Declaration Patterns in Dependency Management

A thesis presented in partial fulfilment of the requirements for the degree of

Master of Science

in

Computer Science

at Massey University, Manawatū, New Zealand.

Jacob Stringer

2020

Abstract

Dependency management has become an important topic within the field of software engineering, where large-scale projects use an increasing number of dependencies to quickly integrate advanced functionality into projects. To take advantage of agile principles - with their fast release cycles - it has become common to delegate the task of dependency management to package managers, whose responsibilities it is to find and download a specified version of the dependency at build time. The principles of *Semantic Versioning* allow developers to specify version declarations that allow package managers to choose from not just one, but a range of versions, giving rise to the automatic updating of dependencies - a convenient but potentially risky option due to backwards incompatibility issues in some updates.

In this thesis, we examine the types of declarations used and their effects on software quality. We find a large variation in practices between software ecosystems, with some opting for conservative, fixed declaration styles, others that prefer *Semantic Versioning* style ranges, and a few that use higher risk open range styles. We then delve into the consequences of these declaration choices by considering how they affect *technical lag*, a software quality indicator, finding that declaration styles can have a significant effect on lag.

In order to avoid technical lag, in all but the most extreme cases (using open ranges), it is necessary to update declarations periodically. In the case of fixed declarations, updates must be made with every change to the dependency - an ongoing challenge and time outlay for developers. We considered this case to find how regularly developers that use fixed declarations update lagging declarations, finding that developers rarely keep up with changes.

The datasets used for these works consisted of large-scale, open-source projects. A developer survey has also been included to contextualise the quantitative results, allowing insight into the intentions of developers who make these declaration choices, and to gain insight on how applicable these findings might be to closed-source projects.

Acknowledgements

I would like to express my gratitude to my supervisor, Dr. Amjed Tahir, who provided immense support and mentorship on my research journey. His unwavering optimism, literature recommendations and timely advice gave me the confidence to branch off to new questions.

I would also like to thank my former supervisor, A/Prof. Jens Dietrich, for starting me along my postgraduate journey and in this topic. His broad knowledge in the field gave me a strong overview to understand the context of these studies within the wider body of knowledge.

Much of the research in this thesis has been done as collaborative studies. I would like to acknowledge and thank Dr. Kelly Blincoe for her insights, work and collegiality in the creation of Chapters 5 and 7, along with Dr. David Pearce for his input into Chapter 5.

I acknowledge and extend my gratitude to my former colleagues within the CSIT department, whose regular conversations and advice have helped me shape the direction of this research.

Finally, I could not have done this without my husband, whose has been instrumental in nurturing motivation, and my parents, who have been a constant source of support and encouragement.

Contents

List of Figures

List of Tables

List of Publications

Chapter 5 consists of a collaborative work resulting in the MSR publication listed below. The author's contribution is estimated at 20% of the work. Chapter 6 was included in the same publication, but primarily represents the work of the author.

Dietrich, J., Pearce, D., Stringer, J., Tahir, A., & Blincoe, K. (2019, May). Dependency versioning in the wild. In 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR) (pp. 349-359). IEEE.

A paper derived from Chapter 7 has been accepted into the APSEC 2020 conference. This chapter represents the work of the author, except two sections (see footnotes within the chapter for details) - these sections have been included for their relevance to the topic.

Stringer, J., Tahir, A., Blincoe, K. & Dietrich, J., (2020, December). Technical lag of dependencies in major package managers. APSEC '20: Proceedings of the 27th Asia-Pacific Software Engineering Conference (pending publication).

Chapter 1

Introduction

Many modern software systems are built from existing packages (i.e. modules, components, libraries - henceforth, just *packages*). This realises a vision almost as old as software engineering itself: that is, in order to be on par with other parts of a modern economy, re-usable software components must be produced and used at scale [McIlroy et al., 1968]. Many common tasks are carried out using third party packages (called *dependencies*): logging, using packages such as Log4j (Java) or Serilog (.NET), quickly creating websites using web frameworks like Spring (Java) or Django (Python), or unit testing using mocking libraries such as Mockito (Java) and Moq (.NET) to isolate sections of code during testing. These dependencies can be extremely widespread across industry: Log4j has over 15,000 usages from other packages hosted on the Maven repository alone (which generally themselves are packages intended for reuse), and Moq has had 134,000,000 downloads from the NuGet repository at the time of writing.

In recent years, many systems have moved to a model where packages are stored in a central repository that is accessed from a client-side *package manager* (e.g. *Maven* for Java, *Cargo* for Rust, *CPAN* for Perl, etc). Such tools access packages as required (e.g. at build time) and, often times, have flexibility over which exact version to use. The use of package managers has considerably simplified the build process (compared with ad-hoc methods used previously) and, furthermore, enabled automatic *package evolution* (e.g. for the controlled propagation of security fixes and other improvements), decreasing the costs associated with updating dependencies. This has important implications on

agile concepts such as continuous delivery, where updates happen regularly, making it almost inevitable for dependencies without automatic updating to become outdated. Allowing dependencies to become outdated has implications not just on the stability and security of an application (due to missing out on bug or security fixes), but can make it impossibly difficult to update the dependency at a later point to take advantage of new features, as exemplified in Section 3.3.2.

Automatic updating of dependencies promises to simultaneously cut down on the time it takes for developers to keep their dependencies up to date, as well as reduce the lag time for bug fixes, security updates, and feature improvements to be integrated into a project (Section 3.5.2). However, it comes with risks, as not all updates are suitable to be automatically used within existing projects due to issues surrounding backwards compatibility. Versioning systems were implemented to order the published releases, but most did not convey any additional meaning beyond the order - they had no *semantics* built into the versioning scheme to explain the type of change made, and specifically if the change should be backwards compatible. An industry-led standard, semantic versioning (also known as semver)[1], was introduced to solve this problem. It provides a framework for developers to categorise the types of changes made, with the idea being that package managers can then automatically update a range of versions that meet specific criteria.

Semantic versioning has one major weakness. Currently, there are no automated ways for a developer to be able to tell what type of changes have been made since the last version of the project. A developer must manually categorise the changes made, which in semver terms, means to make a micro change (bug or performance fix), minor change (new feature), or major change (backwards breaking change). This can be a surprisingly difficult task, particularly when considering whether changes both preserve API stability, semantic contracts (Section 3.1), and binary compatibility (Section 3.2). Studies have found that potentially 1 in 3 version updates have been misclassified, by introducing a backwards incompatibility in an update which should preserve backwards compatibility (Section 3.3.1), an issue which has serious implications on automatically updating dependencies.

Given that there is no automated way of correctly updating version num-

[1] `https://semver.org`

bers, client projects automating dependency updates enter into a *social contract* with the developers of the dependency - a contract where client developers have to place trust that the dependency developers correctly use semantic versioning and correctly signal when a breaking change is being made. This is challenging, as there are few ways of incentivising the dependency's developer(s) to uphold this contract - potential reputation hits (amongst a potentially anonymous clientele) and increased bug reports are the only major downsides to getting it wrong. Allowing automatic updates means handing over a measure of control of a programme to a set of external developers who are generally not known to the developer in question, and who have limited incentives to care about not breaking a client programme. Therefore, the option of automatic updating comes down to a question of how much risk a developer is comfortable with, along with the reputation of reliability and backwards compatibility of the third-party project.

Despite the limitations of current approaches to automatically update dependencies, the results from this thesis indicate that developers do indeed find updating external libraries to be important. Given this tension, this thesis seeks to explain the ways in which developers update their dependencies - do they update their dependencies automatically or manually, or do they not bother at all with updates? How do these strategies affect project quality, and how common are each?

We use a mixed-method approach to answer the above questions. The bulk of the studies in this thesis centre around an open-source dataset from *libraries.io* (Section 5.2.1) which provides over 80 million dependency declarations from projects across a wide variety of software ecosystems.[2] This has allowed this study to make generalisations about the majority of open-source projects. A survey was undertaken that asked software developers currently working in industry several open ended questions about their approach to dependency management. There were over 170 completed responses from a variety of software ecosystems, experience levels, and geographical locations, providing further insights into the empirical results.

This thesis is structured as follow: it begins by introducing several key concepts within dependency management. Chapter 2 explains a number of key

[2]An ecosystem is defined as "a collection of software projects, which are developed and co-evolve in the same environment" [Lungu, 2009].

concepts related to these studies - semantic versioning, package managers, what declarations and updates are and how we have chosen to categorise them, and some background on the two datasets used. Chapter 3 guides the reader through current literature on dependency management and understanding the contracts that exist between upstream developers[3] and downstream developers[4]. It spends a significant amount of time discussing what backwards compatibility between projects looks like and discussing how to minimise disruption from breaking changes, before diving deeper into dependency networks, declaration patterns, technical lag[5], and update strategies - the latter three being our main topics of investigation. It concludes with an overview of what tools are currently available to automate the processes within dependency management.

Following this introduction, there are four research chapters presenting our findings. Chapter 4 begins as a prelude to the original work presented in this thesis - it represents the earliest experiments into declaration patterns. These experiments showed significant differences in declaration styles between the three package managers tested, and led us to delve further into this topic. Chapter 5 expands the empirical side of Chapter 4's findings by moving to the *libraries.io* dataset, which included 17 package managers, allowing much more generalisable conclusions on declaration patterns across open-source software. Chapter 6 qualitatively follows up on Chapters 4 and 5 by asking developers for their reasoning about dependency management strategies, giving us some insights as to why such significant differences exist between ecosystems. Finally, Chapter 7 investigates the consequences of these declaration patterns by considering how declaration types affect technical lag (a software quality metric), and how developers update their most restrictive declarations, fixed declarations.

We conclude this work with a discussion of the results in Chapter 8. There is significant scope to improve and simplify dependency management for the average developer, with stronger tooling integrated into software workflows being a major recommendation. We also suggest that additional understanding of backwards compatibility would improve semantic versioning adherence, and improve the efficacy of automated dependency updating.

[3]Upstream developers are those who produce the project used as a dependency.

[4]Downstream developers are those who consume a specific dependency.

[5]Technical lag is a software quality measure that determines how far out of date a project's dependencies are.

Chapter 2

Background

This chapter covers two key areas required to understand the section of dependency management that this thesis builds upon. The first topic is semantic versioning, which forms the basis for project versioning, and therefore the basis of this thesis. The second topic introduces package managers, and specifically the role that they play in dependency resolution.

These two concepts, one which defines versions, and the other which chooses one specific version out of multiple possible versions, will be used to understand how different flexible version declaration patterns will impact on the final version chosen by package managers.

2.1 Semantic Versioning

Semantic versioning[1] (abbreviated *semver*) is an industry-led standard for how versions should be updated. It encourages developers to adopt a standardised format for project version numbers, and to increment them in a way that carries meaning about the type of update that has occurred. In this way, developers that are planning to use this project as a dependency (*downstream* developers) are given a summary of the type of update made, and signals if the developer of the project that is a dependency (*upstream* developer) believes that this update can be used without any changes to the dependent project.

When we discuss versions in this work, we refer to semantic versions, as

[1] `https://semver.org/`

they standardise how versions are numbered. This not only allows for a sensible ordering to be enforced on a range of versions, but it additionally conveys information about the types of updates that projects undergo over their lifetime.

At the heart of semantic versions are three sets of numbers separated by periods. Each of these three numbers represents a different type of version update when changed, as shown in Figure 2.1:

1. MAJOR - Incrementing the left-most, *major* number signals that this update includes backwards compatibility breaking changes. There are likely to be many new features associated with these versions, along with extensive API redesigns which can cause downstream projects to require significant changes to their source code. A well known example of a major version change is when the Python language moved from version 2 to 3.

2. MINOR - Incrementing the middle, *minor* version signals that new features have been added to the programme. According to semver specifications, minor updates should not break backwards compatibility.

3. MICRO - Incrementing the right-most, *micro* version signals that bug fixes or security updates have been made. According to semver principles, it should not include backwards compatibility breaking changes or new features.

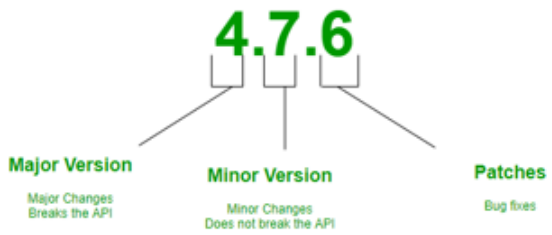

Figure 2.1: Semantic Version Example

In addition to major, minor and micro updates, it is possible to include a *pre-release tag* at the end of the version, separated by a hyphen. These tags are considered to be 'sub-micro' and are recommended for use in internal development. However, in practice they are occasionally included in published releases, where tags such as *-alpha*, *-beta*, *-release*, *-stable*, or the published timestamp

of the release are common. There is little agreement across package managers or even across projects with how these tags can be compared with each other, forming an ad hoc system chosen by each development group. These sometimes need to be ordered internally when there are multiple tags within a single micro version. In this thesis, where ordering tags has been necessary, the published timestamp has been used[2] - all other methods based on the alphabetisation and other lexicographic information suffered from accuracy issues in such a varied dataset.

It is possible that there are additional numbers after the micro number, but before the tag. For example, in the Rubygems ecosystem, it is quite common to have four numbers - the fourth being smaller bug fixes. These are accounted for in our thesis in terms of ordering, but no semantic information is drawn from these extra numbers, as the finest grain information we harvest are micro updates. Instead they are considered to be multiple versions of the same micro number in the same way tags are.

Update Type	**Old Version**	**New Version**
Major	4.7.3	5.0.0
Minor	4.7.3	4.8.0
Micro	4.7.3	4.7.4

Table 2.1: Incrementing Versions

As shown in Table 2.1, incrementing versions is done in the following way: whenever a number is incremented, all numbers to the right of it should be reset to zero. For example, if the minor version is increased, the micro version will be reset to 0, but the major version will remain unchanged. Some tools, such as npm's *semver* tool[3], will automatically update the version number for a developer, but the developer must still decide if it is a major, minor or micro change, which requires knowledge of the changes made and how they affect the syntactic and behavioural contracts that the project has implicitly or explicitly created.

[2]Due to published timestamps sometimes being erroneous (see Section 7.8), a heuristic was added for Chapter 7 where any published versions with a timestamp difference of less than 24 hours would be ordered alphabetically.

[3]`https://www.npmjs.com/package/semver` [Accessed 18 March 2020]

Updating versions in this manner allows ranges of similar releases to form. When working with version `4.7.0`, semver allows you to assume that `4.7.2`, `4.7.4`, and even `4.7.102` will all be similar to the version currently in use. The only difference is that they will have bug fixes or security fixes. This predictability allows automated package managers to make decisions about which versions would be suitable to use, simply based off a declaration, such as one that allows future micro versions within the same major and minor version to be acceptable (Section 5.2.3's *micro* range).

Versions of projects that are in pre-development stages have additional rules attached to them, according to semver principles. The most common of these is that the major version is set to 0, and then the minor is considered a breaking change (like a major update usually would be), and micro is considered a feature improvement (like minor updates usually are). The public APIs of `0.x.x` versions are considered inherently unstable (see Section 3.1), so declaring version ranges on pre-`1.0.0` versions entails additional risks. Note that some well-known projects (such as pandas[4], a widely used statistical package in Python) may continue in the 0 major range for quite some time, and be used by many dependent projects before changing to version 1.0.0.

Even though each ecosystem has their own semver conventions, semantic versions are ubiquitous across package managers, which are discussed in the following section.

2.2 Package Managers and Dependency Resolution

Modern software systems rely on package managers to automatically build and package their source code on demand. Agile methodologies require these builds to happen regularly, so the builds must be fast, efficient and as automated as possible. Some package managers allow dependencies to be downloaded from online repositories at need. Most that have this ability also are able to note if there are newer versions available, and take a declaration that defines a range of possible versions and choose from those. We focus on the package managers

[4] `https://pandas.pydata.org/`. They have only recently moved to using semver `https://pandas.pydata.org/pandas-docs/version/1.0.0/whatsnew/v1.0.0.html` [Accessed 22 Mar 2020], after years of using their own ad hoc versioning system.

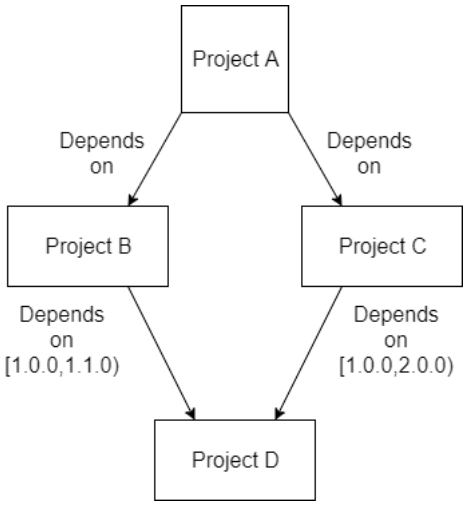

Figure 2.2: Diamond Dependency with Issues

able to source dependencies from online repositories in this work.

One of the important jobs of a package manager is to decide which versions of a dependency to use. Unlike in the case of a fixed declaration, where only one version can be chosen, flexible declarations allow for a range of versions to be chosen, so it is up to the package manager to choose which one will best suit the project's needs - this process is called *dependency resolution*.

Generally, when there are a number of versions from which a version of the dependency is chosen, the package manager will choose the latest (stable release) of these. Therefore, in this work, where a range of versions are possible, it is assumed that the latest will be used. This assumption is particularly important in Chapter 7, which deals with quantifying technical lag - the versions and time that the chosen dependency version is behind the latest version. This practice can vary by package manager however.

The most common instance of when a version chosen is not be the latest version is when it is a transitive dependency - a dependency declared by a dependency that the project relies on (this dependency could be any number of steps away from the downstream project as each dependency must download its own dependencies, who download their own, etc.). When transitive dependencies occur, it is possible to get a situation such as Figure 2.2 where a project indirectly relies on a dependency through two other dependencies. In this ex-

ample, Project A relies on both B and C, who in turn rely on D. The problem here is that Project B relies on D within the `1.0.0` *micro* range, while Project C relies on D within the `1.0.0` *minor* range.

$$D_B = \{version \mid 1.0.0 \leq version < 1.1.0\}$$
$$D_C = \{version \mid 1.0.0 \leq version < 2.0.0\}$$

Suppose we have the above dependency declarations for B and C, along with the following ordered versions of D, there are two *main*[5] options that the package manager might choose:

$$D = \{\ldots, 1.0.0, 1.0.1, 1.0.2, 1.0.3, 1.0.4, 1.0.5, 1.1.0, \ldots, 1.6.2, 2.0.0, \ldots\}$$

- Version `1.6.2` gets chosen. It satisfies C's declaration, but not B's. This is the most recent version that can be chosen[6], but it lies outside of B's declaration - a potential risk.
- Version `1.0.5`, the last version within the `1.0.0` micro range, gets chosen. This fulfils both B and C's declarations, but results in an older version of D being used than C would allow.

While it seems obvious that the second option is the better choice (after all, it satisfies both declarations), it may not be chosen. Dependency resolution is an NP-complete problem [Abate et al., 2012] and, even with optimisations, a complete search can be impractical in reality [Jenson et al., 2010]. Linux package managers (Debian and RPM) have developed complete dependency resolution solvers based off SAT-solver algorithms [Vouillon and Cosmo, 2013]. While these are potentially computationally expensive, for the majority of cases, graph preprocessing (ordering, flattening, etc.) allows this process to be completed quickly, albeit at the cost of increasing complexity and a less transparent dependency resolution process (as greedy strategies might still be needed in the cases that these optimisations do not simplify the problem sufficiently). In fact,

[5]There is nothing to stop a package manager choosing some other option, such as `1.0.0` for example. The version chosen is dictated by the dependency resolution policy of each package manager.

[6]Note that 2.0.0 and beyond exist, but they fit neither D_B nor D_C so would not be chosen.

currently, most package managers employ a greedy strategy, such as Maven's nearest-wins[7] strategy, which uses a breadth first search where the first version found is the version downloaded (even if deeper transitive dependencies needed a different version). When using this strategy, the package manager 'hopes for the best', knowing that the version satisfies the dependency closest to the root project, and hoping that it works for any subsequent dependencies as well. If the version chosen does not have the method or class required, this can cause run-time errors to be thrown, such as Java's `LinkingError` subclasses. Another strategy is its *fail-fast* equivalent, a *dependency conflict*. With this strategy, if the version chosen cannot satisfy later dependencies, the build fails and the developer must manually choose a version of the dependency in question[8].

In addition to the above two options, there is a third strategy to dependency resolution - download both versions. This is the strategy chosen by npm[9], which would allow B to download its own version of D if C had already downloaded D at a version not in D_B. This duplication generally works[10] but at the expense of very large distributions where dependencies are regularly duplicated.

Note that npm's nested dependency strategy is quite unusual across package managers. The general strategy of using global classloaders naturally limits programmes to only one version of any given class (or file) at a time. There are some frameworks that get around this issue, such as Java's OSGi[11] model, which are mostly used in large-scale projects as they add a significant amount of complexity to a project.

Once dependency resolution is considered for transitive dependencies, it is not immediately clear which version will be chosen - this comes down to how each package manager implements its own strategies to conflicts and the path the dependency resolver takes through the dependency directed acyclic graph (DAG). That is why, while it is assumed that the newest version will be chosen if it matches the declaration, this is not necessarily true. Since this work

[7] `https://maven.apache.org/plugins/maven-dependency-plugin/examples/resolving-conflicts-using-the-dependency-tree.html/` [Accessed 28 Jan 2020]

[8] This situation is commonly referred to as 'DLL Hell' after the Windows dynamic link library folder - each DLL file in this folder contains a project, and only one DLL file for each project may exist in the DLL folder, causing the same version conflict issue.

[9] `https://lexi-lambda.github.io/blog/2016/08/24/understanding-the-npm-dependency-model/`

[10] This strategy becomes more difficult when custom objects are passed across package boundaries, but this practice is not common in JavaScript.

[11] `https://www.osgi.org/`

deals primarily with direct dependencies (ignoring transitive dependencies), this assumption is considered reasonable.

As a note, NuGet in fact resolves to the *oldest* dependency that will satisfy its declarations[12]. This behaviour is worthy of note as developer behaviour in NuGet differs from many other package managers in Chapters 5 and 7 - this unusual resolution strategy may contribute to the behaviour differences.

Jezek and Dietrich discuss the problems that can be caused by automated dependency resolution in greater detail [Jezek and Dietrich, 2014]. In addition to an incorrect version of a dependency being chosen, it is also possible to have namespace collisions, where two classes of the same name from different packages are required (classloaders will generally only load one of the two). Redundancy can also occur, where classes are downloaded but are never used in runtime (due to only a fraction of the dependency's functionality being needed). Redundancy will not break applications, but they do make distributions unnecessarily large, and is a particular issue in npm builds as described above.

[12] `https://docs.microsoft.com/en-us/nuget/concepts/dependency-resolution/` [Accessed 6 Feb 2020]

Chapter 3

Literature Review

Dependency management issues have become more prominent with time, especially with the increased use of package managers. The main reason for this increase is the growing size of modern ecosystems, and the increasing speed of change in packages.

Projects are relying more on external packages than in the past. Gonzalez-Barahona et al. studied the growth of the Debian Linux distribution, finding that the average size of packages remained stable, while the overall size of the distribution has been doubling every 2 years and the number of dependencies increased exponentially [Gonzalez-Barahona et al., 2009][1]. A recent whitepaper survey of 400 developers reported that there were an average of 73 dependencies used per project by these developers, indicating that dependency management is an extremely important aspect of software development and maintenance [Vanson_Bourne, 2018].

There are some challenges with managing dependencies. Using fixed declarations is directly correlated with dependencies being out of date (shown in Chapter 7), which leads to issues of security vulnerabilities (Section 3.5.3) and difficulties to update later to take advantage of new features (such as the case study discussed in Section 3.3.2). Allowing for automatic updating of dependencies solves most issues of dependencies going out of date[2], but adds an element

[1] This is consistent with our findings in Section 5.4 on how the number of dependencies increased over the lifetime of a project.

[2] If semver-compliant ranges are used, there is still a possibility that there will be some issues with outdated dependencies.

of risk, as dependencies are updated without testing that they have not broken compatibility. For this reason, a key prerequisite of successfully automating dependency updates is ensuring that automatic updates only happen with new versions of the dependency are backwards compatible. Sections 3.1 - 3.3 discuss various aspects of backwards compatibility and measures developers can take to minimise the number of backwards incompatibilities, along with limiting the negative impact of those updates when they do occur. With a solid grounding in issues with version compatibility, we are ready to discuss topics specifically involved in dependency management. Section 3.4 discusses the topology of dependency graphs and how transitive dependencies can affect dependency management and automatic updating. Section 3.5 considers the current research into how developers are actually managing dependencies - these works form the basis of our research in the following chapters. Finally, Section 3.6 considers which parts of dependency management can currently be automated, or are on the cusp of automation.

3.1 API Stability

In order for two projects to communicate with each other, there needs to be connection points between the two. All projects that intend for others to use them in some way should define a set of classes and methods that can be used by others. These are marked as `public` in languages with built-in visibility modifiers - such as Java which allow encapsulation principles to be enforced at compile-time - or are implicitly public by means of naming conventions and documentation in languages such as Python.

Where a class, method or field is not public, it is intended for use inside the package only, termed here `private`. While the definition of `private` classes, methods and fields differs by language, here we define it in terms of whether it is intended for use by other projects, or only for intra-package use. This distinction between `public` and `private` is the basis for managing backwards compatibility between projects that share code.

An Application Programming Interface (API) is the collection of `public` classes, methods and fields within a package. In order to maintain backwards compatibility, there are limitations on what changes can be made to an API. *API stability* describes how well a project adheres to syntactic limitations imposed by

backwards compatibility requirements. API stability is one aspect of backwards compatibility - an unstable API means introduced breaking changes that will force downstream developers to refactor their code to maintain compatibility.

Over time, evolving conditions and demands on software provide impetus for change to cope with additional complexity and reduce technical debt. In the process of maintaining a mature project, developers usually face the choice of keeping an API stable, or introducing changes needed to bring the project up to date and solve underlying issues that collect over time.

Position	**Change Type**
1	Method removed
2	Class removed
3	Field removed
4	Parameter type changed
5	Method return type changed
6	Interface removed
7	Number of arguments changed
8	Method added to interface
9	Field type changed

Table 3.1: Breaking Changes to an API

Table 3.1 is derived from a study by Raemaekers et al. [2014], ordering the most common types of backwards breaking changes to an API. It is difficult to keep an API stable - doing so often can involve compromising decisions such as keeping methods or classes that have major flaws (bugs, designs inconsistent with the rest of the API, or inefficiencies) in order to maintain stability (see Section 3.3.3).

In addition to API instability, there are other factors that can cause breaking changes for dependent projects:

- The software licence could change, making a dependent project no longer able to use the dependency without changing part of its business practice, e.g. a dependency of a closed-source project moving to a GPL licence, which would require the dependent project to become open-source.
- The semantics of the dependency could change. The behaviour of a

method could change without any changes to its API - for example, if a method throws an exception for negative inputs when it accepted them before, this changes the semantics and may break the dependent project. It could also be more insidious, for example if the results of a call differ between versions due to a new implementation, this would instill a form of non-determinism into the project, constituting a type of breaking change.

- In compiled code, binary compatibility is an additional consideration, as API changes that do not break source code compatibility can still break binary compatibility, for example where 'hot updates' are done - programmes that are recompiled in parts rather than a whole. Section 3.2 discusses this in more detail.

Semantic Compatibility

In order to ensure semantic compatibility (also known as *behavioural compatibility*), Beugnard et al. [1999] suggested basing components on contracts. In this work, they cover four types of contracts, syntactic, behavioural, synchronisation and quantitative. A syntactic contract is the idea of API stability discussed above. Behavioural contracts encompass *design by contract* ideas using pre- and post-conditions, guaranteeing certain behaviours of a piece of code (which languages like Eiffel[3] are based on) - see also the book *The Pragmatic Programmer* [Andrew and David, 2000, p. 109]. Synchronisation contracts ensure that the component will deliver the correct result under multithreaded conditions, and quantitative contracts guarantee that a component will do all of the above efficiently, taking into consideration quality-of-service indicators. All four types of contracts must be maintained in order to keep backwards compatibility.

Dietrich et al. investigated the use of contracts in code across an evolutionary dataset of Java programmes harvested from the Maven repository [Dietrich et al., 2017]. They found no evidence of widespread usage of pre-condition checks, but that projects who use them continue to use them and expand on them as their project grows, indicating that projects which use them continue to promote them through their codebase. They also found a large number of backwards incompatible contract changes, such as strengthening pre-conditions (breaking Liskov's Substitution Principle), indicating that behavioural violations are a major concern in terms of backwards incompatibil-

[3] `https://www.eiffel.org/`

ities.

API Stability and Project Success

The API stability of dependencies can have a significant impact on the quality of the dependent project's code. Linares-Vásquez et al. conducted a study of Android applications, looking for differences between the API stability of dependencies in successful apps versus unsuccessful apps[4] [Linares-Vásquez et al., 2013]. They found that high-rated apps were much less likely to have dependencies with unstable APIs with fewer bugs, with only 20-33% as many API changes and faults as low-rated apps. Many of the comments for low-rated apps were to do with bugs and other faults, which the authors attributed in part due to faults inherited through dependencies and API changes.

API Stability for Pre-`1.0.0` Versions

As mentioned in Section 2.1, semantic versioning principles recommend that projects with versions below `1.0.0` be considered to have unstable APIs. Semver recommends that minor updates be treated as backwards breaking, while only micro updates should still keep backwards compatibility. However, this recommendation has not been implemented consistently across projects or ecosystems. In a bid to improve clarity about which version updates will introduce backwards incompatibility, npm has taken the step of recommending that developers start their versioning at `1.0.0`[5]. It remains to be seen if the pre-`1.0.0` semver-compliance improves, or if the convention becomes avoided in other package managers as well.

3.2 Binary Compatibility

A less obvious backwards compatibility concern in compiled languages is *binary compatibility*. Binary compatibility allows for linked modules to be recompiled separately before recombining (a process called *relinking*) and expecting them to continue to work with the dependent project [Drossopoulou et al., 1998]. Java introduced this as a formal feature of its language specification, allowing for situations such as making modifications to modules where recompilation is not practical (such as where the source code is not available) or the size of the project

[4] They rated success based on user ratings - highly successful apps were the highest quintile of ratings, and unsuccessful aps were the lowest quintile of ratings.

[5] `https://docs.npmjs.com/about-semantic-versioning` [Accessed 4 Feb 2020]

discourages recompiling all units simultaneously. Compiled files are written in either an intermediate language (such as bytecode for JVM languages or IL for .NET), or in machine code (e.g. C) - in either case this compiled code can vary in subtle ways from source code which introduces its own compatibility issues if the phase of compiling projects together is skipped.

At first glance, it may be assumed that binary compatibility would be implied if the projects are source compatible (by having a stable API and semantics). However, there are instances where projects can be source compatible, but not binary compatible, and vice versa. Traditionally, languages have underspecified the requirements for binary compatibility, so formal definitions have been provided and increasingly implemented in modern enterprise language design [Drossopoulou et al., 1998, Drossopoulou et al., 1999]. The listing below derived from Dietrich et al. [2014] gives examples of differences between source compatibility and binary compatibility (using Java-based examples):

1. Source compatible, Binary incompatible changes:

    - The return type of a method gives a subtype of the previous method (e.g. a `List` is promised instead of a `Collection`). This is considered as safe subtyping by the Liskov's Substitution Principle, and the compiler will be able to reason this and determine the source code to be compatible, despite their binary incompatibility.
    - The parameter type of a method requests a supertype of the previous method (e.g. a `Collection` is now requested instead of a `List`).
    - Changing between primitives and their corresponding object type in languages that have both primitive and object types, e.g. `int` and `Integer`.
    - When constants are changed, inlined constants are not updated, leading to incorrect values being used.

2. Source incompatible, Binary compatible changes:

    - When a generic type is changed, it does not show at a binary level due to *erasure*, infamously used in Java. Therefore, a method return that changes from `List<String>` to `List<Integer>` will be binary

compatible, even though the source code is not, as generic type information is removed at compile-time, and the bytecode will only show that it is a `List`.

In general, there are more examples of code being source compatible but binary incompatible than the reverse. Any of the above examples (and more) could lead to run-time exceptions, often with puzzling stack traces for the developer to solve. Dietrich et al. used static analysis to study how common binary compatibility issues are when projects are relinked rather than recompiled [Dietrich et al., 2014]. They found that breaking changes are common (they estimated that up to 75% of versions introduce binary incompatible changes), but that this rarely affects dependent projects in reality, as only a subset of the dependency's code gets used.

Dietrich et al. went on to gather survey responses from 414 developers about what they know of binary compatibility [Dietrich et al., 2016]. The survey was structured as a quiz, testing the developers' understanding of what changes would cause binary incompatibility. They found that, while many developers understood the rules for source compatibility, very few understood the rules for binary compatibility, even amongst developers who considered themselves *experienced* or *expert*. The fact that so few of the surveyed developers understood binary compatibility makes trusting semantic version numbers alone to convey compatibility difficult.

The study also counted how regularly relinking exceptions occurred, based on GitHub and StackOverflow hits. They found that a relinking exception, `NoClassDefFoundError`, had almost as many questions as the extremely common `NullPointerException`. While it is possible that this is a more puzzling exception and thus gets asked about more regularly than the straightforward reasons why `NullPointerException`s occur, it still indicates that binary incompatibility impacts on software quality and developer productivity significantly.

Jezek and Dietrich suggested a modified Java compiler that avoids some binary incompatibility issues, but they note that this involves invasive changes to how the standard Java compiler works and, while technically feasible, may not be chosen due to business concerns [Jezek and Dietrich, 2016]. This work points to how compiler design may be able to limit binary incompatibility in future language and tool design.

Another approach to improving binary compatibility - automating refactor-

ing to maintain binary compatibility - has been proposed by Şavga and Rudolf [2007]. In their example, they formalised the process required for projects relying on a .NET framework to be automatically refactored to maintain binary compatibility, using automatically generated adapter layers at the border between modules.

Binary compatibility is an often overlooked but major issue within the topic of API stability. Changes which are source compatible, and therefore considered compatible by most developers and researchers, can still cause problems for downstream developers attempting to update dependencies. Increased awareness of these issues would benefit both upstream developers, where binary incompatible changes could be postponed until major updates (and therefore limit the number of times downstream developers have to recompile their projects), and downstream developers, where run-time exceptions related to relinking can be quickly understood and solved with recompilation. Tool usage can help developers to understand binary incompatibility issues as well, such as Eclipse's *API Tools* discussed in Section 3.3.3.

3.3 Managing Breaking Updates

Breaking updates of APIs are an ongoing issue for developers who rely on dependencies. Xavier et al. estimated that, across 1,000 Java projects studied, 28% of API changes were breaking, with the majority of API changes not breaking backwards compatibility - operations such as additional implemented API methods[6], increasing visibility or removing deprecated features [Xavier et al., 2017].[7] Interestingly, mature APIs were more likely to break - likely due to accrued technical debt or necessary changes that develop over time. The impact of any one breaking change on clients is relatively small, however, with only 2.5% of clients being affected by the median breaking change.

[6] Adding implemented methods to concrete classes does not break API compatibility, unlike adding methods to an interface, which forces a client who implements this interface to create a new method implementation to maintain compatibility.

[7] Xavier et al. considered removing deprecated features to not break compatibility as users have been given advance notice [Xavier et al., 2017]. However, removing API methods, deprecated or not, does in fact break downstream projects if they have not adjusted their projects.

3.3.1 Using Semantic Versioning

Semantic versioning has given developers a framework to think about updates that are done in projects and help with isolating API instability to major versions, where client developers expect such changes to be made. However, semver is a social contract and comes down to developers following these principles. It turns out that this is quite a difficult task to do manually, and there are currently few tools that can help automate this process (discussed in Section 3.6).

Updates often exhibit backwards incompatibility, even in cases where they should be compatible. Raemaekers et al. showed that many semver versions are incorrectly updated - 23% of micro updates, and 36% of minor and major updates broke backwards compatibility [Raemaekers et al., 2014]. This is problematic for flexible version declarations, as they are based on the assumption that micro and minor updates do not have any backwards incompatibility. While not all code within a dependency will used by every dependent project, the projects that exhibited backwards incompatibility averaged 30 compatibility issues even in micro and minor updates, so there is a good chance that there will be problems in reality. This study by Raemaekers et al. focused on source compatibility - the large amount of binary incompatibility (Section 3.2) is an additional challenge that developers face.

The same study found that breaking changes in minor and micro releases decreased over time from 28.4% in 2006 to 23.7% in 2011, indicating that developers are becoming more aware about semver ideas. Another point is that semantic versioning has become much more entrenched in software engineering since this study took place (see Section 3.5.1), so it is possible that if this study were repeated again now, backwards compatibility in minor and micro updates may have improved.

3.3.2 Client Project Risk Management Strategies

One of Lehman's eight *Laws of Software Evolution* is that programmes must continuously change, or progressively become less relevant [Lehman, 1980]. As change happens at a high cost to projects, there are several strategies prevalent in industry to help mitigate the effects of API instability and cope with this external change impetus:

1. Semantic versioning provides a framework for developers to signal that

they are creating a breaking change, in the form of major updates, as discussed in Section 2.1, however this is not always foolproof, but it provides information of the upstream developer's intent.

2. Design patterns that minimise the places where a project connects to an external API, e.g. Adapters (*Design Patterns*, [Gamma et al., 1993]), a practice that is recommended by many, including Martin [2009].

3. Implement unit tests based on the external API. This way, when the API or its behaviour changes in some way, there is instant and useful information available to the developer, suggested both in *Clean Code* [Martin, 2009] and in a case study by Raemaekers et al., [2012].

4. Consider in-project solutions when the resulting code is not overly complex, rather than adding a dependency. When a dependency must be used, be conservative about which dependency to rely on, taking into account its historical API stability [Bogart et al., 2015, Bogart et al., 2016].

5. Avoid early adoption, unless aware of the risks of greater instability - Espinha et al. noted that early versions of projects have particularly unstable APIs [Espinha et al., 2014].

As noted above, Lehmann's laws of software evolution involve continuing change and growth [Lehman, 1980]. Mens et al. searched for evidence of Lehmann's laws in the Eclipse ecosystem [Mens et al., 2008], finding that Eclipse bundles indeed do continue to change and grow over time. Keeping these laws in mind, adopting an architecture that insulates projects from external change is a sensible choice to protect projects.

Delaying updating dependencies for a long period of time can cause major issues, not only in terms of opportunity costs (Section 3.5.2), but also it can complicate the updating process when it finally occurs. Raemaekers et al. provide a case study about a project that avoided updating its dependencies for seven years, deferring them due to breaking changes that had been made in the dependencies [Raemaekers et al., 2012]. By the time this project was forced to update to a new version for new required functionality, the dependencies that were changed required their entire framework to be updated. They also needed to update their Java Server Pages (JSPs) because its syntax had changed during

a minor update. The process of updating this dependency took a week of developers' time due to the cascading changes, and it was commented on that this update might have been impossible if there had been a lack of regression tests to be able to understand the scope of the changes. It showed that accumulating technical lag (the measure of how outdated dependencies are) can lead to large amounts of refactoring to be done in one go if delayed - very similar to what happens when technical debt is accumulated. It also showed an instance of a minor update introducing breaking changes (Struts), and how updated transitive dependencies can increase the amount of work required to update.

Bogart et al. interviewed 28 developers from three ecosystems, Eclipse, CRAN and npm [Bogart et al., 2016] to check how they managed dependency updates. A few interviewees actively monitored dependencies to check for changes, using GitHub feeds (to varying success, due to overwhelming amounts of information). Some developers actively contributed to their dependencies with features and fixes that they wanted for their own project downstream. Others tried to keep an eye on changes more broadly, often using *Twitter updates* (in npm especially). Many projects use continuous integration to detect compile-time errors - this gives a quick feedback loop for API changes in upstream dependencies, allowing developers to become aware of changes quickly[8]. Additional information can be harvested from continuous integration builds when automated tools are used, as discussed in Section 3.6.

In the absence of information about whether updates are going to include backwards breaking changes, care must be taken by developers. To supplement semantic versioning, additional metrics could help developers understand the risks involved with updating a particular dependency. Bogart et al. has proposed a measure based on the past binary and source compatibility history of a project in order to give developers more confidence in updating projects which have a history of stable APIs [Bogart et al., 2015]. Decan et al. call for similar metrics through their discussions with industry developers, which they term the *health* metrics of a dependency [Decan et al., 2018]. For both direct and transitive dependencies, this information would be useful, as an unhealthy transitive dependency can still have negative implications for a project (Section 3.5.3).

[8]This assumes problems can be picked up at compile-time. It would not spot behavioural changes, or help in non-compiled languages

3.3.3 Strategies for Signalling Pending Breaking Changes

Managing APIs carefully can have very positive effects for upstream developers. Zibran et al. completed a study of over 1,500 bug reports, finding that as many as one third of bugs were related to API issues [Zibran et al., 2011]. These generally related to breaking changes to the API, but in some cases also involved transitive dependencies - dependencies of the upstream project were causing conflicts for downstream developers, or optional dependencies were requested to give greater flexibility for managing version conflicts.

When breaking changes must be implemented, there is mixed evidence that upstream developers can help downstream developers that rely on their project to adjust to future breaking changes by using deprecation patterns.

Robbes et al. undertook a study on the effectiveness of deprecation patterns in the Smalltalk (specifically Squeak and Pharo) ecosystem [Robbes et al., 2012]. It is a small ecosystem consisting of 2,600 projects. They found that roughly 40% of deprecations caused downstream developers to react in *active projects*. For those who did react, it was generally within the first two months of the deprecation, and most followed the developer's recommendation for what to change to (when the developer did suggest an alternative).

A follow up study on deprecations was undertaken by Sawant et al. to consider the effects of deprecation on a mainstream ecosystem, Java. It included several methodological improvements over the Robbes et al. study which were possible due to Java being a statically typed language with deprecation annotations [Sawant et al., 2016]. Analysing over 2,600 client projects of 5 major APIs (Guava, Guice, Hibernate, Easymock and Spring), it was found that deprecation patterns do help downstream developers to update their code. However, their results came with some major caveats. First, the majority of the projects involved never updated the dependencies in question, so deprecations were irrelevant for many of the studied projects. Of those that were affected by the deprecations, less than a quarter updated them. In general, the ones that did update had less changes to make - one project would have had to make over 17,000 method changes to update[9]. Of those that did choose to make updates, the most common strategy was to delete the method rather than to update it to the replacement. This contrasts with the Smalltalk study, where developers

[9]Needless to say, it was a project that did not update in the study.

tended to replace based on the upstream developer's recommendations.

One interesting point that the study noted was that over 95% of the projects studied *added* calls to already deprecated methods. At the same time, the authors noted that almost all moves away from deprecated methods happened within days of the deprecation being added. If the goal is to move clients away from deprecated methods before deletion, there is a case for only keeping deprecated methods for a short time, if at all (Google Guice was noted as not using deprecation patterns at all).

Sawant et al. followed up this quantitative study by interviewing Java developers who were involved with API maintenance [Sawant et al., 2018]. They found that about half of API producers will usually remove features two or more releases after deprecation. From the study, it seems that developers are reluctant to remove deprecated features - only 25% of respondents said they always will seek to remove a deprecated feature eventually. It is likely that this lack of follow through with regard to removing deprecated features contributes to developers often not updating away from deprecated methods.

Hora et al. followed on from the previous study by Robbes et al. by looking more widely at how API changes affect projects in the Smalltalk (Pharo) ecosystem [Hora et al., 2015]. They found further evidence that deprecation strategies work in that ecosystem, but also that it takes some time for developers to respond to changes - a month on average. They also found that in all analysed cases of projects, clients were using the internal API of dependencies, causing problems (as internal APIs are not intended to stay stable)[10]. Finally they found that the reactions to API changes can be partially automated. The idea of automated refactoring to manage breaking updates is discussed further in Section 3.6.

Bogart et al. studied the weighing up of the relative costs of breaking APIs among upstream developers through a series of developer interviews, by looking at three ecosystems with very different philosophies on change - Eclipse (Java), CRAN (R), and npm (JavaScript) [Bogart et al., 2016]:

- Eclipse developers - predictably as major Java projects - focused strongly on backwards compatibility, including using sophisticated tools such as

[10]This is made possible by the lack of visibility modifiers in Smalltalk allowing a compiler to enforce encapsulation principles - a shortcoming of a number of common languages.

API Tools[11] to compare APIs in an update and spot binary and source incompatibilities. They engage in simultaneous releases across the ecosystem once a year, with minor updates in between. They were willing to accept large amounts of technical debt in order to keep backwards compatibility, and several interviewees cited duplicating classes and interfaces to allow for changes to be made while still keeping the old versions available for backwards compatibility purposes. Another cited example was a method that had been deprecated for 11 years and still was not removed. Note that we do not study the Eclipse ecosystem directly in this work, but the Maven ecosystem (which we do study) overlaps with the Eclipse ecosystem in terms of tooling, developers involved, and programming languages served.

- CRAN developers placed a large emphasis on forewarning downstream developers that changes would need to be made in their projects within the coming fortnight or month. In this way downstream and upstream changes are synchronised and breaking updates are closely managed, but at a significant effort for both sets of developers. Daily manual and automated checks (using unit and integration tests) are done to ensure consistency throughout the ecosystem between the newest releases of each project. In this way, the ecosystem as a whole remains stable, but closely controlled. In CRAN, new versions must have a higher version number than the version they replace, so parallel development is not supported in this ecosystem - delaying responses to breaking updates is not a viable option by design.

- Out of the three ecosystems, npm developers most value *moving fast*. As such, breaking changes are seen as less problematic than in the previous two ecosystems, and the developers are willing to make breaking changes as necessary to fine-tune their APIs. The developers in this ecosystem emphasised the role of semantic versioning in signalling breaking changes to downstream developers. In order to lower the burden of managing change on downstream developers, parallel development is common in npm, with a quarter of the 100 most *starred* projects on GitHub having released maintenance releases for old major versions, acknowledging that

[11] `https://www.eclipse.org/pde/pde-api-tools/`

many downstream developers have stayed on an old major version to delay adjusting to the breaking updates introduced in the new major version.

The above highlights the strategies that upstream developers may employ to manage breaking changes. Maintaining old interfaces (leading to duplication), using parallel releases (to allow downstream developers to delay updating), collaboratively planning new releases together with downstream developers (such as in CRAN and Eclipse), and communicating with downstream developers (through use of email lists, forums, GitHub feeds, or documentation of what methods to change to) are all potential strategies to mitigate the negative impact of breaking updates.

Across ecosystems, the primary reason for introducing breaking updates from an upstream developer's perspective was due to fixing technical debt, rather than bug fixes or efficiency reasons (although these were sometimes the concern). Bogart et al. suggested that this is because the developers have thought about this issue in depth - technical debt is something they are faced with on a daily basis, and changes must be thought out in detail before implementation [Bogart et al., 2016].

3.3.4 Web APIs

Up until now, this chapter has focused on statically linked APIs - dependencies that can be downloaded as a local copy and run on the same machine as the downstream project. This downloaded dependency remains constant until such a time as the downstream developer chooses to update to a new version, or the package manager automatically downloads a new version such as the declaration allows. There is a second way of linking to the API of a dependency though - using web APIs such as Google Maps. Instead of downloading the dependency, the downstream project links to the upstream project remotely, by sending HTTP requests to a remote server where the upstream project is being hosted. The difference between web APIs and locally available dependencies could be considered similar to serverless architecture versus traditional architectures as the infrastructure requirements in both web APIs and serverless architectures are abstracted away, simplifying the process but at the cost of losing a measure of control.

This represents a paradigm shift in dependency management - the depen-

dency used are almost exclusively the newest version in web APIs. Some cases involving paid APIs offer multiple versions, but this is the exception rather than the rule. For those that do not offer the older version of APIs, whenever breaking changes are implemented, the downstream developer must immediately react - the stalling tactic of remaining at an older version is not possible. Espinha et al. studied how the Google Maps, Twitter, Facebook and Netflix web APIs evolved over time [Espinha et al., 2014]. Several interesting points emerged out of that study. First, deprecation policy varied significantly from project to project. The Netflix and Twitter APIs did not have a deprecation policy, meanwhile Facebook pushed breaking changes every three months, and Google had a one year deprecation policy that in some cases stretched out to over three years - in the case of Google deprecations, many developers left it until the last possible moment to upgrade. Communication between upstream and downstream developers was discussed in detail in this study - much like in Section 3.3.3, communication was seen as playing a key role in helping developers to stay up to date with breaking changes. Requiring an API key[12] (and therefore an email or other contact information) in order to use the web API allowed upstream developers a method of communicating upcoming changes to the downstream developer - this system gives an upstream developer more information about who the downstream developers of a project are than what is generally available to upstream developers. Despite the differences with standard dependencies and the requirements to stay completely up to date with web APIs, many of the strategies are still the same - keep good lines of communication between upstream and downstream developers, unit test the external APIs, and keep the boundary between internal logic and the external API small.

3.3.5 Summary

Breaking updates happen regularly in projects, even during minor or micro updates, although these do not always break client code. For an upstream developer, communicating with downstream developers about upcoming breaking changes is helpful, such as through forums, Twitter or by deprecating features before removal. When deprecating features, the research recommends making documentation that informs how downstream developers should respond to the

[12] An API key is a token that a developer is issued upon registering for the API service which allows the upstream project to monitor and restrict usage to client projects.

change, and that this should be followed up by a complete removal within a period of a few months to avoid new calls being made to the deprecated method. For downstream developers, when including a new project as a dependency is advantageous, including unit tests for the API to check for contract changes (in terms of syntax or behaviour) is recommended, as is using design patterns, such as adapters, to limit the size of the boundary between the internal logic of a programme and external logic. Active monitoring of upstream dependencies and continuous integration can allow downstream developers to be alerted to breaking updates more quickly, so can form part of an effective dependency management strategy. Moving forward, several sources have called for the development of *health* metrics that allow developers, at a glance, to gauge a dependency's activity and history of breaking changes, which will allow developers to make informed decisions when choosing dependencies.

From here, the chapter moves on from compatibility issues to surveying the state of dependency management and what practices have been observed across the industry.

3.4 Dependency Graphs

It is normal for most projects to have dependencies. When a dependency itself has dependencies, this creates a chain of transitive dependencies, all of which can in some measure affect the end project. To visualise these connections between projects, a dependency graph is used. The dependency graph is an abstract representation of project dependencies within an ecosystem. The topology of each graph differs by ecosystem, but generally consists of deep structures (representing long chains of transitive dependencies) containing a number of dominating nodes (representing a project that many others depend on).

Dependency graphs are complex and often contain dominant nodes - projects depended on directly or transitively by a large number of other projects (as seen in Figure 3.1). Removing any of these projects which form dominant nodes would cause a significant portion of the ecosystem to fail (such as the infamous *left-pad* incident in npm, whose removal caused thousands of npm builds to

fail[13]). Kikas et al. studied the dependency graphs of three major ecosystems - JavaScript, Ruby, and Rust [Kikas et al., 2017]. They found that over two-thirds of Ruby projects and over half of JavaScript projects indirectly depend on a single dependency, and that in JavaScript, transitive dependencies have rapidly grown, with the number of transitive dependencies growing 60% in a single year. The study mused that JavaScript has more transitive dependencies than the other ecosystems (a mean of 55 transitive dependencies per project), but that this may occur because the npm dependency resolution model allows multiple versions of the same project to be included through transitive dependencies. Decan et al. noted that, in order to lower the risk of dominating packages breaking a large portion of the ecosystem, Cargo, NuGet and now npm (since the *left-pad* incident) prevent packages from being removed out of their respective ecosystems [Decan et al., 2018].

Decan et al. [2018] studied the topology of dependency graphs in seven ecosystems: Cargo, CPAN, CRAN, npm, NuGet, Packagist, and RubyGems. Like the work of Kikas et al., this study found a significant number of dominator nodes within the dependency graph, with all package managers studied having at least 40 projects that were transitively depended on by at least 5% of nodes in the graph.

Several authors investigated the tooling aspect of precisely extracting and representing dependencies from a dependency graph. This includes the work of Lungu et al. [2010] for the Smalltalk ecosystem and German et al. [2007] who worked on the Debian ecosystem. These works allow models to be built for transitive dependencies, and to resolve version conflicts more elegantly.

Claes et al. [2018] conducted a study on the R and Debian ecosystems. Both ecosystems seek to provide snapshots of packages that are compatible with each other, working towards a stable graph. Unfortunately, in some cases, strong conflicts exist that stop two components from being able to be used together under any circumstance (meaning that only one of the two components can be installed), an issue this study calls a problem of co-installability. They looked

[13] `https://www.davidhaney.io/npm-left-pad-have-we-forgotten-how-to-program/` provides an interesting account of the incident - the *left-pad* project is 11 lines of straightforward code used for string manipulation. This heavy reliance on dependencies for basic functionality is completely at odds with some other ecosystems, such as R, where developers will actively copy code between projects to avoid dependencies [Bogart et al., 2015]. It also highlights why the suggestion to limit the number of dependencies in Section 3.3.2 is relevant.

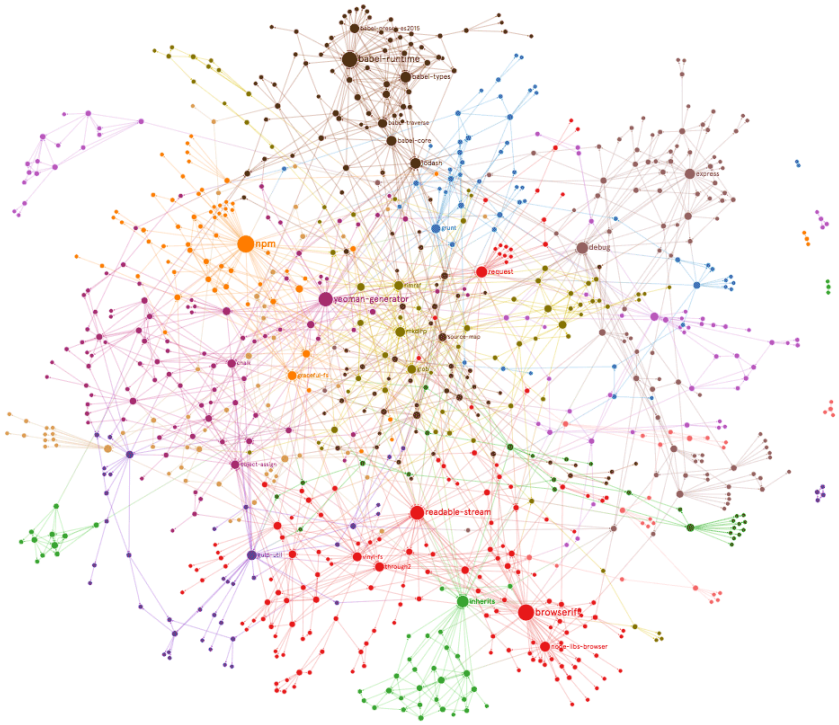

Figure 3.1: An Example Dependency Graph from npm
`https://medium.com/graph-commons/analyzing-the-npm-dependency-network-e2cf318c1d0d` [Accessed 12 Apr 2020]

further into CRAN (R's main repository) where daily continuous integration checks are run by the repository to check for interoperability of the most recent versions of projects in the repository (in CRAN, only the latest version is available for download). This study found that over half the errors fixed by the package maintainers were errors introduced via dependencies, an even higher amount than the 33% reported by Zibran et al. [Zibran et al., 2011].

The Claes et al. and Zibran et al. studies show that developers have a significant amount of work to keep their projects up-to-date. This may shed further light on why developers are often so keen to avoid dependencies [Bogart et al., 2015, Decan et al., 2018].

While our work focuses on direct dependencies, it is important to understand this in the context of that these dependencies usually will have their own dependencies, leading to additional transitive dependencies being downloaded and consumed by the project. Understanding the topology of dependency graphs gives a greater appreciation of why limiting dependency usage can have a large impact on the overall complexity of dependency management. It also helps understand how version conflicts occur in the face of co-installability issues, along with that version constraints may have to be ignored by the package manager in order to create a successful build, an issue discussed in Section 2.2.

3.5 Dependency Management Trends

This section discusses the trends in dependency management from the perspective of downstream developers. We analyse the literature around declaration patterns (which can also be considered a version constraint in that they can limit the potential versions chosen). Because there are often constraints on the versions chosen, dependencies can get out of date, a phenomenon called technical lag. We consider how prevalent technical lag is, outline its cost in terms of security vulnerabilities, and if there are specific ways in which developers act to reduce this lag in the form of updates.

This section directly relates to our research in Chapters 4 - 7. Where applicable, there have been connections made between what has been studied previously along with where our work lies within the field.

3.5.1 Version Constraint Trends

Prior to our work in Chapters 4 and 5, there had been little related work into the types of version constraints that are used in different package managers. Since our publication [Dietrich et al., 2019], there has been a second group of work that has considered dependency declarations and how they change over time.

Decan and Mens studied four package managers, also using the *libraries.io* dataset - Cargo, npm, Packagist and Rubygems - in order to understand semantic versioning declaration choices [Decan and Mens, 2019]. They analysed five years of data (2013-2018) and considered how declaration styles changed over time. They grouped the declarations in one of three ways:

- Restrictive - Considered *fixed* in this study, along with ranges that form a proper subset of a micro or minor range.
- Semver Compliant - Considered *micro* or *minor* ranges in this study (note that custom ranges that are semver compliant would be considered compliant, unlike in our work).
- Permissive - Considered *open ranges* by this study, or ranges that are more permissive than minor ranges.

One result they found was that pre-`1.0.0` releases are generally *permissive*, indicating that developers recognise and accept the risk of these pre-release dependencies, and also that Cargo had a very high proportion of these versions compared with other studied package managers (over 70% vs 20%)[14]. They also checked for how semver compliance was trending over time, finding a significant increase in semver-compliant declarations across all four package managers, with between 30-40% of all declarations changing to being semver compliant over the studied time frame. In Cargo, almost all declarations were initially permissive, and then within a space of months, went to being almost all compliant[15]. Finally, Decan and Mens found anecdotal evidence of developers varying their

[14]Interestingly, the proportion of pre-`1.0.0` versions in npm has decreased dramatically over the past years, which may be due in part to npm recommending projects to start versions only at `1.0.0` as discussed in Section 2.1.

[15]This was due to the package manager no longer supporting the Cargo *any* classification, '*', with a strong recommendation to move to semver-compliant syntax - a good example of how a package manager shapes an ecosystem's behaviour with its policies.

declaration strategy based on a dependency's reputation of maintaining semver compliance, using *lodash* and *underscore*.

3.5.2 Technical Lag

Technical lag is a term used to measure how far out of date dependencies are [Gonzalez-Barahona et al., 2017]. It can be considered an analogue of technical debt for its ability to cause decreased quality and increasing complexity to update dependencies within a project. There had been a handful of small-scale studies quantifying lag in specific ecosystems, such as a subset of Maven projects, or across npm. Zerouali et al. studied technical lag in npm (including for transitive dependencies) [Zerouali et al., 2018].

Raemaekers et al. studied the version and time lag on 2,984 dependency updates from projects on the Maven repository, finding that most projects did not have large amounts of technical lag - the median project had no major, minor or micro lag, but that minor and micro lags were present above the 75^{th} percentile of projects, i.e., at least a quarter of projects contained minor lag and a quarter contained micro lag [Raemaekers et al., 2014].

Zerouali et al. also used *libraries.io* to quantify technical lag, focusing on npm dependencies [Zerouali et al., 2018]. They found that the median technical lag was one minor and three micro versions, and that much of the technical lag was due to inheriting lag transitively. Zerouali et al., in further studies, showed that javascript-based docker images generally included some lag, particularly in micro versions [Zerouali et al., 2019a, Zerouali et al., 2019b].

While there have been a few studies on the quantity of technical lag which we build on, our work represents the first large-scale evaluation of technical lag.

3.5.3 Security Vulnerabilities of Technical Lag

There have been several studies aimed at understanding the correlation between technical lag and security vulnerabilities. While technical lag has multiple costs, including a lack of bug fixes, feature improvements and efficiency improvements, security vulnerabilities are among the most urgent reasons to update, and are the easiest to quantify, which is perhaps why this particular aspect of technical lag costs has been investigated the most.

Developers are often not aware that their dependencies are outdated and

contain security vulnerabilities. Kula et al. analysed over 2,700 library dependencies from GitHub projects, finding that 81.5% of the studied projects had outdated dependencies, and that 69% of interviewees were unaware of vulnerable dependencies in their projects [Kula et al., 2018]. So, while developers may be consciously choosing to delay updating dependencies at times, there are other times where they are simply unaware that updates are available, a finding that agrees with the discussion in Section 3.3.2. Section 3.6 presents some tools that help bridge this information gap, so developers can make conscious and timely decisions about whether to update or delay updating dependencies.

Technical lag has a major effect on project vulnerabilities. Lauinger et al., using 72 libraries harvested from GitHub, pinpointed technical lag as a major source of security vulnerability, and that libraries included transitively are more likely to be vulnerable. They found that the time lag in projects depending on these libraries can often be measured in years, which increases their security vulnerabilities [Lauinger et al., 2018]. This increased vulnerability has a major effect - Cox et al. found in a study of 74 projects that projects using outdated dependencies were four times more likely to have security issues than those with up-to-date dependencies [Cox et al., 2015].

One piece of good news about security vulnerabilities is that not all security vulnerabilities in dependencies will cause problems in client code. This mirrors Section 3.3, where only a portion of breaking changes affecting downstream projects. Zapata et al. found in an npm based study that 73% of projects did not use the vulnerable functions inherited from their dependencies [Zapata et al., 2018].

Keeping dependencies up to date may result in most vulnerabilities being avoided. Pashchenko et al. [2018] studied the impact of vulnerabilities in open-source projects, along with selected commercial SAP products to cross-reference. They noted that many, but not all, vulnerabilities can be fixed by simply updating dependencies. Derr et al. studied library dependencies in Android apps, and found a large number of outdated versions of libraries being used that could be easily upgraded - in many cases the outdated versions had known vulnerabilities [Derr et al., 2017], echoing the results of Pashchenko et al.

3.5.4 Update Strategies

Several studies have sought to find out at what points developers update lagging dependencies, and if there are specific triggers that cause developers to update.

Decan and Mens found that declarations are updated every 3-7 versions (this result is higher than our and other studies), with more restrictive declarations and pre-`1.0.0` versions being updated more regularly than older, more flexible declarations [Decan and Mens, 2019].

Cox et al. showed that projects are most likely to make micro updates [Cox et al., 2015]. In the instance that downstream developers are not updating their declarations regularly, it is likely that they will have a relatively large amount of micro lag compared to minor or major lag.

Kula et al. investigated latency when adopting library releases and found that developers were more likely to adopt updated versions later into a project's lifecycle [Kula et al., 2015], which according to Espinha et al., generally have more stable APIs [Espinha et al., 2014]. They also found that the trend is for developers to automatically go to the newest version when introducing new libraries.

Raemaekers et al. studied the version and time lag on 2,984 dependency updates from projects on the Maven repository, finding that major changes to dependency versions were usually included in major updates of a project [Raemaekers et al., 2014]. They also checked if the presence of breaking changes influenced lag, finding small but statistically significant evidence of this behaviour, where developers will avoid updating due to the existence of breaking changes.

Roseiro Côgo et al. looked at backwards changes to version declarations in the npm package manager, explaining that downgrades were caused by either moving away from buggy versions to a stable release, or as a preventive measure, going from a version range to a fixed version [Roseiro Côgo et al., 2019]. They found that most downgrades happened to only one dependency at a time, indicating that this is usually done as a fix rather than a policy change, and that the median downgrades were significant, skipping back a major, minor and 3 micro versions.

With fixed declarations, there is mixed evidence showing that developers are being conscious with their updating strategy. Bavota et al. [2015] studied 147 Apache projects which collectively had 1,964 releases, finding that projects do

not automatically update dependencies, but make a conscious decision whether the update works for them, and only upgraded about 60% of the time. When the dependency has bug fixes, it is more likely to be adopted, but they also noted that in most cases, changes to the dependency did not require large code changes to the downstream project.

Salza et al. studied updating patterns in mobile apps [Salza et al., 2018]. They found that only 2% of commits updated dependencies, and up to 70% of dependencies were out of date. Interestingly, they found a strong correlation between developers regularly updating their dependencies and their mobile app being highly-rated. While it is not possible to say if there is a causative relationship between the two, it does agree with other studies that have shown updating dependencies to be an effective strategy for reducing security vulnerabilities and bugs (a correlation discussed in [Linares-Vásquez et al., 2013]).

In general, there is a leaning towards developers updating if the update is simple, such as updating to a new micro version, and putting off updates that introduce breaking changes. It is from this basis of (mostly small-scale) studies that we investigate further the types of updates that developers perform across package managers.

3.5.5 Summary

There have been various small-scale studies that have considered how prevalent technical lag is in projects. In most studies, slightly less than half of projects were out of date, with most being behind by a single major or minor version, or a few micro versions. Having technical lag has been pinpointed as a major security vulnerability by several studies listed in Section 3.5.3, and in many cases studied, developers were unaware that they had been using an outdated and vulnerable dependency. There is a dearth of literature comparing technical lag to bugginess in programmes - the only hint that there may be a correlation is a study showing a strong correlation between low technical lag and high ratings in mobile apps.

Updates tend to occur sporadically, bringing projects up to date in most cases. It has been noted that more restrictive version constraints lead to developers updating more regularly - an interesting but unsurprising conclusion. It has also been noted that developers update mature dependencies more regularly, a practice that correlates with mature dependencies having more stable APIs.

There has also been some evidence that developers will delay major updates or updates with known breaking changes until a major revision of their own project.

3.6 Automating the Update Process

Previous sections have discussed that developers often delay updating their dependencies. This is mainly due to the fact that updating dependencies is a task that must be done manually, and at times can require a significant amount of effort. Lowering the costs of updating dependencies by automating this process has been the topic of several studies over the past decade, and holds promise for reducing the workload involved in the update process.

Several tools are available that assist with automating the process of updating declarations as needed. Greenkeeper[16], Dependabot[17], and Dependencies.io[18] are some of the services that sit within a continuous integration service or as a git repository plugin, sending pull requests to update declarations whenever dependency updates occur. This allows developers to test the branch containing the dependency update (some services build the project and run its regression tests automatically to further automate the process), and merge it if there are no issues. Automating this process gives instant feedback to developers when dependency updates have occurred, meaning that developers can at least *know* that the dependency update exists and decide if they want to update immediately or delay the update. Another tool, Snyk[19], works in a similar way to alert developers of security vulnerabilities in their dependencies. For both types of tools, the emphasis is on providing a quick transfer of information, solving the issue that researchers have pointed out about developers often being unaware of updates or vulnerabilities.

There are a number of tools that can detect breaking API changes, including Clirr[20] (widely used in research but not updated since 2005) and Revapi[21] for

[16] `https://greenkeeper.io/`
[17] `https://dependabot.com/`
[18] `https://www.dependencies.io/`
[19] `https://www.snyk.io/`
[20] `http://clirr.sourceforge.net/`
[21] `https://revapi.org/`

Java, or NDepend[22] for .NET. In Section 3.3.3, *PDE API Tools*[23] was also discussed as a tool used by the Eclipse community which does a similar job. These are helpful for detecting API changes, and pinpoint where developers must look closely at the dependency updates, and in some cases can be plugged into tools such as Maven, Visual Studio or cloud-based platform tools. This integration with the automated build cycle is desirable - making the build process as automated as possible [Humble and Farley, 2010], and is considered best practice within DevOps. Jezek and Dietrich benchmarked nine API compatibility tools against a suite of API breaking changes, finding that there are now some highly usable and accurate tools available [Jezek and Dietrich, 2017]. For all API compatibility tools, it should be noted that they only focus on API changes, and ignore that semantic changes within projects can also be breaking, so they only cover one aspect of compatibility.

Foo et al. report on an API incompatibility checker that uses static analysis [Foo et al., 2018]. A statically constructed callgraph is used to detect deep changes that can effect compatibility. The tool works for the Maven, PyPi, and RubyGems ecosystems, and the authors report that based on the experiments with the tools, 26% of library versions are in violation of semantic versioning. The authors caution that this analysis suffers from the imprecision of the static analysis being used (VTA) though, with a large amount of false positives being associated with this method.

According to Dig and Johnson [2006], over 80% of breaking changes are refactoring based. As such, they recommended that a refactoring-based tool be created to limit the developer workload when updating, a tool called *Catchup*, created by Henkel and Diwan [2005], which records refactorings, and create a tool that takes the refactorings and adapts client code. Subsequently, Cossette and Walker performed a study to test the upgrade techniques that had been previously suggested by literature and found that only about 20% of refactoring changes would be automated correctly. So, while refactoring is a significant source of the work required for updating client code, it is not trivial to perform correctly.

[22] `https://www.ndepend.com/`
[23] `https://www.eclipse.org/pde/pde-api-tools/`

3.7 Summary

Dependency management is an important part of managing highly effective software projects. With such large volumes of dependencies being used in an average project, and many more added into the project via transitive requirements, there is a strong potential for errors and issues that the developer must spend time monitoring and fixing as they arise.

Backwards compatibility is a major concern when considering updating dependencies. There are four main contracts which a dependency must uphold for backwards compatibility to be ensured - syntactic contracts (which we broke down into API stability and binary compatibility sections), behavioural contracts (pre- and post-conditions which define the semantics of a code snippet), synchronisation contracts and quality of service contracts. Syntactic contracts are the type that is most often considered - when there are violations, it will either cause compile time errors or run time errors. Checking that these contracts hold is also something that is within the reach of automation, with API comparison tools and automatic refactoring tools being used or in development. Behavioural contracts are much harder to reason about and spot, particularly as few languages enforce the pre- and post-conditions needed for tools to automate checks. Yet breaking behavioural contracts is problematic, leading to issues that are difficult to debug.

We discussed declarations and how they are *version constraints* - limiting the possible dependency versions that can be chosen to a subset of the versions available. Increasingly, ecosystems are moving to semver-compliant ranges, which provide a mixture of automated dependency updates and some protection from breaking updates - assuming upstream developers respect semver guidelines and can identify all types of breaking changes. Several studies highlighted in this chapter note, however, that this does not yet happen industry-wide and is, realistically, very complicated. Yet, at this point, both open ranges and fixed declarations are still widely used in some ecosystems, a key finding from Chapters 4 and 5. Understanding the exact reasoning for why developers choose specific styles of version constraints is an open question, with a few hints being found in research in Section 3.5. Chapter 6 looks closely at why developers choose specific versioning styles.

Keeping dependencies up to date and free of technical lag is a proven way to

reduce security vulnerabilities and fault-proneness. All non-open-range declarations must be updated from time to time in order to stay up to date. Current research points to updates happening every handful of published versions, but this varies wildly by ecosystem, with a study suggesting as few as 1 in 50 versions of mobile apps update dependencies. There have been some tools that help developers know when new dependency versions are available and nudge them in the direction of updating, which has helped to close the information gap that earlier research noted was a partial cause of technical lag. In general, we can say that small amounts of technical lag exists in the average project, but that it incurs major costs. Chapter 7 examines the topic of lag and updates in great detail, providing the first large-scale study across most major package managers.

Chapter 4

Prelude: Declaration Classifications using GitHub

In this chapter, we discuss a pilot experiment to ascertain the relative frequency of fixed declarations versus flexible declarations. It produced interesting results which led to further investigations in the form of a developer survey in Chapter 6, and this study was extended with a larger dataset which covers more package managers, as discussed in Chapter 5. Since these results were used to cross validate the results from Chapter 5's experiments, this chapter has been included, in a very short form, as a prelude to the main research chapters that follow. The goal of this study was to answer the following question, which we expected would lead to further questions:

RQ1: What is the overall proportion of fixed vs flexible dependency declarations in a given project?

4.1 Methodology

In order to gather the initial data required to find out how dependencies are declared, projects were harvested off GitHub, the largest repository of open

source projects available online. This first dataset was custom-built, gathered by scraping projects from GitHub over a two month period at the beginning of 2018, and consisted of all publicly visible projects that used npm, Maven, Gradle or Ant as their package managers at that time.

4.1.1 Custom GitHub Dataset

At the time of the study, we were not aware of any pre-made data sets containing dependency information, so a custom-built dataset was created by collecting projects from online repositories and extracting the declaration strings needed to answer the research question.

Gathering data by scraping GitHub or other online repositories is a common methodology, and Kalliamvakou et al. [2014] discusses the pitfalls of scraping GitHub projects. Some of the points from that paper were not applicable here, however, there were a few points which will affect the data and should be kept in mind:

- Most projects have very few commits.
- Many projects are no longer under active development.
- Some projects are personal projects - for example using GitHub as a backup only.

While these do not necessarily invalidate the data from this dataset, the dataset gathered for this study includes all available projects that used our target package managers - everything from small-scale, personal projects right up to highly influential, open-source projects. For our analysis, projects without dependencies have been excluded (after reporting how many do not have dependencies) - most of these are small projects. Projects that are forked (and therefore are potentially duplicated projects) have also been excluded, but all others have been included by default.

The scraping tool targeted open source projects using the npm, Gradle, Maven or Ant package managers. When such projects were found, the configuration files for that package manager found in the master branch were downloaded - this generally represents the most recent version of the project. The Ant files were later discarded from the dataset, as those with dependency declarations (using Ivy) were found to be a very small overall proportion of the

total projects, indicating that most projects using Ant used dependencies located locally on file systems rather than using online fetching methods required for flexible declaration patterns.

Package Manager	**Metadata File(s)**	**Parsing Method**
npm	package.json	Document Tree
Gradle	build.gradle, settings.gradle	Custom parsing algorithm
Maven	pom.xml	Document Tree

Table 4.1: Configuration Files and Parsing Methods.

After harvesting the configuration files from GitHub, the next step was to parse the files to extract the declaration strings. The configuration files listed in Table 4.1 were parsed according to each package manager's syntax. Maven and npm files were parsed using an object tree model, allowed by their XML and JSON file structures (misformed files were discarded). This allowed declarations to be extracted cleanly, with the complication that Maven files allow for variables to be where declarations are needed. This involved a recursive resolution strategy, as it is possible for multiple levels of indirection to exist, but a near complete variable resolution count was achieved.

Gradle was much more difficult to parse, as its configuration files are written in the Groovy language, rather than a structured markup file. The method chosen to extract declaration strings was a manual traversal combined with regular expresions, which proved more error-prone - sampling indicated that 3% of Gradle files were parsed incorrectly using this method. It was subsequently found that Gradle commonly used submodules in its projects, so submodule configuration files (`build.gradle` and `settings.gradle`) were gathered at a later point, about three months after the initial files. This is a threat to the validity of the data, as in some cases, the root configuration file will be from a different published version than the submodule configuration files.

The submodule files in Gradle often contained variables that were needed for resolving Gradle declaration variables. It was also possible for variables to exist in plugins linked to Gradle, so some variables were not able to be resolved. The process for resolving the variables was done in a similar way as Maven, except that the searching process was more involved due to the less structured

nature of the files. After multiple iterations of validation and improvements, about 2.5% of Gradle variables remained unresolved.

Overall, four million projects were gathered and analysed for this dataset, providing a useful comparison for later results. It also represents a wider dataset than used in subsequent chapters.

4.1.2 Analysis

Once the version declaration data was collected, we classified each version in one of five ways, based loosely on semantic versioning principles but being flexible enough to include other styles of version ranges:

- Fixed version - only 1 single version can be accepted
- Micro range - multiple versions accepted but only within the same minor range
- Minor range - multiple versions accepted but only within the same major range
- Major range - multiple versions accepted within at least two possible major versions
- Other - examples include urls or local files, generally equivalent to fixed versions

These categories are *loosely* based on semver, but differ slightly from the categorisations used in later chapters. In particular, micro ranges and minor ranges in this chapter are subsets of micro and minor ranges in the following chapters, i.e.,

$$micro_{thischap} \subseteq micro_{laterchaps}$$

$$minor_{thischap} \subseteq minor_{laterchaps}$$

Two examples are shown of this in Table 4.2 where an npm micro range looks like `1.7.3 - 1.7.6` (excluding any micro updates that come after 1.7.6) and a minor range looks like `1.7.3 - 1.9` (where any minor updates after 1.9 are out of range).

When further aggregations are used, micro, minor and major ranges map to `flexible` declarations. The `other` classifications are ignored except when considering total numbers of dependencies, as they do not adhere to the fixed versus flexible dichotomy that we are investigating in this study.

Package Manager	**npm**	**Gradle**	**Maven**
Fixed Version	1.7.3	1.7.3	1.7.3
Micro Range	1.7 1.7.+ 1.7.* ˜1.7.3 1.7.3 - 1.7.6	1.7.3+ 1.7.+	[1.7.3, 1.8)
Minor Range	1 1.+ 1.* ˆ1.7.3 1.7.3 - 1.9	1.7+ 1.+	[1.7.3, 2)
Major Range	* + <1.7.3 >= 1.7.3 1.7.3 - 2.4.8	1+ +	[1.7.3,), [1.7.3, 2.5.0) latest

Table 4.2: Example Patterns Mapped to Categories

Regular expressions were used to categorise the string version declarations, attempting to match the version declarations to one of the example patterns noted in Table 4.2. In some cases, this became complicated due to package managers having several possible declaration syntaxes - with npm there were 19 different classifications used, which have been mapped to the above 5 categories for reporting purposes. For syntactic patterns where multiple possible classifications exist (e.g. the range syntax in Maven), the versions were extracted and programmatically examined to determine if the declaration was fixed, or a micro, minor or major range.

4.2 Results

RQ1: What is the overall proportion of fixed vs flexible dependency declarations in a given project?

Table 4.3 shows the number of projects that contain dependencies, which ranges from 73% of npm projects to almost 92% of Gradle projects - those without dependencies are filtered out of the study.

	Projects	**Dependencies Exist**	**Fixed**	**Flexible**
npm	2 575 654	1 883 482 (73.1%)	514 154 (27.2%)	1 788 206 (94.9%)
Gradle	851 981	782 729 (91.8%)	748 282 (95.6%)	114 386 (14.6%)
Maven	746 562	623 406 (83.5%)	620 587 (99.5%)	15 155 (2.4%)

Table 4.3: Summary of Declarations by Package Manager

Of the projects that do use dependencies, Table 4.3 shows how many include fixed or flexible declarations. Where a project uses both, it is counted in each column. There is a sharp divide between the npm package manager and the JVM ecosystems, with Gradle and Maven extremely likely to use fixed versions, while almost all npm projects use some flexible declarations. There are also a significant amount of projects that use both - roughly one in five npm and one in ten Gradle projects will use both fixed and flexible version declarations.

	Fixed	**Micro**	**Minor**	**Major**	**Total**
npm	1.17	1.03	5.58	0.39	8.21
Gradle	5.42	0.05	0.15	0.05	6.84
Maven	8.28	0.003	0.005	0.036	8.33

Table 4.4: Mean Dependency Declarations per Project

The mean sum of direct dependencies are similar in all three package managers studied. Table 4.4 agrees with the previous findings that npm projects generally use flexible declarations while Maven and Gradle predominantly use fixed declarations. However, npm uses a lot more fixed declarations overall than the other package managers use flexible declarations, indicating that npm devel-

opers may switch between the two styles more readily. Gradle has a large number of *other* classifications - averaging more than one declaration per project.

The following results look further into the usage of flexible declarations, filtering projects that do not use flexible declarations.

	Projects	**Micro Ranges**	**Minor Ranges**	**Major Ranges**
npm	1 788 206	369 758 (21%)	1 557 683 (87%)	147 885 (8%)
Gradle	114 386	26 409 (23%)	76 373 (67%)	21 681 (19%)
Maven	15 155	1 277 (8%)	1 387 (9%)	12 987 (86%)

Table 4.5: Flexible Styles Used in Projects

Flexible declarations can be broken down into micro, minor and major ranges, as described in Section 4.1. Table 4.5 shows the proportion of projects (containing flexible declarations) that use each type. Semantic versioning principles recommend developers use minor ranges - maximising the opportunity to receive automatic updates while (hopefully) avoiding breaking updates - or micro ranges as a safer option given that semantic versions are not always updated correctly. npm developers seem to follow this recommendation, with 87% of npm projects that use flexible declarations contain minor ranges, and 21% contain micro ranges. Gradle developers who use ranges also follow this recommendation, pointing to an overall awareness of semver principles within its ecosystem. Semantic versioning recommends that developers do not use major ranges, as by definition they allow breaking changes. The few Maven projects that use flexible declarations tend to ignore this advice - 86% of them include major ranges.

	Projects	**Mixes Ranges**	**Inc Fixed**
npm	1 788 206	14.6%	23.8%
Gradle	114 386	8.3%	89.3%
Maven	15 155	2.9%	81.4%

Table 4.6: Do Projects Use Multiple Declaration Styles?

Initial results showed that developers will sometimes use more than one declaration style (micro, minor, major, fixed) in a given project. Table 4.6

looks at how common it is for projects to mix flexible declaration types, and those that use both flexible and fixed declarations together. While most Maven users stick to one specific flexible style (generally the Maven keyword *latest*), it is somewhat more common for Gradle and npm developers to mix and match, with 8% and 15% of projects using more than one type of flexible style respectively.

A significant number of projects using flexible declarations also use fixed declarations. In Gradle and Maven, over 80% of projects containing flexible declarations will also contain fixed declarations. That said, Gradle and Maven projects only include flexible declarations 15% and 2% of the time respectively, so the average project in both package managers will likely contain only fixed declarations.

Based on these initial results showing npm having substantially more flexible declarations, we hypothesise that npm developers, who come from a fast-moving web background, are more comfortable with risk than Gradle and Maven developers. This may also reflect a difference in the cost-benefit considerations faced. Websites (npm's main use case) can be changed quickly, as there are copies of the programme in relatively few places - usually under the developer's direct control. When a problem is discovered, the fix affecting all users can be rolled out in a matter of minutes. In the case of mobile apps (Gradle's main market) or desktop applications, once a specific version has been downloaded by a client, they may be slow to update, or may not update at all. This simple difference in distributing updates could be a contributing factor of why some ecosystems are more risk-averse than others.

Another factor that may affect declaration usage is convenience. Both Gradle and npm use shortcuts that allow micro and minor semver-compliant declarations to be typed within seconds, as opposed to Maven's verbose range notation. On the other hand, Gradle and Maven both rely heavily on the Maven repository, which generates code stubs containing fixed declarations that can be added to the configuration files. If these code stubs are widely copy-pasted within these ecosystems, this would influence the average project to lean heavily to fixed declarations, potentially without the developer making a conscious choice to do so.

The other interesting result from this initial experiment is the idea that developers mix and match dependency declaration styles. It could be that multiple developers work on a single project, each having a different style, or it

could be that the one developer working on this does not have a consistent style themselves. A more interesting possibility, however, would be that developers adjust their versioning strategy based on the perceived 'trust' that is associated with the dependency's history of backwards compatibility.

4.3 Summary

This pilot study reported in this chapter described the state of declaration patterns for dependencies in three major package managers (npm, Gradle and Maven), pointing to some interesting trends, in particular the strong division of the use of semantic versioning principles and more generally fixed versus flexible declarations between package managers, with npm users tending to use flexible declarations, while Gradle and Mavem users primarily use fixed declarations. The exploratory nature of this chapter created numerous questions, such as:

1. Do other major package managers also have quite distinct styles of declaration usage from one another as observed between npm and Gradle/Maven?
2. What causes developers and ecosystems to adopt such distinct dependency management strategies?
3. How often do developers use more than one declaration pattern on purpose, and why?
4. How widespread is semantic versioning knowledge amongst developers?

Chapter 5 seeks to answer the first question with quantitative measures by looking at the declarations developers use in a large-scale open-source dataset. Chapter 6 focuses on the developer knowledge about semver and their usage practices, by asking developers directly about questions relating to the second, third and fourth questions.

Chapter 5

A Large-Scale Study on Declaration Classifications

Many modern software systems are built on top of existing packages (modules, components, libraries). The increasing number and complexity of dependencies has given rise to automated dependency management where package managers resolve symbolic dependencies against a central repository. When declaring dependencies, developers face various choices, such as whether or not to declare a fixed version or a range of versions. The former results in runtime behaviour that is easier to predict, whilst the latter enables flexibility in resolution that can, for example, prevent different versions of the same package being included and facilitates the automated deployment of bug fixes.[1]

We study the choices developers make across 17 different package managers, investigating over 70 million dependencies. We find that many package man-

[1]This chapter consists of a collaborative work between the author, supervisors (A/Prof. Jens Dietrich, Dr. Amjed Tahir), and collaborators from University of Auckland (Dr. Kelly Blincoe) and Victoria University of Wellington (Dr. David Pearce). The author's contribution is estimated at 20% of the overall work. Sections 5.2.1 - 5.2.4 have been added to provide further background to the reader of the thesis, and other sections have been changed for this thesis. The full work was published in the Mining Software Repositories (MSR) Conference 2019 [Dietrich et al., 2019], however the sections the author did not personally contribute to have been removed, and the remaining sections have been edited to tie in with the wider body of work. The developer survey the author undertook with A/Prof Dietrich (Chapter 6) was published in conjunction with this study, however, logically it is presented as a stand-alone component.

agers support - and the respective community adapts - flexible versioning practices. We see some uptake of semantic versioning in some package managers, supported by tools. However, there is no evidence that projects switch to semantic versioning on a large scale.

The results of this study can guide further tooling support for automated dependency management, and aid the adaptation of semantic versioning in practice.

5.1 Introduction

One challenge now faced by software developers is deciding, for a given package, on which version to depend. For example, one can depend upon a *fixed version* of a given package (i.e. "use only version 1.2.2") or on a *version range* (i.e. "use version 1.2 or higher"). With fixed versions, builds are more deterministic[2], but critical fixes in later versions of the package will not be automatically included [Derr et al., 2017]. In contrast, version ranges have the disadvantage that builds can now fail if changes between versions are not backwards compatible [Dietrich et al., 2014, Raemaekers et al., 2017, Xavier et al., 2017]. On the upside, version ranges allow the package manager to select the "best" version with respect to some metric (e.g. the latest stable version meeting all constraints). This means new versions which fix bugs, address security vulnerabilities, or improve performance are automatically included whenever a project is rebuilt. Another benefit of version ranges is that the package manager can handle packages that are included multiple times by intersecting all constraints to find a single match, thereby preventing multiple inclusions of the same package. This problem has become exasperated in recent times as packages have more and more (transitive) dependencies. While some techniques exist to separate those packages at runtime, such as the use of class loaders in OSGi[3] or JavaScript programming patterns to avoid conflicts in the global namespace like jQuery's `noConflict()`[4], many systems are still prone to runtime version conflicts and the DLL-hell-style bugs [Szyperski, 1999] resulting from them.

[2] Even when only fixed version dependencies are used, builds are not necessarily guaranteed to be completely deterministic, e.g. a dependency may itself declare further dependencies using ranges, therefore the transitive dependencies would not be deterministic.

[3] `https://www.osgi.org/`

[4] `https://api.jquery.com/jquery.noconflict/` [Accessed 6 Apr 2020]

On the other hand, in order to correctly signal incompatible changes to client projects, developers also must understand how incompatible changes between versions arise (as discussed in Sections 3.1 - 3.3, this can be complicated). Packages are governed by multi-faceted contracts that are often only implicitly stated [Beugnard et al., 1999]. This may include *API signatures*, *semantic contracts* (expressed informally or formally through pre- and post-conditions), expectations on *performance* and *resource usage*, and non-technical aspects such as *licenses*. For instance, API signatures are often considered as easy to reason about but, even for well-specified, statically typed languages like Java, the situation is surprisingly complex: most developers don't understand the compatibility rules [Dietrich et al., 2016], tools that try to detect incompatible evolution are incomplete [Jezek and Dietrich, 2017], and incompatible evolution that can break client packages is common [Dietrich et al., 2014, Raemaekers et al., 2017, Haney, 2016]. Semantic contract violations present an even bigger challenge for detection. For example, updating an API so that some parameter no longer accepts null is not backwards compatible as it is strengthening a precondition, as it would previously allow some inputs that it now refuses. Whilst such a violation could conceivably be detected using some form of non-null static analysis [Ekman and Hedin, 2007, Male et al., 2008, Chalin and James, 2007, Fähndrich and Leino, 2003][5], things are less clear for arbitrary contracts (e.g. JML [Jacobs and Poll, 2001]). Of course, testing provides some capability here [Claessen and Hughes, 2000] but, due to its inherent unsoundness, may easily miss violations. Therefore, many issues caused by incompatible evolution of packages may only be detected after deployment and, hence, can be particularly damaging.

In practice, developers have to make trade-offs between those two strategies, balancing the opportunities of optimised systems with the risks of incompatibility errors. *Semantic versioning* has arisen as a popular approach for managing package evolution which uses structured versions of the form "`major.minor.micro`" (see Section 2.1). The idea is to associate certain compatibility guarantees with changes to parts of this structure. For example, when increasing the `minor` version of a package (e.g. `1.2.3` $\implies$ `1.3.0`), all changes should be backwards compatible with previous versions at the same `major` level.

[5] Nullaway (`https://github.com/uber/NullAway`) and Checkerframework (`https://checkerframework.org/`) are both widely used packages for null checking

In contrast, incompatible changes are only permitted between versions at the `major` level (e.g. `1.2.0` $\Longrightarrow$ `2.0.0`). The challenge for developers, however, lies in correctly following this protocol. This is because (as discussed above, along with in Section 3.3) incompatible changes are sometimes subtle and hard to spot, even for seasoned developers. Likewise, there is limited tooling available for checking adherence to the protocol (Section 3.6).

We are particularly interested in the adaptation of semantic versioning. That is, given the above difficulties in sticking with the semantic versioning protocol, *what do developers do?* They might, for example, simply eschew semantic versioning altogether in favour of fixed versions; or, they might throw caution to the wind, and fully embrace semantic versioning despite the challenges, or potentially use some other flexible declaration scheme other than semantic versioning. This study can be considered complementary to the study by Decan and Mens [2019] (outlined in Section 3.5.1), who take a time-sensitive approach that looks at how general declaration styles have changed over time in a subset of package managers, in contrast with our focus on more fine-grained declaration styles across more package manager, and how the number of dependencies change over time. The aim of this study is to investigate which choices developers make across different package managers. Looking at different package managers gives insight into the ecosystems of specific languages (e.g. Java versus JavaScript) and language features (e.g. static versus dynamic typing).

More specifically, we (1) set out to capture the current practice regarding how developers declare dependencies, and (2) we also investigate whether and how developers change their approach as projects mature. This is of particular interest as it will provide some evidence about whether a certain approach is working or not. Therefore, we ask the following research questions in this chapter:

RQ1 **How do projects declare dependencies?**

RQ2 **Do projects change their approach as they evolve?**

These two questions together provide an overview of how common specific declaration styles are, and an indication of how they are changing over time.

5.2 Methodology

In this section, the methodology used to acquire and analyse data is discussed. We will discuss limitations and threats to validity within the subsections as necessary.

5.2.1 Dataset Acquisition

We used the *libraries.io* dataset for this study, using the `v1.2.0` (March 2018) dump. The data set contains dependency data in CSV format that we imported into a PostgreSQL database for further analysis and processing. In particular, the dataset contains a *dependencies* table which has versioned dependencies of packages to other packages. This table has 71,884,555 records with dependency information for packages from 17 different package managers, listed in Section 5.2.2.

For each project, the dataset contains information about each version, including a publication timestamp, the repository where it was found, and importantly for this study, the dependency declarations (saved as raw strings) and the project name of the dependency for each version. Note that unlike with the GitHub dataset from Chapter 4, Maven variables were not resolved in the *libraries.io* dataset, leading to roughly 13% of declarations being classed as *unresolved*. A comparison of results between the two chapters show that the presence of these unresolved variables do not introduce noticeable bias into the results.

Atom is a special case as it also allows users to specify dependencies to npm packages. However, Atom packages are managed in a different repository[6], and we therefore decided to keep those packages in the data set.

5.2.2 Package Managers Covered

As we study dependencies in this work, and not all package managers deal with dependency resolution, only a subset of the available package managers were chosen. The dataset used in this study, *libraries.io*, contained 17 package managers which supported dependency management. Table 5.1 provides an overview of the package managers used.

[6] `https://atom.io/packages` [Accessed: 16 Jan 2019]

Package Manager	**Primary Language(s)**	**Notes**
Atom	JavaScript	Uses npm for dependency management
Cargo	Rust	
CPAN	Perl	
CRAN	R	
Dub	D	A successor of C++
Elm	Elm	Functional front-end language
Haxelib	Haxe	ActionScript successor
Hex	Erlang	
Homebrew	Ruby	Package installer for MacOS and Linux
Maven	JVM languages	Includes Gradle and Ivy
npm	JavaScript	
NuGet	.NET languages	
Packagist	PHP	See footnote[7]
Pub	Dart	Client-optimised for multiple platforms
Puppet	Ruby	Configuration management tool
PyPI	Python	
Rubygems	Ruby	

Table 5.1: Package Managers Used

As shown in Table 5.1, most major package managers are included in this work, along with a variety of much smaller package managers. The table also includes notes about some of the lesser known languages, for reference. Most package managers focus on building projects, and have specific languages that they work with - in most cases a package manager is tied to one specific language or language family. For this reason, when talking about a package manager's community, we often refer to them as an *ecosystem*, a self-contained unit, distinct from other ecosystems, that uses one language (or a few related languages), a package manager, and a set of tools that work with those technologies.

Maven is unusual in that it combines several related JVM package managers. It supports Maven, Gradle, and Ivy declarations as part of the same classifica-

[7]Packagist is Composer's default repository - it has been listed as *Packagist* by the *libraries.io* dataset, but the more appropriate label may be *Composer* instead, which is the main PHP package manager.

tion. As shown in Chapter 4, the Gradle community has a slightly different approach to dependency management compared with Maven, so that should be kept in mind when reading Chapters 5 and 7. The *sbt* package manager is also included in the Maven classification through its use of Ivy (a dependency management add on originally confined to Ant but has since been used in sbt and Gradle) for dependency management.

Most package managers studied are intended to be used to build projects in specific languages. Homebrew and Puppet are unusual in that they focus on installing packages within a MacOS and Linux environment and as a configuration management tool respectively, rather than collecting dependencies for a specific application build. Both use Ruby for defining tasks, and therefore the primary language was noted as Ruby, but unlike Rubygems, they are not intended to build Ruby-based projects.

There are some notable exceptions that have not been studied. Major missing ecosystems include C/C++ (with their plethora of build tools), Haskell (Cabal), Go and Swift (package management can be done directly through tools built into the language), Objective C (CocoaPods), and Chocolatey (Windows). The choice of including or excluding package managers ultimately came down to the data available - we did not find easily accessible data for these ecosystems.

5.2.3 Categories

Projects will generally publish multiple versions over their lifetime, each distinct from the next. When a project depends on another project, in order to avoid compatibility issue, it is necessary to declare the version of the dependency required. The declaration can come in one of two general flavours, *fixed* or *flexible*. Fixed declarations allow package managers to choose only one possible version of the dependency to satisfy the declaration, whereas flexible declarations will allow more than one.

The flexible declarations can be further categorised depending on how many versions could possibly satisfy the declaration. In general, we have categorised these according to semantic versioning. Table 5.2 lists the main declaration types used in this study.

Chapter 5 further specifies the classifications in Table 5.2 by including the following:

Classification	**Explanation**	**Example *Satisfies* Set S**
Fixed	Only one version	$S = \{x \mid x = 1.3.3\}$
Micro	Any later version in the same micro range	$S = \{x \mid 1.3.3 \leq x < 1.4.0\}$
Minor	Any later version in the same minor range	$S = \{x \mid 1.3.3 \leq x < 2.0.0\}$
At-Least	Any later version	$S = \{x \mid 1.3.3 \leq x\}$
At-Most	Any earlier version	$S = \{x \mid x \leq 1.3.3\}$
Any	Any version	$S = \{x \mid x \in DepVersions\}$
Range	Some custom range	$S = \{x \mid 1.3.3 \leq x < 1.6.0\}$

Table 5.2: Declaration Styles

1. *soft* - a type of *fixed* declaration that some package managers use to assist dependency resolution of conflicts.

2. *latest* - a type of *any* declaration that specifically chooses the newest version (as pointed out in Section 2.2, although we assume that the highest version that satisfies a declaration will be chosen, this may not always happen).

3. *not* - a type of inverse *range*, where instead of defining a satisfies set S by inclusions, it defines it by exclusions, e.g. $S = \{x \mid x \neq 1.3.3\}$.

4. *unresolved* - some package managers allow variables to be used in declarations. These were not always resolved before being included in the dataset, so were not able to be analysed.

5. *other* - a particularly unusual pattern which did not fit into any other category, or one that did not follow semver versioning styles, but is otherwise something that we expect to find in the dataset and can create parsing rules for, e.g. a GitHub repository URL.

6. *unclassified* - a declaration that didn't conform to any parsing rule for that package manager.

In Chapters 5 and 7, these categories are regularly grouped further, as different types of analysis are undertaken.

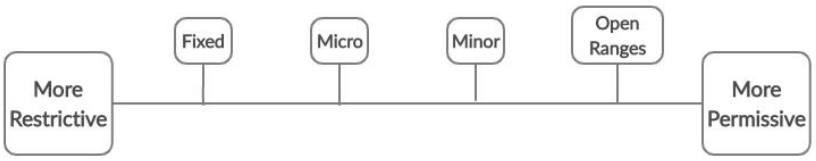

Figure 5.1: Declaration Range Continuum

5.2.4 Declaration Parsing

Each package manager has its own syntax for declarations. The declarations must be parsed in order to categorise them, which means that each package manager requires parsing rules to take a declaration in the form of a raw string, and produce a categorisation.

This process was done over the course of Chapter 5 in a collaborative effort that resulted in an MSR publication [Dietrich et al., 2019]. This section outlines the parsing process, which was a significant proportion of the work required for this study. The repository for that study[8] has the exact parsing rules for each package manager.

For each package manager, a list of rules were created. As shown in Figure 5.2, each rule consisted of a regular expression that encompassed one type of syntax and mapped to one category. For a given raw string containing a declaration, the string was progressively tested against each rule within the package manager until it found a rule that satisfied it, or it exhausted the rules without a match and received an *unclassified* categorisation.

For validation purposes, each rule is accompanied by several test strings that satisfy that regular expression. Some tests were created from the specifications of that package manager, and others were added during the validation process as false positives were spotted in the dataset, checked by both the author of the rules and a reviewer. At each test, a sample of declarations were chosen and classified that would give a 95% confidence level and a 5-10% confidence interval, along with additional classifications for under-represented classifications to ensure each sample had at least 10 declarations of each classification. The sample would then be checked by the author and the reviewer for false posi-

[8] `https://bitbucket.org/jensdietrich/lib.io-study/src/master/mapping-rules/`

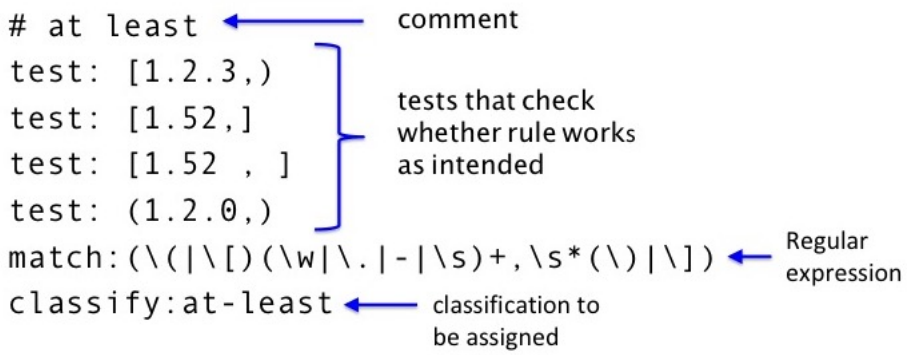

Figure 5.2: Example Rule [Dietrich et al., 2019]

tives. Where issues were found, the rules would be updated and the sample was regenerated. This continued iteratively until both the author and reviewer were satisfied that all possible declarations in the sample were classified correctly.

I created rules for Atom, npm and Rubygems. I then reviewed rules created for Cargo, Elm, Haxelib and Maven. Later, in the process of the Chapter 7 study, I made further alterations to the Packagist rules.

An interesting sidenote is that due to the rule reliance on regular expressions, there were in practice some exponential complexity parsing operations. In both Chapters 5 and 7, it was necessary to account for this and avoid parsing some declarations. The approach in Chapter 5 was to monitor threads that were parsing declarations and terminate any that took an abnormal amount of time to terminate. In Chapter 7, where multithreading[9] was not used, the approach was to blacklist specific offending projects - a total of four Packagist project pairs.

Table 5.3 shows example syntax patterns for micro, minor and range classifications, which were the categories that varied most by package manager. In addition to the declarations shown in Table 5.3, almost all languages had options for *at-most* and *at-least* classifications, along with fixed classifications. With minor differences, these generally looked similar (`<4.0.0`, `>=4.0.0`, or `4.0.0` respectively). It was also common for package managers to allow com-

[9]There were two reasons for not using multithreading: 1. Originally one of the classes was incompatible with concurrency (this was later changed), and 2. the computational load was not high enough to warrant the additional effort.

[10]Note that this should be classified as a minor range

binations of the above rules, creating composite declarations or finer grained detail where necessary. It is worth noting that some package managers did not allow for micro, minor or range declarations, in which case the relevant Table 5.3 cells were left blank.

In most cases, the rules allowed for additional whitespace and superfluous characters such as `=` or `v` before the version.

While many package managers had similar syntax, care was taken to look at the finer details. It was common for the declarations to look similar but for the package manager in question to interpret them differently. This happened often in the pre-`1.0.0` versions, where package managers inconsistently applied semver's guidelines about what minor or micro ranges look like for these versions. At other times, whitespace had meaning in some package managers and not others, and the process of joining multiple conditions was not consistently an `AND` or `OR` operation (an important point when parsing to a *satisfies set*, as discussed in Section 7.2.2).

Syntactic Mapping vs Semantic Mapping

The approach taken in this study was to parse according to a syntactic mapping strategy. This means that the categorisation was chosen solely based on the syntax chosen for the declaration, rather than any underlying semantics that could be embedded in the declaration. For example, while the Elm range in Table 5.3 fits the description of a *minor* range, it has been classified as a (custom) *range*. This approach was chosen for two reasons:

1. It is quite difficult to create a semantic mapping, and practically impossible when using regular expressions and rules as structured in Figure 5.2. It is possible to achieve using the programmatic abstractions created in Section 7.2.2, but this abstraction was created almost a year after the categorisation study. While it would be interesting to consider how many of the ranges are micro or minor ranges, particularly in the package managers that do not support micro or minor ranges, this remains as a next step for the study.

2. When package managers have shortcuts for micro and minor ranges, it explicitly shows the developer's intent to follow a semver-compliant range. While it could be inferred that generic ranges are intended to be a micro or a minor range, there is nothing to say that that was the developer's

intention.

This reliance on using syntax rather than semantics to categorise declarations mostly affects *range* classifications. Some range classifications could be considered *micro*, *minor*, or even *fixed* classifications if semantics were considered. The result of this choice is that, in ecosystems where *range* declarations are common, it understates the amount of time *micro* or *minor* ranges are used (as, in this study, *range* classifications are considered custom ranges without necessarily following semver principles).

5.2.5 Classification Aggregation

While the classification scheme provides a fine-grained view on the various patterns used, it is sometimes useful to consider dependency versioning from a more abstract point of view where we are interested to distinguish between the declaration of fixed versions and variable versions of some kind. Since one of the goals of this study is to investigate the uptake of semantic versioning, we also consider syntax that directly supports semantic versioning practices.

The aggregation of classification categories is defined by the following set of rules, using a simple rule syntax:

```
SEMVER := var-micro | var-minor
FLEXIBLE := range | soft | any | latest | not | at-least | at-most
FIXED := fixed
OTHER := other | unresolved | unclassified
```

The semantics of the rules are straight forward: if a dependency is classified using any category in the body (right side) of the rule, then it is classified in the category in the head (left side) of the rule.

The above mapping considers the *soft* classification to be flexible. As noted in Section 5.2.3, this interpretation was later updated to being a variant of fixed classifications, one which gives additional options to the package manager for resolving version conflicts. As such, Table 5.5 in RQ1 should be viewed as Maven having almost entirely fixed declarations, rather than having predominantly flexible declarations.

5.2.6 Version Ordering

In order to answer RQ2, it was necessary to identify the first and the last version of each project in the dataset. Using a naïve lexicographical order of version strings is not sufficient to achieve this with a sufficient level of accuracy, for instance, while this would yield `1.2.3 < 1.2.4` as expected, this method would also result in `1.2.10 < 1.2.9`. We therefore opted for a more accurate approach to first sanitise version strings (removing leading "`r`" or "`R`" preceding version strings), then to tokenise leading substrings matching `\d+(\.\d+)*` and comparing versions by comparing those numerical tokens from left to right. This was then implemented in a script that produced a table consisting of project name, first version, and last version for each package manager. Those tables were then sampled and peer-reviewed in order to ensure a sufficient level of accuracy.

We did not consider the semantics of additional strings following the numeric parts of the version, (such as `-alpha`, `-beta`, or `-rc1`), and used the lexicographical order for those suffixes. The reason for this decision was that there are a large number of custom prefixes (including hashes referring to commits), that are often platform and project specific. This can lead to cases where our method may not be able to extract the very first or the very last version in the dataset. For instance, we infer `1.2.3-ga < 1.2.3-rc` which is incorrect if one takes the meaning of the respective suffixes (`-ga` – general availability, `-rc` – release candidate) into account.

5.3 How Projects Declare Dependencies

Table 5.4 shows how the version constraints were classified for each package manager using the methodology described in Sections 5.2.3 and 5.2.4. The table also contains the number of records in the dataset for each package manager in the second column, where each record represents a single dependency of a version of a package to some version of another package. It is notable that the number of records varies by a factor of over 10^4 between the package managers with the smallest (Homebrew, 4,886) and the the largest (npm, 52,886,593).

Considering the data in Table 5.4, the kind of dependency versioning syntax used differs significantly between package managers, and no one common pattern

emerges, indicating that the divide between ecosystems in terms of using fixed, ranged and semver-compliant declarations spotted in Chapter 4 are not unique to the JavaScript and JVM ecosystems. Table 5.5 provides a more abstract view on the data, using the aggregation rules discussed in Section 5.2.5. It appears that there is a preference towards some kind of flexible dependency versioning declaration in all package managers, with significant uptake of a semantic-versioning style syntax in Atom, Cargo, Hex, npm and Rubygems.

We note however that we only measured the syntax being used, not the intent of the developer. In particular, there is one package manager where those two aspects may not be aligned - Maven. In Maven, 85.7% of dependencies are declared using the soft version syntax. We think that many developers look up libraries using the maven repository search engine[11], and copy and paste dependency declaration snippets into the project's `pom.xml` or equivalent build files (for Gradle, Ivy, etc.). These snippets use the soft version syntax, and it is not clear (1) how many developers actually understand that this is not a fixed version and (2) how often Maven resolves this to a different version than what is declared in actual builds using dependency management, mediation and exclusions[12]. Anecdotal experience makes us suspect that many developers are not aware of the difference and Maven will, in most cases, resolve the reference to the very version declared, indicating that Maven is actually an example of a system where developers take a conservative approach that favours fixed versions.

Elm and Homebrew stand out as both use only one particular versioning strategy. All Elm dependencies are declared using the version range syntax. This is the only syntax supported[13], and the dependency version is generated by the `elm package install` command. This is consistent with the overall approach of Elm to automate versioning and to limit the control developers have.

All Homebrew packages use the *any* syntax, granting full flexibility to the package manager to resolve dependencies. This is despite Homebrew offering a syntax for versioned dependencies - a minimum version can be declared as

[11] `https://mvnrepository.com/`

[12] `https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html` [Accessed: 21 April 2020]

[13] Although there is no schema that formally defined the syntax of the package manifest `elm-package.json`

additional dependency information[14].

As discussed in Chapter 3, the CRAN repository only allows the latest version of a project to be downloaded and used. It makes sense then that declarations are virtually all open ranges (*any* or *at-least*).

RQ1 How do projects declare dependencies?
All package managers investigated use predominantly some form of flexible dependency version syntax except Maven, where there is a strong tendency to use *soft* declarations. Some systems make extensive use of semantic versioning syntax. For Maven, it is unclear whether developers understand the difference between the soft and fixed declaration styles available to them.

5.4 Changing Dependency Versioning Practices as Projects Evolve

In order to answer RQ2, we extracted the first and the last version of each project, which were identified using the methodology described in Section 5.2.6. We then compared the dependencies declared in the first and last version, excluding projects that had less than two versions - allowing us to investigate evolution of declared dependencies over time. Table 5.6 describes attributes of typical packages within an ecosystem, in particular how many packages only contained one recorded version, and how many dependencies were declared. The majority of projects have more than one version in the dataset, with the exception of the projects using Homebrew[15].

Table 5.6 also shows the average number of versions per project and their respective standard deviations. Those numbers indicate that projects are typically represented by large version ranges, with a significant variation between projects. For instance, there are 33 npm projects with 1,000 or more versions in the dataset, and a further 2,584 projects with between 100 and 999 versions. The project with the most versions is *wix-style-react* — it provides common React UI components, with 3,550 versions (ranging from 1.0.0 to 1.1.3547). The large version ranges reflect the trend towards shorter, often highly automated

[14] `https://docs.brew.sh/Formula-Cookbook` [Accessed: 21 April 2020]

[15] Homebrew, being a configuration tool, lends itself to a 'set-and-forget' style of usage, explaining why many projects only have one release version.

build and release cycles.

Finally, Table 5.6 compares the number of declared dependencies the first and the last version for each project, computed using the methodology described in Section 5.2.6. The data indicates that the number of dependencies significantly increases over time for projects in all package managers except Homebrew, where the number stays constant. If we consider external dependencies as a source of complexity of a system, this confirms Lehmann's first and second law of software evolution: projects evolve and become more complex by doing so [Lehman, 1980]. Note that there is some additional hidden complexity as we only measure direct, not transitive dependencies. The average project in the dataset has at least one additional dependency by its last recorded version than its first. Compared to Chapter 4, the Maven projects studied here have slightly fewer dependencies, while the npm projects have slightly more - overall, the results between the two chapters corroborate.

5.4.1 Project Level Analysis

First, we examined the dependency strategies at a high-level by considering the strategies used across all dependencies for each project. The results are summarised in Table 5.7. We report the number of projects that use at least one flexible or semantic version style dependency in the first version, and add or drop those dependency versioning strategies, shown by its presence or absence in the last version. This is based on the aggregated classification scheme discussed in Section 5.2.5.

As shown in Table 5.7, projects tend to stick to their way of declaring dependencies, and generally resist change, with very few projects introducing new dependency versioning strategies or completely removing existing strategies. When projects do change their strategies, they more often move towards using semantic versioning or otherwise flexible dependency declarations, although there are exceptions (notably, *Maven*[16]).

[16]Here this would refer to Gradle, as Maven itself does not have shortcuts for var-micro or var-minor declarations as are required in this study to be considered *semver*.

5.4.2 Individual Dependency Level Analysis

To complement this coarse, project-level analysis, we also analysed how individual dependencies change over time as we hypothesized that projects will change their versioning practice for some, but not all, of their dependencies.

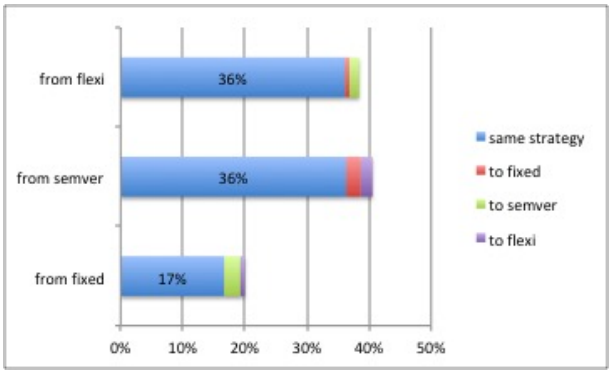

Figure 5.3: Changes in Dependency Declarations from First to Last Version

Again, we found no general trend towards or away from flexible or semantic-versioning style dependency versioning. Once a project chooses a dependency strategy for a particular dependency, it is very unlikely that they will change that strategy. Figure 5.3 shows that nearly 90% of projects keep the same dependency strategy from their first recorded version until their last recorded version. This shows that it is very important for projects to consider the implications of these decisions when adding a new dependency.

The results of this section contrast with the work by Decan and Mens [2019] discussed in Section 3.5.1, who found that there has been a significant shift in some ecosystems (particular Cargo and npm) towards semver-compliant declarations. A difference in that study was that it was an aggregation of declarations, rather than of projects. Another difference is that they filtered and split their results by time intervals. As our results include many projects that happened before their time period, it is possible that the large shift towards semver-compliant ranges is a recent phenomenon. It is also possible that projects which favour semver-compliant ranges have faster release cycles (it may not be unreasonable to assume that projects who embrace semver also embrace continuous delivery ideals) which would overstate the move towards semver-compliant ranges, and that newer projects use semver more than older projects (our analysis bases

comparisons with itself, not other projects).

RQ2 Do projects change their approach as they evolve?
The number of dependencies increase as projects develop across all package managers investigated except for Homebrew, where it stays constant. Projects both adapt and drop flexible and semantic version-style dependency version declarations, although the number of projects changing strategy is relatively small.

5.5 Conclusion

We have studied how developers declare dependencies across 17 different package managers, investigating over 70 million dependencies. We find that many package managers support, and the respective communities use, flexible declarations. We see uptake of semantic versioning in some package managers, supported by tools. However, there is no evidence that projects switch to semantic versioning on a large scale after they have chosen another declaration strategy.

Given that many projects use either fixed or semver-compliant declarations, which are susceptible to lag, an interesting follow up question is how much technical lag could we expect to find in a declaration, given its type. Related to this, how much lag can semver-compliant ranges help projects to avoid? Furthermore, given that fixed declarations must be updated regularly to keep up to date, how often are updates *actually* occurring, and what sort of load does this place on developers? Each of these questions are considered in the follow up study, reported in Chapter 7.

Other interesting topics for future research include more detailed analysis of what the technological and social barriers to the wider adaptation of semantic versioning are, and how particular communities deal with this. We touch on these topics in the developer survey next in Chapter 6.

Package Manager	**Micro**	**Minor**	**Range**
CPAN			<2.0, >3.0
CRAN			
Cargo	~1.2.3	^1.2.3	<0.6.0, >= 0.4.2
Dub	~> 1.30.11	~> 1.2	>1.2 <2.0
Elm			3.0.0 <= v <4.0.0
Haxelib			
Hex	~> 1.30.11	~> 1.2	>2.0.0 and <2.1.0
Homebrew			
Maven (Gradle-like) (Ivy-like)	1.2.+	1.+	[1.2.0,2.0.0) [0.0.0, 1.0.0[
NPM (or Atom)	~1.2.3	^1.2.3	>=1.2.3 <2.0.0 1.2.3 - 1.2.8
NuGet	[1.1.*]	[1.*]	[1.0,2.0) >= 3.2.0.11 <3.3.0 ^1.5.0
Packagist	1.0.* ~1.2.3	1.* ^1.2.3	>=1.0 <1.1 ||> = 1.2 1.0 - 2.0
Pub			>=2.3.5 <2.4.0 ^1.0 ([10])
Puppet	1.2.x	1.x	>=1.2.3 <2.0.0
Pypi	~=1.2.1	~=1.2	<0.6.0,>=0.4.2
Rubygems	~> 1.30.11	~> 1.30	<0.6.0, >= 0.4.2

Table 5.3: Example Declarations

	total	fixed	soft	var-micro	var-minor	any	at-least	at-most	range	latest	not	other	unresolved	unclassified
Atom	215,433	17.69%	0%	18.53%	57.26%	1.76%	2.32%	0.1%	0.08%	0.93%	0%	1.26%	0%	0.07%
CPAN	2,406,593	0%	0%	0%	0%	63.14%	36.84%	0%	0%	0%	0%	0.01%	0%	0%
CRAN	277,856	0%	0%	0%	0%	80.41%	19.58%	0.01%	0%	0%	0%	0%	0%	0%
Cargo	350,862	2.92%	0%	72.86%	16.32%	6.37%	1.2%	0.02%	0.3%	0%	0%	0%	0%	0%
Dub	11,410	6.92%	0%	23.07%	2.15%	8.23%	37.09%	0%	13.94%	0%	0%	8.59%	0%	0.01%
Elm	16,450	0%	0%	0%	0%	0%	0%	0%	100%	0%	0%	0%	0%	0%
Haxelib	5,776	39.87%	0%	0%	0%	60.13%	0%	0%	0%	0%	0%	0%	0%	0%
Hex	50,227	7.24%	0%	36.81%	44.72%	0%	6.99%	0.02%	0.36%	0%	0%	3.86%	0%	0%
Homebrew	4,886	0%	0%	0%	0%	100%	0%	0%	0%	0%	0%	0%	0%	0%
Maven	3,592,035	0.03%	85.7%	0.05%	0.03%	0%	0.37%	0%	0.77%	0.01%	0%	0.01%	13.04%	0.01%
npm	52,886,593	16.48%	0%	21.61%	56.69%	2.92%	0.72%	0.01%	0.08%	0.8%	0%	0.58%	0%	0.1%
NuGet	3,097,666	6.91%	0%	0%	0%	0.01%	87.14%	0.02%	5.91%	0%	0%	0%	0%	0%
Packagist	4,178,062	11.41%	0.02%	14.21%	8.02%	7.9%	3.39%	0.11%	54.29%	0%	0%	0.66%	0%	0%
Pub	119,810	1.69%	0%	0%	0%	17.1%	5.92%	0.1%	73.61%	0%	0%	1.57%	0%	0%
Puppet	57,292	5.79%	0%	0.86%	3.11%	0%	56.64%	0.35%	33.04%	0%	0%	0%	0%	0.21%
Pypi	126,536	11.78%	0%	0%	0%	49.51%	33%	0.71%	4.97%	0%	0%	0%	0%	0.03%
Rubygems	4,487,068	4.59%	0%	14.87%	29.42%	0%	49.25%	0.23%	1.62%	0%	0.02%	0%	0%	0%

Table 5.4: Dependency Version Classification

	TOTAL	FIXED	FLEXIBLE	OTHER	SEMVER
Atom	215,433	17.69%	5.19%	1.33%	75.78%
CPAN	2,406,593	0%	99.99%	0.01%	0%
CRAN	277,856	0%	100%	0%	0%
Cargo	350,862	2.92%	7.89%	0%	89.19%
Dub	11,410	6.92%	59.26%	8.6%	25.21%
Elm	16,450	0%	100%	0%	0%
Haxelib	5,776	39.87%	60.13%	0%	0%
Hex	50,227	7.24%	7.37%	3.86%	81.52%
Homebrew	4,886	0%	100%	0%	0%
Maven	3,592,035	0.03%	86.85%	13.05%	0.07%
npm	52,886,593	16.48%	4.54%	0.68%	78.3%
NuGet	3,097,666	6.91%	93.09%	0%	0%
Packagist	4,178,062	11.41%	65.7%	0.66%	22.23%
Pub	119,810	1.69%	96.73%	1.57%	0%
Puppet	57,292	5.79%	90.03%	0.21%	3.97%
Pypi	126,536	11.78%	88.19%	0.03%	0%
Rubygems	4,487,068	4.59%	51.12%	0%	44.29%

Table 5.5: Aggregated Dependency Version Classification

	PROJ	ONE	AVG	STDEV	AVG1	STDEV1	AVGL	STDEVL
Cargo	11,251	3,236	6.13	9.56	3.85	3.54	4.86	4.39
Maven	63,497	16,952	9.96	23.23	5.03	6.3	5.3	7.01
CRAN	11,646	3,223	5.56	8.75	3.54	3.87	6.05	5.98
Pypi	4,083	935	8.76	14.87	2.71	2.85	3.15	3.25
CPAN	28,015	5,055	7.49	15.04	7.24	9.33	10.87	14.34
Elm	1,273	352	4.43	6.36	2.5	1.75	2.54	1.79
Homebrew	1,806	1,784	1.01	0.13	2.77	2.61	2.77	2.61
npm	547,338	153,412	7.34	22.52	8.75	16.16	9.76	15.78
Atom	3,845	600	11.18	22.11	2.92	3.79	4.08	5.43
Haxelib	470	188	6.06	9.84	1.8	1.26	1.89	1.31
NuGet	76,775	19,860	12.28	48.76	2.77	3.54	2.96	3.77
Dub	550	144	8.69	18.76	1.58	1.18	1.85	1.8
Packagist	104,585	28,340	7.59	16.48	3.37	3.97	4.04	4.6
Rubygems	119,942	35,671	6.41	15.35	4.19	3.41	4.89	3.95
Hex	3,667	1,248	5.4	8.01	2.14	1.56	2.33	1.7
Pub	2,867	688	9.06	17.72	3.08	2.45	3.95	3.34
Puppet	3,703	956	5.61	8.08	2.01	1.83	2.29	2.31

Table 5.6: Dependency Numbers At Start and End of Lifetime

*Project evolution by package manager (PROJ - project count, ONE - projects with only one version, AVG/STDEV - average / standard deviation of number of versions per project, AVG1 / STDEV1 - average / standard deviation of number of dependencies in first version, AVGL / STDL - average / standard deviation of number of dependencies in last version

	Projects	SEMV1	SEMV+	SEMV-	FLEX1	FLEX+	FLEX-
Cargo	11,251	6,981	751	30	7,857	111	16
Maven	63,497	324	23	238	43,133	5	5
CRAN	11,646	0	0	0	8,422	0	0
Pypi	4,083	0	0	0	2,908	81	34
CPAN	28,015	0	0	0	22,960	0	0
Elm	1,273	0	0	0	921	0	0
Homebrew	1,806	0	0	0	22	0	0
npm	547,338	336,963	10,793	5,680	370,102	5,786	5,208
Atom	3,845	2,698	151	51	2,928	90	37
Haxelib	470	0	0	0	219	20	23
NuGet	76,775	0	0	0	54,981	285	624
Dub	550	158	54	28	342	12	7
Packagist	104,585	27,295	6,140	4,860	71,314	2,593	541
Rubygems	119,942	54,001	5,503	3,200	83,207	474	263
Hex	3,667	2,096	80	50	2,213	56	9
Pub	2,867	0	0	0	2,052	24	7
Puppet	3,703	154	28	46	2,660	41	9

Table 5.7: Adaption of Semantic Versioning and Flexible Versioning

* Adaptation of flexible dependencies by package manager (PROJ - project count, SEMV1 / FLEX1 - projects using semantic versioning / flexible dependency syntax in first version, SEM+ / SEM- - projects adapting / dropping semantic dependency versioning syntax between first and last version, FLEX+ / FLEX- - projects adapting / dropping flexible dependency syntax between first and last version

Chapter 6

Developer Survey on Dependency Management

To complement the results of Chapters 4 and 5, which showed that ecosystems approach dependency management in vastly different ways, we created a survey to investigate how developers approach dependency management. This survey informs our understanding on how these trends form, and highlights decision making processes that developers in different ecosystems face when maintaining dependencies.

6.1 Survey Design

The survey was circulated by email during August 2018 to the authors' industry contacts, and often was further disseminated by the participating developers to others in their companies and beyond. It was also presented to professional developer groups, which prompted many of the developers present to participate.

6.1.1 Survey Participants

The 170 responses came from a broad range of locations predominantly across Europe, North America and Australasia. There was a broad range of experience levels, as seen in Figure 6.1.

We also asked participants which package managers they had used (results

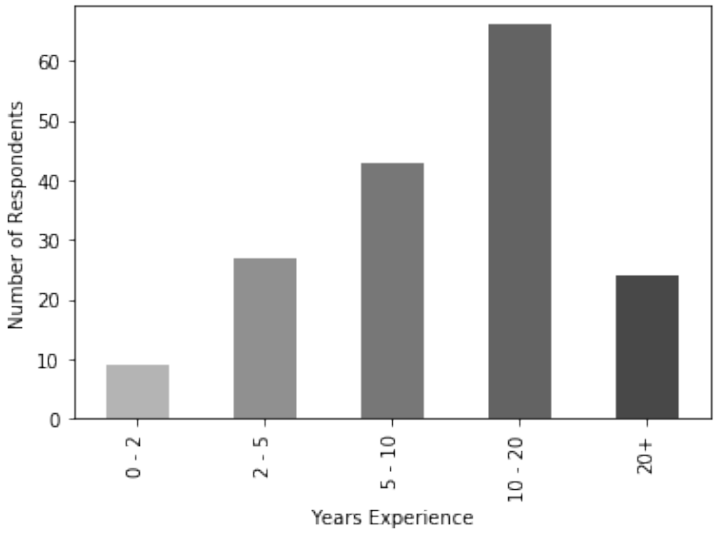

Figure 6.1: Developer Experience in Years

in Table 6.1), which shows good coverage of the package managers represented in the `libraries.io` dataset. There is only a single system which none of the participants use (Dub). Many respondents reported they use multiple package managers (86 use 3, 48 use 4, and 20 use 5), likely due to developers developing programmes in multiple ecosystems and programming languages.

6.1.2 Survey Design

In addition to the developer meta data discussed above, the survey then asked the following questions:

1. How familiar are you with Semantic Versioning? (On a scale of 1-5)
2. How do you declare dependencies to libraries?
   (a) Always using fixed declarations.
   (b) Always using version ranges.
   (c) Both depending on the context.
   (d) Adopting the styles of others (e.g. copy pasting).
3. Has your approach to declaring dependencies changed over time?

Package Manager	# Participants	Package Manager	# Participants
NPM	90	Atom	8
Pypi	56	CRAN	7
Homebrew	55	Elm	4
Maven	46	Puppet	4
Rubygems	29	Hex	2
Cargo	22	Haxelib	1
NuGet	22	Pub	1
CPAN	15	Dub	0
Packagist	11	Other	50

Table 6.1: Package Managers Used by Survey Participants

4. Do you use any tools to help you version your code?

5. Additional comments about your approach to dependency management.

Free form responses were provided to answer those questions in more detail where the respondent wished to elaborate. These proved valuable in gaining insight into the thought processes that underpin the decisions developers make when managing dependencies.

6.2 Survey Results

The developer survey, described in Section 6.1, provided further insight into how versioning strategies are used in practice. The majority of developers surveyed were familiar with semver, as seen in Figure 6.2, with 73% responding as either familiar or very familiar. Only 9% responded as being not familiar with it at all.

Interestingly, as shown in Figure 6.3, developers tended to self-identify as varying their dependency declaration strategy between fixed and range declarations depending on the situation (45%). 32% of developers always used fixed declarations, 11% always used ranges, 6% followed the styles of others (e.g. copy-pasting declarations from Maven repository), and 5% followed other strategies.

Table 6.2 breaks down the dependency management styles by package manager. An important point when reading this table is to note that most respon-

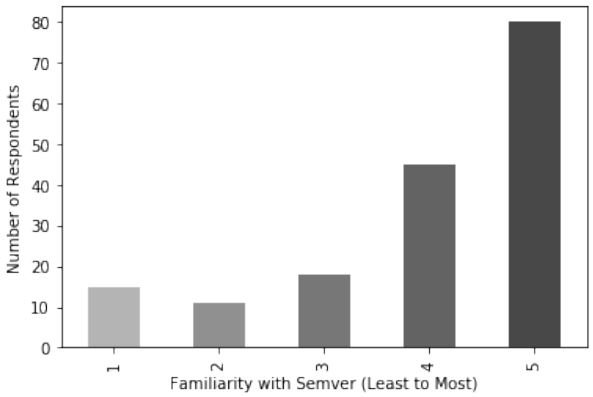

Figure 6.2: Semver Familiarity (Least to Most Familiar)

dents use multiple package managers. This has led to situations like Homebrew having a large proportion of fixed dependency responses despite syntactically only allowing ranged dependencies. The more common package managers most closely mirror the data in Table 5.5 - indicating that the use of multiple package managers is the reason for differences between these tables. It also implies that developers change their declaration style based on the conventions each package manager's community and syntax creates.

Further comments showed that sometimes developers vary their strategy at a library level, for example if some libraries are perceived as better at maintaining backwards compatibility than others, but also that developers change their strategies between types of projects, such as commercial vs. open-source projects. This result contrasts with the analysed dependency declarations in Chapter 5's Tables 5.4 and 5.5, which show most package managers having one main declaration style used by convention.

Developers were also asked if they have changed their approach to declaring dependencies. In this survey, 42% of respondents had changed their method of dependency declarations. However, when analysing the direction of change, no clear shift to or from semver was discernable. This closely mirrors the results discussed in Section 5.4.

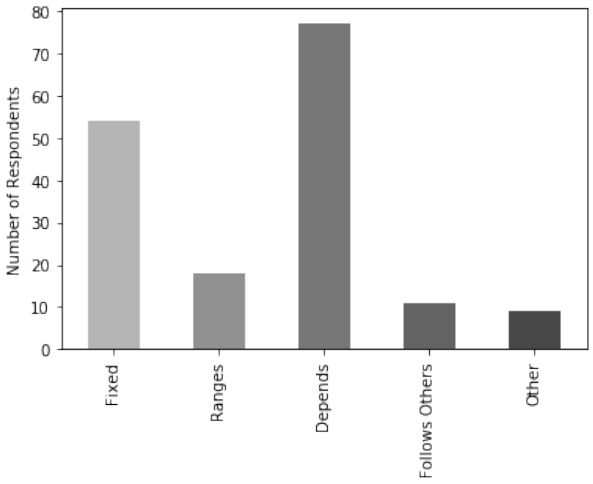

Figure 6.3: Self-Described Dependency Declaration Styles

6.3 Developer Perspective

Interestingly, the number of survey respondents who report to have changed their dependency declaration approach was rather high (42%). This seems to contradict the results from Chapter 5, although not the study by Decan and Mens (Section 3.5.1). We attribute this to the nature of survey participants – we believe that most participants had an above average level of experience and interest in managing dependencies.

In general, a number of respondents noted that they cared more about dependency management as their experience grew. Although changes towards and away from semver-compliant declarations were reported, changes towards the use of fixed versions were slightly more common[1].

Changes Towards Fixed Declarations

Developers who reported moving away from flexible dependency versioning commonly cited concerns about build stability and the introduction of compatibility-related bugs. The following quotes are illustrative of these concerns[2]:

[1] In the survey, we have only asked "Has your approach to declaring dependencies changed over time?", and respondents had the option to provide additional text to elaborate, so the above statement is based on a qualitative analysis of the comments.

[2] Each quote is followed by some summary data about the respective respondent: years of experience, package managers used, and level of familiarity with semver

Package	Fixed	Ranges	Depends	Follows	Other	# Responses
Atom	29%	0%	71%	0%	0%	7
CPAN	20%	7%	47%	20%	7%	15
CRAN	67%	17%	17%	0%	0%	6
Cargo	26%	4%	57%	4%	9%	23
Elm	25%	25%	25%	0%	25%	4
Hex	0%	0%	100%	0%	0%	2
Homebrew	38%	6%	45%	3%	8%	64
Maven	44%	0%	45%	8%	1%	71
NPM	26%	9%	54%	6%	5%	104
NuGet	21%	4%	64%	4%	4%	28
Packagist	17%	33%	42%	0%	8%	12
Pub	0%	0%	100%	0%	0%	1
Pypi	31%	5%	57%	3%	3%	58
Rubygems	18%	6%	71%	0%	3%	34
Puppet	0%	0%	75%	25%	0%	4

Table 6.2: Self-Declared Developer Declaration Styles by Package Manager

> "I have been burned too many times by so-called point releases on npm."
>
> [10 - 20 years experience, uses npm, NuGet, Pypi]

> "Taking end-to-end responsibility for software conception, development, and deployment requires predictable outcomes. If you do not use fixed versions, then rebuilding an artifact to resolve an issue identified during QA testing can cause unrelated changes that can manifest in production."
>
> [10 - 20 years, uses Homebrew, Maven, very familiar with semver]

> "You begin to realize how sloppy upstream people are, and the issues it causes, so you get a bit better about it."
>
> [10 - 20 years, uses Homebrew, npm, NuGet, Pypi, Rubygems, very familiar with semver]

> "I used to declare version ranges, then I realized that even ver-

sion ranges cannot capture the full 'compatibility range' of a dependency."

[10 - 20 years, uses npm, Nix, very familiar with semver]

"Packages not following semver caused issues with repeatable builds."

[10 - 20 years, uses npm, Nix, very familiar with semver]

"Updated versions of packages broke functionality when using ranged version, so [I] switched to fixed versioning to prevent this. We now manually update dependencies in order to roll back more easily."

[5 - 10 years, uses Homebrew, Maven, npm, Other, very familiar with semver]

"Version ranges were useful but some libraries do not practice semver very well."

[10 - 20 years, uses CPAN, Cargo, Maven, npm, Other, very familiar with semver]

"Usually due to package maintainers not conforming 100% to semver resulting in them shipping code that isn't compatible with the previous version. This is most noticeable on npm; if I pick up a client project that is older than six months its 50/50 wether (sic) npm will complete the install and even if it does its 50/50 that the tests will complete."

[10 - 20 years, uses npm, Composer, very familiar with semver]

"I changed from ranges to fixed versions, since I noticed libraries broke compatibility even when it shouldn't given their semver."

[5 - 10 years, uses Homebrew, Maven, Pypi, Other, very familiar with semver]

One developer explicitly stated mistrust for his package manager, a sensible sentiment given that greedy approaches to dependency resolution are used:

"This is a hard problem, [I] always choose fixed to prevent the package manager trying to be smarter than it is."

[5 - 10 years, uses Homebrew, Maven, 5]

Different Approaches Depending on the Ecosystem

Some experienced developers report differences between package managers, and adopt a horses for courses approach:

> "As much as Python packages claim to implement semver, they always find a way to break something in a minor release. No issues with Rust though."
>
> [5 - 10 years, uses Cargo, Pypi, very familiar with semver]

> "Due to package maintainers not conforming 100% to semver resulting in them shipping code that isn't compatible with the previous version[, I have changed to restrictive declarations]. This is most noticeable on npm. "
>
> [10 - 20 years, uses npm, Packagist, very familiar with semver]

This is sometimes also extended to the usage context, with a more conservative approach taken in commercial projects, or from organisations imposing rules:

> "In serious production code, always fixed version... in open source code, more relaxed."
>
> [20+ years, uses Homebrew, Maven, npm, Rubygems, very familiar with semver]

> "At work, some packages we are told to keep under a certain range because it is more supported."
>
> [2 - 5 years, uses Homebrew, Maven, npm, not familiar with semver]

Changes Towards Flexible Declarations

There were also cases where developers reported their rationale for changing towards flexible dependency versioning practices:

> "Because while exact versions give you predictability, they're difficult to keep up to date in manually when you have a lot of dependencies (particularly with pip; pipenv improves on that)."
>
> [10 - 20 years, uses npm, Pypi, very familiar with semver]

> "Used to be fixed, but I wanted to keep up to date and using version ranges helped to do that automatically."
>
> [5 - 10 years, uses npm, NuGet, Pypi, Others]

> "At the start, it felt the easiest to just use a library and keep the fixed version. However, it ended up being quite limiting (especially when there's 'so many cool new features' that I couldn't use). I therefore prefer accepting a certain range of versions to keep the software 'fresh' for a longer time."
>
> [5 - 10 years, uses CRAN, NuGet, not familiar with semver]

> "I now prefer declaring patch versions over just having a fixed value. Have yet to be burnt by a patch version update, but the ideals of keeping up where possible is a good place to be."
>
> [10 - 20 years, uses Homebrew, Maven, npm, Rubygems, familiar with semver

> "Frequently updates on the minor version of libraries have some major bug fixes. This requires frequent change in library versions. Declaring them using ranges makes this easy."
>
> [10 - 20 years, uses Homebrew, Maven, npm, NuGet, Other, not familiar with semver]

Tooling Supporting Flexible Declarations

Some developers have moved to more permissive declarations as they have adopted stronger tooling such as testing, CI and lockfiles:

> "I used to be very careful with specifying ranges (e.g. 3.3.x only, not 3.x) but from my experience, any type of package change is prone to changing behaviour or breaking things. I now rely on testing (automated and QA) to catch any changes when updating packages."
>
> [10 - 20 years, uses npm, Composer, Rubygems, very familiar with semver]

> "First, realizing that without fixed versions things can break, second change was when I started using Cargo which introduced lockfiles."
>
> [10 - 20 years experience, uses Cargo, Pypi, very familiar with semver]

Legacy Code Dependency Management Issues

A couple of developers noted how not being specific enough with declarations have hurt them in legacy code maintenance:

> "For Haskell (Cabal) I used to not declare versions very often, e.g. just use 'some-library'. After coming back to some projects after a few years, and experiencing how difficult it is to figure out which versions it was actually built for, I now tend to declare minimum versions, e.g. 'some-library >= 3'. I usually don't force upper bounds, or specific versions, unless I encounter an actual breakage."
>
> [10 - 20 years, uses Pypi, Other, 5]

> "The more I interact with old projects, the more I see the need to be more explicit."
>
> [10 - 20 years, uses NuGet, Pypi, Other, very familiar with semver]

Minimising Dependency Numbers

One developer noted about minimising dependency numbers so that security vulnerabilities could be more actively monitored, a strategy discussed in Sections 3.3 and 3.5.3. Another does the same just to keep on top of dependency issues in general.

> "Mainly looking at the security checks of older dependencies and figuring out how to minimize the amount of dependencies so that the dependencies that are used can be scrutinized for security vulnerabilities along with using the latest versions (typically those two things correlate)."
>
> [2 - 5 years, uses Homebrew, Maven, Other, very familiar with semver]

> "I usually keep an eye on the packages I use and update versions when I see it to be advantageous - e.g bug fixes, security patches, new features that are useful. To this point I try to keep the number of dependencies down so the administration of them is less of a concern. Both npm and composer have tools that list the packages you have installed and whether there are any updates available; as

part of our CI system we have a task that runs those and raises a warning when dependencies are getting 'out-of-date'."

[10 - 20 years, uses npm, Composer, very familiar with semver]

Updating Patterns

Some developers noted the need to keep dependencies up to date, citing similar issues to those discussed in literature from Sections 3.3 and 3.5.

> "I try very hard to keep all dependencies up to date with the latest release version, as I've found this is the surest way to ensure compatibility. Falling behind can be very expensive in terms of effort when an upgrade becomes necessary. Most small open source projects do not do security releases for older versions."
>
> [10 - 20 years, uses Homebrew, Maven, very familiar with semver]

> "Versioning libs is a pain. The best approach is to try to keep up to date. Avoid bit rot and avoid the same infrastructure requiring different versions."
>
> [5 - 10 years, uses CPAN, npm, somewhat familiar with semver]

Miscellaneous

Several respondents pointed to using a package manager called *nix*, which allows npm-style isolated dependency resolution for Linux packages. There were also a large number of responses that mentioned package locks in npm and Rust as giving them the confidence of reproducibility when moving to flexible declarations or in the face of breaking updates.

Finally, one developer makes the same conjecture as this study - semver is first and foremost a social construct, where trust is required in order for it to succeed:

> "Semver is more of a social contract than anything technically precise. In Rubygems and npm, you never really know what a minor or patch version is — it's down to the author/publisher who often have their own idiosyncratic ideas about what to communicate through version numbers. Some claim to use semver, but you still sometimes

have to look at changelogs and code diffs to really understand the impact of a change on the way you use the code.... ”

[10 - 20 years, uses Atom, Homebrew, npm, Pub, Rubygems, very familiar with semver]

6.4 Summary

This chapter described the results of a dependency management survey sent out to developers. It gathered information from 170 developers across a broad geographical area and wide range of experience levels.

This survey showed that most developers surveyed were very familiar with semver, with less than 10% having no familiarity with it. When asked what style of declaration patterns they used, almost half responded that they switched between fixed and flexible declarations as the situation called for it. One third always used fixed declarations. Very few always used flexible declarations (11%) or copy-pasted / followed the style of other developers (6%).

Almost half of developers have changed their approach to dependency declarations over time. There are examples of developers going both more towards fixed and flexible styles, with anecdotal evidence of most of the advantages and disadvantages discussed in this work being shared by developers. Broadly speaking, developers want automated updates but note that upstream developers do not consistently follow semver standards. Many of the more experienced developers stress the need for reproducible builds, and have either moved towards fixed declarations or using lockfiles to achieve this. The free form responses give a wide agreement - industry developers want to use semver, but trust between developers is problematic, and better tooling is required to allow the system to function.

6.5 Conclusion

This survey was conceived after the discovery in Chapter 4 that npm and Maven had such different approaches to declaration patterns. Both Chapter 4 and the follow on study in Chapter 5 highlighted the heterogeneity of declaration styles across ecosystems. Given the empirical data, we expected to find the responses in this survey to differ based on the ecosystem the developer comes from. In the

responses, we only found hints of this - certainly not enough to fully explain the empirical data. However, many respondents used multiple package managers, so it is possible that their style in each differs and follows the *wisdom of the crowd*. This idea is supported in part by the qualitative analysis conducted, as well as the responses summarised in Figure 6.3, where almost half of developers self-described their declaration style as varying based on the context.

This survey highlights that developers understand many of the challenges involved with dependency management, and that, these challenges present regular issues for developers. The responses also present further validation to existing literature - the responses for updating patterns, minimising the quantity of dependency, and upstream developers violating semver patterns were all discussed in Chapter 3. Quite specifically to our work, the qualitative analysis mirrors the Chapter 5 finding that developers move both from flexible to fixed declarations, and vice versa, with no clear bias towards either direction - both options currently have significant drawbacks. This survey gives an insight into the thought processes of developers, supplementing the empirical data in Chapters 4, 5 and, later in Chapter 7. Overall, it comes to a similar conclusion about the state of semver as the empirical chapters - semver is a good idea, but one that is not fully realised and is currently plagued with problems.

Chapter 7

A Large-Scale Study on Technical Lag and Update Patterns

This large-scale study on technical lag in open source projects analyses how often dependencies are kept up-to-date and, when lagging, how far outdated dependencies get. It uses open-source projects from 14 package managers (see Section 5.2.2) using the *libraries.io* dataset (see Section 5.2.1). We find that, in the projects analysed, the majority of fixed declarations, along with a significant number of flexible declarations, are outdated. When fixed declarations become outdated, this study considers how developers respond - do they update the dependencies or let them fall further out of date, and are they more likely to make smaller updates or wait until larger updates become available (e.g. a micro update, or a major update as discussed in Section 7.2.5)?

7.1 Introduction

Most modern software systems are built from existing packages (e.g. modules, components, libraries - henceforth termed *dependencies*) that allow complex functionality to be delivered easily. These libraries are generally created by third-party developers and are linked to a project via a symbolic dependency

declaration resolved by modern package managers, such as *Maven* for JVM languages, *Cargo* for Rust, or *npm* for JavaScript. These package managers allow dependencies to be downloaded from a remote repository at build time by declaring constraints which describe the versions of a dependency compatible with the project, giving significant flexibility and agility to the dependency management process compared to the ad-hoc methods that preceded this system.

Dependencies are generally active projects which publish updates intermittently, so it is easy for dependency declarations to become outdated over the life cycle of a project if the version requested by the package manager does not get updated regularly to the latest available version of the dependency. Keeping dependencies up to date can cause a significant overhead on a developer's time (particularly if it is not automated), but it is important for the security and overall health of the project [Lauinger et al., 2018, Cox et al., 2015, Pashchenko et al., 2018, Salza et al., 2018].

While flexible declarations are widely used, a significant number of projects still employ fixed version dependencies [Dietrich et al., 2019] - this was discussed in detail in Chapter 5. This has the advantage of greater control - dependencies only get updated after being integration-tested with the rest of the project. The downside of fixed version declarations is that it makes the updating process less agile. It requires developers to manually find updates and change the relevant configuration files by hand, which can be really hard and time consuming especially in the presence of large number of dependencies in modern systems. When manually updating dependencies, they are often not updated immediately, potentially resulting in *technical lag* [Gonzalez-Barahona et al., 2017], where newer versions - with their bug fixes, security fixes and other improvements - are available but not used. This technical lag potentially has a high associated cost associated with it in the form of security vulnerabilities, bugfixes and missed improvements [Kula et al., 2015, Lauinger et al., 2018, Cox et al., 2015, Derr et al., 2017]. See Sections 3.5.2 and 3.5.3 for more details about recent technical lag literature.

There have been some studies on technical lag and updating strategies in recent literature, however they have all focused on one specific package manager (e.g. npm or Maven), making them hard to generalise across the industry. This study builds on previous studies such as [Raemaekers et al., 2014, Zerouali et al., 2018] highlighted in Section 3.5.2 by quantifying lag across pack-

age managers in the first large-scale, cross-ecosystem study on lag that we are aware of. Our goal is to describe how much technical lag exists across open-source projects in general, and how developers update declarations to manage technical lag. In the process, we consider practical techniques of how technical lag might be reduced, and how effective they could be.

This chapter studies, in detail, this technical lag, and more specifically, sets out to answer the following five research questions:

- **RQ1**: How often do dependencies lag?
- **RQ2**: How much lag is there in dependencies when they are not up-to-date?

We then focus on how developers update fixed version declarations, by studying the following questions:

- **RQ3**: What type of updates are most common, and what type of project releases contain dependency updates?
- **RQ4**: How often do developers perform updates when they lag behind, and do those updates bring them up to date?
- **RQ5**: How often do developers make a backwards change to their dependencies, and why?

7.2 Methodology

This chapter uses the `v1.4.0` (December 2018) dump of the *libraries.io* dataset [Katz, 2018], containing over 2.7 million unique open source projects from 37 package managers and 235 million dependencies between projects, introduced in Section 5.2.1. The way that this dataset is structured means that not only can we take each declaration individually and classify it, but also take a longitudinal approach to project timelines, to gather how projects change over subsequent versions of a project. This makes it suitable for the declaration update research questions posed in RQ3 - RQ5.

7.2.1 Declaration Classifications

In this chapter, we focus on a subset of the declaration classifications discussed in Section 5.2.3. For reference, Table 7.1 demonstrates the main types of de-

pendency version declarations that we focused on in this study. All declarations that allow for more than one version are termed *flexible*, while declarations that only allow one version are termed *fixed*. We further break down the flexible declarations into semver-compliant ranges, *micro* and *minor*, and other ranges listed in the table. In principle, the *at-least* and *any* (*open ranges*) classifications should not suffer from technical lag[1], but risk compatibility issues due to allowing major updates which are expected to include breaking changes.

Classification	Explanantion	Example
Fixed	One specific version	`1.3.0`
Micro	Micro versions from min	`[1.3.2, 1.4.0)`
Minor	Micro or minor versions from min	`[1.3.2, 2.0.0)`
At-Most	Any versions up to a limit	`[0.0.1, 1.3.2]`
At-Least	Any versions from min	`[1.3.2, ∞)`
Any	Any versions	`[0.0.1, ∞)`
Range	Any custom range	`[0.3.2, 0.7.1]`

Table 7.1: Dependency Declaration Types

7.2.2 Parsing Declarations to Satisfies Set S

The parsing strategy discussed in Section 5.2.4 is enough to be able to categorise the type of declarations, but it lacks the ability to tell if the declaration is satisfied by a given version. This is a fundamental step in ascertaining the existence and quantity of lag within a given declaration. Without this abstraction, this chapter would be limited to fixed declarations, without being generalised to other declaration categories.

In order to overcome this problem, a second abstraction for parsing declarations was developed, which created a set S that encompasses the versions that could satisfy the declaration. This was done programmatically, producing *Declaration* objects (henceforth, the italicised *declaration* will refer to the object derived from the declaration string) that consisted of closed ranges, where

[1]We note that technical lag is still possible, depending on the package resolution strategy used by the respective package manager, as discussed in Section 2.2 - for example NuGet resolves to the oldest possible version that satisfies the declaration, not the newest one.

any version within that closed range would match, allowing for a method inside *Declaration* with the method signature shown in Equation 7.1. In other words, given a *declaration*, we can tell if a *Version* object (an abstraction of a semantic version as described in Section 2.1) satisfies this declaration.

```
fun matches(version: Version): Boolean
```
(7.1)

This *declaration* needed to be general enough to work with any arbitrary declaration mentioned in Sections 5.2.3 and 5.2.4. As it is possible for composite declarations to be used (e.g. Equation 7.2), two linked lists were used internally, connecting the *declaration* to 0 or more other *declarations*, one via an `AND` operator, the other via an `OR` operator. Two other details were subsequently added, one to deal with *not* classifications, and the other to deal with cases that only allowed for pre-releases in a specific version (this is a requirement in Rubygems when ranges are specified along with a starting version that contains a pre-release tag). This reflected an operator tree approach, which mirrors how declaration strings are written, and therefore virtually all declaration strings could be parsed to a *declaration*.

$$S = \{x \mid (1.0.0 \leq x < 1.1.0) \vee (1.2.0 \leq x < 1.3.0)\} \tag{7.2}$$

This abstraction can be used to categorise how declaration ranges have changed (see Section 7.10). However, it is possible that the operator tree approach used could flag two declarations expressing the same version intervals but expressed in different syntactic methods as being different if the approach is not careful. The best way to manage this may be to transform this operator tree object into a series of maximal disjoint intervals for comparison.

Parsing Method

Parsing the declaration strings to a *declaration* required a complete overhaul of the categorisation method described in Section 5.2.4. For each package manager, a class was created with a method `getDec` (Equation 7.3) to take a declaration string and the classification derived from Section 5.2.4, and return a *declaration*.

```
fun getDec(class: String, dec: String): Declaration
```
(7.3)

There is one class per package manager, each of which contains the logic required to convert a string declaration to a *declaration*, given the rules of that

package manager. In the case of some package managers, the classification was used as a 'hint' to route the declaration to specific logic. In other cases, the string was parsed without other assistance. The rules created in the categorisation parsing step were used as a guideline when creating the programmatic parsing logic, although it was still necessary to refer back to the documentation, particularly for how to correctly deal with composite declarations - this is not a detail needed for classification, but is important for creating a satisfies set S.

Not all declaration strings could be parsed. There were two instances of declarations (one Rubygems *at-least* classification, and one NuGet *fixed* classification) that could not be parsed. Additionally, a call was made in some package managers not to parse some types of patterns with complex syntax and limited numbers; the *other* classification in Dub, Maven and Pub, and the *unresolved* classification in Maven, which cannot be parsed due to being unresolved variables. These unsupported *other* patterns amounted to 8.5% of Dub, 1.6% of Pub[2] and 0.01% of Maven declarations.

Validation

Due to each package manager having its own set of logic that was independent to other package managers, it was important to test the programmatic parsing methods for each package manager. For this, test cases were chosen from the rule tests (such as in Figure 5.2) - these test cases were chosen because they represented both the standards for that package manager, as well as more exotic cases found in the dataset. Choosing test cases in this manner has increased our confidence that the created logic is valid for this dataset.

For each package manager, a file of test cases were created (see Figure 7.1) to test the class. Each category would have at least two declarations - often more where syntax could vary, and each declaration would be given versions that they would match or not match, testing boundary cases. In this manner, 50-100 test cases were created for most package managers, depending on the complexity of the syntax and number of possible classifications.

[2]Dub and Pub are very small parts of the dataset, so this only represented about a hundred declarations.

```
1  classification,declaration,test,result
2  fixed,^0.0.1,0.0.1,match
3  fixed,^0.0.1,0.0.2,no_match
4  var-micro,^0.1,0.1.2,match
5  var-micro,^0.1,0.2.2,no_match
6  var-minor,^1.1,1.5.2,match
7  var-minor,^1.1,2.2.2,no_match
```

Figure 7.1: Example Test File

7.2.3 Classification and Filtering Process

Due to requiring metadata about both the downstream and upstream project history to calculate lag and update values, which was not always available, not all pairs of downstream and upstream projects could be included. From all possible pairs of projects A and B, where A depends on B, pairs were classified into one of the following categories:

Fixed and Flexible

If a dependency declaration could be satisfied with more than one version of the dependency, it is a flexible dependency declaration. Any pairs with at least one flexible declaration are used for answering questions about lag, but have been excluded from the research questions RQ3 - RQ5 that focus on the updating strategies used for fixed declarations. To check for the classification of the declaration, the process in Section 5.2.4 was used. The *FIXED* and *FLEXI* columns in Table 7.2 show the number of pairs considered fixed or flexible, respectively.

Not Semver-Compliant Syntax

The *NOT_SV* column in Table 7.2 shows the number of dependency pairs filtered out due to semver violations. Projects with versions that could not be parsed to the semver format specified in Section 2.1 were discarded, as we could not rely on automated version ordering. Similarly, versions with unusually large numbers (with values over 100,000 - usually related to timestamps) were discarded. The

Packagist numbers are particularly high due to common shortcut commands such as `@dev` and `@stable` being filtered by this check[3].

Missing Project Information

Projects sometimes include dependencies to other projects which were not included in the dataset. This meant that information about their version history was not available, making them unsuitable for our study. The *MISSING* column in Table 7.2 shows the number of pairs filtered out due to lack of data - this is about 1% of the total set of dependencies considered.

Groups of Packages with Coordinated Releases

Often, the developers working on project A also work on its dependency B. This is most regularly associated with the two projects being components of a single overarching project. Often these dependencies are updated as part of the internal release procedures (e.g. coordinated product release, Apache Lucene is an example of such coordinated releases[4]). Including them would be highly likely to lead to under-reporting the level of technical lag in independent projects. The *SUBCMP* column in Table 7.2 shows the number of pairs filtered out due to being suspected subcomponents. For the purpose of this study, pairs are considered subcomponents of a wider project if either the first half of their names (minimum 4 characters) are the same, or their entire project names are the same. To ensure that this only captures subcomponent pairs, a manual validation on the first 300 pairs per package manager was conducted by the authors and cross-validated the results by a collaborator - we found a low false positive rate ($< 1\%$) from the validation, having a confidence interval of less than $\pm 6\%$, with a 95% confidence level.

There were initially 17 package managers with dependencies in the dataset, listed in Section 5.2.2. Three package managers (CPAN, CRAN, Homebrew) were ruled out of scope as their declaration patterns are entirely open range (the *any* and *at-least* classifications) which means they cannot have technical lag in the sense studied here (see Section 7.2.1). Table 7.2 shows how the pairs in the remaining 14 package managers were classified. These package managers were used to answer the questions about how much technical lag is present in

[3]These declarations are considered unresolved variables.

[4]https://mvnrepository.com/artifact/org.apache.lucene

PM	FIXED	FLEXI	NOT_SV	SUBCMP	MISSING
Atom	2276	17565	2	642	147
Cargo	1237	81543	90	4030	42
Dub	43	999	0	109	128
Elm	0	3988	0	253	0
Haxelib	192	568	0	266	5
Hex	698	10531	0	592	88
Maven	389044	23860	7358	106631	47015
npm	876534	7449356	2254	335976	19130
NuGet	2010	245725	738	64038	13568
Packagist	19250	474455	19784	70975	19886
Pub	157	16142	0	809	672
Puppet	204	7210	0	1800	242
Pypi	3300	25418	31	1185	569
Rubygems	14940	606994	93	21931	1586

Table 7.2: Project Pairs Included by Package Manager

dependencies. However, only 9 of the 17 package managers (from the most fixed pairs to the least - npm, Maven, Packagist, Rubygems, Pypi, Atom, NuGet, Cargo and Hex) were included in answering the questions based around update strategies, as the remaining package managers have too few fixed declaration pairs to compare results against other package managers in a meaningful way.

7.2.4 Quantifying Technical Lag

For each project, the versions are ordered according to semver principles (discussed in Section 2.1). In order to quantify technical lag, we work with a timeline of two projects. For every version of project A (the client project), A_i, that has a dependency to a version of project B, *Bdep*, there is technical lag present if *Bdep* is not the latest version of project B. We measure technical lag in two ways, the number of versions behind (*version lag*) and the time behind (*time lag*). Figure 7.2 provides a working example of how technical lag is quantified. **Version lag** is formalised below, where *Bnewer* is any dependency version

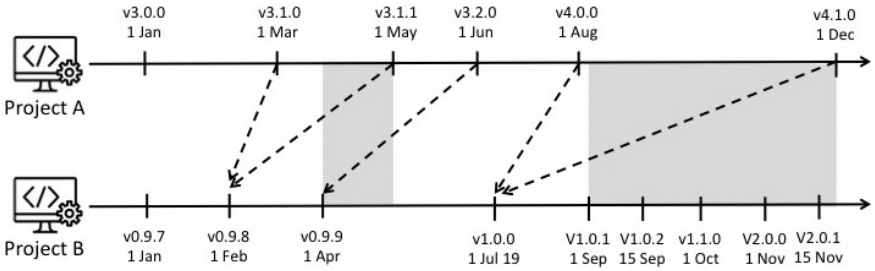

Figure 7.2: Quantifying Technical Lag

The arrows indicate the dependencies. As depicted, technical lag (grey shaded regions) occurs when Project A depends on a *Bdep* that is not the latest version of B available at the time A_i was released. Figure courtesy of Dr. Kelly Blincoe.

with a higher version numbers than *Bdep*, and B_{major}, B_{minor}, and B_{micro} are the major, minor or micro numbers extracted out of the overall version number (e.g. $Bnewer = 1.2.3$, then $Bnewer_{major} = 1$, $Bnewer_{minor} = 2$, and $Bnewer_{micro} = 3$):

$$MAJOR\_LAG = count(\{Bnewer_{major} \mid Bnewer_{major} > Bdep_{major}\})$$

$$MINOR\_LAG = count(\{Bnewer_{minor} \mid Bnewer_{minor} > Bdep_{minor} \wedge Bnewer_{major} = Bdep_{major}\})$$

$$MICRO\_LAG = count(\{Bnewer_{micro} \mid Bnewer_{micro} > Bdep_{micro} \wedge Bnewer_{major} = Bdep_{major} \wedge Bnewer_{minor} = Bdep_{minor}\})$$

$$A_i\_TOTAL\_LAG = (MAJOR\_LAG, MINOR\_LAG, MICRO\_LAG)$$

Major lag is calculated as the distinct major versions later than *Bdep*. *Minor lag* is calculated as the distinct minor versions later than *Bdep within the same major range*, and *micro lag* is calculated as the distinct micro versions later than *Bdep within the same minor range*. This means that there could be a great deal more newer minor and micro versions than reported, but within other major or minor version ranges. Limiting micro and minor lag to the these narrower version ranges allows direct comparisons to be made with semver compliant micro and minor ranges - the most common alternative declarations to fixed declarations [Dietrich et al., 2019]. Total lag is defined as the major lag, minor lag and micro lag, following the semver idea that any major lag is

more significant than all minor lag, and any minor lag is more significant than all micro lag.

In Figure 7.2, project A `4.1.0` depends on `1.0.0` of project B. Since this was not the latest version of project B at the time of release of project A `4.1.0`, there is technical lag. Both the major and minor lag are 1 since there was one later major release and one later minor release available when `4.1.0` was released. The micro lag is 2 since there were two later micro releases available within the `1.0` minor release. Note that there is also a micro release `2.0.1` available, but since it is not within the `1.0` release, it is not considered since most flexible versioning strategies would not automatically update to this micro release.

Time lag is calculated as the number of days before the release of A_i that *Bdep*'s first subsequent update was available. Time lag is also calculated in three parts: *major*, *minor*, and *micro*. As mentioned above, `4.1.0` has major, minor, and micro lag in Figure 7.2. Since this release was made on 1 December, the time lags are:

- 91 day micro lag since `1.0.1` was released on the 1st of September.
- 61 day minor lag since `1.1.0` was released on the 1st of October.
- 30 day major lag since `2.0.0` was released on the 1st of November.

Note that the release of `1.0.2` on the 15th of September does not impact the micro time lag since this is not the first micro update after *Bdep*. The dependency began to be out of date the moment `1.0.1` was released, so the time lag is calculated based on this release date.

Lag is counted for every version A_i which has a *Bdep*. Therefore, project pairs with long version histories of dependencies are represented multiple times in the aggregated results. This event-based method, focused on counting lag at each version of A, follows previous studies on technical lag [Zerouali et al., 2018, Zerouali et al., 2019a, Zerouali et al., 2019b, Raemaekers et al., 2014].

This study is based around the use of semantic versions (Section 2.1), and in particular major, minor and micro versions. Tag updates, considered pre-release by semver principles, and further sub-micro numbers are not considered as relevant changes when counting technical lag (due to varying semantics of these sub-classifications, and the overriding convention that these are very small

changes). However, as dependencies are sometimes updated within tag or sub-micro releases, they are still included in the analysis as *no-change* updates of a project or dependency.

7.2.5 Update Classification

For RQ3-5, we look at how developers update fixed declarations. When a declaration is fixed, updates could be classified in seven ways, shown in Table 7.3. When looking at updates in more detail, the interesting part is to understand how much the downstream developer had to change. While it is impossible to say for certain how widespread the changes were without closely inspecting the diffs between published versions, semver standards give us hints in the form of the major, minor and micro update patterns. If a declaration is changed to a newer major version, we can infer that the effect on the dependent project (in terms of benefits - additional and hopefully better functionality, as well as costs - time spent refactoring and ensuring integration) will be larger than a minor change, and that the effects of a minor change will be larger than a micro change.

FORWARDS_MAJOR	Matches a later major version
FORWARDS_MINOR	Matches a later minor version in the same major range
FORWARDS_MICRO	Matches a later micro version in the same minor range
NO_CHANGE	Dependency stays the same or matches a different pre-release tag
BACKWARDS_MICRO	Matches an earlier micro version in the same minor range
BACKWARDS_MINOR	Matches an earlier minor version in the same major range
BACKWARDS_MAJOR	Matches an earlier major version

Table 7.3: Update Categories for Fixed Declarations

Between any two versions, A_i and A_{i+1}, if there exists a declaration to project B in both, the declarations are compared. This results in either classifying the declaration change as no change, a forwards change, or a backwards

change. The forwards and backwards changes are further split into major (if the major version changed), minor (if the minor version changed but not the major version), or micro (if only the micro version changed). In the case that one of the two contiguous versions did not have a declaration, no update data is recorded.

In cases where A_i is before A_{i+1} in both version and chronological date, the results from the preceding comparison make sense. This is the case for any two versions in the same micro range, due to the semver practice of always incrementing numbers by one. However, at the border between minor and major ranges (e.g. `3.5.2` and `3.6.0`), it is possible that A_{i+1} has been published before A_i, a practice called *parallel development*. Where A_i is a lower version than A_{i+1}, but published later, the update result is not always sensible.

To illustrate why the chronological order is important, suppose a given project A has versions $S = \{\dots, 3.5.2, 3.5.3, 3.6.0, 3.6.1, \dots\}$. `3.6.0` was published after `3.5.2`, but before `3.5.3`. After `3.6.0` was published, a security hole was found in a dependency, so it was upgraded to a newer version that fixed the vulnerability and released as `3.5.3`, to support downstream projects using the `3.5.x` micro range, and `3.6.1` for those using the `3.6.x` micro range. Now, comparing `3.5.2` with `3.5.3` will show a forwards update to the dependency, while comparing `3.5.3` with `3.6.0` will show a backwards update to the dependency. Logically, it does not make sense to categorise `3.6.0` as a backwards update, when there was no change between it and `3.5.2`, the version chronologically preceding it. To account for this case, where A_{i+1} is published before A_i, the version *chronologically* preceding A_{i+1} is selected instead.

7.2.6 Validation

There have been two main processes used to transform the data in the *libraries.io* dataset for analysis. For RQ1 - RQ2, *declaration* objects were used. The parsing and validation method used to convert these objects from strings are shown in Section 7.2.2.

For RQ3 - RQ5, validation was done in an iterative manner. The entire state of 10 random pairs from each package manager were manually checked for data validity, ordering and categorisation by the author. Where there were issues, tests were created before fixing the code. Then, another 10 pairs were reviewed, until there were no further issues found. Rare cases such as pairs updating to an

older dependency version, or outlier lag values have been considered separately by the same process.

As noted in Section 7.2.5, backward updates had additional complications due to simultaneous development. To validate these heuristics, we manually analysed a random sample of 95 pairs with backward changes, looking for false positives where our heuristics identified a backward change but one did not occur. This sample size gives us a confidence interval of 95% with a 10% margin of error. The manual analysis was done by the author and a collaborator. For each pair of a backward change, a search was done of the commit history in the related repositories to identify whether it is actually a true backward change. We excluded samples where repository information is not publicly available.

Of the 95 manually analysed backward changes identified through our heuristics, 5 or 5.3% were false positives (meaning no backward change had occurred). There were two reasons for the false positives. In one case, the dependent project had changed its dependency strategy from a date based versioning system to semver. Thus, the dependency changed from `2015.03.12` to `1.4.6`, which was incorrectly identified as a backward change by our heuristics. The other four false positives occurred due to the projects performing parallel release development. In some of these cases, the projects appeared to explicitly maintain a parallel release to continue to offer support for a prior version of a dependency. For example, one project released `v0.7.0` with a dependency on project B `v2.15.0`. Shortly after that release, another release was made called `v0.7.0-projB-2.12` which depended on `v2.12.4` of project B. Our heuristics incorrectly classified these as backward changes since the release of the parallel version depending on the older version of project B occurred after the release which was updated to the new version of project B.

Since our heuristics were able to correctly classify 90 of the 95 (94.7%) backward changes, and the other cases could not be easily identified automatically, we used these heuristics to identify backward changes for the remaining analysis.

7.2.7 Identifying reasons for backward changes

To understand the reasons why developers make backwards changes to dependencies, we used the random sample of 95 backward changes identified by the

heuristics.[5]

For each backward change, a search was done in the project repositories to look for the related commits, any associated issues or discussions, and release notes to look for mentions of reasons why the backward change was being made. We were able to find information on the reason behind the change for 66 of the backward changes. We used Thematic Content Analysis [Braun and Clarke, 2006] to identify the main themes from these reasons. Three of the authors jointly discussed the reasons to develop the themes.

[5]Please note that Section 7.2.7 is the work of Dr. Amjed Tahir from Massey University and Dr. Kelly Blincoe from the University of Auckland, which was conducted to complement the empirical investigation for RQ5.

7.3 Lagging Dependencies

RQ1: How often do dependencies lag?

7.3.1 Which Declaration Types Lag Most?

Table 7.4 shows the proportion of declarations that lag in each package manager, based on classifications discussed in Section 7.2.1. It was found that the more restrictive the declaration is, the more likely it is to lag. Fixed declarations are more likely to lag than micro ranges, which are more likely to lag than minor ranges. At-most declarations tend to lag even more than fixed versions. Open range declarations have been excluded from this table, as by definition, they are not expected to lag.

In most package managers, ranges form a small minority of the overall number of declarations - often a range differs from micro or minor ranges to cover a set of versions that do not have a specific semver meaning. However, in Elm, Packagist and Pub, ranges are the most common way to set up flexible declarations, even where micro or minor ranges are intended. Perhaps due in part to these factors, there is little correlation between range declarations and the likelihood of technical lag across package managers.

Across most package managers, minor range declarations have a low probability of being outdated, with npm at a high of 26% to Cargo with a low of 3% lagging minor range declarations. The outlier here is Rubygems in which a staggering 61% of its minor ranges are lagging. About 30% of Rubygems declarations are minor ranges, so this is a widespread issue in Rubygems projects where semver compliant declarations are not being kept up to date.

Micro range declarations tend to be between 10-20 percentage points less current than minor range declarations, and there is about the same difference between micro ranges and fixed declarations. In general, fixed versions are more often out of date than they are current.

Some package managers are much more likely to include out of date dependencies. This is especially true for Maven (63% lagging), which both primarily uses fixed declarations, and those fixed declarations lag more than average. Rubygems (54%), NuGet (49%) and Pypi (46%) are relatively outdated as well, but in all three, the most common declarations are any or at-least, so overall, dependencies are more current than suggested in Table 7.4. In the other

PM	Fixed	Micro	Minor	At-Most	Range	Overall
Atom	49.9%	40.7%	17.4%	34.0%	17.4%	28.3%
Cargo	36.3%	13.6%	2.8%	51.0%	19.8%	11.7%
Dub	29.9%	13.1%	10.3%	-	12.0%	15.4%
Elm	-	-	-	-	18.1%	18.1%
Haxelib	8.1%	-	-	-	-	8.1%
Hex	34.6%	21.5%	9.8%	50.0%	11.0%	16.2%
Maven	63.2%	29.2%	7.8%	75.0%	14.8%	63.0%
npm	51.6%	34.6%	26.5%	47.2%	38.6%	32.2%
NuGet	39.2%	-	-	69.8%	54.0%	49.4%
Packagist	71.7%	47.7%	23.1%	80.0%	35.9%	31.9%
Pub	57.5%	-	-	11.0%	18.4%	19.0%
Puppet	33.6%	52.7%	8.5%	47.5%	13.0%	15.7%
Pypi	51.5%	-	-	69.8%	23.5%	45.8%
Rubygems	56.6%	39.5%	61.0%	72.0%	22.0%	53.5%

Table 7.4: Percentage of Dependencies that Lag
*This excludes declarations that cannot lag, such as *'at-least'* or *'any'*

packages, less than one third of non-open-range dependencies lag.

7.3.2 Most Common Types of Lag

In addition to considering how dependencies lag based on the type of declaration used, we also considered what type of lag is present in the dependencies. Micro version lag may not mean a significant difference between the dependency used compared to the newest dependency available, but lagging behind by a major or minor version generally implies that there are more improvements that have been missed out on.

Using all declarations (including the *'any'* and *'at-least'* classifications filtered from Table 7.4), Table 7.5 categorises the type of lag found in each declaration, along with the percentage of declarations without lag in each package manager. As noted from Table 7.4, Maven has a significant percentage of technical lag within its ecosystem, having twice as much lag as the next ecosystems

(Packagist, npm and Atom). We also see the effects of including the declarations that cannot lag. Three package managers with high levels of lag in Table 7.4 - NuGet, Pypi and Rubygems - show very low levels of lag once accounting for their *'any'* and *'at-least'* declarations. So while their levels of lag are low overall, when using declaration styles that can lag, there is a higher probability that they will not be kept current.

Table 7.5 splits how a declaration lags into seven categories - the seven possible combinations of major, minor and micro lag. When a dependency has *major* lag, it is behind by at least one major version from the newest available, but it is up to date within its declared major version - it has no minor or micro lag. In the case of simultaneous development branches of projects where multiple major versions of a project are being maintained, this set of dependencies can be considered to be up to date, but in the older development branch. In several package managers, such as Atom, Maven, npm and Packagist, there are a disproportionate number of dependencies that lag only by major versions and therefore are up to date within that specific major range, mirroring anecdotal accounts that projects in these ecosystems choose to keep old major versions updated with bug and security fixes to avoid dependent projects having to deal with breaking updates (see Section 3.3.3). This could be the result of minor ranges keeping them up to date within that major version, or because the project has made a conscious decision to stay within that specific major range of the dependency - this remains a question for follow up research.

In Cargo, there is a significant amount of minor lag. The Cargo ecosystem is relatively new with many major packages having versions within the `0.x.x` range (over 70% are in the zeroth major range, according to Decan and Mens [2019], compared to less than <20% in the mature package managers they studied). Semver-compliant projects consider the zeroth major version to be under heavy development. Semver suggests that a minor update in the zeroth major range be considered backwards incompatible in the same way that major updates are for post-`1.0.0` versions. Therefore, these minor lag numbers in Cargo can be considered similar to the major lag found in mature package managers.

Most package managers have a spread in the types of lag encountered. A significant proportion have major, minor or micro lag, but not a mixture. This is most likely to occur because they are only lagging by a single version, and generally signals a shorter amount of time lag. Those that have a mixture of

lags have had multiple versions released without being updated, and represent longer periods of technical lag.

In general, there are three distinct groups of package managers separated by the amount of lag present and the styles of declarations present in Chapter 5. One group consists of the mature package managers that primarily use fixed or semver compliant ranges. In this group, comprised of Maven, Packagist, npm and Atom, between one-third and two-thirds of declarations lag. A second group consists of other mature package managers that primarily use open ranges - often relying heavily on *'any'* and *'at-least'* classifications. Due to this reliance on open ranges, this group, consisting of NuGet, Pypi and Rubygems, has low levels of lag (<15%). The third group consists of newer package managers. This group has extremely low levels of lag (<12%) with most lag being micro and minor lag. While the author believes that much of these differences in lag types and quantities are explained by the age of the ecosystem, it remains as a follow up question to consider if this is the main reason, or if there are other factors.

7.3.3 Would Semver Declarations Reduce Lag?

Semver is intended to make automatic updates easier, which in turn can reduce technical lag. The question is, how large is the effect on technical lag between using a fixed declaration versus using a semver-compliant declaration. To measure the effect that moving from fixed declarations to semver declarations would have, projects with fixed declarations were considered to be replaced with the most permissive semver-compliant declaration that encompassed that fixed declaration, a minor range, to see what effect this would have on technical lag across the ecosystem, e.g. instead of $B_{dep} = 4.7.3$, this section assumed that $B_{dep} = [4.7.3, 5.0.0)$, the minor range equivalent of the listed fixed declaration, and considered the effect this had on technical lag.

Table 7.6 notes the impact that switching from fixed declarations to a semver-compliant minor range would have on technical lag. While it makes little difference for some package managers that already have a high uptake of semver-compliant or open range declarations, the 'Eliminates Lag' column shows that in most package managers it would eliminate technical lag in roughly 10% of dependencies, a significant proportion of the dependencies which have technical lag. Maven, shown above to have abnormally high amounts of technical lag, is an outlier in this case - some 56% of dependencies would benefit from

a reduction in technical lag if declarations were changed to minor ranges. The 'Lag Using Semver' column shows how many dependencies would still lag with semver declarations, compared to the 'Current Lag' column showing how many currently lag. In most cases, it would resolve between one-third and two-thirds of technical lag.

There are downsides to using minor semver-compliant ranges, in terms of potential breaking updates from mislabeled updates. However, this section shows that if the tooling aspects of dependency management improve, semantic versioning provides a significant part of the answer to technical lag. For tooling, smart API incompatibility checkers and semantic reasoning (where possible) being included in the build cycles would allow upstream developers better hints about what version numbering to make, and could allow package managers to automatically check compatibility before they download dependency updates. This would make backwards compatibility issues a problem much less of the time[6], leading to better trust and fewer problems with dependency updates.

RQ1 Summary:
Technical lag is common, although the quantity varies widely by package manager, from two-thirds in Maven lagging, down to less than a quarter in others. The more permissive open range declarations are much more likely to be current than semver compliant ranges or fixed declarations. When closed range declarations are used, most mature package managers exhibit technical lag in >30% of dependencies. Increased usage of semver-compliant ranges would decrease technical lag significantly.

[6]But not all of the time, as there are aspects of compatibility that cannot be automatically checked given current language constraints, such as many of the behavioural contracts that exist.

Contains lag	Major	Major & Minor	Major & Micro	Major & Minor & Micro	n
Atom	15.93%	2.76%	0.47%	0.37%	240653
Cargo	1.81%	0.24%	0.26%	0.00%	521006
Dub	2.56%	0.42%	0.54%	0.01%	7616
Elm	4.21%	0.00%	0.03%	0.00%	17613
Haxelib	0.36%	0.13%	0.15%	0.00%	5286
Hex	6.47%	0.61%	0.03%	0.01%	64621
Maven	9.80%	3.67%	4.39%	2.57%	3467909
npm	20.24%	2.22%	0.64%	0.49%	75733660
NuGet	1.33%	0.20%	0.13%	0.04%	2748879
Packagist	19.84%	2.01%	0.39%	0.29%	4562384
Pub	2.85%	0.08%	0.02%	0.02%	128692
Puppet	5.13%	0.30%	0.05%	0.05%	50281
Pypi	1.44%	0.95%	0.30%	0.31%	459160
Rubygems	6.96%	0.87%	0.17%	0.11%	4765613
	Minor	Minor & Micro	Micro	No Lag	n
Atom	6.87%	0.90%	2.49%	70.22%	240653
Cargo	8.91%	0.07%	0.37%	88.33%	521006
Dub	6.21%	0.05%	2.60%	87.61%	7616
Elm	0.01%	0.00%	0.09%	95.66%	17613
Haxelib	0.79%	0.09%	1.84%	96.63%	5286
Hex	5.45%	0.38%	0.89%	86.15%	64621
Maven	19.85%	12.23%	13.08%	34.40%	3467909
npm	5.49%	0.88%	1.59%	68.45%	75733660
NuGet	0.48%	0.07%	0.11%	97.64%	2748879
Packagist	6.82%	0.82%	1.05%	68.79%	4562384
Pub	4.07%	0.36%	0.56%	92.04%	128692
Puppet	1.26%	0.16%	0.31%	92.74%	50281
Pypi	4.96%	1.70%	2.26%	88.07%	459160
Rubygems	4.74%	0.44%	0.57%	86.13%	4765613

Table 7.5: Most Common Types of Lag

	Adopting Semver		Lag Totals	
PM	Eliminates Lag	Major Lag Remains	Lag Using Semver	Current Lag
Atom	10.3%	3.6%	19.5%	29.8%
Cargo	9.4%	0.5%	2.3%	11.7%
Dub	8.9%	1.0%	3.6%	12.4%
Elm	0.1%	0.0%	4.2%	4.3%
Haxelib	2.7%	0.3%	0.7%	3.4%
Hex	6.7%	0.7%	7.2%	13.9%
Maven	45.2%	10.6%	20.4%	65.6%
npm	8.0%	3.4%	23.5%	31.5%
NuGet	0.4%	0.7%	2.0%	2.4%
Packagist	8.7%	2.7%	22.1%	31.2%
Pub	5.0%	0.1%	3.0%	8.0%
Puppet	1.7%	0.4%	5.5%	7.2%
Pypi	8.9%	1.6%	3.0%	11.9%
Rubygems	5.8%	1.2%	8.2%	13.9%

Table 7.6: Lag Reduction Using Semver

7.4 Lag Quantity per Dependency

RQ2: How much lag is there in dependencies when they are not up-to-date?

Version Lag

Figures 7.3 and 7.4 show the mean value of technical lag (in terms of number of versions) and time lag (in terms of days) by package manager, based on the approach described in Section 7.2.4. Despite dependencies often having technical lag, the median declarations do not have lag for major, minor and micro versions. The data is quite skewed, where a few percent of dependencies are heavily outdated, with the vast majority behind by 0 - 2 versions. Once dependencies without the respective types of lag have been excluded, the standard deviation of the data is generally similar in size to the mean, indicating a predictably right-skewed distribution, as seen in Table 7.7.

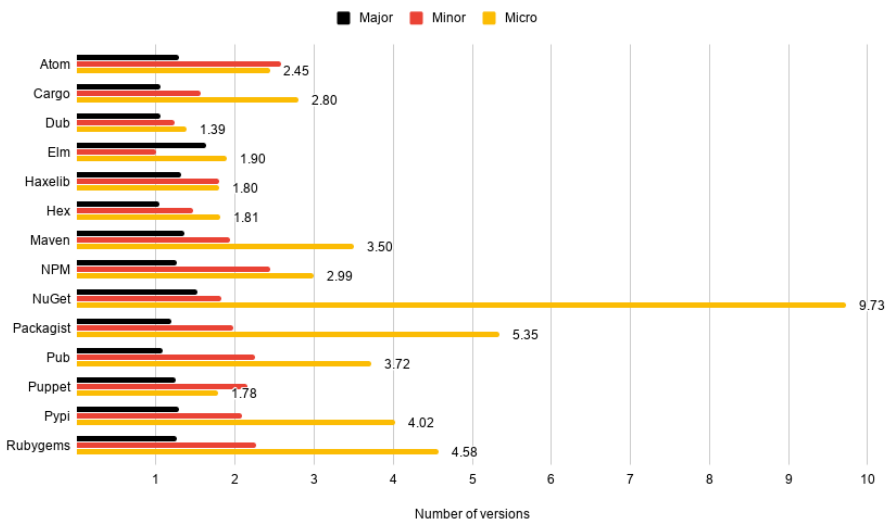

Figure 7.3: Version Lag by Package Manager

As shown in Figure 7.3, when there is a lag in major releases, all package managers have an average of 1 to 2 major versions lag. The values increase for micro releases, where lag can be over 5 (Packagist, NuGet). Micro releases

	Major		Minor		Micro	
PM	Mean	StdDev	Mean	StdDev	Mean	StdDev
Atom	0.25	0.67	0.66	2.13	0.56	3.42
Cargo	0.00	0.07	0.06	0.38	2.02	10.44
Hex	0.04	0.20	0.27	0.67	0.37	1.12
Maven	0.28	0.78	0.74	1.79	1.12	3.90
npm	0.30	0.70	0.64	2.38	0.62	2.30
NuGet	0.35	0.76	0.26	0.73	0.33	2.26
Packagist	0.27	0.57	0.77	2.01	2.04	4.76
Pypi	0.16	0.89	0.79	1.70	0.98	4.26
Rubygems	0.18	0.88	0.78	2.00	0.67	1.81

Table 7.7: Means and Standard Deviation of Version Lag

happen much more frequently than minor or major releases, accounting for 67% of the releases in this dataset (2% are major, 16% are minor and 15% are tag updates), so more micro lag is to be expected.

NuGet contains a high micro lag, coming from only 0.35% of dependencies with a standard deviation of 3.5 times the mean. This indicates that a small handful of outlier projects with large micro lags are responsible for its large result. This results in NuGet fixed declarations needing to be updated much more frequently than other package managers in Figure 7.5 - dependencies accrue technical lag quickly in NuGet. Note that its micro time lag is similar to other package managers, indicating that the reason for the increased micro version lag is that NuGet micro releases seem to happen at a quicker pace. Rubygem's micro lag results are similarly right-skewed as NuGet's, while the others (including Packagist's high micro lag result) exhibit a standard deviation similar to the mean. In all cases, the outlier values are not significant, with the 95th percentile being less than 10 and in all but a few cases, the maximum lag being under 100.

Time Lag

For time lag (Figure 7.4), the lag in number of days for major releases is higher than minor and micro releases for most package managers, which could point to a reluctance to update, as noted in the literature (Section 3.5.4). There

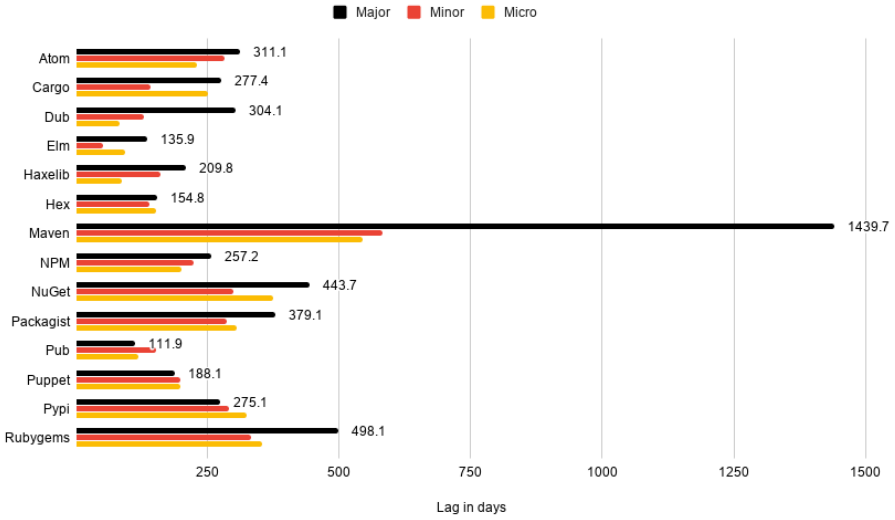

Figure 7.4: Time Lag by Package Manager (in days)
* mean values where the relevant type of lag is present in the declaration

is a range between Pub, which has an average of 112 days lag, to Rubygems, which has an average of 498 days lag between major versions. In most mature package managers, when lag exists, it is between half a year to a year out of date. The outlier lag values are higher in mature package managers, with the maximum lag in Maven being 13 years, but most are below 10 years of lag.

Maven is an interesting outlier for time lag, as the value of time lag for major releases is much higher than other package managers (1439 days). This could be due to the Maven ecosystem's long history of simultaneous development in prominent packages, where developers can continue to use an outdated major version and still receive the necessary updates to keep it secure. The Maven ecosystem tends to be conservative (it is the only major package manager that primarily uses fixed declarations, see Chapter 5, and works within the Java philosophy of maintaining backwards compatibility as much as possible[7]). It also suffers from binary compatibility issues during updates that may dissuade developers from updating. Other package managers such as npm, on

[7]To quote Martin Buchholz: 'Every change is an incompatible change. A risk/benefit analysis is always required.' (`https://blogs.oracle.com/darcy/kinds-of-compatibility:-source,-binary,-and-behavioral`)

the other hand, inform developers of vulnerabilities in dependencies, motivating them to update dependencies faster. Other generic tools, such as Snyk[8], also provide similar functionalities of warning developers of potential vulnerabilities in libraries.

RQ2 Summary:
When lag exists, it is generally in small amounts. Most dependencies do not get more than 1-2 major or minor versions behind, or 3-5 micro versions, owing to that micro updates represent two-thirds of project updates. The time lag for most package managers sits between half a year to a year and a half.

[8] `https://snyk.io`

7.5 Update Frequencies

RQ3: What type of updates are the most common, and what type of project releases contain dependency updates?

This research question investigates update frequencies for fixed declarations only, as discussed in Section 7.2.3.

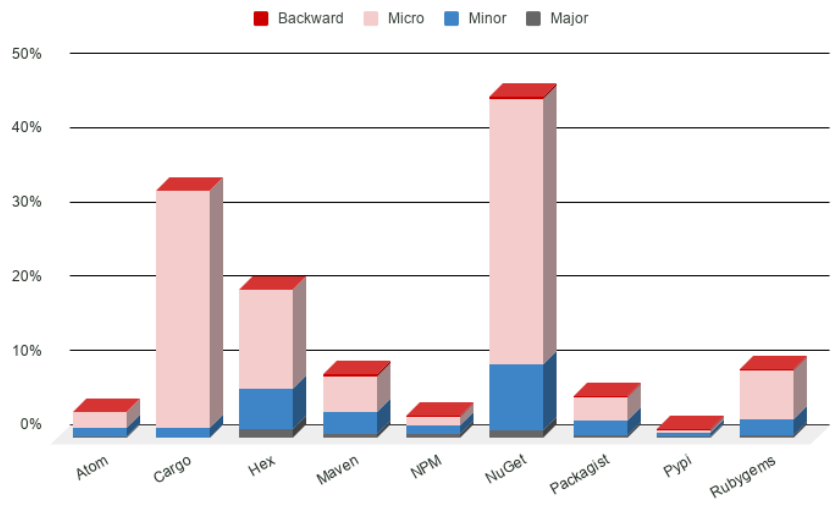

Figure 7.5: Updates by Package Manager

Figure 7.5 shows how often a given dependency is updated, and the types of dependency updates made - whether the dependency is increased a micro version, a minor version or a major version.

The predominant type of dependency update is the micro update, where only the micro number is increased, e.g. `1.0.1` → `1.0.2`. In most package managers, micro declaration updates are two to three times more likely to occur than minor updates. Despite this, due to the high number of micro updates in projects (67% of updates), this increased number of micro declaration updates is unable to keep up with the dependency development, leading to the higher levels of micro lag noted in Section 7.4. However, micro updates are less likely to cause compatibility issues. Semver mandates that they never cause compatibility issues, although in practice this does not always happen [Raemaekers et al., 2014].

Across all ecosystems, major declaration updates, where we expect backward breaking changes to occur, are observed in less than 1% of updates.

The amount developers update their dependencies varies significantly by package manager, with dependencies being updated less than 5% of the time in Pypi, npm and Atom, through to over 45% of the time in NuGet.

Project A Update Type	Dependency Update Type		
	Major	Minor	Micro
Major	5.16%	5.43%	3.83%
Minor	0.96%	4.34%	3.15%
Micro	0.27%	0.92%	2.19%
Tag update	0.28%	2.12%	2.70%

Table 7.8: Frequency of Declaration Updates

The types of declaration updates that developers do varies by the type of update in their own project, as shown in Table 7.8. Across package managers, major updates to dependencies most often coincide with major changes to the dependent project A. Minor updates are more likely in major or minor updates of project A, while micro updates tend to happen in micro or pre-release changes (the *Tag update* row, e.g. `1.4.0-beta.1` → `1.4.0-beta.2`). These results make sense within the semver construct, as major updates to dependencies are more likely to require a significant amount of refactoring or redesigning of a project if the external APIs are not well insulated from the internal logic, so it may be easier to make these disruptive dependency updates when the dependent project is already undergoing significant changes.

RQ3 Summary:
In most package managers, fixed version dependencies are not updated regularly. NuGet (46%), Cargo (33%) and Hex (20%) are the only package managers where fixed dependencies are updated in over 10% of version releases. On the other extreme, in Pypi dependencies are only changed in 1.15% of version releases. Developers tend to update fixed dependencies much more often in major changes to their own projects than in minor or micro updates. They are also more likely to update to a new major version of the dependency at this point as well.

7.6 Update Strategies

RQ4: How often do developers perform updates when they were behind, and does this bring them up to date (or are they intentionally staying behind in some way)?[9]

Table 7.9 shows the amount of time that dependencies are updated based on if they are outdated. The first two columns together show the dependencies that are outdated. The second column shows the ones that are out of date and did not have any changes made to them - these are the dependencies that automatic updates would reduce technical lag on. Notice that in most package managers, over half of fixed dependencies fall within this category. The first column are the dependencies that have had some type of change, but are still out of date. These ones are the most interesting, as they indicate that the developer is choosing an outdated version (potentially for a specific reason), rather than going to the newest version. While this practice of updating to another outdated version varies between ecosystems, it happens from 5% of the time in Cargo, up to 51% of the time in Packagist. The reasons for this are unclear and present an interesting direction for future study.

The final column of Table 7.9 shows dependencies that were already up to date, so did not require developer intervention. The second-to-last column shows the amount of time dependencies are updated and are now current (without the developer's intervention, they were or would have become outdated), e.g. $B_{dep_{i-1}}$ is `3.7.2` and B_{dep_i} has been changed to `3.7.3` - the newest version of B available. Together these two columns make up the fixed declarations that do not have technical lag. In most cases, declaration updates are not made when lagging (being updated <10% of the time). However, in Cargo and NuGet, over half the time that a declaration needs updating, it is updated.

The definition of being outdated is quite coarse - if there is *any* lag (major, minor or micro), the dependency is considered outdated. However, a common situation in some ecosystems is to have a project with simultaneous development on two major versions (Python 2 and Python 3 being a very well known example). This analysis was rerun, considering major lag as not being outdated, and only considering minor and micro lag to be outdated, allowing for projects

[9] This question also focuses on fixed declarations.

in a lagging major version to still be counted as up to date if they are at the newest minor and micro version. This is reported in the two right-most columns of Table 7.9. Comparing the second half of Table 7.9 with the first half shows that over 10% of projects using fixed declarations in Atom, Maven, npm and NuGet are lagging overall but are up to date within that old major version.

RQ4 Summary:
Developers in some package managers update fixed dependencies regularly to stay up-to-date, with NuGet and Cargo updating their lagging dependencies most of the time. In most package managers, however, less than 10% of dependencies that are out-of-date get updated with a new version. Anywhere between 5-51% of updates do not default to the latest version. Projects in NuGet, Maven and npm have a higher probability that fixed declaration dependencies are up-to-date in a lagging major version.

	Updated & Outdated	No Update & Outdated	Updated & Current	No Update & Current
	(With any lag being considered as outdated)			
Atom	0.6%	52.1%	2.9%	44.4%
Cargo	1.8%	23.0%	31.5%	43.7%
Hex	2.3%	35.4%	17.8%	44.5%
Maven	3.8%	62.7%	4.8%	28.7%
npm	0.7%	49.2%	2.3%	47.9%
NuGet	7.3%	28.9%	38.6%	25.2%
Packagist	2.9%	62.0%	2.8%	32.3%
Pypi	0.3%	54.0%	0.9%	44.9%
Rubygems	2.2%	50.7%	7.1%	40.1%
	Allowing for major lag to be classified as *current*[*]			
Atom	0.5%	41.9%	3.0%	54.6%
Cargo	1.8%	22.7%	31.5%	43.9%
Hex	2.2%	33.2%	17.9%	46.7%
Maven	2.8%	53.8%	5.8%	37.7%
npm	0.4%	38.8%	2.5%	58.2%
NuGet	4.5%	19.7%	41.5%	34.4%
Packagist	2.5%	57.3%	3.2%	37.0%
Pypi	0.2%	49.2%	0.9%	49.6%
Rubygems	1.7%	45.1%	7.5%	45.7%

Table 7.9: Updates vs Lag

[*]with only minor or micro lag being reported as *outdated*

7.7 Backwards Updates

RQ5: How often do developers make a backwards change to their dependencies?

In the fixed declarations analysed, backwards changes, where a project deliberately increases the technical lag in a dependency, were a rare case. Table 7.10 reports the frequency of backwards changes across all package managers. The results range from zero found in Cargo, up to 0.33% found in Maven (aggregating the 3 columns together) - representing 1 in 300 declarations. There are no major trends towards making a particular type of backwards change based on the dependency's version change.

PM	Micro		Minor		Major	
Atom	2	0.01%	3	0.01%	0	0.00%
Cargo	0	0.00%	0	0.00%	0	0.00%
Hex	0	0.00%	2	0.08%	0	0.00%
Maven	1357	0.05%	7453	0.25%	899	0.03%
npm	1561	0.02%	2828	0.04%	1890	0.03%
NuGet	5	0.02%	41	0.13%	37	0.12%
Packagist	176	0.15%	53	0.05%	49	0.04%
Pypi	17	0.03%	16	0.03%	13	0.02%
Rubygems	25	0.02%	33	0.03%	13	0.01%

Table 7.10: Frequency of Backwards Changes

Reasons for backwards changes:[10]

Through the manual review of backward changes and Thematic Content Analysis described in Section 7.2.7, we identified several reasons why backward changes were made in 66 separate cases:

- *Project A goes stable.*

The most common time we saw backward changes in dependencies is when

[10]Please note that the following section presented is the work of Dr. Amjed Tahir from Massey University and Dr. Kelly Blincoe from the University of Auckland, which was conducted to complement the author's empirical investigation for RQ5 and has been included due to relevance.

project A moves from an unstable release to a stable release (26 of the 66 backward changes). These downgrades were not explicitly discussed in the project repositories, but it is likely they tried to update their dependencies to the latest versions in the unstable release candidates, but experienced some problems and reverted when releasing the stable version.

- *Compatibility issues.*

  The next most common reason for backward changes were compatibility issues (14 of 66). The project A developers often discussed the compatibility issues in the new version before downgrading the dependencies. Some examples include[11]:

  > *"...many of our production systems use OTP20 still and this change is not backwards compatible."*

  > *"Downgrade to [project B] 2.7.5 until [our project] is compatible with 2.8."*

- *Bug in project B.*

  Another common reason (11 of 66) was that the backward change was made because of a bug introduced in the project B release. For example, after encountering issues after the project B dependency upgrade, one developer identified the problem stems from project B and explains[12]:

  > *"I made the following change to [project B]'s source code ... can we push this issue upstream ... could we downgrade [project B] until an upstream fix can be applied?"*

- *Release not available.*

  Another reason for backward changes (7 of 66) was that the project B release was no longer available in the dependency manager.

- *Performance issues.*

  The new release of project B introduced performance issues in project A (2 of 66).

[11] `https://github.com/AdRoll/erlmld/pull/9`
[12] `https://github.com/renovatebot/renovate/issues/196`

- *New stable version of project B.*

  Project A depended on an unstable new minor release of project B and project B released a new stable micro release of the previous minor version (2 of 66).

- *Consistency.*

  One of the backward changes was made to ensure consistency across the components in project A.

  > *"update [project B] version to match other components"*

- *Mistake.*

  One of the backward changes was made by mistake. The project went from depending on version 1.0.1 to version 0.0.1. When this was questioned, one of the developers said *"Thank you; it's a typo. We'll fix it."*[13]. The change was reverted in the next release.

The remaining two backward changes with explanations did not provide enough details for us to categorize them into one of our themes. They both mentioned that the change was being made as a fix, but did not provide additional details on what was being fixed. It is likely these were compatibility issues or bugs in project B, but we did not include them in the categories above due to lack of available information.

RQ5 Summary
Backwards updates are uncommon across package managers, with the highest rate being 1 of 300 dependency changes in Maven.
Through qualitative analysis of a sample of backward changes, we identified several common reasons for the backward changes in dependencies including: project A moving from an unstable to stable release, project A not being compatible with the newer version of project B, and project B introduced a bug.

[13] `https://github.com/pouchdb/pouchdb/issues/5430`

7.8 Threats to Validity

There are two main threats that can affect the validity of the results reported in this chapter:

Construct Validity.
This threat is mainly related to our automatic approach that used to mine data from the *libraries.io* dataset. We developed a number of scripts that extracted and then processed data obtained from *libraries.io*. We built our scripts in iterations, first testing them on small samples size (from a selected set of package managers) that could be manually verified before employing them on the larger dataset in order improve precision. Our scripts and filtered dataset are publicly available for replication purposes[14].

External Validity.
This study used a data dump from *libraries.io* that contains data from over 2.7 million projects from 37 package managers. We consider this to be a representative sample size for a large scale empirical study. However, we cannot claim that the results can be generalised for other package managers that are not investigated in this study or for closed source projects. Different package managers might have different approaches in handling dependencies, which might result in different conclusions.

The libraries.io dataset has been validated in each of the experiments. Most of the data has been deemed to be of high standard. There have been some times where timestamps are off by a small amount - an artefact of the web scraping process - and there are occasional missing data points. These inconsistencies are not deemed to affect the overall validity of the results. There are also a few places where data has been incorrect, such as timestamps from the year 1900. The following outlines the places we noticed issues:

- The dependency declaration data for some versions is missing from the dataset. This occurs roughly 5% of the time. No data for these versions were included.
- While the timestamps of when versions were published are intended by the dataset to be the published time, in some cases, they were not accurate. When checking for lag in the *at-least* declaration classifications,

[14] `https://github.com/jacobstringer/masters/`

there existed lag in 40% of Pub, 8% of npm, 3% of Maven, and <1% of other package managers. Since it seems unlikely that developers would choose a declaration that cannot be satisfied by any version, a more logical explanation is that the published timestamps on some versions are not accurate. Informal tests showed this accuracy issue to be a matter of hours to a few days in most cases, so further investigations were not carried out. A second issue with timestamps is in NuGet, where 275 pairs have default timestamps (`1900-01-01`). These pairs were removed from the time lag analysis in Section 7.2.4.

7.9 Conclusion

In this chapter, we investigated technical lag in a wide range of package managers. We also studied the frequency of updates, updating strategies, and the reasons for backward updates.

The results of this analysis show that technical lag is common. Many dependencies lag, but this varies widely by package manager, as declarations can be anywhere on a continuum from *fixed* (where lag is avoided by regular developer intervention), to *open ranges* (where the latest version is always able to be used). In between these two extremes are semver compliant ranges, *micro* and *minor*, which have less lag than fixed declarations, but still can lag when the dependency has minor or major updates respectively. Moving from restrictive declarations to semver-compliant ranges could solve a significant amount of lag, avoiding lag in one-third to two-thirds of dependencies which currently lag.

When lag exists, it is generally in small amounts. Most dependencies do not get more than 1-2 major or minor versions behind, or 3-5 micro versions behind. In terms of the frequency of which developers update their dependencies, in most package managers, fixed version dependencies are not updated regularly. When they are updated, as much as 50% of the time they are not updated to the most recent versions.

Developers tend to update fixed dependencies much more often in major changes to their own projects than in minor or micro updates. They are also more likely to update to a new major version of the dependency at this point as well. Developers in some package managers (e.g. NuGet and Cargo) update their fixed dependencies regularly to stay up-to-date. However, this is not

common across other package managers.

Backwards updates are uncommon across package managers - the package manager with the highest rate being 1 in 300. When backward changes happen, the most common reasons for them are that project A moves from an unstable to a stable release, project A is not compatible with the newer version of project B, or that project B introduces a bug.

This study only analysed update changes for fixed declarations. An interesting follow up question would be if these results can be generalised to flexible declarations - a topic which is discussed further in Section 7.10. This might also help shed light on why update patterns vary so wildly by package manager. This study also found that in particular ecosystems, such as Maven and npm, a significant number of projects are up-to-date in an old major version. It would be interesting to know how often this is done deliberately, and why developers have avoided migrating their projects to new major versions of dependencies.

This study also points to how effective semantic versioning ranges are in reducing technical lag in dependencies. We believe that improved tooling for detecting incompatible updates and increased integration into build cycles are still needed to shield developers from breaking changes, and for them to have the necessary trust in upstream developers to use these more permissive declarations.

7.10 Future Work

Classifying Update Types for Flexible Declarations

One of the future directions directly in the line of this work would be to understand how developers choose to update flexible declarations - expanding on the work in this chapter to understand how fixed declarations are updated.

If we wish to extend updates to flexible declarations, it is possible with the *declaration* object discussed in Section 7.2.2, but requires different categorisations for updates. Categorising changes to flexible declarations is more complex than for fixed declarations, as changes come in many forms - the starting version can change, as can the ending version, additional ranges can be added or removed, and all of these can happen independently of one another. There are a large number of possibilities, some of which are outlined below:

Classification Based on Ranges

S_1 = Satisfies set of the penultimate declaration
S_2 = Satisfies set of the latest declaration
$S_{\#min}$ = First version in set
$S_{\#max}$ = Last version in set

1. Declarations are two disjoint sets: $S_1 \cap S_2 = \emptyset$
   (a) S_2 consists of older versions
   (b) S_2 consists of newer versions
   (c) Both or neither of the above[15]
2. Declarations overlap but differ: $S_1 \cap S_2 \neq \emptyset, S_1 \neq S_2$
   (a) S_2 includes newer versions: $S_{2max} > S_{1max}, S_{2min} = S_{1min}$
   (b) S_2 excludes newer versions: $S_{2max} < S_{1max}, S_{2min} = S_{1min}$
   (c) S_2 includes earlier versions: $S_{2max} = S_{1max}, S_{2min} < S_{1min}$
   (d) S_2 excludes earlier versions: $S_{2max} = S_{1max}, S_{2min} > S_{1min}$
   (e) S_2's range is further forward: $S_{2max} > S_{1max}, S_{2min} > S_{1min}$
   (f) S_2's range is further back: $S_{2max} < S_{1max}, S_{2min} < S_{1min}$
   (g) S_2 has a wider range: $S_{2max} > S_{1max}, S_{2min} < S_{1min}$
   (h) S_2 has a narrower range: $S_{2max} < S_{1max}, S_{2min} > S_{1min}$
3. Declarations are the same: $S_1 = S_2$

There are further classifications that could be made in item 2 due to declarations possibly consisting of multiple ranges, and item 1c encompasses several options, but even the above classifications are likely to be too fine-grained. Each of the three main classifications give interesting information, but by themselves do not fully describe what choices developers are making. I would hypothesise that 1a, 1b, 2a, 2b, 2e, 2h, and 3 are the most common patterns. That accounts for micro to minor range changes (2a, 2e) and minor to micro range changes (2b, 2h), with the choices depending if the developer takes the time to update the minimum to the newest version simultaneously. 2d would be the equivalent

[15] If either S_1 or S_2 is a disjoint set where the other fully lies within a gap of its range, this condition is possible, although I would hypothesise that this is exceedingly rare in practice based on the results from Chapter 5.

of 2e and 2h in the case where the developer decides that the newest version is compatible with the project and sets it as the new minimum but no other changes are made. It would be interesting to know how often developers increase the minimum version (2d, 2e and 2h combined) when they already had set a range.

In addition to the above set of classifications being quite unwieldy, it gives little information about the motivation of why developers made the declaration update. Given that we have a model to think about declarations, and in particular because semver-compliant ranges are widely used (as shown in Chapter 5), it would be more interesting to frame the updates in terms of moving between different styles of declarations.

Classification Based on Semver Declarations

Another possible way to describe declaration updates would be to use a composite classification, based on three orthogonal classifications:

1. Declaration Type Changes
   (a) Declaration type remains the same
   (b) S_1 was micro, S_2 is minor
   (c) S_1 was minor, S_2 is micro
   (d) S_1 was a semver-compliant range, S_2 is fixed
   (e) S_1 was fixed, S_2 is a semver-compliant range
   (f) S_1 was a custom range, S_2 is a semver-compliant range
   (g) S_1 was a semver-compliant range, S_2 is a custom range
   (h) Other style change
2. Range Changes
   (a) No change, e.g. $S_1 = S_2$
   (b) S_2 is newer, e.g. $S_{2max} > S_{1max} \vee S_{2min} > S_{1min}$
   (c) S_2 is older, e.g. $S_{2max} < S_{1max} \vee S_{2min} < S_{1min}$
3. Sets Overlap
   (a) Overlap, e.g. $S_1 \cap S_2 \neq \emptyset$

(b) No Overlap, e.g. $S_1 \cap S_2 = \emptyset$

The above set of classifications would focus first on style changes - movements along the declaration risk continuum in Figure 5.1. These types of updates are easy to understand and interpret, as they build on a layer of abstraction (by first classifying the type of range), unlike the previous set of categories. Similarly, custom ranges which could be replaced with a semver-compliant range need not be treated differently at this stage, giving a much more generalisable set of data. While it is possible for other changes to exist (e.g. going between fixed and custom ranges), unless it turns out that 1h gets a significant portion of the changes, there is a risk of going too fine grained.

The range changes in the second part of the classification give a coarse indication if the change is to newer versions or older versions. Since these three options do not fully account for range changes (for example, it is possible that 2b and 2c could be true simultaneously, such as when a range is widened), it is possible to substitute them for the range classifications in the previous classification set. However, based on the results from previous chapters, that level of fine grained results is not likely to provide too much additional information despite its added complexity.

The third criteria, checking if sets overlap, provides a sense of scale to the change. A change that adds versions to the satisfies set will tend to be smaller than one that completely replaces the set (e.g. updating a minor range from version 3 to version 4). Combining this information with part two will provide additional insights regarding the size and direction of the updates.

Chapter 8

Conclusions

Dependency management is a necessary part of any modern, large-scale project. It can also be complex - dependencies are usually under development at the same time as client projects, and therefore linking a project to a compatible version of a dependency is imperative to creating a stable build. However, keeping a dependency at a specific, compatible version means missing out on any fixes (bugs, security patches etc.) that are released. This missing out of dependency updates is a phenomenon called technical lag, which literature has repeatedly shown to negatively affect the quality of a project.

Semver was released 20 years ago as a framework for dependency management automation. It allows upstream projects to signal to downstream projects what sort of changes they are making, and specifically, if they believe the update is going to introduce breaking changes - the idea being that package managers can automatically download new versions of dependencies as long as they do not break, leading to smaller amounts of technical lag and requiring a smaller time commitment from developers. In order for package managers to choose between multiple versions, a developer only needs to create a declaration that allows any minor versions or micro versions of a given major range. This declaration is called a flexible declaration, as multiple versions would satisfy it, as opposed to fixed declarations which only allow one specific version to be used. While recent studies have pointed to semver-compliant ranges being used more in declarations in some ecosystems, the work reported in this thesis shows that there is a long way to go for its full potential to be realised.

Upon starting these studies, we expected to find that, while semver was a great idea, it did not have great uptake in industry - a view that was perhaps shaded by our backgrounds using Maven, an ecosystem where fixed version declarations are commonplace and the community is dominated by conservative practitioners. After the initial experiment reported in Chapter 4, we realised that the picture was much more complicated, with each of the three ecosystems studied having quite different approaches to dependency management and semver uptake. This led to a much larger study in Chapter 5 which found that this was not unique to npm versus Maven, but that almost all package managers have different declaration patterns, and therefore different views on automating dependency management - although one interesting finding is that the Maven ecosystem is unique in just how conservatively it approaches dependency management. Chapter 6 cross-validated our findings, when developers gave free form answers of how they practiced dependency management - while there were some themes which we will discuss below, the most interesting part was the heterogeneity of the responses, mirroring the data captured in Chapter 5.

With the findings showing little agreement in dependency management styles and declaration patterns between ecosystems, our focus turned to the ramifications of these results. Each declaration style has to weigh up two competing factors - on one hand, being more restrictive with the declaration means investing more developer time into maintenance (or alternatively increasing technical lag with each update of the dependency), and on the other hand, being more permissive leaves a project more vulnerable to breaking changes released in updates of the dependency. There have been studies looking at the cons of being too permissive (see Section 3.3), and others that have looked at the cons of having technical lag (see Section 3.5.3 in particular), however, we are not aware of any studies that allow us to generalise how much technical lag is present, or give us an indication of how declaration patterns will affect lag. It also quantifies the choice that a developer makes when a declaration is fixed - the most restrictive declarations of all - how often do they invest time into updating, versus how often do they ignore it and allow technical lag to increase.

The results corroborated previous research findings and added additional nuances. Overall, fixed declarations incurred technical lag a great deal of the time, and by and large developers are not good at keeping up with updates

when it has to be done manually. They tend to leave updating dependencies until major milestones in development (such as major releases of their own project), unless the change is trivial. We also looked into how much would semver-compliant ranges help reduce technical lag. The results were staggering - depending on the ecosystem, up to two-thirds of lagging dependencies could be up-to-date if using minor ranges, and in all ecosystems, at least three-quarters of dependencies would be up-to-date.

8.1 Future Work

Given our results, it is clear that semver offers a way to reduce technical lag, allowing for higher quality projects. In the course of these studies though, we often ran into reasons why it *currently* cannot fulfil its potential.

In order to understand why semver has been a mixed success so far, it is necessary to realise that semver is a social contract between developers rather than a technical issue to be solved. The system relies on upstream developers (1) understanding what types of updates can cause breaking changes, and (2) following an implicit contract in the form of semver principles when updating their version numbers. Both in the literature and from developers' experiences shared in Chapter 6, it is clear that each point is problematic. For downstream developers, semver relies on trust - trust that upstream developers do both. Because a notable number of upstream developers either ignore or are unaware of all the subtle ways that a project can break backwards compatibility (which is an extremely complex process), downstream developers often lose trust in the upstream developers' ability to protect them from breaking updates, causing the contract to fall apart.

No social contract can ever be fool-proof. However, most successful social contracts build in incentives for agents to keep the contract, or penalties for agents who break the contract. An example of a similar social contract is bullying on social media. There are generally no explicit rules that can stop this behaviour, but for any agents who bully others, there is a penalty in the form of social policing. This penalty usually helps to avoid this negative social behaviour as the potential bully does not wish to be ostracised by their social group.

For upstream developers, breaking the social contract they have with (po-

tentially anonymous) downstream developers by introducing breaking changes carries fewer penalties than the social media bullying example. There may be some reputation damage, and downstream developers may increase their workload by reporting more issues, but the penalties involved for upstream developers who introduce breaking changes tend to be small. While the cost of introducing breaking changes for upstream developers is low, it can be high for downstream developers who must immediately fix the issues created by the dependency update.

Some ecosystems have attempted to increase cooperation between upstream and downstream developers, in order to heighten both the benefits of keeping the contract, and increasing the penalties of ignoring it. CRAN is a good example of this, where the repository runs daily integration tests to ensure that projects can build successfully.[1] To supplement this, it requires both upstream and downstream developers to work together on breaking changes, in order to keep projects up-to-date. Eclipse[2] employs a similar tactic, where its yearly releases give a consistent snapshot of projects that work with each other [Tahir et al., 2017]. However, the costs of maintaining consistency is high, so these models are not common throughout the industry.

Another approach that has been suggested [Decan et al., 2018] is to increase the penalties of non-compliance by creating health metrics for dependencies - making reputation highly visible. If projects are rated on their activity levels and ability to keep compliance with semver, this would make it easier for downstream developers to make choices about which dependencies to include. Well respected dependencies would be used more, and ones that often break contracts with their downstream users would be shunned by their communities. Social problems tend to have social solutions, and this social pressure may prove to be an effective strategy if it became widely used.

While semver may never be perfect, due to its reliance on social contracts, there are also technical measures that we can use to improve it:

1. Improve the information that upstream developers have access to. For upstream developers, having tools that analyse programme changes and make smart suggestions for what the semantic version should be updated

[1] See `https://cloud.r-project.org/web/checks/check_summary_by_maintainer.html` for daily updates of `R CMD check` integration tests done by the CRAN maintainers.

[2] `https://www.eclipse.org/`

to would improve the fidelity of the versioning numbers and decrease the amount of accidental breaking changes that projects make (which is currently very high).

2. Improve existing tools which check updates for breaking changes and avoid automatic updates where issues are detected. For downstream developers, because it is not possible to trust that upstream developers have correctly incremented their version number, having additional, independent information is essential. Integrating a tool that scans for breaking updating into a package manager or continuous integration cycle would be the best case, as the survey has shown that while some sophisticated API checking tools are available, none of the respondents currently use any of them. For this case, it is both about having high quality tools available, and rolling them out widely.

Unfortunately, the process of automatically spotting breaking changes is exceedingly difficult. Not only must syntactic contracts be kept (there are currently tools that automate this in statically typed languages), but behavioural contracts (pre- and post-conditions) and other non-technical elements such as licences must also be kept. A few behavioural contract checks can be automated (such as nullability and errors thrown), but without the adoption of languages that enforce pre- and post-conditions, most behavioural contracts remain out of the reach of automation.

In the meantime, there are tools that can help bridge the communication gap between upstream and downstream developers. The literature confirms that a good portion of technical lag stems from developers not being aware of dependency updates - tools like Greenkeeper[3] or Dependencies.io[4] which check for new dependencies and let developers know of possible updates are a light-weight first step to keeping technical lag down in an environment where automatic updating via semver-compliant declarations is not safe enough for many developers.

Automation is the future for dependency management, but that is still some way from occurring, as tools must be improved and be helped by advances in language design. In the immediate future, education for both upstream and downstream developers is imperative to ensure the social contracts between

[3] `https://greenkeeper.io/`
[4] `https://dependencies.io/`

them can be as stable as possible. Social pressure is likely to be effective as well, with the first step being making a project's reputation and semver-compliance highly visible - the community's social policing may well make breaking changes have much stronger repercussions than currently, making upstream developers more careful with their version numbering schemes.

Bibliography

[Abate et al., 2012] Abate, P., Di Cosmo, R., Treinen, R., and Zacchiroli, S. (2012). Dependency solving: a separate concern in component evolution management. *Journal of Systems and Software*, 85(10):2228–2240.

[Andrew and David, 2000] Andrew, H. and David, T. (2000). The pragmatic programmer: From journeyman to master.

[Bavota et al., 2015] Bavota, G., Canfora, G., Di Penta, M., Oliveto, R., and Panichella, S. (2015). How the apache community upgrades dependencies: an evolutionary study. *Empirical Software Engineering*, 20(5):1275–1317.

[Beugnard et al., 1999] Beugnard, A., Jézéquel, J.-M., Plouzeau, N., and Watkins, D. (1999). Making components contract aware. *Computer*, 32(7):38–45.

[Bogart et al., 2015] Bogart, C., Kästner, C., and Herbsleb, J. (2015). When it breaks, it breaks: How ecosystem developers reason about the stability of dependencies. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, pages 86–89. IEEE.

[Bogart et al., 2016] Bogart, C., Kästner, C., Herbsleb, J., and Thung, F. (2016). How to break an api: cost negotiation and community values in three software ecosystems. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 109–120.

[Braun and Clarke, 2006] Braun, V. and Clarke, V. (2006). Using thematic analysis in psychology. *Qualitative research in psychology*, 3(2):77–101.

[Chalin and James, 2007] Chalin, P. and James, P. R. (2007). Non-null references by default in Java: Alleviating the nullity annotation burden. pages 227–247.

[Claes et al., 2018] Claes, M., Decan, A., and Mens, T. (2018). Inter-component dependency issues in software ecosystems. *Software Technology: 10 Years of Innovation in IEEE Computer*.

[Claessen and Hughes, 2000] Claessen, K. and Hughes, J. (2000). QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the International Conference on Functional Programming (ICFP)*, pages 268–279. ACM.

[Cox et al., 2015] Cox, J., Bouwers, E., Van Eekelen, M., and Visser, J. (2015). Measuring dependency freshness in software systems. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 109–118.

[Decan and Mens, 2019] Decan, A. and Mens, T. (2019). What do package dependencies tell us about semantic versioning? *IEEE Transactions on Software Engineering*.

[Decan et al., 2018] Decan, A., Mens, T., and Grosjean, P. (2018). An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering*, pages 1–36.

[Derr et al., 2017] Derr, E., Bugiel, S., Fahl, S., Acar, Y., and Backes, M. (2017). Keep me updated: An empirical study of third-party library updatability on android. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2187–2200. ACM.

[Dietrich et al., 2014] Dietrich, J., Jezek, K., and Brada, P. (2014). Broken promises: An empirical study into evolution problems in java programs caused by library upgrades. In *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 64–73.

[Dietrich et al., 2016] Dietrich, J., Jezek, K., and Brada, P. (2016). What java developers know about compatibility, and why this matters. *Empirical Software Engineering*, 21(3):1371–1396.

[Dietrich et al., 2017] Dietrich, J., Pearce, D. J., Jezek, K., and Brada, P. (2017). Contracts in the wild: A study of java programs (artifact). In *DARTS-Dagstuhl Artifacts Series*, volume 3. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[Dietrich et al., 2019] Dietrich, J., Pearce, D. J., Stringer, J., Tahir, A., and Blincoe, K. (2019). Dependency versioning in the wild. In *Proceedings of the 16th International Conference on Mining Software Repositories*, pages 349–359.

[Dig and Johnson, 2006] Dig, D. and Johnson, R. (2006). How do apis evolve? a story of refactoring. *Journal of software maintenance and evolution: Research and Practice*, 18(2):83–107.

[Drossopoulou et al., 1999] Drossopoulou, S., Eisenbach, S., and Wragg, D. (1999). A fragment calculus-towards a model of separate compilation, linking and binary compatibility. In *Proceedings. 14th Symposium on Logic in Computer Science (Cat. No. PR00158)*, pages 147–156. IEEE.

[Drossopoulou et al., 1998] Drossopoulou, S., Wragg, D., and Eisenbach, S. (1998). What is java binary compatibility? In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 341–361.

[Ekman and Hedin, 2007] Ekman, T. and Hedin, G. (2007). Pluggable checking and inferencing of non-null types for Java. 6(9):455–475.

[Espinha et al., 2014] Espinha, T., Zaidman, A., and Gross, H.-G. (2014). Web api growing pains: Stories from client developers and their code. In *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 84–93.

[Fähndrich and Leino, 2003] Fähndrich, M. and Leino, K. R. M. (2003). Declaring and checking non-null types in an object-oriented language. pages 302–312.

[Foo et al., 2018] Foo, D., Chua, H., Yeo, J., Ang, M. Y., and Sharma, A. (2018). Efficient static checking of library updates. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 791–796.

[Gamma et al., 1993] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1993). Design patterns: Abstraction and reuse of object-oriented design. In *European Conference on Object-Oriented Programming*, pages 406–431. Springer.

[German et al., 2007] German, D. M., Gonzalez-Barahona, J. M., and Robles, G. (2007). A model to understand the building and running interdependencies of software. In *Reverse Engineering, 2007. WCRE 2007. 14th Working Conference on*, pages 140–149. IEEE.

[Gonzalez-Barahona et al., 2009] Gonzalez-Barahona, J. M., Robles, G., Michlmayr, M., Amor, J. J., and German, D. M. (2009). Macro-level software evolution: a case study of a large software compilation. *Empirical Software Engineering*, 14(3):262–285.

[Gonzalez-Barahona et al., 2017] Gonzalez-Barahona, J. M., Sherwood, P., Robles, G., and Izquierdo, D. (2017). Technical lag in software compilations: Measuring how outdated a software deployment is. In *IFIP International Conference on Open Source Systems*, pages 182–192.

[Haney, 2016] Haney, D. (2016). Npm & left-pad: Have we forgotten how to program ? `https://www.davidhaney.io/npm-left-pad-have-we-forgotten-how-to-program/`.

[Henkel and Diwan, 2005] Henkel, J. and Diwan, A. (2005). Catchup! capturing and replaying refactorings to support api evolution. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 274–283. IEEE.

[Hora et al., 2015] Hora, A., Robbes, R., Anquetil, N., Etien, A., Ducasse, S., and Valente, M. T. (2015). How do developers react to api evolution? the pharo ecosystem case. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 251–260. IEEE.

[Humble and Farley, 2010] Humble, J. and Farley, D. (2010). *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Signature Series (Fowler). Pearson Education.

[Jacobs and Poll, 2001] Jacobs, B. and Poll, E. (2001). A logic for the Java modeling language JML. pages 284–299.

[Jenson et al., 2010] Jenson, G., Dietrich, J., and Guesgen, H. W. (2010). An empirical study of the component dependency resolution search space. In *International Symposium on Component-Based Software Engineering*, pages 182–199. Springer.

[Jezek and Dietrich, 2014] Jezek, K. and Dietrich, J. (2014). On the use of static analysis to safeguard recursive dependency resolution. In *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 166–173. IEEE.

[Jezek and Dietrich, 2016] Jezek, K. and Dietrich, J. (2016). Magic with dynamo–flexible cross-component linking for java with invokedynamic. In *30th European Conference on Object-Oriented Programming (ECOOP 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[Jezek and Dietrich, 2017] Jezek, K. and Dietrich, J. (2017). Api evolution and compatibility: A data corpus and tool evaluation. *Journal of Object Technology*, 16(4):2–1.

[Kalliamvakou et al., 2014] Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D. M., and Damian, D. (2014). The promises and perils of mining github. In *Proceedings of the 11th working conference on mining software repositories*, pages 92–101.

[Katz, 2018] Katz, J. (2018). Libraries.io Open Source Repository and Dependency Metadata.

[Kikas et al., 2017] Kikas, R., Gousios, G., Dumas, M., and Pfahl, D. (2017). Structure and evolution of package dependency networks. In *Proceedings of the 14th International Conference on Mining Software Repositories*, pages 102–112. IEEE press.

[Kula et al., 2015] Kula, R. G., German, D. M., Ishio, T., and Inoue, K. (2015). Trusting a library: A study of the latency to adopt the latest maven release. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 520–524.

[Kula et al., 2018] Kula, R. G., German, D. M., Ouni, A., Ishio, T., and Inoue, K. (2018). Do developers update their library dependencies? *Empirical Software Engineering*, 23(1):384–417.

[Lauinger et al., 2018] Lauinger, T., Chaabane, A., Arshad, S., Robertson, W., Wilson, C., and Kirda, E. (2018). Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web. *arXiv preprint arXiv:1811.00918*.

[Lehman, 1980] Lehman, M. M. (1980). Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076.

[Linares-Vásquez et al., 2013] Linares-Vásquez, M., Bavota, G., Bernal-Cárdenas, C., Di Penta, M., Oliveto, R., and Poshyvanyk, D. (2013). Api change and fault proneness: a threat to the success of android apps. In *Proceedings of the 2013 9th joint meeting on foundations of software engineering*, pages 477–487. ACM.

[Lungu et al., 2010] Lungu, M., Robbes, R., and Lanza, M. (2010). Recovering inter-project dependencies in software ecosystems. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 309–312. ACM.

[Lungu, 2009] Lungu, M. F. (2009). *Reverse engineering software ecosystems*. PhD thesis, Università della Svizzera italiana.

[Male et al., 2008] Male, C., Pearce, D., Potanin, A., and Dymnikov, C. (2008). Java bytecode verification for @NonNull types. pages 229–244.

[Martin, 2009] Martin, R. C. (2009). *Clean code: a handbook of agile software craftsmanship*. Pearson Education.

[McIlroy et al., 1968] McIlroy, M. D., Buxton, J., Naur, P., and Randell, B. (1968). Mass-produced software components. In *Proceedings of the 1st International Conference on Software Engineering*, pages 88–98.

[Mens et al., 2008] Mens, T., Fernández-Ramil, J., and Degrandsart, S. (2008). The evolution of eclipse. In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pages 386–395. IEEE.

[Pashchenko et al., 2018] Pashchenko, I., Plate, H., Ponta, S. E., Sabetta, A., and Massacci, F. (2018). Vulnerable open source dependencies: Counting those that matter. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, page 42. ACM.

[Raemaekers et al., 2012] Raemaekers, S., Van Deursen, A., and Visser, J. (2012). Measuring software library stability through historical version analysis. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 378–387. IEEE.

[Raemaekers et al., 2014] Raemaekers, S., Van Deursen, A., and Visser, J. (2014). Semantic versioning versus breaking changes: A study of the maven repository. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pages 215–224.

[Raemaekers et al., 2017] Raemaekers, S., van Deursen, A., and Visser, J. (2017). Semantic versioning and impact of breaking changes in the maven repository. *Journal of Systems and Software*, 129:140–158.

[Robbes et al., 2012] Robbes, R., Lungu, M., and Röthlisberger, D. (2012). How do developers react to api deprecation?: the case of a smalltalk ecosystem. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 56.

[Roseiro Côgo et al., 2019] Roseiro Côgo, F., Oliva, G., and Hassan, A. E. (2019). An empirical study of dependency downgrades in the npm ecosystem. *IEEE Transactions on Software Engineering*, PP:1–1.

[Salza et al., 2018] Salza, P., Palomba, F., Di Nucci, D., D'Uva, C., De Lucia, A., and Ferrucci, F. (2018). Do developers update third-party libraries in mobile apps? In *Proceedings of the 26th Conference on Program Comprehension*, pages 255–265. ACM.

[Şavga and Rudolf, 2007] Şavga, I. and Rudolf, M. (2007). Refactoring-based support for binary compatibility in evolving frameworks. In *Proceedings of the 6th international conference on Generative programming and component engineering*, pages 175–184.

[Sawant et al., 2018] Sawant, A. A., Aniche, M., van Deursen, A., and Bacchelli, A. (2018). Understanding developers' needs on deprecation as a language feature. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 561–571.

[Sawant et al., 2016] Sawant, A. A., Robbes, R., and Bacchelli, A. (2016). On the reaction to deprecation of 25,357 clients of 4+ 1 popular java apis. In

2016 IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 400–410. IEEE.

[Szyperski, 1999] Szyperski, C. (1999). Greetings from dll hell. *Software Development*, 7(10).

[Tahir et al., 2017] Tahir, A., Licorish, S. A., and MacDonell, S. G. (2017). Feature evolution and reuse-an exploratory study of eclipse. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, pages 582–587. IEEE.

[Vanson_Bourne, 2018] Vanson_Bourne (2018). White paper: The trials and tribulations of component security; are organizations at risk? Technical report, Veracode, CA Technologies.

[Vouillon and Cosmo, 2013] Vouillon, J. and Cosmo, R. D. (2013). On software component co-installability. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(4):1–35.

[Xavier et al., 2017] Xavier, L., Brito, A., Hora, A., and Valente, M. T. (2017). Historical and impact analysis of api breaking changes: A large-scale study. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 138–147.

[Zapata et al., 2018] Zapata, R. E., Kula, R. G., Chinthanet, B., Ishio, T., Matsumoto, K., and Ihara, A. (2018). Towards smoother library migrations: A look at vulnerable dependency migrations at function level for npm javascript packages. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 559–563.

[Zerouali et al., 2018] Zerouali, A., Constantinou, E., Mens, T., Robles, G., and González-Barahona, J. (2018). An empirical analysis of technical lag in npm package dependencies. In *International Conference on Software Reuse*, pages 95–110.

[Zerouali et al., 2019a] Zerouali, A., Cosentino, V., Mens, T., Robles, G., and Gonzalez-Barahona, J. M. (2019a). On the impact of outdated and vulnerable javascript packages in docker images. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 619–623.

[Zerouali et al., 2019b] Zerouali, A., Mens, T., Robles, G., and Gonzalez-Barahona, J. M. (2019b). On the relation between outdated docker containers, severity vulnerabilities, and bugs. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 491–501.

[Zibran et al., 2011] Zibran, M. F., Eishita, F. Z., and Roy, C. K. (2011). Useful, but usable? factors affecting the usability of apis. In *2011 18th Working Conference on Reverse Engineering*, pages 151–155. IEEE.