Reference Coupling
A Method for Identifying Software Ecosystems of Technically Dependent Projects

by

Francis Harrison
B.Sc., University of Victoria, 2015

A Thesis Submitted in Partial Fulfillment
of the Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

© Francis Harrison, 2015
University of Victoria

All rights reserved. This thesis may not be reproduced in whole or in part, by photocopy or other means, without the permission of the author.
Supervisory Committee

Reference Coupling
A Method for Identifying Software Ecosystems of Technically Dependent Projects

by

Francis Harrison
B.Sc., University of Victoria, 2015

Supervisory Committee

Dr. Daniela Damian, Supervisor
(Department of Computer science)

Dr. Kelly Blincoe, Co-Supervisor
(Department of Computer science, AUT New Zealand)

Dr. Sudhakar Ganti, Departmental Member
(Department of Computer science)
Abstract

Software projects are not developed in isolation. Open source software projects encourage a networked collaboration and interdependence across projects and developers. Recent research has shifted to studying software ecosystems, communities of projects that depend on each other and are developed together. However, identifying technical dependencies at the ecosystem level can be challenging. In this dissertation, we propose a new method, known as reference coupling, for detecting technical dependencies between projects. The method establishes dependencies through user-specified cross-references between projects. We use our method to identify ecosystems in GitHub hosted projects, and we identify several characteristics of the identified ecosystems. Our findings show that most ecosystems are centered around one project and are interconnected with other ecosystems. The predominant type of ecosystems are those that develop tools to support software development. We also found that the project owners’ social behavior aligns well with the technical dependencies within the ecosystem, but project contributors’ social behavior does not align with these dependencies. We conclude with a discussion on future research that is enabled by our reference coupling method.
# Table of Contents

Supervisory Committee ................................................................................................... ii

Abstract ............................................................................................................................... iii

Table of Contents ............................................................................................................... iv

List of Tables ....................................................................................................................... vii

List of Figures ...................................................................................................................... viii

Acknowledgments ............................................................................................................. ix

Dedication ............................................................................................................................ x

Introduction .......................................................................................................................... 1

  1.1 Problem Statement ........................................................................................................ 3

  1.2 Research Goals and Methodology ................................................................................. 3

  1.3 Summary of Contributions ............................................................................................ 8

  1.4 Thesis Organization ....................................................................................................... 8

Identifying Technical Dependencies using Reference Coupling ........................................ 10

  2.1 Background and Related Work ..................................................................................... 11

  2.2 Methodology and Approach ....................................................................................... 14

  2.3 Results .......................................................................................................................... 16

  2.4 Conclusion of Study ..................................................................................................... 18

Visualizing an Ecosystem based on Technical Dependencies .......................................... 20
Bibliography
List of Tables

Table 1 - Definitions of Software Ecosystems relative to its Domain ..................22
Table 2. Node table Structure ...........................................................................25
Table 3 - Edge table structure ...........................................................................26
Table 4 - General statistics of the dependency network ......................................32
Table 5 - Ecosystems in GitHub. Details of the Most Well-Connected Node in Each Ecosystem .........................................................................................33
Table 6 - Project owners: correlations between technical dependencies and social behavior .................................................................................................44
Table 7 - Project contributors: correlations between technical dependencies and social behavior .................................................................................................45
List of Figures

Figure 1 - Example depicting cross-referencing on GitHub........................................11

Figure 2 - A link is created between tsugiri/axiom and ninenines/cowboy due to
cross-referencing........................................................................................................15

Figure 3 - All GitHub projects with cross-references. The largest connected
component (or giant component) is easily identified as the well-connected
subgraph appearing in the center of the graph.............................................................29

Figure 4. Ecosystems in the largest connected component of Github-Hosted
projects. Project names follow the pattern user/repository where user is the
owner’s Github login and repository is the name of the project repository..............30

Figure 5 - Twbs/bootstrap Ego Network. Portraying a sample star pattern in the
network..........................................................................................................................32

Figure 6. Jsdoc3/jsdoc Ego Network. Portraying another variation of the star
pattern in the network..................................................................................................34

Figure 7 - The Rails Ecosystem ....................................................................................37

Figure 8 - The Owner Follows Network $Gof$ ...............................................................42

Figure 9 - The Contributors Follow Network.................................................................43
Acknowledgments

I would like to thank:

My Supervisor, DR. Daniela Damian, for giving me the opportunity to pursue a graduate degree, and giving me every chance to excel. For her enthusiasm, support, and encouragement, and for helping me grow personally and professionally.

My Co-supervisor, Dr. Kelly Blincoe, for always being nearby when I needed help, for her insights and support, and her thoughtful editing of my thesis and for being a great mentor.

Members of SEGAL LAB, for all the laughs and for making this journey worthwhile.

My family, for their well-wishes and support throughout my graduate study.

“That’s the thing about goals, when we focus too much on our end game we can miss the fun of the journey. We can miss the detour that would take us somewhere even more rewarding.”

- Carrie Bradshaw
Dedication

To my Mom
And
Late Sir. F.G. Harrison
Chapter 1

Introduction

In recent years, *global software development* (GSD) has been supported by the improvements in PC power, network transfer speed, and data storage costs. This has led to the deployment of standalone software projects as collections of software systems composed in large source code repositories across the web.

Software projects that co-exist and co-evolve together in an environment, are referred to as being part of a *software ecosystem* (SECO) [1]. These projects share software components and functionalities that are often implemented by the distributed intelligence of developers in Internet communities. Consequently, software dependencies span the boundaries of standalone projects and connect similar and interdependent SECO’s. In this thesis, we explore these software dependencies as they affect the Open-source model of software development. Specifically, we investigate a method for identifying these technical dependencies across Open-source software (OSS) projects and the visualization of dependent projects in an ecosystem.

However, dependencies are not easily identifiable at the ecosystem level. Literature on open source development reports that in many OSS projects, developers coordinate almost exclusively over the internet, making use of tools such as version control, mailing lists, comments and even chat [73]. At the premise is the notion of social coding. Social coding is a form of software development which provides developers with an unprecedented level of transparency allowing them to view the actions of other developers on public or
shared artifacts [37]. Take the example of the successful social coding site called GitHub\(^1\), a web-based, code-hosting repository, based on the Git version control system. Embedded in GitHub are large repositories of inter and intra dependent software projects. The size of these repositories and the amount of source code found in them, makes it very difficult to identify dependencies between projects using existing methods such as extracting dependencies from source code [3,4].

Our main assumption in this thesis is that dependencies across projects can be identified by considering the coordination practices of OSS developers. Through effective communication and making use of the existing tools at their disposal, developers find ways to manage dependencies in their OSS project space. Previous studies found that OSS developers use various functionalities from other projects in their own work and watch the projects they depend on [37], use asynchronous communication channels such as comments for providing and collecting awareness information [74], and attract and encourage new participants to contribute by ensuring that awareness information is readily available [64, 66]. During our study we found that, nested in the coordination practices of developers, are dependencies between projects; specified by the developers themselves. A method to extract these dependencies would provide an easier way to track software dependencies across projects. This will result in a better understanding of how OSS projects are connected and where they belong in a software ecosystem. Consequently, this knowledge can help reduce work when multiple projects are faced with the same issue of stemming from their use of a share repository. It can also help the owner of a repository become

\(^{1}\) http://github.com
aware of what other projects depend on their project or specific issues within the project.

1.1 Problem Statement

Our motivation for this study is supported by the opinion that software projects are not developed in isolation. Instead, they often depend on and coexist with other projects in the same space. However, challenges exist when trying to identify technical dependencies at the ecosystem level. These challenges include the scalability of existing methods to identifying technical dependencies across a large number of projects, and so forth. Without a way to establish dependencies between projects, we can not fully understand a project’s ecosystem because the technical dependencies that exist between projects define the structure of the ecosystem [2]. Understanding a software projects ecosystem opens an avenue for analysis into the quality, design and governance of the project. It also creates awareness on collaboration opportunities with other projects in the same ecosystem.

1.2 Research Goals and Methodology

Our goal was to investigate a method for identifying cross-project dependencies in open-source projects, and subsequently visualize the ecosystem of technically dependent OSS projects. We analyzed open-source projects on the popular social coding platform GitHub, with the aim that this would provide us with better insight into how projects are connected and what types of dependencies exist. GitHub is a popular social code hosting platform built to provide the basic requirements for distributed revision control as found in Git, as well as other features (e.g. pull requests, issues, forks). GitHub provides its users with social networking features that allows users to share repositories with others, watch or access another users’ repository, and most importantly make use of open source
projects found on GitHub. Issues, commits and pull request descriptions are written using the *GitHub flavored Markdown*\(^2\), and comments allow for collaborators or team members to discuss about an implementation, or provide feedback to the author. GitHub allows for users to insert cross-references to another repository’s issue, commit or pull request. These cross-references use the pattern User/Project#Num for issues or pull requests (e.g rails/rails#234) or User/Project@SHA for commits. With the GitHub flavored Markdown, these cross-references are automatically linked to the other project.

We obtained data on the cross-references from the GHTorrent [12] project, which contains a mirror of the GitHub API data. GHTorrent obtains its data by monitoring and recording GitHub events as they occur. We used the MySQL 2014-04-02 dataset to obtain information on the projects. This dataset contains data on 2,399,526 repositories, 3,426,046 users, and their events - including commits, issues, pull requests and comments. Since the MySQL database contains only the first 256 characters of comments, we obtained all comments from GHTorrent’s main MongoDB server in May 2014. The MongoDB contains the full text of all comments.

Our research focused on three major aspects:

**Identifying technical dependencies:** Technical dependencies exist across projects and identifying these dependencies on a large scale has proven to be difficult [3]. Different methods such as extracting dependencies from a projects source code [2, 3, 4] have been proposed. However, due to the large amounts of projects in OSS systems such as GitHub, these methods are not scalable enough and

\(^2\) [https://help.github.com/articles/github-flavored-markdown/](https://help.github.com/articles/github-flavored-markdown/)
therefore cannot be adequately employed to identify dependencies. As such, a method that takes into account the dependencies that exist across projects and addresses the issue of scalability is desirable. Therefore, we investigated the research question:

RQ1. “Do cross-references to other projects in issue, pull request, and commit comments indicate the existence of a technical dependency between the two projects?”

Our objective was to establish and examine a method for identifying technical dependencies across a large number of projects. To answer this research question we developed a method that extracted and leveraged cross-project technical dependencies as embedded in cross-references between projects. We make use of a grounded theory approach to identify these dependencies, grouping conceptually similar cross-references into categories. The method makes use of the GitHub flavored Markdown and creates a link to a repository whenever a user cross-references another repository in a pull request, issue or commit comment. Our investigation focused on identifying ecosystems, based on technical dependencies in software projects which may be nested in these cross-references.

We found that 90% of the links described technical dependencies between projects. This conceptualization of technical dependencies between projects is referred to as Reference coupling. The results of our first study provided a platform for and directly influenced our second area of our investigation.

**Structure of the software ecosystem:** With an ability to identify technical dependencies between a large number of projects, we can also identify ecosystems of densely connected projects. A way to identify and visualize these
ecosystems is important for developers as it gives an understanding of how their projects fit into the big picture, as well as who they can coordinate with at the ecosystem level [1, 11]. To achieve this, we explored the research question:

RQ2. “What ecosystems exist across GitHub-hosted projects and what is their structure?”

Our second objective was to use the identified dependencies from our first study to visualize the ecosystem of OSS projects on GitHub and also examine the properties of these ecosystems. We created a visualization that depicts the ecosystem of OSS projects in the GitHub environment. GitHub encourages collaboration between users both within and across projects through its transparent interface and built in social features. By leveraging these technical relationships between projects, we try to identify communities that exist based on technical dependencies between projects. Using techniques for community detection, we created a software ecosystem and identified communities, which represent sets of projects that are tightly connected based on technical dependencies.

Social behavior of project owners and contributors: Awareness facilitates collaboration. Social network analysis has been used to create awareness and facilitate collaboration in software development. One notably important line of inquiry in this work has been to understand the influence and motivation of OSS developers [76], as well as the alignment between their communication & coordination (C&C) and development activities [75]. For example, an analysis of the email social networks in OSS projects show that the more software development activities performed by an individual, the more C&C activity the same individual must undertake [75]. Previous studies have also found that when
a developer’s coordination aligns with their technical dependencies, productivity and quality increases [59]. This led us to investigate the social behavior of OSS developers on GitHub, in order to determine if there is an awareness of software dependencies by examining their C&C activities. The goal was to answer the following research questions:

RQ3. *Do the project owners’ social behaviors align with the technical dependencies?*

RQ4. *Do the project contributors’ social behaviors align with the technical dependencies?*

In our final study, we set out to understand how the social behavior of project owners and contributors relates to the identified ecosystems by examining the alignment between social and technical connections around the projects. Examining the correlation between dependencies and C&C activities, allows us to establish the degree of awareness about the technical dependencies we found.

To achieve this, we constructed links between projects based on social behaviors such as following[^3] and starring[^4] on GitHub. In order to fully understand the relationships between the technical dependencies and these social connections, we ran some correlations between edge weights in the dependency network with the edge weights in each social network. The study findings indicate that: while project owners seem to follow the right people, and are aware of the right

[^3]: https://help.github.com/articles/be-social/
projects based on the technical dependencies that exist in the ecosystem, the social behavior of project contributors is not aligned with project dependencies.

1.3 Summary of Contributions

Our overall objective is to study and understand software ecosystems and how open-source projects depend on each other. We developed a method that makes it easier to identify and visualize technical dependencies across projects. This is beneficial for both software developers and researchers.

For software developers, our study provides a new and holistic view into collaboration in their practice. Primarily, we provide a view of how they are connected to other projects, which communities they belong to and where they fit in at the ecosystem level. This knowledge would aid collaboration among developers in practice by providing an avenue for more streamlined coordinated efforts.

For software engineering researchers, our study expands the existing and budding knowledge about software ecosystems. First, we suggest a method for detecting technical dependencies across projects which could be very useful for studying software ecosystems. Our method captures dependencies which are more logical and cannot be easily detected by looking at source code and also enables the researcher identify unknown software ecosystems. Additionally, our study opens avenues for various interesting research paths such as the study of socio-technical congruence across a larger set of projects and their stakeholders.

1.4 Thesis Organization

The purpose of this chapter is to introduce the reader to our research, allowing the user understand the context, and our motivation for studying technical dependencies across projects. The remaining chapters all carry an overall review
of existing literature as they apply to and motivate each study. In Chapter 2, we discuss our method for identifying technical dependencies and provide our methodology for examining the validity of the reference coupling method. We also categorize the dependencies we found based on how they affect the projects.

In Chapter 3, we describe the steps we took to visualize the ecosystem of software projects on GitHub. We also discuss our analysis of the ecosystems we found by highlighting their properties.

In Chapter 4, we examine the social behaviours of projects owners and contributors to determine if they align with the technical dependencies we found.

In Chapter 5, we discuss the implications and contributions of our study as well as some threats to validity. We also suggest future work in identifying technical dependencies and software ecosystems.
Chapter 2

Identifying Technical Dependencies using Reference Coupling

Today’s software projects are not developed as monolithic components; rather they are distributed and inter-connected by networks of dependencies. As projects grow larger, it becomes increasingly difficult to identify technical dependencies. Most research in this area is focused on identifying the technical dependencies within a project (intra-project) using techniques such as, analysis of project source code. However, these techniques do not scale up to identifying dependencies between projects (inter-project). Additionally, collecting source code data across multiple projects is not always feasible.

With our focus on identifying inter-project dependencies, the question we investigated in this study was:

RQ Do cross-references to other projects in issue, pull request, and commit comments indicate the existence of a technical dependency between the two projects?

This chapter explains the approach used to identify cross-references between projects in GitHub. The process involves examining user-specified cross-references to determine if there was an indication of technical dependencies. We found that these cross-references do indicate the existence of technical dependencies and we were able to highlight the different categories of technical dependencies we found.
The GitHub hosting site incorporates a number of social features, allowing for visibility of information about its users and their activities within and across open source software projects. One of such features is the GitHub flavored Markdown, and it allows for a link to be automatically created whenever a user cross-references another repository in a pull request, issue or commit. These user-specified cross-references are made as part of a comment as developers coordinate and manage their work dependencies, and consequently link two projects together.

![Example depicting cross-referencing on GitHub](image)

**Figure 1 - Example depicting cross-referencing on GitHub**

2.1 Background and Related Work

Research has tried different methods for identifying technical dependencies between projects on a large scale. However, this has proven to be difficult. On the technical side, analysis of a project’s source code is a common technique to identify technical dependencies within a project (intra-project). However, these techniques do not scale up to identify dependencies between projects (inter-project). Ossher et al. [4] used a technique that analyzes import statements by cross-referencing a project’s missing types in Java source code to resolve inter-project dependencies. Businge et al. [5] used a similar technique when studying the Eclipse ecosystem. Lungu et al. [2] considered external methods and class
calls in a project's source code for extracting inter-project dependencies. However, these techniques all require a large amount of source code, and a large amount of memory. Collecting source code data across any versioning system would require multiple TBs of data and more than a year in processing time [6]. As such, the number of projects that can be studied using these techniques are limited.

Studies have also examined different ways to identify the technical dependencies without relying on analysis of source code. One of the methods used involves examining declared dependencies found in a project's configuration files or its dependency management tool like Maven[^5] to identify technical dependencies [2, 7, 8, 9, 10]. Bavota et al. [9] used a code analyzer to extract dependencies existing between releases in Java Apache projects. They found that 37% of releases showed no indication of dependencies. Syeed et al. [15] collected metadata from published specifications of technical dependencies at rubygems.org in their study of the Ruby on rails ecosystem. However, this approach is specific to only projects that publish dependency specifications.

Gall et al. [19] proposed a method called *logical coupling* which is analogous to ours. This method detects intra-project dependencies when artifacts have been worked on together by examining the information in the release history of a system. By comparing the change report analysis of a product release with logical couplings of both inter and intra-project dependencies, the *logical coupling* approach exposes unknown dependencies in software modules and also identifies the evolution pattern of a software

[^5]: http://maven.apache.org
ecosystem. Our method differs in that it detects inter-project dependencies when issues, pull requests or commits have been worked on in conjunction with another projects (as evidenced through user-specified cross-references).

Largely, these studies have either mostly focused at a very high and technical level of software development (source code and releases), or have looked at project specific dependencies. Our approach does not rely on analyzing source code and makes use of cross-references that can be made in comments on GitHub. These cross-references are user-specified links between a pair of projects. They can be found in comments on issues, pull requests and commits and developers make use of them to coordinate and manage their work dependencies.

**Software reuse and technical dependencies**

Identifying and managing dependencies between software systems is essential to ensuring the quality of these systems. In open source software systems, dependency relationships can exist between projects and the resources they need, or just between the resources [63]. In some cases, these dependencies are created by the assignment of tasks to developers (e.g. two developers working on the project may face restrictions on the kind of changes they are allowed to make without directly interfering with each other). One train of thought on managing dependencies is to minimize dependencies by creating components that can function and be worked on independently [31].

Some of these dependencies may arise as a result of software reuse. Software reuse involves the process of making software systems from existing software as opposed to creating these systems from scratch as defined in [33]. Software artifacts such as components (bits of software that typify usefulness and have
been created particularly with the end goal of being reused) and snippets (various lines of codes from existing frameworks), are commonly reused in open source software development [32, 33]. Other studies on software reuse in OSS suggest that software reuse is very popular among open source software projects and has contributed to its success [34, 35]. Developers perceive efficiency and effectiveness as benefits of software reuse, and tend to practice it more on OSS projects so as to focus on their preferred tasks [36]. Defects in software are frequently and effectively recognized and discussed by the open-source community as a result of software reuse.

Nonetheless, dependencies have their drawbacks. When these dependencies occur within a code base or across projects in open-source development, they are harder to identify due to the magnitude and amount of the projects involved. Complexity increases as the amount of code reused by a project increases. As such, it becomes very difficult for developers to fully understand all the dependencies their software relies on.

2.2 Methodology and Approach

Extracting Cross References

In order to answer our research question, we investigated if cross-references in GitHub comments indicate a technical dependency between the two projects by examining a set of these cross-references to understand why users create cross-references between repositories. We call this method for identifying technical dependencies Reference coupling.
Figure 2 - A link is created between tsugiri/axiom and ninenines/cowboy due to cross-referencing

To identify cross-references, we performed pattern matching on all comments in the GHTorrent database. Since we were interested only in relationships between projects, we filtered the cross-references to ignore the relationships between a pair of projects if one project is a fork of the other project since we consider a repository and all of its forks as a single project as recommended by [21]. We were able to identify 89,784 unique comments with a cross-reference to another project.

Verifying Cross-references

To verify that these cross-references are a valid conceptualization of dependencies, we examined a random set of comments that cross-referenced another project. A total number of 300 random comments were manually examined. We classified a comment as a technical dependency if the comment described a work dependency, either direct or indirect, between the two projects. We looked for indications in the comments around these links to understand the context in which they were created and why they linked to the second project.
Since the comments contain the information of who created them, we were able to verify that they were unique cross-references. Of these cross-references, we found that 90% of the comments described technical dependencies between the two projects. Of the 10% where no clear technical dependency could be established, the comments did not contain enough details for validating the dependency. Many of these comments only referenced another repository using the pattern described above, without any additional text provided. This was particularly prevalent when we examined commit comments. As previous research has suggested, there is a problem with the lack of context when creating or updating commit comments [38].

In the cases where technical dependencies were described in the comments, different types of dependencies were discovered. We examined these comments to identify the types of dependencies that exist through these cross-reference relationships by using a grounded theory approach [22]. We conducted open coding on the cross references, grouping conceptually similar comments into categories. We stopped when we achieved saturation after 49 comments. We also tried to categorize these dependencies we found in order to give an accurate description as to which dependencies exist through these cross-references.

2.3 Results

Dependency Categorization

Through our analysis, we were able to identify two main types of dependencies:

**Dependency between two projects:** This was the most common type of dependency we found. This type of dependency occurred when the comments described a direct technical dependency between two projects, which resulted in
the creation of the cross-reference. For example, there were cases where an
issue created in one project depended on a fix/update from another project or a
project needed an update because there were changes made in another project
that affected it. We also found cases where commits in one repository were
created to directly fix an issue created in another repository. GitHub allows for
automatic closure of issues through commit comments, even when the commit is
in a different repository\(^6\). Below, we provide some examples of the comments
which shows the types of dependencies that were found through cross-
references.

- Issue #449 on the sensu/sensu project describes an issue that is the result of the
  interaction between the sensu/sensu code and the ruby-amqp/amq-client library.
The comment references a commit on the ruby-amqp/amq-client that fixes the
issue.

  “I verified that the problem is still the one referenced in ruby-amqp/amq-
client#14. This fix is not merged with amq-client’s ‘0.9.x-stable’ branch.
This is why I am still hitting it. The commit ruby-amqp/amqclient@60f1c59
is the fix but it resides only in the master branch.”

- Issue #8 on the tsujigiri/axiom project notes that changes must be made to
  the code base to allow an upgrade to the latest release of the
ninenines/cowboy project.

  “Upgrade Cowboy: After Cowboy 0.6.1 Cowboy’s http req record was
made opaque and can not be used directly anymore. I didn’t really have

\(^6\) https://github.com/blog/1439-closing-issues-across-repositories
the time yet to look into it, but it looks like we just need to remove all references to the record from the documentation and add directions on how to access cowboy req:req() via the cowboy req functions. See ninenines/cowboy#266 and ninenines/cowboy#267.”

- Commit 81bbbec21c04b6392f6892f7735243387d295337 on the joyent/node project closes isaacs/node-graceful-fsissue #6, which describes a problem in the isaacs/nodegraceful-fs code stemming from the use of joyent/node.

  “This fixes isaacs/node-graceful-fs#6.”

**Two projects depend on a third project:** Another category of dependency we discovered, involved cases where there was a cross-reference between two projects that both depended on another projects. For these cases, there were some instances where the third project was not cross-referenced but mentioned in the comments. For example, everzet/capifony’s pull request #376 cross-references composer/composer’s issue #1453, but the problem stems from the use of the symfony/symfony project. After identifying the source of the problem, a new issue (#411) is created on the everzet/capifony project that identifies the changes that need to be made to the way the symfony environment is set so that the composer/composer code executes correctly. “As described in #376 capifony should execute composer with the right symphony environment set. Currently, with --no-scripts option removed in #376, composer is always executing symfony scriptswith default dev environment.”

**2.4 Conclusion of Study**

We developed a way to detect technical dependencies between projects by considering the cross-references made in comments on GitHub. We found that
cross-references are commonly included in GitHub comments. By analyzing the content of these cross-references, we showed that they are a valid conceptualization of technical dependencies. This conceptualization of dependencies, we call Reference Coupling.

Our method is analogous to the logical coupling method that detects dependencies within a project as proposed by Gall et al. [19]. However, reference coupling detects inter-project dependencies in comparison with logical coupling which only identifies intra-project dependencies. As such, reference coupling is highly scalable, and allows for considering a large number of projects. Additionally, the dependencies we identified were created as part of the coordination practices of developers. With an ability to identify technical dependencies between a large number of projects, ecosystems of densely connected projects can be identified.
Chapter 3

Visualizing an Ecosystem based on Technical Dependencies

As a result of study 1, a second study was conducted. In study 1, we were able to identify technical dependencies between a large number of projects using our method called reference coupling. We followed up this result by trying to visualize an ecosystem of GitHub-hosted open source projects, which was based on the dependencies we found. This study uses visualization tools and community detection algorithms to identify these ecosystems.

Projects in an ecosystem depend on one another [1]. The dependencies that exist between projects in an ecosystem define the structure of the ecosystem [2]. Thus, identifying dependencies between software projects is a useful way to identify ecosystems. By using our method called reference coupling, we were able to identify technical dependencies between projects and visualize the corresponding ecosystem. Recent research has shifted to studying software ecosystems (SECOs), which are defined as “a networked community of organizations or actions, which base their relations to each other on a common interest in the development and use of a central software technology” [1]. Since software projects are not typically developed as isolated, monolithic projects, studying a software project without examining its surrounding ecosystem is incomplete [64]. Literature suggest that these ecosystems can be effective in constructing large software systems on a software platform by composing software components developed by actors both internal and external [39, 40]. In these ecosystems, all the actors coexist interdependently, sometimes without
realizing what roles they play as part of the larger software system. A way to identify ecosystems would be beneficial to these actors, by giving them an understanding of how their tasks fit into the big picture and understanding who they need to coordinate their changes with at the ecosystem level [41], [42].

Based on the technical dependencies we found, we asked the research question:

RQ2: *What ecosystems exist across GitHub-hosted projects and what is their structure?*

### 3.1 Background and Related Work

Software ecosystem is a term introduced in 2003 by Messerschmitt, in which he describes “a collection of software products having some degree of symbiotic relationships” [43]. It is a derivation from the natural biological system, since living life forms collaborate with and rely upon one another in a mutual domain. Depending on the domain, the definitions of a software ecosystem takes on different meanings from the social perspective, to the business aspects and finally to interdependency between projects as highlighted by Table 1 below. The current software development projects involve distributed and inter-connected network. In this thesis we focus on inter-dependent projects and the technical dependencies that create these ecosystems. We make use of the definition by Lungu [1] which describes a software ecosystem as

> “*a collection of software projects which are developed together and co-evolve together in the same environment*”.

An environment can be

- a technological platform
- a company
- a software community – e.g. the open-source community.

Table 1 - Definitions of Software Ecosystems relative to its Domain

<table>
<thead>
<tr>
<th>Domain</th>
<th>A Software Ecosystem is . . .</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Technology- and Market-based</td>
<td>“a set of businesses functioning as a unit and interacting with a shared market for software and services, together with the relationships among them. These relationships are frequently underpinned by a common technological platform or market and operate through the exchange of information, resources and Artifacts” [23] by Jansen et al.</td>
<td>Google/Android, Apple/iOS apps, Eclipse</td>
</tr>
<tr>
<td>Social and Business Communities</td>
<td>“a set of software solutions that enable, support and automate the activities and transactions by the actors in the associated social or business ecosystem and the organizations that provide these solutions” [39] by Bosch et al.</td>
<td>Facebook API, Dropbox API</td>
</tr>
<tr>
<td>Third-party, Internal and External Developers</td>
<td>“a software platform, a set of internal and external developers and a community of domain experts in service to a community of users that compose relevant solution elements to satisfy their needs” [42] by Bosch et al.</td>
<td>EchoNest API, Spotify API</td>
</tr>
<tr>
<td>Inter-dependent projects</td>
<td>“a collection of software projects that are developed and evolve together in the same environment” [1] by Lungu et al.</td>
<td>SqueakSource, Apache, Gnome, Ruby Gem repository, Java Maven repository etc.</td>
</tr>
</tbody>
</table>
Werner et al [13] tried to classify current research into software ecosystems by creating a systematic mapping to produce a visual summary of how different dimensions may affect a SECO. These dimensions include the technical, business and social aspects of a software ecosystem. Studies of open source software ecosystems have mostly focused on the analysis of well-defined OSS ecosystems, such as Eclipse [5, 14], Ruby on Rails [15, 16], or Apache [7, 9]. In these well-defined ecosystems, the relationships between the actors are explicit and easily traceable in comparison to loosely defined ecosystems. However, challenges arise in trying to replicate the methods used in analyzing these well-defined systems on a large number of projects. We introduce a method identifying unknown ecosystems that exist by using community detection.

In previous studies around software ecosystems, networks have been used to understand the patterns of interaction between elements. These dependency networks find their origins in social network analysis and provide a distinct representation of correlation based networks, with the ability to uncover hidden information about the structure of an ecosystem. Baldwin et al [45] classify network graphs as the largely adopted method for visualizing ecosystems and their components. In these visualizations, projects are represented as nodes, and the dependencies between them are depicted by the edges. Other existing studies have suggested that, the use of network graphs can provide an understanding of the existence of hubs (central projects that drive the ecosystem), the size of an ecosystem, and how an ecosystem changes over time by comparing different network graphs of the same ecosystem [45, 46].

Aside from aiding visual perception and observation, these network graphs are subject to analysis using various methods [45, 47, 48, 49]. Network analysis involves a plethora of quantitative measures that portray the structure of a
network [50], all influenced by the level of centrality of a node as defined by [51]. Previous case studies have led to the visualizations of ecosystems such as contributors to open source projects on SourceForge [52, 53], the ecosystems of popular software firms such as IBM, Microsoft and SAP [47], and in evaluating the ecosystem health of cloud PaaS providers [54]. These studies focus on specific attributes of the ecosystems rather than the overall structure of the network.

Several studies [17, 18] have used community detection algorithms to detect communities across GitHub projects, but they have focused on dependency relationships between developers rather than technical dependencies between projects. Thung et al. [19] constructed project-to-project networks for GitHub-hosted projects, but edges between projects in their network represent a single developer contributing to both projects. This method can not be used to detect dependencies between projects since developers can often work on multiple independent projects and, thus, sharing developers is not an indication of a technical dependency. We use technical dependencies for community detection since the structure of an ecosystem is defined by its technical dependencies [3].

3.2 Methodology and Approach
In this study, the environment is GitHub as we are considering projects hosted on GitHub. GitHub encourages collaboration between users both within and across projects through its transparent interface and built-in social features.

To create this visualization, we constructed a network of the technical dependency relationships established through reference coupling as described in Chapter 2. The Dependency Network we constructed, is defined as a directed graph
$$G_D = \langle V, E \rangle.$$  

The set of vertices, denoted by $V$, is all the GitHub projects involved in at least one cross-reference.

**Table 2. Node table Structure**

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Id</th>
<th>Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>Astropy</td>
<td>19810</td>
<td>astropy/astropy</td>
</tr>
<tr>
<td>Mozilla</td>
<td>866182</td>
<td>mozilla/friendlycode</td>
</tr>
<tr>
<td>Blackberry-Webworks</td>
<td>843</td>
<td>blackberry-webworks/BB10-WebWorks-Framework</td>
</tr>
<tr>
<td>Zendframework</td>
<td>2598049</td>
<td>zendframework/Component_ZendDb</td>
</tr>
<tr>
<td>Jquery</td>
<td>1965156</td>
<td>jquery/jquery-wp-content</td>
</tr>
</tbody>
</table>

The set of edges, denoted by $E$, is a set of node pairs

$$E(V) = \{(x, y) | x, y \in V\}.$$  

If the project represented by node $x_i$ cross-referenced the project represented by node $y_j$, there is a directed edge from $x_i$ to $y_j$.

A mysql database was used to store the node and edges of the dependency network. Table 2 shows a subset of the node table. The nodes denotes the owner of the project e.g. zendframework, each with a unique ID. The label follows the pattern Owner/Project, indicating the name of the owner (e.g. jquery), and the distinct repository included the ecosystem (e.g. jquery-wp-content). We

---

⁷ https://github.com/zendframework  
⁸ https://github.com/jquery/jquery-wp-content
used this juxtaposition to distinguish between repositories because an owner can work on more than one repository, each repository having different names. The ID’s are also unique to the repository to ensure we ignored duplicates. Table 3 shows a subset of the edge table.

Table 3 - Edge table structure

<table>
<thead>
<tr>
<th>Source</th>
<th>Target</th>
<th>Type</th>
<th>Id</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>19810</td>
<td>6013</td>
<td>Directed</td>
<td>61964</td>
<td>9</td>
</tr>
<tr>
<td>1938026</td>
<td>25326</td>
<td>Directed</td>
<td>61965</td>
<td>2</td>
</tr>
<tr>
<td>181909</td>
<td>52321</td>
<td>Directed</td>
<td>61966</td>
<td>39</td>
</tr>
<tr>
<td>62373</td>
<td>128593</td>
<td>Directed</td>
<td>61969</td>
<td>17</td>
</tr>
</tbody>
</table>

The source and targets makes use of the unique ID’s of the projects that they connect. These relationships as found in Table 3, illustrate the couple references we found through our method. We make use of the open source software Gephi [16] to create our visualizations.

Gephi automatically assigns these source-target relationships with type ‘Directed’ for any directed graph. The weight of each edge indicates the count of cross-referenced comments for the pair of projects. We filtered out edges between nodes if there was less than two cross-references between a pair of projects. This was to ensure that we captured only the stronger dependencies in the network.

**Identifying Ecosystems:** To identify ecosystems across GitHub projects, we used the popular Louvain community detection method [11] on the Dependency Network. The algorithm works by optimizing and calculating the density of edges
(modularity) around groups of nodes. This method is based on the concept of modularity optimization of networks, which aims to partition a network into communities of densely connected nodes and optimize the modularity of a network [10]. Modularity is defined as “the number of edges falling within [communities] minus the expected number in an equivalent network with edges placed at random [25].” The Louvain method is comprised of two steps. It first optimizes modularity locally by looking for small communities. Then it aggregates the nodes in each small community and builds a new network with these aggregated nodes. It iterates on these two steps until the modularity is maximized. The Louvain method outperforms all other community detection methods in terms of both the modularity that is achieved and the computation time [10].

**Modularity:** High modularity scores indicate that there are dense connections within communities but sparse connections across communities, validating that an optimal solution has been found. Modularity scores range from -1 to 1. High modularity scores indicate that the detected communities are much more tightly connected by technical dependencies than would appear in a random graph. When high modularity scores are obtained, the communities possess a significant real world meaning [11]. On the first run of the algorithm, each node is placed into its own community. Subsequently, the algorithm computes the modularity of all communities in the network as well as the gain in modularity for each neighbor in a community. The highest detected modularity gain is used to merge two neighbors. Finally, the algorithm recalculates the modularity for all communities until there are no modularity gains possible. In our dependency network, the communities we identified represent sets of projects which are densely connected by technical dependencies. Since dependencies that exist
between projects define the structure of an ecosystem [1], we believe that these communities represent the software ecosystem of GitHub-hosted projects. To display nodes that are more prominent in each ecosystem, nodes were sized according to their authority.

**Visualizing Ecosystems:** We used various filtering algorithms available in Gephi to display our network due to its size and complexity including:

The Giant component which may be viewed as the largest cluster of nodes in a network. The filter works by eliminating nodes that are not connected to the largest cluster. The connected components isolated from the giant component are primarily comprised of same owner communities in which all nodes in the connected component are projects owned by the same GitHub user or organization.

Secondly, we used the edge weight filter on the network so as to only view nodes with more than two edges. Using the Fruchterman-Reingold layout algorithm for drawing force directed graphs [55], we created the network shown in Figure 3. For visibility, our final network model consisted of 1857 nodes connected by 4012 edges. Fruchterman-Reingold makes sure that nodes are placed within a predefined circular space, allowing for minor variations in the distances between nodes based on the weight of the edges [55]. This results in graphs that are easier to create and control, especially for large networks. Basole used this same algorithm in his visualization of the mobile ecosystem [56].
Figure 3 - All GitHub projects with cross-references. The largest connected component (or giant component) is easily identified as the well-connected subgraph appearing in the center of the graph.
3.3 Results

3.3.1 Visualization

Figure 4. Ecosystems in the largest connected component of Github-Hosted projects. Project names follow the pattern user/repository where user is the owner's Github login and repository is the name of the project repository.
Since we are most interested in studying the popular GitHub ecosystems, we focus our analysis on the interconnected part of the network or the giant component. Figure 4 shows the giant component. Of the projects in our dataset, 10,484 (57%) are a part of the largest connected component which is the largest subgraph in which every node in the network is connected to every other node by some path. For example, the second largest connected component is comprised of 65 nodes, of which, all but two are owned by GitHub user deathcap\(^9\).

This diagram was created using the following criteria: The node size is equal to the degree (which indicates the amount of incoming and outgoing connections with the node), and the edge size is equal to the edge weight. The color of the nodes represent communities as detected by the Louvain method. When a node has a high number of cross-references linking to it, it has a high authority value [25]. Table 5 shows the project nodes with the (highest authority value) in each of the ecosystems. We obtained a modularity score of 0.913 (out of a possible range of -1 to 1). This high modularity score indicates that the detected communities are much more tightly connected by technical dependencies than would appear in a random graph. There were 43 ecosystems identified in this network. Nodes are sized according to their authority to display the nodes that are more prominent in each ecosystem. When a node has a high number of cross-reference relationships pointing to it, it has a high authority value [18]. Table 5 shows the most well-connected project node (highest Authority value) in each of the ecosystems.

\(^9\) https://github.com/deathcap
Table 4 - General statistics of the dependency network

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Average Degree</td>
<td>1.414</td>
</tr>
<tr>
<td>Average Weighted Degree</td>
<td>7.857</td>
</tr>
<tr>
<td>Network Diameter</td>
<td>22</td>
</tr>
<tr>
<td>Graph Density</td>
<td>0.015</td>
</tr>
<tr>
<td>Modularity</td>
<td>0.913</td>
</tr>
</tbody>
</table>

3.3.2 Properties.

Ecosystems revolve around one central project

As depicted in Figure 4, each ecosystem appears to revolve around one main project. In Table 5, the most well-connected project node in each ecosystem is listed along with a description of the project, the number of stars the project has, the size of the associated ecosystem, and the node’s degree. Each of these

Figure 5 - Twbs/bootstrap Ego Network. Portraying a sample star pattern in the network projects has a higher in-degree than
<table>
<thead>
<tr>
<th>Project</th>
<th>Description</th>
<th>Stars</th>
<th>Ecosystem Size</th>
<th>Degree (in,out)</th>
</tr>
</thead>
<tbody>
<tr>
<td>joyent/node</td>
<td>Framework</td>
<td>39,373</td>
<td>10.08%</td>
<td>69 (53,16)</td>
</tr>
<tr>
<td>symfony/symfony</td>
<td>Framework</td>
<td>10,985</td>
<td>8.46%</td>
<td>93 (53,40)</td>
</tr>
<tr>
<td>rails/rails</td>
<td>Framework</td>
<td>29,744</td>
<td>7.92%</td>
<td>93 (65,28)</td>
</tr>
<tr>
<td>JuliaLang/julia</td>
<td>Programming Language</td>
<td>5,531</td>
<td>6.74%</td>
<td>51 (35,16)</td>
</tr>
<tr>
<td>rubygems/rubygems</td>
<td>Package Manager</td>
<td>1,304</td>
<td>6.04%</td>
<td>22 (14,8)</td>
</tr>
<tr>
<td>mxcl/homebrew</td>
<td>Package Manager</td>
<td>13,723</td>
<td>3.94%</td>
<td>48 (21,27)</td>
</tr>
<tr>
<td>zendframework/zf2</td>
<td>Framework</td>
<td>5,841</td>
<td>3.88%</td>
<td>72 (65,7)</td>
</tr>
<tr>
<td>travis-ci/travis-ci</td>
<td>Development Tool (Continuous Integration Platform)</td>
<td>3,693</td>
<td>3.50%</td>
<td>70 (54,16)</td>
</tr>
<tr>
<td>wet-boew/wet-boew</td>
<td>Framework</td>
<td>688</td>
<td>3.34%</td>
<td>19 (15,4)</td>
</tr>
<tr>
<td>twbs/bootstrap</td>
<td>Framework</td>
<td>41,828</td>
<td>3.29%</td>
<td>9 (9,0)</td>
</tr>
<tr>
<td>dbashford/mimosa</td>
<td>Development Tool (Browser development)</td>
<td>472</td>
<td>2.43%</td>
<td>25 (20,5)</td>
</tr>
<tr>
<td>h5bp/html5-boilerplate</td>
<td>Framework</td>
<td>31,926</td>
<td>2.37%</td>
<td>19 (15,4)</td>
</tr>
<tr>
<td>mitchellh/vagrant</td>
<td>Framework</td>
<td>9,274</td>
<td>2.10%</td>
<td>23 (15,8)</td>
</tr>
<tr>
<td>libgit2/libgit2</td>
<td>Library</td>
<td>5,161</td>
<td>2.05%</td>
<td>20 (11,9)</td>
</tr>
<tr>
<td>Behat/Mink</td>
<td>Development Tool (Testing)</td>
<td>673</td>
<td>1.99%</td>
<td>13 (9,4)</td>
</tr>
<tr>
<td>OCamlPro/opam</td>
<td>Package Manager</td>
<td>118</td>
<td>1.89%</td>
<td>9 (8,1)</td>
</tr>
<tr>
<td>basho/riak</td>
<td>Database</td>
<td>2,520</td>
<td>1.83%</td>
<td>27 (18,9)</td>
</tr>
<tr>
<td>Polymer/polymer</td>
<td>Library</td>
<td>8,787</td>
<td>1.83%</td>
<td>16 (11,5)</td>
</tr>
<tr>
<td>mapnik/mapnik</td>
<td>Development Tool (Toolkit for developing mapping applications)</td>
<td>1,003</td>
<td>1.78%</td>
<td>20 (12,8)</td>
</tr>
<tr>
<td>mozilla/rust</td>
<td>Programming language</td>
<td>5,604</td>
<td>1.78%</td>
<td>36 (29,7)</td>
</tr>
<tr>
<td>alphagov/static</td>
<td>Other (GOV.UK static files/resources)</td>
<td>67</td>
<td>1.73%</td>
<td>13 (10,3)</td>
</tr>
<tr>
<td>adobe/brackets</td>
<td>Development Tool (code editor)</td>
<td>23,921</td>
<td>1.46%</td>
<td>26 (16,10)</td>
</tr>
<tr>
<td>CocoaPods/CocoaPods</td>
<td>Development Tool (dependency manager)</td>
<td>5,711</td>
<td>1.46%</td>
<td>14 (9,5)</td>
</tr>
<tr>
<td>yeoman/yeoman</td>
<td>Development Tool (web development tools)</td>
<td>7,246</td>
<td>1.46%</td>
<td>18 (13,5)</td>
</tr>
<tr>
<td>angular/angular.js</td>
<td>Framework</td>
<td>42,950</td>
<td>1.40%</td>
<td>12 (8,4)</td>
</tr>
<tr>
<td>dotcloud/docker</td>
<td>Development Tool (application container engine)</td>
<td>14,270</td>
<td>1.35%</td>
<td>24 (19,5)</td>
</tr>
<tr>
<td>emberjs/ember.js</td>
<td>Framework</td>
<td>14,185</td>
<td>1.29%</td>
<td>20 (12,8)</td>
</tr>
<tr>
<td>owncloud/core</td>
<td>Other (personal cloud storage tool)</td>
<td>3,222</td>
<td>1.19%</td>
<td>26 (13,13)</td>
</tr>
<tr>
<td>typhoens/typhoens</td>
<td>Library</td>
<td>2,465</td>
<td>1.19%</td>
<td>6 (4,2)</td>
</tr>
<tr>
<td>facebook/nhvm</td>
<td>Other (Virtual machine)</td>
<td>11,506</td>
<td>1.08%</td>
<td>15 (10,5)</td>
</tr>
<tr>
<td>celluloid/celluloid</td>
<td>Framework</td>
<td>2,855</td>
<td>0.86%</td>
<td>9 (6,3)</td>
</tr>
<tr>
<td>xp-framework/rfc</td>
<td>Framework</td>
<td>0</td>
<td>0.86%</td>
<td>16 (14,2)</td>
</tr>
<tr>
<td>rogerwang/node-webkit</td>
<td>Framework</td>
<td>19,737</td>
<td>0.86%</td>
<td>16 (11,5)</td>
</tr>
<tr>
<td>documentcloud/underscore</td>
<td>Library</td>
<td>7,135</td>
<td>0.81%</td>
<td>6 (4,2)</td>
</tr>
<tr>
<td>middleman/middleman</td>
<td>Development Tool (website generator)</td>
<td>4,179</td>
<td>0.75%</td>
<td>8 (5,3)</td>
</tr>
<tr>
<td>elasticsearch/elasticsearch</td>
<td>Other (search and analytics tool)</td>
<td>10,700</td>
<td>0.70%</td>
<td>11 (11,0)</td>
</tr>
<tr>
<td>chapmanb/bcbio-nextgen</td>
<td>Other (RNA-seq analysis tool)</td>
<td>173</td>
<td>0.59%</td>
<td>10 (9,1)</td>
</tr>
<tr>
<td>wp-cli/wp-cli</td>
<td>Development Tool (command line interface for WordPress)</td>
<td>1,968</td>
<td>0.59%</td>
<td>13 (9,4)</td>
</tr>
<tr>
<td>cucumber/cucumber</td>
<td>Development Tool (Testing)</td>
<td>5,142</td>
<td>0.49%</td>
<td>7 (4,3)</td>
</tr>
<tr>
<td>jsdoc3/jsdoc</td>
<td>Development Tool (API documentation generator)</td>
<td>2,909</td>
<td>0.49%</td>
<td>6 (3,3)</td>
</tr>
<tr>
<td>propelorm/Propel</td>
<td>Development Tool (Object-Relational Mapping)</td>
<td>893</td>
<td>0.49%</td>
<td>7 (7,0)</td>
</tr>
</tbody>
</table>
the out-degree with the exception of the mxcl/homebrew\textsuperscript{10} project. On the other hand, low-degree project nodes are four times as likely to be dependent on another project than they are to have a project depend on them.

This shows that ecosystems are being formed around a central project with the other projects in the ecosystem mostly depending on that central project. This results in a star pattern. The twbs/bootstrap\textsuperscript{11} ego network (Figure 5) clearly depicts this pattern within the graph.

\begin{figure}
\centering
\includegraphics[width=\textwidth]{ego_network}
\caption{Ego Network.}
\end{figure}

The jsdoc3/jsdoc\textsuperscript{12} ego network provides another representation of the star pattern in the network. Jsdoc is an API documentation generator built for JavaScript. In this case, while the other projects in the ecosystem depend on

---

\textsuperscript{10} http://brew.sh/
\textsuperscript{11} https://github.com/twbs/bootstrap
\textsuperscript{12} https://github.com/jsdoc3/jsdoc
jsdoc (and maintain the star pattern), the tool itself is largely dependent on its documentation (found in the repository jsdoc3/jsdoc3.github.com) and vice-versa. This is evident by the weight and direction of the edge between the two repositories. These central projects are the hubs of the ecosystems and provide a basis for projects in that ecosystem to coexist. Open source practices ensure that these central projects enjoy a high level of freedom and can influence the development of the surrounding ecosystem. Generally, these projects are responsible for the further development of the base technology used by the projects that are part of the ecosystem.

*Predominant type of ecosystems is software development support.*

Interestingly, nearly all of the ecosystems are centered around projects whose purpose is to support software development, such as frameworks, libraries and programming languages. In fact, of the 43 ecosystems we found, there are only 5 whose purpose is not to support software development. Frameworks are essential to the software development process as they provide an effective way to improve variables such as productivity and product quality [57]. There are 13 frameworks, 5 libraries, 3 package managers, 2 programming languages, 1 database, and 14 other tools that support software development like a testing tool, a continuous integration platform, and an API documentation generator. The framework joyent/node\textsuperscript{13} commands the largest ecosystem and is responsible for 10.8% of the dependency network. Node.js\textsuperscript{14} was created as a JavaScript framework for building fast, scalable network applications, and has become one of the most popular code repository on GitHub (Popularity is determined by the

\textsuperscript{13}https://github.com/joyent/node

\textsuperscript{14}https://nodejs.org/
number of watchers on a repository)[70]. This is not entirely unexpected, as the JavaScript programming language is the most popular language on GitHub, judging by the amount of repositories that actively use the language [58]. The project alphagov/static\textsuperscript{15}, controls the largest ecosystem whose purpose is not software development support. The application “static”, used to define global templates for pages on the GOV.UK\textsuperscript{16} website and also for managing static files and resources is controlled by the same GitHub organization. One of the reasons for the size of this ecosystem may be the need for coordination across the different components of the website.

\textit{Ecosystems are interconnected}

The graph in Figure 3 shows two types of communities that occur in GitHub-hosted projects, those that are part of the largest connected component and those that are isolated from the largest connected component. The majority of project nodes, 10,484 or 57%, are involved in the largest connected component, indicating that many ecosystems are connected to each other across the projects in our Dependency Network. The next biggest connected component in the graph is only 65 nodes indicating that the ecosystems that are isolated are small and have not attracted public attention. Figure 4 displays the interconnected part of the network, and the connections between the ecosystems are apparent.

\textsuperscript{15} https://github.com/alphagov/static
\textsuperscript{16} https://www.gov.uk/
Figure 7 - The Rails Ecosystem

As an example, Figure 7 shows the rubygems/rubygems\textsuperscript{17} ego network, clearly depicting its connection to the rails/rails\textsuperscript{18} project. This is not surprising, since the rubygems project is a package management framework for the Ruby programming language and rails/rails is a web application framework written in Ruby. There is a direct connection between the rubygems/rubygems and rails/rails nodes. In addition, there are projects, like carlhuda/bundler\textsuperscript{19} and airblade/paper_trail\textsuperscript{20}, which connect the two projects.

\textsuperscript{17} https://github.com/rubygems/rubygems
\textsuperscript{18} https://github.com/rails/rails
\textsuperscript{19} https://github.com/bundler/bundler/
\textsuperscript{20} https://github.com/airblade/paper_trail
Open-source software practices bolster an overall, organized coordinated effort and an interdependence among autonomous developers. Developers are free to contribute to as many projects, and projects can feed off other projects. This results in a high level of interconnectedness between projects as is apparent in the ecosystem. The isolated ecosystems tend to contain projects owned by the same GitHub user or organization.

3.4 Conclusion of Study
In this study, we explored the visualization of the GitHub ecosystem, focusing on the technical dependencies we discovered in study 1. To achieve this, we created a dependency network of GitHub projects and applied the Louvain community detection algorithm to identify densely connected communities of dependent projects. We identified 43 interconnected ecosystems in our network and inspected the visualization of the network to discover patterns.

Our findings indicate that: projects in an ecosystem co-exist around a single project which are mainly projects that support software development processes e.g. Frameworks, and while small projects remain in isolation, the majority of ecosystems are interconnected.
Chapter 4

Social behavior of project owners/contributors

In this chapter, we examine the impact of project owners and contributors on technical dependencies and the ecosystem. Using statistical methods and predefined social relationships on GitHub, we analyze social behavior in a variety of ways, including the level of awareness with regards to the existence of technical dependencies.

To complement our investigation of technical dependencies and connectedness of projects in GitHub, we also sought to understand the social behavior of project owners and contributors in relation to their software ecosystem. We studied two of GitHub social relationships, following users and starring projects. On GitHub, users can follow other users to receive notification on their activity and star a repository to bookmark it or indicate interest in the project. To understand how the social behavior of project owners/contributors relates to the identified ecosystems, we examine the alignment between social and technical connections between the projects.

To adequately direct our inquiry, we ask the following research question.

RQ3: Do the project owners’ and contributors’ social behaviors align with the technical dependencies?

4.1 Background and Related Work

Recent studies have tried to understand the behavioral patterns of developers in social coding platforms. Specifically, research has tried to examine how project management occurs in GitHub. Dabbish et al [37] suggest some project
management activities that occur include recruitment, identifying user needs and most importantly, managing dependencies. They suggest that owners attend closely to changes on the project they are dependent on, by either subscribing or watching for commit events. Transparency in GitHub supports this behaviour because a project owner or contributor can watch or follow any open source project which supports knowledge sharing and community building. Tsay et al [71] evaluate the influence of social signals and cues on contributions. They find that developer engage both social and technical information when evaluating potential contributions to open source projects. Yu et al [72] explored social behaviour in GitHub by constructing follow-networks using the follow behaviors among developers in GitHub. The networks show that developers either follow acquaintances in an independence pattern, or follow other developers in order to collaborate on the same project. They also find that the more developers are followed, the faster their project grows. In our study, we focus constructing networks that illustrate the follow relationship of project owners and contributors on GitHub.

In this study we evaluate the social behavior of project owners and contributors in comparison to technical dependencies. We use statistical methods to confirm if there is an alignment between the social behaviour of developers and the technical dependencies we discovered using our method. The existence of an alignment indicates that developers may be aware these dependencies and make use of the available social features to manage them.

4.2 Methodology and Approach
To answer our research question, we construct project-to-project networks based on the following and starring activity of the project owners and contributors. We ran correlations between the edge weights in the Dependency Network with the
edge weights in each social network to determine if there is a relationship between the technical dependencies and the social connections. Pearson correlations were used since the data was normally distributed.

**Project Owners**

We constructed two networks using the following and starring relationships by considering the actions of the project owners. The Owner Stars Network,

\[ G_{os} = \langle V; E \rangle \]

and the Owner Follows Network,

\[ G_{of} = \langle V; E \rangle \]

are both undirected graphs whose set of vertices is all GitHub projects involved in at least one cross-reference. For the Owner-Follows Network, there is an edge from nodes \( x_i \) to \( y_j \) if the owner of project \( x_i \) follows the owner of project \( y_j \). There is an edge from \( x_i \) to \( y_j \) in the Owner Stars Network if an owner of any project in our dataset has starred both project \( x_i \) and project \( y_j \). To compare the social connections with the technical dependencies, we compare the edge weights of these two networks with the edge weights of the Dependency Network described earlier. The edge weights of these three networks represent the following:

- **Dependency Network** \( G_D \): Number of technical dependencies, measured through reference coupling, between the two project nodes.

- **Owner Follows Network** \( G_{of} \): 0 if neither project owner follows the other, 1 if one project owner follows the other project owner, and 2 if both project
owners follow each other.

- Owner Stars Network $G_{os}$: Number of project owners who have starred both projects.

![Figure 8 - The Owner Follows Network $G_{of}$](image)

**Project Contributors**

We constructed two additional networks using these following and starring relationships by considering the actions of the project contributors (users who have made commits on the project or are members of the project).
The Contributor Stars Network, $G_{CS} = (V; E)$, and

The Contributor Follows Network, $G_{CF} = (V; E)$,

are also undirected graphs whose set of vertices is all GitHub projects involved in at least one cross-reference. The Contributor Follows Network has an edge from nodes $x_i$ to $y_j$ if a contributor of project $x_i$ follows a contributor of project $y_j$. The weight of each edge is the count of contributors with following relationships for the pair of projects. The Contributor Stars Network has an edge from $x_i$ to $y_j$ if a contributor to any project in our dataset has starred both project $x_i$ and project $y_j$. The weight of each edge is the count of project contributors who have starred both projects.

Figure 9 - The Contributors Follow Network
4.3 Results

Project Owners. Table 6 shows strong, positive correlations between the technical dependencies and the social behavior of the owners. Along with these strong correlations, Figure 8 shows a pronounced star pattern in the unfiltered Owner Follows Network. This indicates that the project owners in an ecosystem tend to follow the owner of the central repository.

Table 6 - Project owners: correlations between technical dependencies and social behavior

<table>
<thead>
<tr>
<th></th>
<th>Pearson Correlation</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Technical Dependencies and Following</td>
<td>0.91</td>
<td>&lt;0.001</td>
</tr>
<tr>
<td>Technical Dependencies and Stars</td>
<td>0.79</td>
<td>&lt;0.001</td>
</tr>
</tbody>
</table>

Project Contributors. As shown in Table 7, the social behavior of project contributors does not align with the technical dependencies. This indicates that, while the project owners seem to follow the right people and are aware of the right projects based on the technical dependencies that exist in the ecosystem, the social behavior of project contributors is not aligned with project dependencies. Figure 9 shows the unfiltered Contributor Follows Network. As shown, the structure is quite different than the Dependency Network. Communities do not have one central project and the network is much more densely connected.
Table 7 - Project contributors: correlations between technical dependencies and social behavior

<table>
<thead>
<tr>
<th></th>
<th>Pearson Correlation</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Technical Dependencies and Following</td>
<td>0.0002</td>
<td>0.98</td>
</tr>
<tr>
<td>Technical Dependencies and Stars</td>
<td>0.001</td>
<td>0.88</td>
</tr>
</tbody>
</table>

4.4 Conclusion of Study

In this study, we analyzed the social behavior of project owners and contributors in the GitHub ecosystem, focusing on the technical dependencies we discovered in study 1. Our analysis involved constructing follow networks of both groups, and applying Pearson’s correlation between the degree of nodes sharing an edge to determine if there was an alignment between the social behavior and technical dependencies. In our case, the direction of the edges does not apply, since we only consider projects that share a link between the. Therefore, our choice of undirected graphs does not affect the Pearson’s coefficient.

We discovered a level of awareness for project owners with regards to the existence of technical dependencies, by studying the social behaviors of owners and contributors. Contributors did not exhibit the same patterns as owners, leading to suggestions that they may not be aware these dependencies exist.
Chapter 5

Discussion

In this dissertation, we describe our exploration into identifying technical dependencies across a large number of projects. While the results of each study is outlined in the chapters above, this chapter focuses on the usefulness of our findings to the research community as well as discussing the implications for future research and for software practitioners.

This chapter includes 4 subsections. The first section discusses the identifying technical dependencies and the lessons learned from applying our method. The second section will address software ecosystems and the importance of understanding the properties of software ecosystems. Furthermore, we highlight the implications of our findings in section three. Finally, we provide some threats to validity to our study.

5.1 Discussion of Software Ecosystem based on Technical Dependencies

Identifying technical dependencies

Previous research has addressed the challenges involved with identifying technical dependencies [3, 4, 6, 8]. While opinions may differ on what approach is best to handle this problem, there is a unifying agreement as it relates to the importance. It has been found that technical dependencies created by syntactic relationships, have a tendency to capture a generally slender perspective of dependencies which does not completely illustrate the essential dependencies that encourage coordination [59]. As such, methods that detect technical dependencies through analysis of source code or configuration files may not be
best suited for identifying software ecosystems. For example, when one project uses another project, it does not necessarily mean the two software projects are evolving together in the same environment, especially when the dependency is to an established, off-the-shelf software package. Thus, identifying all relationships that manifest in the source code or configuration files may result in dependencies that are not important for the identification of ecosystems.

Our method detects technical dependencies that may not manifest themselves in source code by identifying issues, pull requests or commits that rely on another project, and, therefore, it can identify dependencies not identified by other methods. Our method is analogous to the logical coupling method that detects dependencies within a project proposed by Gall et al. [19] except at the ecosystem level. Where logical coupling detects dependencies when artifacts have been worked on together, our method detects dependencies when issues, pull requests or commits have been worked in conjunction with another project (as evidenced through user-specified cross-references). Thus, the dependencies established through our method are those that are logical.

Our method also allows for the identification of dependencies across all projects hosted on GitHub. Other methods that detect dependencies are limited to analyzing a given project or set of projects. Analyzing the dependencies of a popular project through its source code or configuration files to identify its ecosystem would not identify projects that rely on that project. We saw that most ecosystems across the GitHub-hosted projects are centered around one main project and many projects depend on that project without a reciprocal relationship. These relationships would be missed if only the dependencies of the main project were studied to identify its ecosystem. Since the ecosystems are not
always well-defined, it would be impossible to know which other projects to consider for analysis. Thus, our method is better suited to identifying ecosystems since it is not limited in the number of projects it can analyze. With reference coupling, we now have the capacity to track and oversee technical dependencies over the new scene of open source software development which spans across numerous projects, repositories, and cross language support.

**Logical Dependencies**

Our method limits the dependencies to those that are logical which is important when using those dependencies to identify ecosystems. Logical dependencies refer to transformative dependencies that are set up among source-code records that frequently change together (albeit not structurally related) [19, 60, 61]. Logical dependencies give a more exact representation of the dependencies that directly influence the development process [59]. This is because logical dependencies are computed between artifacts that have developed and been altered ordinarily together, despite the fact that these artifacts have no unequivocal association [61]. It is important to note that logical dependencies are not totally unique in relation to syntactic dependencies. As we discovered from the dependencies we found, they can range from an explicit syntactic relationship (e.g. a cross-reference is created in project A with a direct reference to a method in the source code of project B), to more unpredictable semantic conditions where an update in one project affects another project or multiple projects that depend on it.

The logical dependency approach is more valuable for ecosystem identification because it can channel syntactic dependencies that may be applicable in regards to the coordination needs of developers. For example, the syntactic dependency
methodology highlights connections among fundamental libraries at a high level (e.g. memory administration), in light of the fact that they contain very coupled files. Analyzing the dependencies of a popular project through its syntactic relationships would not identify projects that rely on that. Essentially, the logical dependency methodology can possibly identify vital dependencies not noticeable in the analysis of code syntax, as in our case where dependencies across projects were established from developer interaction on the code base.

**Social Coding**

As we observed in study 3, when dependencies exist between a pair of projects, the project owners tend to be following the owner of the other project. Conway was the first to describe the possibility of an alignment between social connections and technical dependencies in software engineering projects, commonly referred to as Conway’s law [21]. Based on existing literature, we expected that when dependencies exist between software artifacts, there would be an inherent creation of dependencies between developers which in turn creates a need for coordination [22, 23, 30]. Therefore, our analysis centered around cross-references created by developers to fulfill coordination requirements. Our findings indicate that these cross-references are driven by the existing dependencies.

Social coding is an approach to software development that allows for coordination among developers by providing an environment that supports discourse and creation in either a formal or informal setting [30, 37]. Our method explores the components of social coding platforms and the dynamic role they play in facilitating coordination. The transparent nature of GitHub as well as the provision of “social-coding” tools, facilitates socio-technical coordination by
creating awareness among developers. Awareness in the case of GitHub involves shared information about projects, people, actions on code artifacts etc. Transparency allows for a continuous interaction between the social and technical aspects of development in a project’s repository [37]. GitHub markdown feature provides a notification service that is aware of the projects being executed, and easily customizable to meet the needs of users. Our method proposed the consideration of developer’s comments in detecting technical dependencies between projects.

Of course, what makes for effective social coding is an open question to consider. A positive influence of awareness for the developer would be a full grasp of the dependencies that exists in the project’s environment which comes with a better understanding of the project. On the negative side, information overload may cause a problem for the developer, and time could be spent on understanding the dependencies as opposed to working on the projects. As stated in [68], developers when given the choice, often prefer productivity over awareness.

### 5.2 Discussion on Properties of Software Ecosystem

Understanding the context and properties of a software ecosystem is an essential part of defining the development process in a software environment. This is because, it is very important to determine the role a particular software system plays in the bigger picture of a software ecosystem. The process requirements for a project providing a platform differs significantly from the process requirements of projects that use these platforms. For instance, the communication practices of the central project in an ecosystem would not be the same as the other projects. Developers of these central projects require feedback and broadcast mechanisms, and may setup communication channels
to receive feedback on their projects as well as announce update information to the ecosystem. The other projects in the ecosystem may choose to set up communication channels to get updates from the central projects, provide feedback, or more importantly collaborate with other projects in the ecosystem.

Our results show that the majority of ecosystems are centered around projects that support the development process, such as frameworks and programming languages. Understanding the properties and practices of these central projects allows for a level of focus on the kind of information a developer needs when working at the ecosystem level. Information on who uses their framework and how these frameworks are used are beneficial to developers.

In one aspect, it provides a level of motivation to the developers, allowing them estimate the impact of code changes to the ecosystem. Developers of these central projects can choose to notify other developers in their ecosystem on how to stay compatible when the code base changes. Additionally, information on how these frameworks are being used would provide insight into unused components. For example, a developer can decide to deprecate an unused component, so as to better distribute time and effort; thus improving efficiency.

Some additional knowledge benefits that can be achieved include license compatibility, commit monitoring, open-source support availability, and the quality of implementation at the ecosystem level. Given that these ecosystems are interconnected, the same applies to other ecosystems and fosters social coding as described in the discussion above.
5.3 Implications for research

The ability to easily identify technical dependencies between a large set of projects opens the door for many interesting avenues of research.

Socio-technical analysis. Studies that have attempted to study how communication aligns with dependencies across projects have been limited to studying well-defined ecosystems where dependency information is publicized in some way. For example, dependencies can be made available through a project’s configuration files, dependency manager files or through publicly available dependency specifications [1, 6, 7, 8, 9, 20]. Our method allows the identification of technical dependencies more broadly across projects and opens the door to continuing the study of socio-technical alignment across a larger set of projects and their stakeholders.

In this study, we found that when dependencies exist between a pair of projects, the project owners tend to be following the owner of the other project. Conway was the first to describe the possibility of an alignment between social connections and technical dependencies in software engineering projects, commonly referred to as Conway’s law [21]. The transparent nature of GitHub could encourage technical connections between projects by providing an awareness of activity across projects. An interesting future research question is understanding how and when these technical dependencies and social connections came to exist. Did the social connections exist first and result in a technical dependency or did the technical dependency exist first and result in a social connection? If the social connections existed first, what was the driver behind the creation of the technical dependency? Perhaps, the awareness of the other project, enabled through GitHub’s notifications, was enough to spur a technical dependency indicating that GitHub’s transparency is changing the
landscape of OSS projects. These research questions could be investigated in future research.

While the project owners’ social behaviors (following users and starring projects) aligned with the technical dependencies in our study, we did not witness such an alignment for all project contributors. The follower network of project contributors showed that there were no clear central projects and communities were densely connected. This is in contrast to the technical dependency network. These results align with recent research that found that the reasons behind following others extends beyond project coordination needs [22]. Future work should investigate the usefulness of following others for coordination purposes.

It is also worth studying in more detail the coordination needs of developers on OSS projects. Perhaps the mere existence of a technical dependency does not imply a coordination need, especially given the transparent environment of GitHub. Previous work [23] have already begun this investigation, but coordination needs at the ecosystem level are also worthy of investigation.

Ecosystem emergence and evolution. The most prominent nodes in Figure 4 are not always the most popular projects on GitHub when considering the number of stars each project has. In fact, the two projects with the most stars, angular/angular.js\textsuperscript{21} and twbs/bootstrap, have significantly smaller ecosystem size and lower degree than other projects. Future work can investigate how and why ecosystems emerge and why some projects become popular without growing a large ecosystem. Such a study could include a temporal analysis of the composition of the ecosystem and density of connections together with a

\textsuperscript{21} https://github.com/angular/angular.js/
temporal analysis of project history information such as number of contributors, forks, stars, etc. It would also be worth triangulating results with other information on important project events now commonly available through blogs and wikis. Such a study on the evolution of ecosystems can be a first step in understanding when and why projects accumulate an ecosystem.

_Ecosystem size and strength of connections and project success._ On many open source projects, volunteers are crucial to project success as they rely on volunteers to submit new features and fix bugs. As a project accumulates more projects in its ecosystem, it is also likely to increase its contributions as developers on dependent projects will be more likely to fix bugs that they encounter through their dependency. Future research could investigate this relationship to identify if the size of a project’s ecosystem is a good predictor of various project health and success metrics like the number of contributions it receives or the number of forks it has.

_Automatic detection of ecosystems._ Another avenue for future research is creating tools to support developers at the ecosystem level. It is important for developers to know who they need to coordinate with across the ecosystem and to understand how their tasks fit into the big picture. A tool could be developed that automatically identifies technical dependencies across projects and provides a visualization of the ecosystem. Such a tool could increase awareness of coordination needs that extend outside project boundaries and help developers gain a better view of the ecosystem surrounding their project.

5.4 Implications for Practice

The ability to easily identify technical dependencies between a large set of projects is also important to software companies and developers.
For organizations and coordinators. Software ecosystem governance is defined as the processes and procedures used by an organization to control, change or maintain its current and future position in a software ecosystem [62]. Software ecosystems are usually governed or driven by one or more coordinating parties who control the underlying technology on which the ecosystem is built e.g. The company Joyent\(^{22}\), controls the node.js platform and therefore controls the node.js ecosystem. In some case, there are single developers or a community of developers with a huge influence on the ecosystem e.g. the Travis-ci\(^{23}\) ecosystem. Our study shows this pattern, as the ecosystems revolve around one major project. The ability to identify the dependent projects and the size of an ecosystem would provide software ecosystem coordinators with valuable information for adequate ecosystem management. These coordinating parties can decide on and implement practices that would ensure the continuous improvement and robustness of the ecosystem. For open source ecosystems, it is also important for attracting new contributors [42, 62] since projects within an ecosystem are more likely to attract attention. For example, the decision on the level of accessibility of a project, plays a major role in determining what types of contributors will play a part in it.

For developers. Ecosystem identification is important to help developers understand how their tasks fit into the big picture and who they need to coordinate their changes with at the ecosystem level [1], [11]. The direct implication of this research to the development process is that of awareness among developers. As was stated, open-source developers can better focus on

\(^{22}\)https://www.joyent.com/
\(^{23}\)https://travis-ci.org/
coordination needs at the ecosystem level, improving efficiency. An indirect implication is that software reuse is made easier. For example, for a new project in the GitHub environment, the owner armed with the knowledge of which tools they require, can easily identify which projects in that ecosystem they can collaborate with and which software is available for reuse.

5.5 Threats to Validity.

Although we tried to efficiently gather data and apply adequate analysis in this thesis, our study suffers from several limitations. As this study is exploratory, our findings may not be entirely representative. We examine the internal and external threats to the validity of our research. Internal validity assesses the extent of bias in a study, to determine how significant and reliable our findings are within the context of the study. While external validity gauges the degree to which our findings are generalized to other populations or contexts [69].

5.5.1 Internal threats to validity

Internal validity is concerned with our ability to influence the analysis of the study and the conclusions derived from that analysis. We identified some internal threats we sought to mitigate.

Firstly, while we achieved saturation in our results, our results could be impacted by selection bias. To mitigate this, I ensured an equal number of comments for each source (commit, issue, pull request) were included in our sample. Further, for each repository, only I performed the manual analysis introducing a possible risk of unreliable results. However, the types of dependencies identified seem reasonable for any software project. Future work can continue this investigation by examining the content of cross-reference comments across a wide range of projects and code hosting environment.
Another threat stems from our selection of the GHTorrent dataset, which may not be a full copy of all GitHub data [12]. Nevertheless, it is a best-effort approach that has been widely accepted in the research community as evidenced by its inclusion as the dataset for the MSR 2013 Mining Challenge [24] and the many recent papers that utilize its data in their analysis. Furthermore, the cross-references we found from the data does not take into account, project communication using private channels. It has been argued however, that discussion on open source projects should be kept public and archived in order to enforce the collaborative nature of this form of software development [65, 66]. This is because private discussions may be detrimental to the process of open participation.

Finally, seeing that most repositories on GitHub are inactive [67], it makes sense that some of the cross-references we found belong to inactive projects. This could potentially introduce a bias in our results. However, we argue that our findings reflect the entirety of the open source projects on GitHub since dependencies would still exist regardless of whether these projects are active or inactive.

5.5.2 External threats to validity

External threats affect the generalizability of our findings. Our manual exploration of cross reference comments illustrates a variety of types of technical dependencies, but these results also can not be generalized. One threat is that our analysis was limited to only projects hosted on GitHub. We do not extend our results to other coding hosting environments. However, we argue that the entirety of GitHub projects accurately reflects the open source landscape. This is based on the relative popularity of GitHub to other code hosting services, as well
as the increase in project transfer to the site [67]. Future work can investigate whether similar cross-reference comments that indicate technical dependencies are found in other coding hosting environments.

Chapter 6

Conclusions

In this thesis we propose a method for detecting technical dependencies across software projects which we call Reference coupling. This method utilizes user-specified cross-references between projects which exhibit dependencies. In the open source software community where software reuse has become the norm via dependencies on frameworks, any project usually has a large number of dependencies. Identifying technical dependencies between a large number of projects is important because it allows for the identifying software ecosystems of dependent projects.

In our first study, we extracted user-specified cross-references made by developers on issue, pull requests or commit comments on GitHub. Next, we applied grounded theory to verify if these cross-reference displayed an existence of technical dependencies between projects. Through our analysis, we found the 90% of the references we examined, were created due to different types of dependencies between projects. Lastly, we categorized the different types of dependencies we found which include; direct dependencies between two
projects and dependencies between two projects created due to their dependency to a third project. Since the dependencies we found were not extracted from the source code or configuration files but through the developer coordination practices, we highlight that these dependencies are more logical. The reference coupling method is able to identify dependencies that are not detectable through source code. This provided the basis for the creation of an ecosystem of dependent projects in Study 2.

In our second study we presented the analysis steps involved in visualizing the ecosystem of open source software projects on GitHub. We made use of an existing community detection algorithm to identify 43 densely connected connected clusters of dependent projects. Further analysis of the identified ecosystems, showed that most ecosystems are centered around a single project. While small, unpopular ecosystems remain isolated, most ecosystems are interconnected. The isolated ecosystems tend to contain projects that are managed by the same GitHub user or organization. We also found that the popular ecosystems are mostly centered around tools that support software development such as frameworks, and programming languages.

This research also addressed the social behavior of project owners and contributors with respect to the technical dependencies. By analyzing two GitHub features, following and starring, we ran some correlations to determine if there was alignment with the identified technical dependencies. Our results suggest that project owners tend to follow the project they depend on. This led us to believe that project owners are aware that these dependencies exist. The same cannot be said for project contributors as we did not find any correlation between their behaviour and the technical dependencies.
The main contribution of this work is an understanding of inter-project dependencies in open-source software development. This understanding is beneficial to the open source community because it provides awareness to developers on how their tasks fit into the big picture, and who they can coordinate with at the ecosystem level. It also creates an avenue for attracting new contributors. Although we do not generalize our results, future research can look into replicating our method on different software development platforms. Additionally, our study opens the door for more research in software ecosystems including the study of socio-technical relationships, evolution, health and governance, and the success of these ecosystems.
Bibliography


[9] D. M. German, J. M. Gonzalez-Barahona, and G. Robles, “A model to understand the building and running inter-dependencies of software,” in


