developer mapping

There is no standard way on finding the programming domains of an API. The Tool developed in the context of this thesis attempts to solve this problem by mining the meta-data provided for the packages. The approach is error-prone and so the results of the Tool show both the domains and the individual modules which it comprises.

2.6 Developer-API-Domain Mapping

To determine which domains a developer is experienced in, the mapping between APIs and programming domains, and between APIs and their use by developers must be combined. A transitive mapping must be made from the developers over the APIs to the domains (or vice-versa). Figure 2.1 shows an example of such a mapping of four developers (rectangles) to eight APIs (hexagons) which belong each to one of four programming domains (ellipses). Note that this example uses Java APIs and not Node.js APIs, which this thesis actually examines.

The example shows that *Jake* has experience with the APIs `SAX`[2] and `DOM`[3]. Both modules are used to parse XML and can thus be considered belonging to the XML domain. *Jake* also uses Java's reflection API, `java.lang.reflect` and thus has experience with with the meta domain. The API-domain mapping in this example has been retrieved from the catalog[4] from [LPS11].

2.7 Git Identity Management

One major challenge when working with authorship-information from version control systems is inconsistency. Oftentimes, developers use multiple e-mail addresses and handles, for example when working from multiple machines. In the context of this thesis, this lead to difficulties with the assignment of expertise to authors. API uses became assigned to the multiple apparent identities of a developer, which affected the results of the Island Finder, because the expertise was reported as shared between the apparent identities and thus not recognized as Island of Knowledge.

To alleviate this problem, **Idman**[5] was used. Idman is a Perl tool that employs various identity merge algorithms to produce a mapping of apparent identities to real identities. The algorithm used here was `default`. In order to be able to use Idman from the Node.js environment of the Island Scanner, **node-idman**[6], a thin npm package wrapping the original program, was created. To get the identity-merged results from git, the Island Finder produces a `.mailmap`[7] from the output of node-idman for each repository that does not already have such a file. The `.mailmap` maps, line by line, one pair of "wrong" ("commit" in git terminology) identity credentials (email-address and name) to a real ("proper" in git terminology) pair of credentials. If a `.mailmap` file is available, most git tools use it to combine the statistics of the given pairs into those of the proper pair. Since it is not known which pair should be considered as real or preferred by the author, the Island Finder just chooses the one with the most commits according to `git shortlog -sne HEAD`. This is considered sufficient because even if the author in question preferred an other pair of credentials, they will recognize the chosen pair.

Note that the author of this thesis is also one of the authors of Idman.

[2] http://www.saxproject.org/

[3] https://docs.oracle.com/javase/8/docs/api/index.html?org/w3c/dom/package-summary.html

[4] http://softlang.uni-koblenz.de/explore-API-usage/

[5] https://github.com/turbopope/idman

[6] https://github.com/turbopope/node-idman

[7] https://git-scm.com/docs/git-shortlog#_mapping_authors

2.8 Terms For Persons

Throughout this thesis, many terms for the persons and roles in a software repository are used. To avoid confusion, they are defined as follows:

- **Developer**: Any person involved in any way in the development of the software.
- **Programmer**: Usually the same as a developer, but specifically in the context of writing code.
- **Contributor**: Usually the same as a developer, but specifically in the context of contributing commits to a project.
- **Identity**: A "handle" for a real world person. Multiple identities may belong to one real person. An identity may have multiple handles or e-mail addresses (identity artifacts).
- **Author**: git-specific term for a person that wrote a piece of code or a patch.
- **Committer**: get-specific term for the person who applied(committed) a piece of code or a patch to a project.

The author and the committer of a commit are usually the same, but can differ in some cases, for example when a developer applies the patch written by an other programmer.

2.9 git blame

In order to find the author of a line of code, the Island Finder uses **git blame**[8]. It is a tool provided with git to "[annotate] each line in the given file with information from the revision which last modified the line". `git blame` can be instructed to detect movement of code snippets within and between files, and to ignore whitespace changes. It is thereby somewhat protected from being confused by code refactorings.

2.10 ASTs

The analysis of the source code and the extraction of API uses is facilitated by the Island Finder through the use of Abstract Syntax Trees (AST).

[8] https://git-scm.com/docs/git-blame

> An abstract syntax tree (AST) captures the essential structure of the [source code] in a tree form, while omitting unnecessary syntactic details. ASTs can be distinguished from concrete syntax trees by their omission of tree nodes to represent punctuation marks such as semi-colons to terminate statements or commas to separate function arguments. ASTs also omit tree nodes that represent unary productions in the grammar. Such information is directly represented in ASTs by the structure of the tree.[Jon03]

The transformation into an AST makes the source code machine-consumable. The AST-parser used by the Island Finder produces plain JavaScript objects with positional information for each node. Listing 1 shows a simple JavaScript program and listing 2 shows its AST represented as JSON. Each node in the AST also has a (positional) `loc` property as shown in listing 3, which can be used to find its position in the original source code.

```
console.log(1+1);
```

Listing 1 A JavaScript statement

2.11 Node.js and npm Ecosystem

The website of the Node.js Foundation states:

> Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient. Node.js' package ecosystem, npm, is the largest ecosystem of open source libraries in the world[20117c].

Indeed, going by the number of opened Pull Requests, JavaScript is the most popular language on GitHub as of 2016 (although this also includes "traditional" non-Node.js repositories like `jQuery`). The TOIBE Index, an programming language ranking based on search results in popular search engines, lists JavaScript at rank eight[20117a].

In keeping with npm terminology, this thesis will from now on refer to modules as packages. A package can usually contain multiple modules, but this distinction will not be made if it is not necessary (which is usually the case with npm packages, since they tend to be small and only contain one module).

Central to the npm ecosystem is the npm package registry[9]. It stores packages, meta-data about the packages and usage-statistics of the packages. The website

[9] https://www.npmjs.com/

serves as a graphical interface for package exploration, while the npm CLI, installed together with Node.js, usually facilitates interaction with the registry, like searching, displaying and installing packages.

Packages can be installed globally on a machine, meaning that the executables which they provide are available from anywhere in the system. For example, installing the **http-server**[10] with `npm install http-server` provides the `http-server` executable, which can be used to serve any directory in a temporary HTTP server without any configuration.

Packages can also be installed locally, as dependencies for a software project. For this, npm records the package name and the required version in a `package.json` file, together with other information about the project, like its own name and a description. For example, `npm install --save lodash` (notice the `--save` flag) installs the **lodash**[11] package as a local dependency. lodash is a utility package which provides functions for functional programming in JavaScript. Subsequently, every time a developer runs `npm install` in the project, all listed dependencies including the newly required lodash are installed. The package is downloaded together with all other dependencies (including transitive dependencies) to the `node_modules` directory in the project directory, next to `package.json`.

To use a module from a package in a source file, it must be required (imported). Listing 4 shows three possible types of such a requiring. In line 1, the module **lodash** is required. It is an *external package*, installed by npm. It is used in line 5 to find all negative numbers in an array. In line 2, the module **fs** is required. It is a *core module* of the Node.js platform and can thus be required without having to be installed through npm. It facilitates access the file system. It is used in line 6 to read the contents of a file into a variable. In line 3, a `math` module is required. A relative path (starting with `./` or `../`) indicates that it is neither a builtin core module nor from an external package but a **local module**, belonging to the project itself. It is used in line 7 to write out π.

2.11.1 ECMAScript

The JavaScript language follows the ECMAScript Language Specification[Fla06]. Its latest version, as of march 2017, is the ECMA-262[12] 7th edition from June 2016. The specification is backwards-compatible to its previous version. The AST parser used by the Island Finder Tool supports this version of the language[13]. This means, that the Island Finder should be able to parse source code from any version of JavaScript up to this version.

[10] https://www.npmjs.com/package/http-server
[11] https://www.npmjs.com/package/lodash
[12] https://www.ecma-international.org/publications/standards/Ecma-262.htm
[13] https://github.com/jquery/esprima/blob/master/README.md

```
{
  "type": "Program",
  "body": [
    {
      "type": "ExpressionStatement",
      "expression": {
        "type": "CallExpression",
        "callee": {
          "type": "MemberExpression",
          "computed": false,
          "object": {
            "type": "Identifier",
            "name": "console"
          },
          "property": {
            "type": "Identifier",
            "name": "log"
          }
        },
        "arguments": [
          {
            "type": "BinaryExpression",
            "operator": "+",
            "left": {
              "type": "Literal",
              "value": 1,
              "raw": "1"
            },
            "right": {
              "type": "Literal",
              "value": 1,
              "raw": "1"
            }
          }
        ]
      }
    }
  ],
  "sourceType": "script"
}
```


Listing 2 The AST of listing 1

```
{
  "type": "Identifier",
  "name": "log",
  "loc": {
    "start": { "line": 1, "column": 8 },
    "end": { "line": 1, "column": 11 }
  }
}
```

Listing 3 An AST node with positional data

```
1 const lodash = require('lodash'); // External Module
2 const fs = require('fs'); // Core Module
3 const math = require('./lib/math'); // Internal Module
4
5 lodash.find([-2,-1,1,2], n => { n < 0; });
6 const text = fs.readFileSync('text.txt');
7 console.log(math.PI);
```

Listing 4 Requiring and using three Node.js modules

Chapter 3

Island Finder

The Node Islands of Knowledge Finder[1] was developed to provide a mostly automated system for the detection of Islands of Knowledge and thus vulnerability of projects due to a low Bus Factor. Its development was explorative in nature: Since there are no (known) similar tools available, development was mostly based on trail and error and the design presented here is only a best effort. An adequate way to extract the required data from the repositories, a way to judge what constitutes as an Island of Knowledge and a useful method to display the results needed to be found. Although the details of the implementation were unclear at the beginning of the development, the basic approach was clearly defined:

1. Extract API uses from the repository
2. Find the author of the API uses
3. Assign one or more domains to the used APIs
4. Report how the authors use APIs in the domains
5. Apply metrics to API use data to find IoKs

3.1 Components

The sequential nature of the approach described above lends itself to a component-based software architecture with a loose coupling. Many of the components produce some output that is then further processed by other components. This makes development easier because the individual components can be implemented and tested independently from each other.

The components are described in this section, roughly in sequential order. Usually, a component has an internal "library" module containing the majority of

[1] https://github.com/turbopope/node-iok-finder

its business logic, and a script that allows using the component from the command line. One of the components described in this section is the *index* script, which combines most components to produce an API use report for a given repository. Section 3.2 describes the most important artifacts produced by the components.

The repository of the Island Finder on GitHub contains a readme that describes how exactly the scripts are to be used.

3.1.1 analyze

The purpose of the *analyze* component is to count how many times the authors of a repository used which APIs.

3.1.1.1 Input Artifacts

- A git repository

3.1.1.2 Output Artifacts

- API usage statistics as table (CSV)
- API usage statistics as map (JSON)

3.1.1.3 Arguments

As arguments it takes the path to a repository, a whitelist, a path to an output directory, a revision identifier and a string parameter to prepend to the names of the output files. The component requires that the repository to be analyzed is available on the local file system, i.e. it needs to be cloned to the file system beforehand. The whitelist is an array of paths of directories, relative to the given repository, which should be analyzed (recursively). If no whitelist is provided, no whitelist is applied, although *analyze* will still try to infer which files in the repository are source files and should be processed (see below). The revision-identifier is a "tree-ish"[2] that tells the component which version of the repository to analyze. This could for example be `master` or a commit sha-1 hash.

3.1.1.4 Operation

1. Check out the target revision.
2. Select files to analyze:

[2] http://stackoverflow.com/a/18605496

(a) Use `linguist`[3] to list JavaScript source files in the repository. Linguist is the library used by GitHub to detect the language of source files and display the language statistics of a repository. It excludes binary files, vendored files (foreign library files that have been committed to the repository) and generated files. It is a heuristics-driven approach that does not claim perfect accuracy, but its results are reasonable for most repositories.

(b) Further sort out files that do not match the whitelist (if provided). Files with paths starting with a path from the whitelist are kept, others rejected.

(c) Furthermore, reject files that do not contain the string `require` within the first 1000 characters. In virtually all Node.js source files the require statements can be found at the top of a file. If a file contains no requires it need not be analyzed because no modules are used.

3. For each file, retrieve the Abstract Syntax Tree. If the file has been analyzed before, load its AST from a cache, else parse the file with `esprima`[4]. Ignore files that cannot be parsed.

4. Instantiate a tree walker that walks over all nodes of the AST. Register observers for relevant node types. Walk the tree and call the observers when visiting nodes of their type:

   - **VariableDeclarator:** Nodes of this type are assignments of a value ("init") to a variable. Check if the init uses the `require` identifier. If yes, record the variable name and the string argument of the `require` call (the module name). Listing 5 shows a collection of such `VariableDeclarator`s.
   - **Identifier:** Nodes of this type represent for example variable names. Get the line number of the line where the identifier was used in the original source file and call `git blame -w -M -C` to determine the author of the line (who presumably used the identifier). Record the identifier name and the author. If a record exists (i.e. if the identifier has been used by the author in a previous node), increment its count. The `blame` flags specify that whitespace-changes and moving lines within and across files should not credit the changing but the original author of the lines. This is important because such refactoring changes attest much less expertise than the original writing of the code.

5. Post-process the recorded requires and identifier uses:

[3] https://github.com/github/linguist
[4] https://github.com/jquery/esprima

(a) Remove all requires that require an internal module. It is assumed that Islands of Knowledge for internal code is not dangerous enough to merit further analysis.

(b) Remove all identifiers that do not have a corresponding require recorded. This excludes uses of functions and variables that are declared in the file itself instead of having been imported from a module. It still retains uses of global objects which don't need to be required before they can be used (examples are `Array`, `JSON` or `Date`).

(c) Map the list of identifiers to the name of the module which it represents.

6. Combine the counts of identifier uses by authors from all files. This results in a data-structure like shown in a JSON representation in listing 7.

7. Save the resulting API usage statistics in a JSON file and a CSV file.

```
const fs1 = require('fs'); // With `const`
let fs2 = require('fs'); // with `let`
var fs3 = require('fs'); // with `var`
const readFileSync
    = require('fs').readFileSync; // Partial require
const util = require('./lib/util'); // Internal module
const chai = require('chai'); // External module
```

Listing 5 A collection of `VariableDeclarator`s with a `require` statement

3.1.2 keywords

The purpose of the *keywords* component is to count the numbers of times a keyword has been used across all packages in the npm registry.

3.1.2.1 Input Artifacts

- A list of packages and their keywords (JSON)

3.1.2.2 Output Artifacts

- A list of keywords and their frequencies (JSON)

3.1.2.3 Arguments

It takes no arguments, but requires that a list of packages and their keywords is available in `out/npm_keywords.json`.

3.1.2.4 Operation

Each package in the npm registry must contain a `package.json`[5]. The file contains a JSON object with data about the package. Among them is the `keywords` field. The documentation states the following about the field:

> Put keywords in it. It's an array of strings. This helps people discover your package as it's listed in npm search.

For example, the `keywords`-field of `left-pad` is shown in listing 6.

```
{
  "...": "...",
  "keywords": [
    "leftpad",
    "left",
    "pad",
    "padding",
    "string",
    "repeat"
  ],
  "...": "..."
}
```

Listing 6 Excerpt of **left-pad**'s `package.json`, showing its keywords

The *keywords* component iterates over the packages listed in the input file and increments a counter for every package. Keywords are normalized to lowercase to avoid multiple entries for keywords that just differ in their capitalization. The component then outputs a list of keywords and their counts, sorted by their counts.

The exact procedure to acquire the input file with the package names and their modules is described in the readme of the Island Finder. In summary, it must be retrieved from the CouchDB database[6] of the npm registry.

An excerpt of the output is shown in listing 8.

3.1.3 modulesToKeywords

The purpose of the *modulesToKeywords* is to map the modules in the output of the analyze component to a domain.

[5] https://docs.npmjs.com/files/package.json
[6] https://couchdb.apache.org/

3.1.3.1 Input Artifacts

- API usage statistics as table (JSON)
- A list of keywords and their frequencies (JSON)
- A list of Node.js builtin modules and their programming domain (JSON)
- A list of Node.js global objects and their programming domain (JSON)

3.1.3.2 Output Artifacts

- API usage statistics with the APIs resolved to domains (CSV)
- A "backwards map" of domains and modules which belong to them (JSON)
- A map of modules in the repositories to their keywords and their frequencies (JSON)

3.1.3.3 Arguments

As arguments it takes a path to a CSV file of the API usage statistics from *analyze*, a string parameter to prepend to the names of the output files and optionally a path to the list of keywords with their frequency statistics produced by *keywords*.

3.1.3.4 Operation

1. Copy the row with the module names for later.
2. Iterate all module names:
   (a) If it is in the list of global objects (listing 9), continue with the programming domain listed there. The programming domains have been assigned manually.
   (b) Else, if it is in the list of built-in modules (listing 10), continue with the programming domain listed there. Like with the global objects, the programming domains have been assigned manually.
   (c) Else, look up the module's keywords in the npm-registry (the readme in the Tool's repository explains how to set up a local CouchDB replicate of the registry, which is used by *modulesToKeywords*). Then, continue with the most frequently used keywords as determined by the input statistics file.
   (d) If the module is not found, continue with just the module name as domain name.

3. If a domain or keyword has been found, replace the module name with the domain or keyword in uppercase.

4. Write out a new table. The first row has the module names replaced by the modules most frequently used keyword or their manually assigned programming domain. The second row contains the original module names. The usage statistics remain unchanged. Table B.2 shows an example of the resulting table.

5. Write out a JSON file with a map back from the chosen programming domain or keyword to the module name. This information is also contained in the table from the previous step, but it is easier to process this file. Listing 11 shows an example of the resulting "backwards map".

6. Write out a JSON file with a map from the module names to all of the module's keywords. This is only done with module names that have been looked up in the registry. Listing 12 shows an example of the resulting map.

3.1.4 condense

The purpose of the *condense* component is to remove the original module names from the table that *modulesToKeywords* generates and to combine the usage statistics of the programming domains.

3.1.4.1 Input Artifacts

- API usage statistics with the APIs resolved to domains (CSV)

3.1.4.2 Output Artifacts

- API usage statistics with the APIs resolved to domains and the counts added together (CSV)

3.1.4.3 Arguments

As arguments it takes the path to the file that should be condensed and optionally a path were the resulting table should be written.

3.1.4.4 Operation

1. Remove the line with the module names from the raw CSV from the input file.

2. Parse the CSV. This automatically combines columns with the same header (key) by summarizing the values of their cells because the table library used

works like a Hash-Map with two-dimensional keys and does not allow a key to exist multiple times (collisions of cells are handled by adding their contents together).

3. Sort the rows of the table in descending order by the sum of the usages in the row. The rows represent an author's usage-statistics of the programming domains. By sorting this way, the statistics of the programmer with the most usages overall will occupy the first row, followed by the programmer with the second-most usages and so on.

4. Write the new table out. Listing B.3 shows an example of the resulting table.

3.1.5 report

The purpose of the *report* component is to generate a human-readable report to present the results of the other components and to guide manual Island of Knowledge identification.

3.1.5.1 Input Artifacts

- API usage statistics with the APIs resolved to domains (CSV)
- API usage statistics with the APIs resolved to domains and the counts added together (CSV)
- A "backwards map" of domains and modules which belong to them (JSON)
- A map of modules in the repositories to their keywords and their frequencies (JSON)

3.1.5.2 Output Artifacts

- A HTML report document (HTML)

3.1.5.3 Arguments

report takes a path to a directory with the intermediate artifacts of the other components for a repository and a path to an output directory.

3.1.5.4 Operation

1. Read and parse the output files of other (previous) components:
    - The condensed usage statistics (from *condense*)
    - The uncondensed usage statistics (from *analyze*)

- The "backwards map" from programming domains to modules (from *condense*)
- The map from the modules in the repository to their keywords and their frequencies (from *condense*)

2. Remove all authors with less than 25 API usages because at lower numbers no expertise should be attributed.
3. Remove all modules with less than 25 API usages because at lower numbers no expertise should be attributed.
4. Produce a "heatmap" table analog to the condensed usage statistics table. Each cell in the heatmap corresponds to a cell in the statistics file and contains the percentage of the author's domain uses versus the total number of uses of that domain. For example, if there were 100 usages of APIs/modules from the *JSON* domain, and *Developer A* used APIs/modules from the domain in 75 instances, the heatmap would contain a "heat value" of 75% for *Developer A* in domain *JSON*.
5. Render and write a report HTML document to the output directory. See 3.2.3 for a detailed description of the report document.

3.1.6 index

The purpose of the *index* component is to orchestrate the other components in order to automate the process of generating a report for a repository.

index takes as arguments the path to the repository to analyze, a path to a working/output directory, a revision identifier and a whitelist. The last two arguments are the same as for *analyze* and are directly passed to it.

1. Execute *analyze*
2. Execute *modulesToKeywords*
3. Execute *condense*
4. Execute *report*

The input and output files of all components are read from/written to the specified output directory.

3.1.7 scan_for_islands

The purpose of the *scan_for_islands* component is to scan the given usage statistics for Islands of Knowledge, according to some metrics which are implemented inside of it. It was developed in conjunction with the Case Study described in chapter 4.

3.1.7.1 Input Artifacts

- API usage statistics (domains or modules) (JSON)
- A "backwards map" of domains and modules which belong to them (JSON, optional)

3.1.7.2 Output Artifacts

- A JSON file with found Islands of Knowledge (JSON)

3.1.7.3 Arguments

- `module_threshold`: Modules with less uses than this threshold get removed before scanning (defaults to 25)
- `author_threshold`: Authors with less uses than this threshold get removed before scanning (defaults to 25)
- `portion_threshold`: How many percent of uses in a module/domain constitute an IoK (defaults to 0.75)
- `carry_threshold`: How many percent more the author with the most domains needs to have than the author with the second-most uses to form an Island of Knowledge (defaults to 0.75)

3.1.7.4 Operation

1. Read and parse the output files of other (previous) components:
   - The usage statistics
   - The "backwards map" from programming domains to modules, if given (from *condense*)
2. Remove all authors with less than `author_threshold` API useges.
3. Remove all modules with less than `module_threshold` API usages.
4. For each domain or module determine the total number of uses and if any authors contributed more than `portion_threshold` percent of these uses. Return such authors as a found Island of Knowledge.
5. For each domain or module determine the author with the most uses (the *carry*) and the author with the second-most uses. Return the *carry* author if he has more than `carry_threshold` more uses in the domain or module than the author with the second-most uses.

The resulting data-structure is described in subsection 3.2.4.

3.2 Artifacts

The components described above produce and consume various artifacts. Some of them have a more auxiliary nature and have been described in place. This section describes the four main artifacts. The term *artifacts* was chosen deliberately to allow viewing data independent from its "physical" representation (e.g. files).

3.2.1 Usage Statistics

Table B.1 shows an example for the API Usage statistics for a repository in the form of a table. Essentially, each cell in the table shows how many times each author has used each module that was used at least once in the repository. If an author has never used an API, a default value of zero is listed.

Listing 7 shows the same example in the form of a json object. The default value is omitted here.

The basic principle of the Island Finder is producing a transitive mapping between authors, APIs and programming domains. This artifact represents the first step of this mapping: The mapping between authors and APIs.

3.2.2 Condensed Usage Statistics

Table B.2 shows an example for the condensed API usage statistics for a repository in the form of a table with two header rows. The table is a processed form of the usage statistics table, with the only difference of an added first row where each cell contains the programming domain of the module which the column represents.

In the mapping between authors, APIs and programming domains, this artifact now contains the complete mapping. But still, the data is hard to process for humans, and so further processing for easier representation is required.

3.2.3 Report

The report is the final product of the Node Island of Knowledge Finder tool. It is an interactive HTML document and it is therefore not practicable to show include it in this thesis document. In lieu thereof, several example reports are available at `https://turbopope.github.io/node-iok-finder/examples/`.

Because it is impossible to find objective criteria for Islands of Knowledge, the report document makes the data produced by the Island Finder available for interpretation through a human (for example a developer in the repository or a client looking for trustworthy projects). The chosen representation is still very close to the original data, but makes the data more immediate and obvious for the interpreter.

The central section of the report document is a modified version of the condensed API usage statistics table. Like the statistics table, it has the programming

domains as column headers, the authors as row headers and the cells contain the number of API uses in a domain by the author. The background color of the cells depends on the value of the corresponding cell in the heatmap and lies on a spectrum between white (`#FFFFFF`) if the cell represents 0% of the uses in the domain and red (`#FF0000`) if the cell represents 100%. Hovering over a cell displays the percentage-value of the heatmap in a tooltip. Hovering over a domain name in the column headers shows a tooltip with the modules in this domain (from the "backwards map"). Clicking on a domain name expands the table to show the usage statistics for the individual modules in this domain.

Below that in the next section, another table shows the complete "backwards map", i.e. the programming domains and the modules which they comprise. Hovering over a module in the table opens a tooltip with all its keywords and its frequencies, if it is an external (npm) module.

Additionally, the report document contains some help texts for the interpreter.

3.2.4 Scan

Listing 15 shows an example for the output of the `scan_for_islands` component. The output artifact contains a list of Islands of Knowledge found by the **portion** metric (`islandsByPortion`) and a list with IoKs found by the **carry** metric (`islandsByCarry`). All islands have a `location` , i.e. the domain or module in which they lie, a `remap` , i.e. which modules make up the domain in `location` (for modules, the list just contains the module again), a field for the offending author (`author` or `carryAuthor`) and a field for the actual metric value (`portion` or `carryToSecond`). Islands by the portion metric also exhibit the total number of uses and the uses by the author in questions. Islands by the carry metric additionally exhibit the uses of the carry author, and the name and the uses of the author with the second-most uses (`secondAuthor`). See chapter 5 for a more in-depth description of the metrics.

While some of these fields are not strictly necessary, providing the information aids in further processing (for example to exclude islands with small uses values) and displaying the found Islands in an understandable manner.

Chapter 4

Case Study

In order to find the answer to the research question ("How well can a MSR tool find Islands of Knowledge in Node.js projects?"), the Island Finder Tool described in chapter 3 was tested in a case study on a number of npm packages. The Case Study comprised two steps: First applying the Island Finder to find Islands of Knowledge in the subject repositories and then conducting a survey with the maintainers of the subject repositories.

Since the Island Finder was primarily designed to provide raw usage data and not definitive answers, the Case Study required defining a metric for Islands of Knowledge first. Since there are no comparable studies and there are no generally accepted criteria as to what constitutes an Island of Knowledge, any metric defined here is explorative in nature and any classification based on them is inherently debatable. Still, they are a best effort and may start a discussion about more complex metrics.

4.1 Metrics

After measuring the numbers of uses by authors in modules and domains with the *analyze* component, the measurements must be classified to determine if they represent an Island of Knowledge or not. Many metrics could be applied to the measurements, but for simplicity, the Case Study limits itself to two metrics and evaluates their utility. Each metric evaluates for each measurement (the number of uses by an author in a module or domain) if it represents an Island of Knowledge within this module or domain:

- **portion:** A measurement represents an IoK if it represents more than threshold t_p percent of all uses in that module or domain.
- **carry:** A measurement represents an IoK if the author (the *carry*) with the most uses has threshold t_c more uses than the author with the second-most uses in that module or domain.

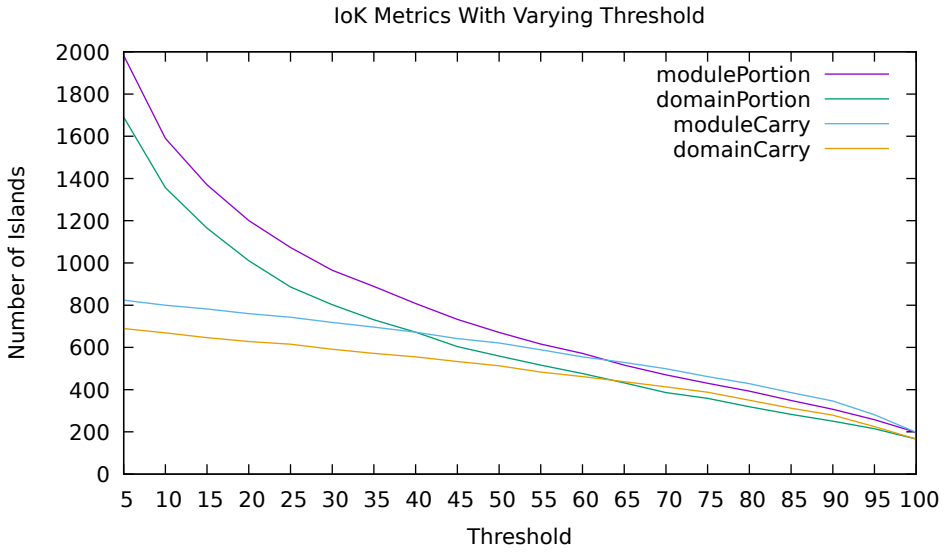

Figure 4.1 Portion and Carry metric with varying thresholds for domains and modules

Both thresholds t_p and t_c are percentages. Figure 4.1 shows the resulting number of Islands of Knowledge from applying both metrics with varying thresholds to the usage statistics of the 178 most popular npm packages (see 4.2), for both domains and modules. It is evident that observing domains vs modules makes only a proportional difference, since the graphs for these metrics run "in parallel". Also, it is noteworthy that while the number of found Islands of Knowledge the portion and the carry metrics is very different for low thresholds, their graphs converge at a threshold of about 65%. This suggests that both metrics are equally suited to find Islands of Knowledge at threshold levels over 65%. Indeed, manual inspection of the data[1] confirms that both metrics report largely the same Islands. Only in rare cases one metric reports an Island that is not reported by the other. This is true for both domains and modules.

To determine the thresholds, a manual inspection of the reports for the ten repos with the most total uses in the subject repositories has been executed. Table 4.1 shows those ten packages.

The following subsection lists striking outliers in the usage data, which will be used as a basis to justify threshold determination. Note that all data shown here has been extracted from publicly available sources. It is intended for purely analytical purposes and in no way poses any personal criticism at the authors.

[1] https://github.com/turbopope/npm-iok-study/blob/master/scans/65.json

Table 4.1 The ten npm packages with the most uses

uses	repoName
4393	wp-calypso
4532	sails
4552	bower
4805	jsdom
4983	webpack
6222	bluebird
6224	eslint
9811	npm
12792	Ghost
17545	mongoose

4.1.1 portion

In **mongoose**, developer Valeri Karpov (who has the most uses in total) has 81% of all uses in the `PROMISE` domain. Valeri Karpov also has 63% of all uses in the `DATA` domain and 65% of all uses in the `MODULE` domain.

In **Ghost**, **npm** (as package) and **eslint**, there are no huge outliers.

In **bluebird**, in almost all domains, the uses by Petka Antonov make out 74% or more. Similarly, in **webpack** most uses by Tobias Koppers make out 63% or more.

In **jsdom**, 95% of all uses in the `TEST` domain are from Joris van der Wel. In the same package, 62% of all `ERROR` uses are by Domenic Denicola.

In **bower**, 70% of all `MODULE` uses, 68% of all `JSON` uses, 68% of all `CORE` uses, 78% of all `ERROR` uses and 62% of all expect.js (which has no domain) are by Andre Cruz, while 71% of all `TEST` uses are by Adam Stankiewicz.

In **sails**, 64% of all `PROCESS` uses, 62% of all `IO` uses, 82% of all `TEST` uses, 79% of all `JSON` uses, 100% of all `TEMP` , 87% of all request (no domain) uses and 95% of all `FILESYSTEM` uses are by Scott Gress, while 68% of all `MODULE` uses, 83% of all `LOGGING` and 100% of all `CLI` uses are by Mike McNeil.

In **wp-calypso**, the uses in most domains are to more than 77% by Andy Peatling.

In conclusion, most of these outliers seem to fall above 65%, so t_p is set to 65%.

4.1.2 carry

In **mongoose**, Valeri Karpov has 72% more uses in `CORE` than Christian Murphy, 70% more uses in `ERROR` than Constantine Melnikov and 90% more in `PROMISE` than Constantibe Melnikov (the latter was also recognized by the portion metric).

In **Ghost**, Hannah Wolfe has 67% more uses in `TEST` than Katharina Irrgang.

In **npm** and **eslint** there are no huge outliers.

In **bluebird**, Peteka Antonov has 94% or more uses more than the author with the second-most uses in most domains, which is also true for most domains in **webpack** for Tobias Koppers at above 77% and **bower** for Andre Cruz at above 69%.

jsdom has no huge outliers.

In **sails**, Scott Gress has 88% more `TEST` uses than Mike McNeil. These potential Islands was also identified by the portion metric.

In **wp-calypso**, like before, Andy Peatling in most domains has more than 76% more uses than the developer with the second-most uses.

In conclusion, the carry metric may reveal some Islands of Knowledge which are not discovered by the portion metric, but also seems to miss some. Thus both should be used in combination. A reasonable threshold for the carry metric is set at $t_c = 70\%$.

4.2 Subject Repositories

In order to evaluate test the Island Finder, the case study applies it to a number of repositories of popular npm packages.

The npm website allows its logged-in users to star packages which they like. To ensure that the analyzed repositories are relevant to the npm community and because it was expected to yield easily processible results, it was decided to use this meta-information to find and analyze the most starred packages. The list of repositories was automatically scraped off the npm website[2] with a Node.js script[3]. The result is shown in listings 13 and 14, together with the revision at which the repository was analyzed. Each page on the npm website lists 36 packages. It was decided to scrape 5 pages for a total of 180 packages, i.e. the 180 most-starred packages were retrieved for analysis. Of these 180, for two packages, no git repository could be found, so 178 repositories were retrieved for analysis in total.

After retrieving the names and URLs of the repositories, they were cloned (or updated on subsequent runs) to a local filesystem and their `master` branch checked out. This process was also carried out by a script[4].

Table 4.2 shows the statistics of the repositories in the Case Study.

4.3 Island Finder Application

After cloning the subject repositories to the local file system, the *index* component (described in section 3.1.6) of the Island Finder was run automatically on each

[2] https://www.npmjs.com/browse/star

[3] https://github.com/turbopope/npm-iok-study/blob/master/starScraper.js

[4] https://github.com/turbopope/npm-iok-study/blob/master/update.js

Table 4.2 Statistics of the repositories in the Case Study. The numbers of files and SLOC were determined with **cloc** (https://github.com/AlDanial/cloc)

Number of Repositories	178
Total number of Files	60,902
Number of JavaScript Files	31,980
Total SLOC	6,527,202
JavaScript SLOC	3,672,040

one. The resulting artifacts were saved for later use. The script that was used to run `index` on all repos can be found at https://github.com/turbopope/npm-iok-study/blob/master/index.js.

Since the `index` component itself does not include any interpretation of the data, subsequently the `scan_for_islands` component was used to find Islands of Knowledge in the 178 subject repositories. The thresholds determined in 4.1 were used. The script for running the `scan_for_islands` component on all target repositories can be found at https://github.com/turbopope/npm-iok-study/blob/master/scan.js.

For the `author_threshold` and the `module_threshold` parameters, the somewhat arbitrary default value 25 was used. Any islands with less uses are considered insignificant, because less than 25 uses of an domain/module hardly constitutes expertise.

The results of the Islands Finder application are shown in chapter 5:

4.4 Survey

In order to evaluate the findings of the Island Finder, the developers of some of the subject repositories that had Islands of Knowledge were contacted. They were shown the results of the Island Finder (domains with `portion` metric) and given a short survey with four questions. The questions were:

1. "Do you think that the listed developers have most of the knowledge of the listed programming domains?"
2. "Do you think that `package_name` may be in trouble if the listed developers leave the project?"
3. "Does `package_name` have some characteristics such as detailed documentation so that the loss of the listed developers would be less troublesome?"
4. "Do you think that the assignment of modules to domains shown above is reasonable?"

The full pug[5] template used to build the surveys can be found at https://github.com/turbopope/npm-iok-study/blob/master/email_survey/proto.pug.

Due to the time constraints of this thesis, it was not possible to send the survey to all subject repositories. It was decided to send it only to interview the developers of the 50 most-starred packages with islands.

In a first round, the repositories of the top 10 most starred packages in the npm registry were contacted. All repositories were hosted on GitHub, so it was decided to use GitHub's issue tracker system to open one issue in each repository. The purpose of doing such a pilot was to detect any problems with the survey early on, for example to see if the questions could be misunderstood and needed rephrasing.

Although the issues immediately received some interesting feedback, it became apparent that use of the issue tracker system violates GitHub's Terms of Service. In the ensuing discussion, a support employee wrote:

> Your account was flagged for opening issues on other users' repositories for the purposes of drawing attention to your research project, which is prohibited by our Terms of Service.

Indeed, the Terms of Service contain a section about advertisement:

> Advertising Content, like all Content, must not violate the law or these Terms of Use, for example through excessive bulk activity such as spamming. We reserve the right to remove any advertisements that, in our sole discretion, violate any GitHub terms or policies.[6]

Thus, the issue tracker strategy was abandoned.

Instead, it was decided to write emails with the survey to the three most active developers in the remaining projects. The developers were determined with `git shortlog -sne`, which lists the number of commits, the name and the email-address of a repository's authors, sorted in descending order by number of commits. `git shortlog` respects the `.mailmap` of repositories, so the `mailmap`s generated by the Idman tool in the preceding *index* run could also be applied here to curate the list of authors and avoid duplicate mails to real persons with multiple developer identities. From this list, the first three authors with a valid email address and more than 100 commits were selected.

Then, a Googlemail[7] address was created and used with nodemailer[8] to send the emails. The used script can be found at https://github.com/turbopope/npm-iok-study/blob/master/email_survey/send.js.

[5] https://pugjs.org/api/getting-started.html

[6] https://help.github.com/articles/github-terms-of-service/#i-advertising-on-github

[7] https://www.google.com/gmail/about/

[8] https://nodemailer.com/about/

The results of the survey are shown in chapter 5.

Chapter 5

Case Study Results

In chapter 4, the application of the Island Finder and the subsequent survey of developers in the affected repos was described. This chapter presents the results of both steps.

5.1 Islands Finder Application

All following statements are derived from the result of the scan script[1] with the thresholds determined in chapter 4. The raw data from the scan script is also available in the repository of the case study[2].

For the evaluation, **jq**[3] was used. jq is a command-line JSON processor. It can be used for example to select specific fields from JSON objects or to filter JSON arrays. Using jq usually requires an input JSON string or file and a "filter" written in the jq language, the simplest of which is `.`, which just returns the unmodified input.

The statements below all relate to the application of one or more jq-filters to the raw data from the scan script. In order to make the results reproducible, the filters are listed line-by-line in listing 16 and the statements refer to the line numbers of related jq-statements in round brackets. If two or more filters are listed, their results have been combined in some trivial way.

In the 178 (2) subject repositories, there were in total 432 programming domains with Islands of Knowledge by the `portion` metric, 413 programming domains with IoKs by the `carry` metric, 527 modules with IoKs by the `portion` metric

[1] https://github.com/turbopope/npm-iok-study/blob/master/scan.js

[2] https://github.com/turbopope/npm-iok-study/blob/master/result_65_70.json

[3] https://stedolan.github.io/jq/

and 506 modules with IoKs by the `carry` metric (1). This means that in the best case, on average each repository had 2.3 Islands of Knowledge (1, 2).

Of the 178 subject repositories, only 106 or 59.6% had domains with Islands of Knowledge by the carry metric (2, 3). Only 77 or 43.3% had two or more Islands and 57 or 32% had three or more Islands (2, 3).

The exact number of domain/ `carry` Islands of Knowledge for each repository is shown in table 5.1 (4).

When examining how the Islands are distributed among a repository's authors, one feature of the data is particularly striking: If a repository has Islands, they all tend to belong to the same author. Upon closer examination, (5) reveals that, going by domain islands and the carry metric, all Islands in 91 of 178 repositories belonged to only one author. Only in 15 repositories the Islands belonged to two or more authors. A similar distribution is exists for module Islands (6).

One possible explanation for this peculiarity is the fact that npm packages tend to have one main contributor who writes a majority of the code. Comparing the total sum of uses in a repository with the highest sum of uses in Islands of Knowledge by a single author (domain/ `carry`) for each repository supports this notion: For example, in **wp-calypso** 1968 of 4393 (45%) of all uses in Islands of Knowledge came from one single author (7). On average across all repositories with at least one Island, 46.87% of all uses in Islands (domain/ `carry`) came from the author with the most uses in Islands (8).

Another peculiarity is the recurrence of the `TEST` domain in the data. A ranking of the most frequent domains in the Islands (9, using the `carry` metric) shows that the domain is indeed the location of most Islands of Knowledge (60 Islands). The `MODULE` domain ranks second with 43 Islands and the `IO` domain third with 34 uses. The top ten domains are, in descending order by number if Islands:

`TEST` (60), `MODULE` (43), `IO` (34), `CORE` (33), `PROCESS` (27), `ERROR` (27), `UTIL` (15), `LOGGING` (15), `JSON` (14) and `DATA` (12).

These are all domains that have been used for builtin modules and global objects (cf. chapter 3, and listings 9 and 10), which is not surprising, because these can be expected to be used most uniformly by Node.js developers.

For modules, the 15 most frequent are:

`exports` (36), `assert` (31), `module` (28), `Error` (27), `process` (24), `path` (24), `Object` (22), `fs` (20), `console` (16), `JSON` (12), `chai` (10), `should` (9), `chalk` (9), `Array` (9) and `Date` (8).

Besides `chai` and `chalk`, this list similarly to the domains mostly contains builtin modules and global objects.

Table 5.1 domain/ `carry` Islands of Knowledge in each subject npm package

0	Font-Awesome, TypeScript, UglifyJS2, angular, autoprefixer, aws-sdk-js, axios, backbone, bootstrap, chalk, classnames, coffeescript, colors, colors.js, commander, commander.js, cordova-cli, cors, cross-env, d3, del, dotenv, esprima, forever, grunt-cli, grunt-contrib-cssmin, grunt-contrib-jshint, grunt-contrib-uglify, gulp-autoprefixer, gulp-babel, gulp-changed, gulp-connect, gulp-htmlmin, gulp-if, gulp-imagemin, gulp-inject, gulp-less, gulp-livereload, gulp-load-plugins, gulp-plumber, gulp-rename, gulp-replace, gulp-uncss, gulp-watch, html-webpack-plugin, http-server, javascript, koa, left-pad, lodash, marked, n, node, node-csv, node-jsonwebtoken, node-mkdirp, node-sanitize-filename, node-schedule, node-semver, node-supervisor, node-uuid, node-xml2js, phantomjs, react-router, redux, request-promise, standard, statsd, stylus, through2, underscore, watchif
1	async, co, compression, connect, cookie-parser, debug, ejs, gulp, gulp-concat, gulp-jshint, gulp-rev, gulp-sass, gulp-sourcemaps, gulp-util, jquery, jshint, less, minimist, mocha, morgan, node-mime, node-optimist, q, run-sequence, sequelize, shelljs, superagent, supertest, validator.js
2	chai, cheerio, eslint, gm, grunt-contrib-watch, handlebars, helmet, js, node-bunyan, node-formidable, node-fs-extra, node-http-proxy, node-sass, npm-check, qs, rimraf, session, shortid, socket, socket.io
3	Ghost, body-parser, gulp-uglify, gulp-useref, joi, multer, node-glob, npm, npm-check-updates, npm-run-all, react, request, should, tape, winston, yo
4	Inquirer, atom, express, grunt, jsdom, log4js-node, meteor, moment, mongoose, mysql, nodemon, openmct, passport, ws
5	gulp-notify, json-server, node-postgres
6	node-mssql
7	jade, js-xlsx, karma, node-restify, node_redis, pug, sails, sinopia, vue
8	nodemailer, webpack
9	selenium
10	istanbul
11	bower, hapi, ionic-cli, keystone, pencilblue
12	browser-sync
13	bluebird, wp-calypso
14	hexo, pm2

5.2 Survey

In combination, eleven answers were collected from the first round of issues and the second round of emails. A lot of emails did not receive an answer or had an answer that ignored the questions. Some emails could not be delivered in the first place, which is not surprising because even though some filtering for valid emails was done, there is no guarantee that the addresses from the git metadata are real.

Note that the verbatim responses will not be made publicly available in order to protect the privacy of the developers.

In order to quantify the answers to the questions from the survey, they have been manually categorized as *positive*, *negative* and *undefined*.

For the question "Do you think that the listed developers have most of the knowledge of the listed programming domains?" five answers were positive, five answers were negative and one was undefined.

One developer answered the question with "Yes, $*developer* knows the code far better than me and $*other_developer* and has written most of it." Another developer wrote (referring to an Island in domain `TEST` with module `assert`) "The domain listed is very generic, but yes, I believe $*developer* is perfectly at ease in the matter." Another Developer strongly disagreed: (regarding the same Island) "No. This seems like a complete miss." Another answer was (referring to an Island in domain `TEST` with modules `tape, assert`) "No, everyone knows how to use an assert library".

While it is hard to extrapolate a definitive assessment from the answers, their mixed tenor suggests that the Island Finder as used in the Case Study can not always tell if a developer has the most knowledge in a domain. The last quote also suggests that even though an Island is found in the code, it does not necessarily signify a low Bus Factor, because developers in a project can have expertise in a domain even though they do not work with that domain in that project.

For the question "Do you think that `repo_name` may be in trouble if the listed developers leave the project?" two answers were positive, eight answers were negative and one was undefined.

For this question, an interesting insight came from this answer: (Referring to an Island in domain `TEST` with modules `should` and `assert`) "This developer has already left the project. His leaving the project did not have a huge impact. This is relative, because $*package* has been 'in trouble' for a long time." So even though the package had problems, the departure of an developer with an Island of Knowledge did not incapacitate the project. Another answer also asserted the risk for the project as low: "No, the project has [eight] developers and it should be easy to add new ones if necessary." The two positive answers were not too convincing either. One developer wrote (referring to an Island in domain `TEST` with module `assert`) "Yes. He's the maintainer now. I'm doing nothing. But the assert module is irrelevant to that ;)". This suggests that even though the Island Finder correctly identified a critical maintainer, it was for the wrong reason. The other positive

answer was: (Referring to Islands where they were the author) "yes, seeing as how its mainly driven by us". This reply came from an email address that, like the message, suggests that there are multiple developers sharing the address when committing to the project. The Island Finder of course is not able to distinguish the developers in this case and thus even the confirmatory answer can not be fully considered a success.

For this question, the image emerging from the case study is clear: The contacted developers do not generally think that the existence of Islands of Knowledge necessarily means that the project is in danger.

For the question "Does `repo_name` have some characteristics such as detailed documentation so that the loss of the listed developers would be less troublesome?" six answers said that there were redeeming characteristics, two said that there were none and three answers were undefined.

From the answers, it appears that generally for developers the most important such characteristic is documentation ("we certainly have documentation for most of the features.", "The docs and tests are great"). Some also mentioned the community or tests ("Yeah, pretty good docs and test suite. So other people could somewhat easily contribute."), code comments or wiki entries ("There's very little internal documentation aside from code comments and wiki entries, a portion of which are out-of-date.").

For the question "Do you think that the assignment of modules to domains shown above is reasonable?" four answers were positive, three answers were negative and four answers were undefined. The question did not seem to spark much discussion, most answers were some variation of "yes" or "no".

All in all, these results are very mixed. On one hand, there are quite some negative answers from developers. On the other hand, some seem to agree with the findings. In conclusion, the Island Finder seems to be able to detect vulnerabilities through Islands of Knowledge at least to some extend. As one developer put it: "Bus factor is potentially $*package*'s greatest challenge at the moment. If you recognize this simply from your algorithm, then you are doing something right."

5.2.1 Additional Remarks

Almost all answers contained some additional remarks with useful insight into how the developers perceived the Case Study.

One confirmatory answer to the second question (whether the project would be in trouble if the developer left) was: "[...] Right now he is the one handling the project. [It] wouldn't be in trouble as it's a stable project but in the future it would probably be noticed." This suggests that looking at the current state alone may not be enough to judge the vulnerability of a project. The Bus Factor might be a much smaller threat to projects that no longer require active maintenance, because there is more time to replace a developer. On the other hand, dormant projects

are in themselves problematic, because software usually requires at least some maintenance, especially in a young ecosystem like npm.

On a related note, one response contained: “I can’t recall exactly, but I don’t believe $*developer* has contributed to $*package* for over a year. It seems like the algorithm should consider "active" or recent contributors.” The Island Finder indeed makes no distinction between code originating from older or from more recent commits.

It was mentioned that Islands of Knowledge stemming from popular modules should not be considered as high a threat as more specific modules, for instance: “I have the most knowledge about $*package* because I am the author of that [package]. There might be a bit of a problem if I go missing, if fixes for that [package] are needed. [...] The other modules are much less of a problem (mocha, chai, assert) because their use is a standard practice in the JS community.” This implies that the severity of on an IoK may not only be dependent on its number of uses, but also the popularity of its constituent modules. This would be an extension of counting the numbers of uses for the modules within the project to counting the number of uses of the modules in the whole npm ecosystem.

Chapter 6

Conclusion

Chapter 1 introduced the concept of the Bus Factor, which describes the minimum number of persons who can depart a project before it is incapacitate. One way of finding such a person or set of persons is finding Islands of Knowledge, which are clusters of specific knowledge or expertise that are exclusive to one person in a project. In chapter 3 a software tool for the automated extraction of knowledge statistics in software projects using API usage was proposed and the implementation of a prototype described. Chapter 4 showed that the raw data about the expertise in a software project must be interpreted – either manually or through the use of metrics – in order to identify Islands of Knowledge.

In an explorative approach, two metrics were defined and subsequently applied to a large set of popular Node.js packages. Doing so showed that, going by the metric with the least severe results, on average each repository had 2.3 Islands of Knowledge, most of which lie in the builtin modules of Node.js. Over half of the packages had at least one Island of Knowledge, with 14 Islands being the highest number measured (in two packages).

The application of the Tool also inadvertently revealed an other property of the npm ecosystem: Most packages have only one main developer, contributing a majority of the code changes. This in some sense overrides the Island of Knowledge metric for the Bus Factor, since the dependence on such developers is trivial and could be recognized by much simpler metrics like the number of commits.

On a smaller scale, the results for some of the packages were shared with their most active developers, together with a survey to assess the validity of the results. The responses were mixed, on one hand indicating that there is value in analyzing the API usage statistics of software repositories, and on the other hand pointing out some weaknesses of the approach. In summary, the weaknesses found by the whole Case Study are:

- Packages should not be considered in isolation, but rather with the whole ecosystem in mind, because even if knowledge about a module is exclusive to one developer in a repository, the module could still be well-known to

developers outside of the project, who could step up to replace a departing project member.

- A developer not using an API does not necessarily mean that they have no experience with it from other sources.
- Choosing the most-frequent keyword of a package as its programming domain can lead to a dubious API-domain mapping.
- Each found Island of Knowledge need further assessment of its severity regarding its context. Some pertain to code that is "stable" i.e. does not need anymore changes. Others point out developers that are already no longer active, which could hide Islands by active developers.
- Wrong assignments of domains to modules might also hide IoKs.
- The API usage extraction is not perfect. It does not recognize uses of module class instances, does not recognize uses of APIs added to existing identifiers rather than to a own module identifier and relies on programmers requiring modules in the beginning of a file in the standart sequential manner.

All together, there is no final answer to the research question ("How well can a MSR tool find Islands of Knowledge in Node.js projects?"). This work represents a first explorative attempt at such a tool. While the Tool certainly does find concentrations of API uses, their assessing their relevance requires further interpretation. The responses to the survey in the Case Study indicate that the basic metrics used for this interpretation are applicable and that the approach merits further effort in order to refine the at this stage somewhat crude findings.

6.1 Threats to Validity

Limiting the Case Study to popular packages might have introduced a selection-bias towards stable projects. Popular packages on npm are usually widely used libraries with a big developer base and a lot of outside interest. For more private projects like specific web applications or corporate projects/closed source projects with few developers, the existence of Islands of Knowledge might be a much higher risk factor.

The assignment process of domains to modules is flawed in two ways: Firstly, the manual assignment of domains to builtin modules and global objects is highly subjective. Secondly, the automated assignment using keywords from the npm registry might produce unexpected results. Because of these two reasons, Islands found in domains should be treated with skepticism.

The determination of thresholds for the IoK metrics in chapter 4 is based on a manual classification of the measurements for a subset of the subject repositories. This process was subjective and other threshold values may be just as valid.

The number of replies to the survey was quite low. While the replies were used for anecdotal evaluation, a truly statistically significant case study should consider more packages or an other communication strategy in order to get more responses. Also, the manual classification of the answers into the categories *positive*, *negative* or *undefined* is prone to personal bias.

Because of privacy concerns, the survey responses can not be made publicly available.

6.2 Future Work

Predictably, this very explorative work came across various previously unknown difficulties. Testing the design revealed several problems and improvement opportunities, as described in the beginning of this chapter. Future works might be able to learn from the experiences made here, improve on the weaknesses listed above and continue the work in multiple ways, namely:

- More specific Islands of Knowledge must be found. Finding Islands in common domains just shows high activity of a developer, not necessarily critical exclusiveness of knowledge.
- Modules need weighting according to their ecosystem-global popularity and depending on whether they are builtin or not.
- More comprehensive study of author knowledge, incorporating experience demonstrated outside of the subject repository (e.g. drawing from all the projects to which they contribute on GitHub). This could be expanded into some kind of "experience-profiling" of developers, for which the Island Finder Tool developed here could be repurposed somewhat easily.
- Local modules could be considered too, although it can be argued that they would form a different kind of Islands of Knowledge.
- Application of the Island Finder on a larger scale and survey of more repositories.
- Definition of better metrics for finding Islands of Knowledge in the raw API usage data.
- More thorough extraction of API usages

Appendix A

Listings

All data shown here has been extracted from publicly available sources. It is intended for purely analytical purposes and in no way poses any personal criticism at the authors.


```
{
  "console": {
    "Hao Chen": 3
  },
  "Object": {
    "Steve Mao": 1
  },
  "String": {
    "Hao Chen": 1
  },
  "benchmark": {
    "Hao Chen": 1,
    "Steve Mao": 1
  },
  "tape": {
    "E.Azer Koçulu": 1,
    "Zehua Liu": 3
  }
}
```


Listing 7 uses.json for **left-pad** (https://www.npmjs.com/package/left-pad)

```
{
  "react": 13105,
  "api": 7181,
  "css": 6858,
  "javascript": 6848,
  "node": 6009,
  "plugin": 5412,
  "json": 5351,
  "yeoman-generator": 5331,
  "cli": 5197
}
```


Listing 8 Excerpt of the ranked list of keywords from the npm-registry

```
{
  "Infinity": "MATH",
  "NaN": "MATH",
  "Number": "MATH",
  "Math": "MATH",
  "null": "CORE",
  "Object": "CORE",
  "Function": "CORE",
  "Boolean": "CORE",
  "Symbol": "CORE",
  "Error": "ERROR",
  "EvalError": "ERROR",
  "RangeError": "ERROR",
  "ReferenceError": "ERROR",
  "SyntaxError": "ERROR",
  "TypeError": "ERROR",
  "URIError": "ERROR",
  "Date": "CORE",
  "String": "CORE",
  "RegExp": "CORE",
  "Array": "DATA",
  "Int8Array": "DATA",
  "Uint8Array": "DATA",
  "Uint8ClampedArray": "DATA",
  "Int16Array": "DATA",
  "Uint16Array": "DATA",
  "Int32Array": "DATA",
  "Uint32Array": "DATA",
  "Float32Array": "DATA",
  "Float64Array": "DATA",
  "Map": "DATA",
  "Set": "DATA",
  "WeakMap": "DATA",
  "WeakSet": "DATA",
  "ArrayBuffer": "DATA",
  "DataView": "DATA",
  "JSON": "JSON",
  "Promise": "PROMISE",
  "Reflect": "CORE",
  "Proxy": "CORE",
  "Intl": "CORE",
  "console": "LOGGING",
  "exports": "MODULE",
  "global": "MODULE",
  "module": "MODULE",
  "process": "PROCESS"
}
```

Listing 9 Global identifiers in Node.js with and their primary programming domain

```
{
  "assert": "TEST",
  "buffer": "CORE",
  "child_process": "PROCESS",
  "cluster": "PROCESS",
  "console": "LOGGING",
  "constants": "CORE",
  "crypto": "CRYPTO",
  "dgram": "IO",
  "dns": "IO",
  "domain": "IO",
  "events": "EVENTS",
  "fs": "IO",
  "http": "HTTP",
  "https": "HTTP",
  "module": "MODULE",
  "net": "IO",
  "os": "UTIL",
  "path": "IO",
  "process": "PROCESS",
  "punycode": "IO",
  "querystring": "IO",
  "readline": "IO",
  "repl": "REPL",
  "stream": "IO",
  "string_decoder": "IO",
  "timers": "CORE",
  "tls": "SSL",
  "tty": "IO",
  "url": "IO",
  "util": "UTIL",
  "v8": "CORE",
  "vm": "CORE",
  "zlib": "IO"
}
```

Listing 10 Built-in modules in Node.js and their primary programming domain

```
{
  "LOGGING": ["console"],
  "CORE": ["Object", "String"],
  "PERFORMANCE": ["benchmark"],
  "TEST": ["tape"]
}
```

Listing 11 The programming domains and their modules used in left-pad

```
{
  "benchmark": {
    "benchmark": 202,
    "performance": 800,
    "speed": 156
  },
  "tape": {
    "tap": 342,
    "test": 4127,
    "harness": 24,
    "assert": 456,
    "browser": 3423
  }
}
```

Listing 12 The keywords of the modules used in left-pad and their frequencys

```
["async@295b307", "session@b5a0437", "chokidar@3f7f113",
"grunt-contrib-jshint@8d38fc2", "node_redis@5650806",
"gulp-sass@25d37e7", "axios@5630d3b", "browser-sync@9e0a170",
"q@d8fd789", "node-http-proxy@c979ba9", "atom@d6f7b98",
"should@4a53d94", "ejs@1c7e365", "karma@f567e20", "co@249bbdc",
"html-webpack-plugin@b8fd142", "request-promise@b5e06f5",
"lodash@f3e0cbe", "js-xlsx@3cacfc4", "gulp-uncss@0432b6c",
"mongoose@ea9ff07", "node-sqlite3@f1456b1", "helmet@ebd0d35",
"gulp-util@28c2aa2", "body-parser@57d237f", "mysql@310c6a7",
"validator.js@f72ec86", "marked@8f9d0b7", "jshint@6c34960",
"ionic-cli@ebde0f5", "node-supervisor@85d92db",
"node-restify@16347ae", "joi@33d45d1", "npm@d081cc6",
"node-mime@9515cda", "gulp-jshint@807bead",
"run-sequence@7e263af", "socket.io@01a4623", "left-pad@aff6d74",
"compression@4bd00d3", "standard@11a9635",
"react-router@5868dc4", "js@09e61e1", "gulp-concat@c179a8c",
"gulp-if@06c0b18", "koa@18d753c", "cross-env@c1a9ed0",
"selenium@24a5055", "vue@2a19f91", "log4js-node@547267f",
"bluebird@eb0d465", "shelljs@b9201eb", "express@efd7032",
"minimist@4cf45a2", "esprima@5a46bf8", "redux@a56339d",
"node-semver@8fff305", "sails@1dbc810", "node-fs-extra@ed5dc63",
"cors@b6dac7f", "npm-check@4c09a09", "pencilblue@24d9379",
"forever@3aa17a1", "gulp-babel@42bcfe3", "npm-run-all@46cfd57",
"sequelize@b31b662", "node-optimist@680451c",
"gulp-less@18d0880", "gulp-htmlmin@e7e5766",
"gulp-plumber@982ec11", "connect@2fa7514", "pug@82f2fcc",
"cordova-cli@035d86c", "qs@64d620d", "meteor@90cb625",
"shortid@7bad247", "multer@7ef2e81", "gulp@62323fc",
"stylus@63e0cbe", "handlebars@d40cbfc", "grunt-cli@7f6298e",
"UglifyJS2@4bceb85", "jade@82f2fcc", "through2@4383b10",
"node-schedule@aa45a76", "TypeScript@b6dfa39",
"node-csv@6523da7", "gulp-useref@58d56c1",
"coffeescript@ca0fd22", "gulp-autoprefixer@cd022c8",
"node-formidable@b81d1c7", "classnames@51b3524", "ws@2ace70d",
"Font-Awesome@d87e4e9", "gulp-replace@824e2e8",
"underscore@d4f52aa", "gulp-livereload@bbf71b1",
"commander.js@89858e5", "node-glob@9b0994e",
"gulp-changed@d836d67", "node-sass@f2f4b96", "colors.js@9f3ace4",
"webpack@98ea823", "watchify@f768b43", "gulp-connect@2027a3c",
"gulp-load-plugins@55cf056", "tape@66519cb",
"grunt-contrib-watch@7f8cf80", "node-xml2js@b5e351a",
"eslint@e4da200", "gulp-notify@658efc5", "node-mkdirp@f2003bb",
"hapi@c8473df", "node-mssql@b0c1a21", "gulp-watch@608c7aa",
"gulp-rename@3b4fdf8", "winston@fcf04e1", "sinopia@3f55fb4",
"nodemailer@9b4f90a", "cookie-parser@513b2fc",
"grunt-contrib-uglify@3a718cf", "angular@d96e58f", "n@cf8b323",
"json-server@f9d4a46", "bootstrap@9b7d140"]
```

Listing 13 The 180 most starred npm packages and their HEAD revision as of 2017-03-21, in alphabetical order, part 1

```
["gulp-imagemin@8bc8f11", "gulp-uglify@4656fe5",
"wp-calypso@6ff1225", "jquery@c1c5497", "gulp-rev@82c185c",
"istanbul@89e338f", "jsdom@e068779", "gm@c6a6c5a",
"autoprefixer@93c2e0c", "http-server@da8bfe5",
"keystone@d6cef03", "less@bfe19b1", "node-uuid@084c3fa",
"openmct@b28eb04", "nodemon@2cd85b1", "del@b33ee97",
"gulp-inject@43de8c2", "debug@27d93a3",
"node-sanitize-filename@ef1e8ad", "superagent@21fab25",
"node-bunyan@994f90e", "gulp-sourcemaps@8166cbe",
"cheerio@51e3645", "Ghost@29511bf", "chai@a7e1200",
"morgan@a6eebc7", "node-jsonwebtoken@2ec4960",
"node-postgres@3de22ba", "socket@e40accf", "moment@f2af24d",
"backbone@bd50e2e", "d3@4227c3c", "javascript@13dc420",
"react@97ab3f5", "grunt@6c596b1", "aws-sdk-js@18deee0",
"chalk@0d21449", "dotenv@825c1b2", "npm-check-updates@5efc915",
"node@cb82c05", "mocha@b4ebabd", "colors@9f3ace4",
"hexo@5a7ea3e", "phantomjs@750d5f3", "pm2@c2b1a89",
"commander@3367806", "bower@b716bc4",
"grunt-contrib-cssmin@e6b2d72", "Inquirer@f925e8d",
"supertest@199506d", "statsd@8d5363c", "yo@ecd7ffa",
"passport@8de1c66", "rimraf@d84fe2c", "request@b12a624"]
```

Listing 14 The 180 most starred npm packages and their HEAD revision as of 2017-03-21, in alphabetical order, part 2

```
{
  "islandsByPortion": [
    {
      "location": "CLI",
      "remap": [
        "chalk"
      ],
      "author": "Mike McNeil",
      "total": 60,
      "uses": 60,
      "portion": 1
    }
  ],
  "islandsByCarry": [
    {
      "location": "TEST",
      "remap": [
        "assert",
        "mock-req",
        "should",
        "supertest"
      ],
      "carryAuthor": "Scott Gress",
      "carryAuthorUses": 783,
      "secondAuthor": "Mike McNeil",
      "secondAuthorUses": 148,
      "carryToSecond": 0.8109833971902938
    }
  ]
}
```

Listing 15 Example output of the `scan_for_islands` component

```
1 {domainPortion: ([.repos[] | .domainIslands.islandsByPortion |
  .[]] | length), domainCarry: ([.repos[] |
  .domainIslands.islandsByCarry | .[]] | length), modulePortion:
  ([.repos[] | .moduleIslands.islandsByPortion | .[]] | length),
  moduleCarry: ([.repos[] | .moduleIslands.islandsByCarry | .[]] |
  length)}
2 [.repos[]] | length
3 [.repos[] | select(.domainIslands.islandsByCarry | length > 0)] |
  length
4 [.repos[]] | group_by(.domainIslands.islandsByCarry | length)[] |
  [.[] | .repoName]
5 [.repos[] | {repoName, uniqueAuthorIslands:
  (.domainIslands.islandsByCarry | group_by(.carryAuthor) |
  length)}] | group_by(.uniqueAuthorIslands)[] | {variety:
  .[0].uniqueAuthorIslands, number: length}
6 [.repos[] | {repoName, uniqueAuthorIslands:
  (.moduleIslands.islandsByCarry | group_by(.carryAuthor) |
  length)}] | group_by(.uniqueAuthorIslands)[] | {variety:
  .[0].uniqueAuthorIslands, number: length}
7 [.repos[] | {repoName, uses, firstCarryAuthorUses:
  (.domainIslands.islandsByCarry | group_by(.carryAuthor) |
  max_by([.[] | .carryAuthorUses] | add) // [] | [.[] |
  .carryAuthorUses] | add), carryAuthors:
  (.domainIslands.islandsByCarry | group_by(.carryAuthor)) |
  length}]
8 [[.repos[] | {repoName, uses, firstCarryAuthorUses:
  (.domainIslands.islandsByCarry | group_by(.carryAuthor) |
  max_by([.[] | .carryAuthorUses] | add) // [] | [.[] |
  .carryAuthorUses] | add), carryAuthors:
  (.domainIslands.islandsByCarry | group_by(.carryAuthor)) |
  length}] | .[] | select(.carryAuthors > 0) |
  .firstCarryAuthorUses / .uses] | add / length
9 [[.repos[] | .domainIslands.islandsByCarry[]] |
  group_by(.location)[] | {length: (length), location:
  .[0].location}] | sort_by(.length)
```

Listing 16 jq-filters used in chapter 5

Appendix B

Tables

All data shown here has been extracted from publicly available sources. It is intended for purely analytical purposes and in no way poses any personal criticism at the authors.

Table B.1 The modules used in **left-pad**, as table with useage statistics

	console	Object	String	benchmark	tape
Hao Chen	3	0	1	1	0
Steve Mao	0	1	0	1	0
Zehua Liu	0	0	0	0	3
E.Azer Koçulu	0	0	0	0	1

Table B.2 The modules used in left-pad resolved to programming domains, as table with useage statistics

	LOGGING	CORE	CORE	PERFORMANCE	TEST
	console	Object	String	benchmark	tape
Hao Chen	3	0	1	1	0
Steve Mao	0	1	0	1	0
Zehua Liu	0	0	0	0	3
E.Azer Koçulu	0	0	0	0	1

Table B.3 The programming domains used in left-pad, as table with useage statistics

	LOGGING	CORE	PERFORMANCE	TEST
Hao Chen	3	1	1	0
Zehua Liu	0	0	0	3
Steve Mao	0	1	1	0
E.Azer Koçulu	0	0	0	1

Bibliography

[20117a] Tiobe index for march 2017. https://www.tiobe.com/tiobe-index/, 2017. Accessed: 2017-03-26.

[20117b] Website of the 14th international conference on mining software repositories. http://2017.msrconf.org/, 2017. Accessed: 2017-03-11.

[20117c] Website of the node.js foundation. https://nodejs.org/en/, 2017. Accessed: 2017-03-12.

[AAB+15] Bruno Almeida, Sophia Ananiadou, Alessandra Bagnato, Alberto Berreteaga, Juri Di Rocco Barbero, Davide Di Ruscio, Dimitrios S Kolovos, Ioannis Korkontzelos, Scott Hansen, Pedro Maló, et al. Ossmeter: Automated measurement and analysis of open source software. *Projects Showcase@ STAF'15*, page 36, 2015.

[Aks15] Hakan Aksu. Evolution-aware api analysis ofdeveloper skills. Master's thesis, University of Koblenz-Landau, Germany, 2015.

[APHV16] Guilherme Avelino, Leonardo Passos, Andre Hora, and Marco Tulio Valente. A novel approach for estimating truck factors. In *Program Comprehension (ICPC), 2016 IEEE 24th International Conference on*, pages 1–10. IEEE, 2016.

[AVH15] Guilherme Avelino, Marco Tulio Valente, and Andre Hora. What is the truck factor of popular github applications? a first assessment. Technical report, PeerJ PrePrints, 2015.

[Awa07] Kailash Awati. Increasing your team's bus factor. https://eight2late.wordpress.com/2008/09/03/increasing-your-teams-bus-factor/, 2007. Accessed: 2017-03-11.

[Fen94] Norman Fenton. Software measurement: A necessary scientific basis. *IEEE Transactions on software engineering*, 20(3):199–206, 1994.

[FL84] Ludwik Finkelstein and MS Leaning. A review of the fundamental concepts of measurement. *Measurement*, 2(1):25–34, 1984.

[Fla06] David Flanagan. *JavaScript: the definitive guide*. " O'Reilly Media, Inc.", 2006.

[Hol17] Burke Holland. The era of micro packages. http://developer.telerik.com/content-types/opinion/era-micro-packages/, 2017. Accessed: 2017-03-11.

[Jon03] Joel Jones. Abstract syntax tree implementation idioms. In *Proceedings of the 10th conference on pattern languages of programs (plop2003)*, page 26, 2003.

[LPS11] Ralf Lämmel, Ekaterina Pek, and Jürgen Starek. Large-scale, ast-based api-usage analysis of open-source java projects. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, pages 1317–1324. ACM, 2011.

[VCBSS96] Catherine E Volpe, Janis A Cannon-Bowers, Eduardo Salas, and Paul E Spector. The impact of cross-training on team functioning: An empirical investigation. *Human factors*, 38(1):87–100, 1996.

[WK02] Laurie Williams and Robert Kessler. *Pair programming illuminated*. Addison-Wesley Longman Publishing Co., Inc., 2002.