Master thesis
Information Science

Radboud University Nijmegen

Measuring dependency freshness in software systems

Author:
J.R. Cox
joel.cox@student.ru.nl
Student No. 4023390

Internal supervisor:
Prof. dr. M.C.J.D. van Eekelen
marko@cs.ru.nl

External supervisor:
Prof. dr. ir. J. Visser
j.visser@cs.ru.nl

August 19, 2014

Abstract

Modern software systems often make use of external dependencies to speed up development and reduce cost. These dependencies have to be updated to ensure the flexibility, security, and stability of the system. In this thesis we analyze the dependency update behavior of industry systems. Several measurements are presented to quantify how outdated an individual dependency is, as well as a benchmark-based metric to rate a system as a whole. The system-level metric is validated through three different methods. First, the usefulness of the metric is validated using interviews with practitioners. Secondly, the metric is checked for flatlining when a system is monitored over time. Finally, the metric's relationship with reported security vulnerabilities in dependencies is investigated. The latter validation step shows that systems with outdated dependencies are more than four times as likely to have security issues in their external dependencies.

Acknowledgements

First of all I would like to thank Marko and Joost for their guidance during my research. Your questions and insights helped me shape my thesis into what it has become. Additionally I would like to thank Joost for giving me the opportunity to write my thesis at the Software Improvement Group (SIG), which has been a great experience.

Eric Bouwers and Dennis Bijlsma also provided valuable feedback during the last few weeks. Thank you for helping me put my thoughts into words. I also want to thank the research team, my fellow interns and everybody else at SIG who made my stay a pleasant one.

Finally, I would like to thank my parents who always supported and enabled me in whatever I wanted to pursue. Thank you.

Joël
Amsterdam
August 2014

Contents

1 Introduction **4**
1.1 Research questions 8
1.2 Methodology 8
1.3 Research context 9
1.4 Application 9

2 Dependency freshness at the dependency-level **11**
2.1 Software dependencies 11
2.2 Dependency versions 12
2.3 Dependency freshness 13
2.4 Version distance 14
2.4.1 Version sequence number 14
2.4.2 Version release date 15
2.4.3 Version number delta 15
2.4.4 Public API delta 16
2.4.5 Measurement overview 17
2.5 Datasets 17
2.5.1 Dataset of industry systems 18
2.5.2 Dataset of dependency versions 19
2.5.3 Descriptive statistics 19
2.5.4 Limitations 20
2.6 Dependency freshness in practice 20
2.7 Discussion 22

3 Dependency freshness at the system-level **23**
3.1 Software metrics 23
3.2 Normative versus descriptive metrics 24
3.3 Benchmark-based aggregation metric 24
3.3.1 Measurement aggregation 25
3.3.2 Transformation to rating 26
3.4 Discussion 29

4 Validation 30
4.1 Usefulness 31
4.1.1 Interview guide 32
4.1.2 Results 33
4.1.3 Analysis 36
4.1.4 Threats to validity 38
4.2 Vulnerabilities detection 38
4.2.1 Methodology 38
4.2.2 Results and analysis 40
4.2.3 Threats to validity 41
4.3 Tracking 42
4.3.1 Methodology 42
4.3.2 Results and analysis 42
4.4 Threats to validity 43
4.4.1 Construct validity 43
4.4.2 External validity 44
4.5 Discussion 45

5 Related work 46

6 Conclusion 48
6.1 Application in practice 49
6.2 Future work and opportunities 49
6.2.1 Dependency context 49
6.2.2 Version distance 50
6.2.3 Update effort 50
6.2.4 Dataset cleanup 51
6.2.5 Impact on software quality 51

Bibliography 51

Appendix A: Software metric catalog format 56

Chapter 1

Introduction

Modern software systems often make use of third-party software components in order to function. Components enable developers to reuse code across systems allowing systems to be developed at a lower cost in a shorter period of time [34]. When a component is added to a software system, the system builds upon the functionality provided by the component and thus starts depending on that component in order to function. The term dependency is used interchangeably to describe both the relationships between the component and the system, as well as the component itself.

Throughout the system's lifetime dependencies have to be kept up-to-date with new releases of the dependency, but updating dependencies can come at a high cost [6]. Keeping up-to-date can be challenging as the team working on the software system may have little influence on the development process of the dependency, yet it relies on the developers of the dependency to provide non-trivial security and bug fixes [39].

The lack of control over the dependency's development process and the trade-offs for updating dependencies make the decision to update a dependency non-trivial. Whether to update a system's dependencies is a trade-off. On one hand there is the effort needed to bring dependencies up-to-date and on the other hand the benefits of having up-to-date dependencies. The benefits include:

Security A dependency may contain security vulnerabilities. When a dependency is not updated to patch such a vulnerability, the system may be at risk.

Flexibility Up-to-date dependencies allows the organization to respond to change more quickly, because recent dependency versions are easier to update than older dependency versions. New functionality in dependencies is already available.

Stability Dependency updates often contain bug fixes, improving the stability and correctness of the overall system.

Compatibility External changes to the system's environment may stop the system from functioning.

There a several reasons why dependencies are not updated, or why it's hard to update a system's dependencies. These reasons can both be attributed to the dependency's developers, the software system's developers as well as the direct management of the latter.

Testing effort When a system has a low test coverage, manual testing is required in order to check for regressions after a dependency update. Because the majority of the dependencies aren't versioned in a way that the version number reveals potential breaking changes [38] testing is required.

Prioritization Bug fixing and adding features to a system is often prioritized over preventive maintenance. Dependency updates may contain trivial changes not directly affecting the system, which makes it hard to justify the update effort at the time of the release of the dependency.

Implicit dependencies Not all systems explicitly declare their dependencies, making it impossible to get a comprehensive overview of the dependencies, increasing the effort when updating the dependencies.

Deciding whether to update a system's dependencies is thus a double edged sword and requires a careful balance of the effort needed to update the dependencies and the benefits gained by updating. In order to make such a decision an objective way to quantify whether a system's dependencies are up-to-date is needed.

Little research has been done to define what good dependency management actually entails. A very low level of freshness is seldom desirable, but a bit might be. To understand why this may be desirable, consider the following software release cycle. Functionality is added during a major release, accompanied by bugs which were introduced while adding a feature. In subsequent minor releases, these bugs are fixed. Not being fully up-to-date might thus be beneficial for the stability of the system as subsequent minor releases are more stable than major releases. A function describing the desirability of a dependency given its version distance is plotted in Figure 1.1.

An exception to this is when a dependency contains a critical security vulnerability. Delaying the dependency update leaves the system in a vulnerable state and a low dependency freshness may make the upgrade process difficult. However, the opposite might also be true; a dependency may be so outdated that vulnerabilities do not apply to the version used by the system. For the remainder of this study we assume that having the most recent release of a dependency is the most desirable.

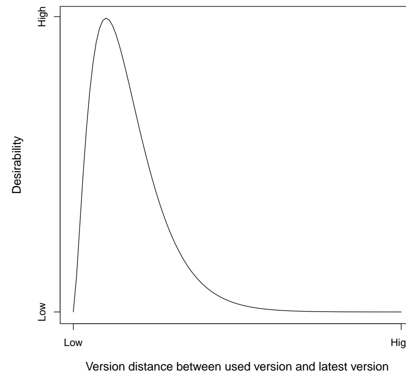


Figure 1.1: Function of version distances between the used dependency and latest dependency, and desirability.

In practice, dependencies are often installed automatically through package managers, which also take care of transitive dependencies. Systems specify their dependencies in a manifest, which are then downloaded and installed by the package manager. This process is repeated until all dependencies are met. There are several intricacies to this process, such as dependency compatibility, circular dependencies and interdependency relations [12][16].

Technical debt

The technical debt metaphor was originally used to describe the problem of not improving less-than-ideal code to non-technical stakeholders; "not quite right code which we postpone making it right" [15]. Over the past decades this concept has been applied to several kinds of debts related to software [28]. The notion of update debt can be defined as the buildup of dependency versions which have yet to be applied to the software system.

In Figure 1.2 a system is depicted with a single dependency, added at t. At $t + 2$, the dependency has seen two releases since t, thus increasing the technical debt of the system. At $t + 3$, the dependencies are updated within the system, after which the debt is paid.

In this study the term dependency freshness is used to express how up-to-date a dependency is. This metaphor is taken from produce; like dependencies they start out fresh, but after a certain period of time they go bad. A system owner may find it desirable to maintain a certain level of dependency freshness to keep the overall system healthy. The exact reasons why this is beneficial are outlined in the previous section. The term freshness thus refers to how recent a version of a dependency is compared to the version that is available.

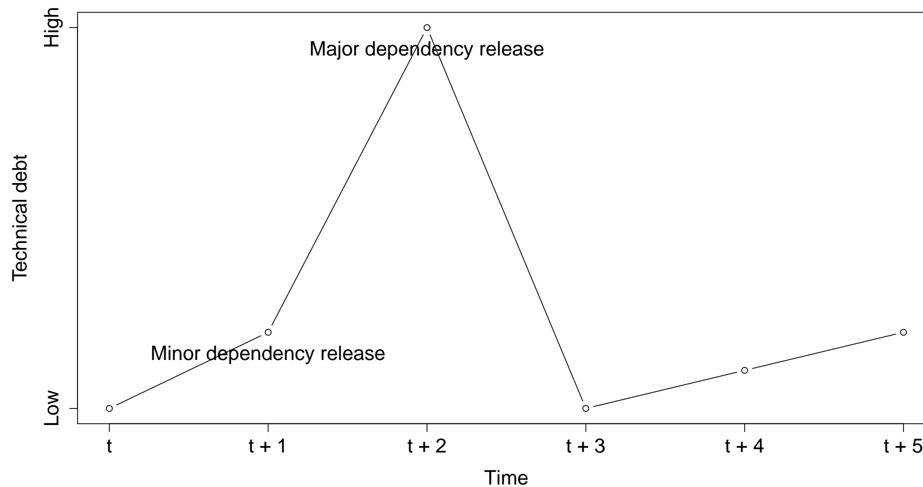


Figure 1.2: Applying the technical debt metaphor to dependency updating.

Relevance

Even though external dependencies are widely used in software engineering, little research has been done on how to manage these dependencies. This means that most of the industry practices are formed through anecdotal experience rather than empirical study. With this in mind, this study could be labelled as fundamental research.

Proper dependency management is not only interesting from a technical perspective, but is also relevant to our society in which software systems take an increasingly prominent role. A prime example is the April 2014 end-of-life of Microsoft's Windows XP, meaning that no new security updates will be released to the general public. Net Market Share [42] still reported a market share of over 26% when support was discontinued for this version of Windows. This leaves more than a quarter of desktop internet users exposed to security vulnerabilities.

Organizations that still haven't upgraded have built up a considerable amount of debt, but these organizations are only partly to blame. It could be argued that Microsoft failed to provide an adequate update path in time, delaying suppliers of the organization to update their applications therefore slowing down the migration. This illustrates the importance of managing dependencies and the risk involved of using third-party dependencies.

1.1 Research questions

To define a system-level dependency freshness metric we first need to be able to measure the dependency freshness of a single dependency. Once a measurement is established a metric for an entire system can be defined and subsequently validated. The following research questions are formulated to guide this research:

RQ1 How can we measure the dependency freshness of a single dependency?

RQ2 How can we measure the dependency freshness of a system as a whole?

The structure of this study is as follows. In chapter two several concepts are defined and measurements are proposed to express the distance between two versions of a dependency. These measurements are then compared and used to analyze a set of industry systems. A single measurement is then used to define a benchmark-based metric, which can be found in chapter three.

This metric is subsequently validated in chapter four, where we look into the usefulness of the metric in practice, its relationship to security and suitability for long-term monitoring. Finally, the related work is covered in chapter five and the conclusion of the study can be found in chapter six, together with future work.

1.2 Methodology

In order to answer the stated research questions the concept of freshness is defined and its different properties explored. Dependencies are mined from a dataset of industry systems in order to gain further insight in how dependencies behave in real life systems. In this research dependencies are considered to be external software libraries which are used by a system. The database of dependencies mined from the industry systems will be enriched with data retrieved from Maven.org, a widely used software repository.

The system-level metric is validated using the criteria from the IEEE Standard 1061-1998 [3]. The general usefulness of the metric is validated through a series of semi-structured interviews with practitioners. A dataset created by Cadariu [14] is used to determine whether dependencies contain reported security vulnerabilities. The metric's performance through time is validated by recomputing the system-level dependency freshness rating for each versions of the system as found in the dataset of industry systems.

1.3 Research context

This study is performed at the Software Improvement Group (SIG), an independent advisory firm that evaluates the quality of software systems. Systems are evaluated using the SIG Maintainability Model [23] and employs a star-based rating system to compare systems to a benchmark of industry systems. Evaluated systems are added to the benchmark repository that is used to calibrate the model on a yearly basis.

1.4 Application

The main goal of the system-level dependency freshness metric is to aid stakeholders in deciding when it is time to update a system's dependencies. This metric should given an indication whether a system is still performing as it should, or requires attention. However, a comprehensive overview of the state of a system's dependencies can't be formed by dependency freshness alone. In order to get such an overview, the dependency freshness metric can be incorporated into a metric system.

A widely-used approach for developing such a software metric systems is the Goal Question Metric approach popularized by Basili et al. [7]. The GQM approach is top-down, rather than bottom-up. The goal is defined first, as the main objective of the metric system.

Two additional aspects related to dependency management where identified as important, namely dependency coupling and security issues within dependencies. Therefore two additional questions are added to the metric system. Finally, metrics are assigned to each individual question in order to complete the metric system. This system is described in Table 1.1.

While this metric can thus be used in isolation, incorporating the metric in a metric system will give stakeholders a more complete view of the state and risks of a system's dependencies.

Goal	Purpose	Make informed decisions
	Issue	about managing
	Object	software dependencies
	Perspective	from the system's owner perspective.
Question	Do any of the software dependencies contain security vulnerabilities?	
Metric	Vulnerability alert ratio [14]	
Question	What is the level of encapsulation and independency of a dependency within a system?	
Metric	Dependency profile [10]	
Question	Are the system's dependencies up-to-date?	
Metric	Dependency freshness	

Table 1.1: A GQM approach to a metric system which could incorporate the dependency freshness metric.

Chapter 2

Dependency freshness at the dependency-level

To gain a better understanding of dependency freshness we first define working definitions for a dependency in general, dependency versions and dependency freshness itself. After defining these terms, several measurements are proposed to capture the dependency freshness concept. A number of these measurements will then be used to analyze a dataset of industry systems and their dependencies. We describe the dataset and how it was obtained.

2.1 Software dependencies

In component-based software development, systems are assembled from multiple components which can be developed independently of one another. These individual components form a cohesive system when put together [34]. In order for the system to function, these individual components have to be able to communicate amongst each other. Such a component is called a dependency. Systems can have several types of relationships with a dependency [8], namely:

Mandatory The dependency has to be present for the system to run.

Optional The dependency is not required for the system to run, but this may impact quality aspects like performance. An example of this would be a compiled database driver versus a driver in a higher level language.

Negative The dependency cannot be present to avoid conflicts with other available components.

Ideally, dependencies are defined explicitly in order to assess fully whether a system functions properly with regards to its dependencies.

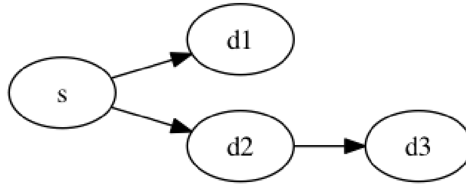


Figure 2.1: Example of a dependency graph of system s with two dependencies, excluding one transitive dependency.

Dependencies can have dependencies on their own, resulting in a recursive structure. One can look at these dependencies on different levels of granularity, such as statements and functions [21]. In this study the focus is on components, or more specifically, libraries. The relations between dependencies and systems can be described in a dependency graph.

More formally, a system s can be described using the following definition in which G describes the dependency graph, V the vertices and E the edges in the graph [9]:

$$G = (V, E)$$
$$V = \{s, d_1, d_2, d_3\}$$
$$E = \{(s, d_1), (s, d_2), (d_2, d_3)\}$$

This system has three components; d_1, d_2 and d_3. Component d_1 and d_2 are dependencies of s, while d_3 is a dependency of d_2.

2.2 Dependency versions

Software version numbers are used to refer to a piece of software at a certain point in time, but they have no universal semantic meaning. The numbers can be an indication of amount of change relative to a previous version, software maturity and even software stability [17]. They can also be used to distinguish between different branches of development. While version numbers may look comparable, their semantic meaning can differ between projects.

The number of versions released within a single time period is different between projects and is likely to be variable. Some projects will only see a handful of releases per year, while other projects may be released several times per day [25]. This makes it impossible to assign an amount of effort based on version number.

The Semantic Versioning[1] standard tries to standardize the meaning of version numbers across projects. In this versioning scheme, version numbers

[1]http://semver.org

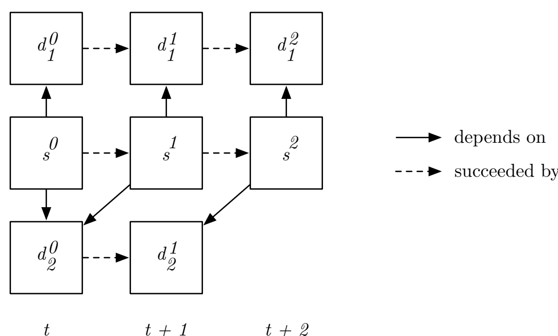


Figure 2.2: Example update path of a system with two dependencies.

signify the stability of a public API, the interface which is specifically designated to be used by third-party developers. The format of these version numbers is a tuple (x, y, z) or the major, minor and patch version numbers, respectively.

When a developer makes a change to the public API that breaks backward compatibility (i.e. the software using the API has to be modified in order to work with a new release of the dependency) the major version number has to be incremented. The minor version number should be incremented when functionality is added to the library, while the patch version number should be incremented when bugs are fixed. In practice, very few projects actually adhere to this versioning scheme [38].

For this study a public release of a dependency with a distinctive version number is considered to be a separate version of a dependency, without making further distinctions.

2.3 Dependency freshness

Dependency freshness can be explained as the difference between a dependency version that is used by a system and the most recent version of that dependency, at that time. More formally; let d be a dependency of system s at time t. At $t+1$ a new version of d becomes available, d^1. However, s at $t+1$ still relies on d; the system relies on an older version of a dependency even though a newer version is available.

Using d at $t+1$ is therefore considered less fresh than using d^1 at $t+1$. In other words, it's less desirable to use dependencies when a newer version can be used. It would be impossible for s to depend on d^1 at t, as d^1 only became available at $t+1$.

A more elaborate example is given in Figure 2.2. In this figure a system with two dependencies is depicted. Dependency d_1 is kept up-to-date with every release of s, while a release of d_2 wasn't updated immediately. This

was corrected in the last version of the system, s^2. Measurements to quantify this distances are presented in the next section.

2.4 Version distance

To express how fresh a certain dependency is, we need a measurement which makes it possible to compare a certain release of a dependency to another release of that dependency. In this section several measurements are explored that use different attributes of a dependency. The measurements are also described using the Software Metrics Catalog Format [11] and can be found in the appendix.

Because the system-level metric will be based on a single measurement, the most appropriate measurement has to be selected. Our criteria for selecting this measurement are based on the criteria described by Heitlager et al. [23].

Indicator of recentness How well the measurement indicates the timespan between two versions of a dependency.

Indicator of change How well the measurement indicates whether a dependency has seen a lot of change.

Ease of implementation How easy the measurement is implemented across languages, as well as additional data needed to compute the distance between two versions.

Sensitivity to outliers How sensitive a measurement is to outliers, such as dependencies with short release cycles or dependencies that see extensive periods of inactivity.

First we present each individual measurements, after which they are scored.

2.4.1 Version sequence number

The difference between two separate versions of a dependency can be expressed by the difference of the version sequence numbers of two releases. This measurement does not necessarily take into account the version number of a dependency, but can employ the dependency's release date to order the difference versions. For a dependency with the versions (d^n, d^{n+1}, d^{n+2}) ordered by release date, the version sequence distance between d^n and d^{n+2} is 2.

Considerations

Dependencies with short release cycles are penalized by these measurement, as the version sequence distance will be relatively high compared to dependencies on longer release cycles.

Version	Version number	Delta	Cumulative delta
d^n	$(1,2,0)$		
d^{n+1}	$(1,2,1)$	$(0,0,1)$	$(0,0,1)$
d^{n+2}	$(1,3,0)$	$(0,1,0)$	$(0,1,1)$
d^{n+3}	$(1,3,1)$	$(0,0,1)$	$(0,1,2)$

Table 2.1: Example of how to compute the version number delta distances between several versions of a dependency.

2.4.2 Version release date

In order to express how recent a version is, the number of days between two releases of a dependency can be calculated. This number expresses the number of days that a new version of a dependency has been available, but not yet updated to.

Let r be a function which returns the release date for a dependency version. The release date distance between $r(d^n) = 10/3/2014$ and $r(d^{n+2}) = 30/6/2014$ is thus 113 days. Note that the difference between d^n and d^{n+2} can be computed directly rather than via d^{n+1}. Unlike the other measures presented in this section, this measurement can be calculated without knowledge of intermediate releases.

Considerations

This measurement penalizes dependencies that release a new version of a dependency after large periods of inactivity. This is often the case for dependencies with a high level of maturity.

2.4.3 Version number delta

Comparing two version number tuples can be done by calculating the delta of all version number tuples between two releases. A version number is defined as a tuple (x, y, x) where x signifies the major version number, y the minor version number and x the patch version number. The function v will return the version numbers tuple for a version of a dependency.

The delta is defined as the absolute difference between the highest-order version number which has changed compared to the previous version number tuple. To compare multiple consecutive version number tuples, the deltas between individual versions are added like normal vectors.

For example, two consecutive versions of a dependency $v(d^n) = (1,2,2)$ and $v(d^{n+1}) = (1,3,0)$ will result in the version delta distance $(0,1,0)$. A more elaborate example can be found in Table 2.1.

Change	$d^n \Delta d^{n+1}$	$d^{n+1} \Delta d^{n+2}$
Classes added	3	0
Classes removed	1	0
Methods changed	5	3
Methods removed	0	2
Delta	9	5
Cumulative delta		14

Table 2.2: Example of how to compute the Public API delta between three versions of a dependency.

Considerations

The main problem with this measurement is that there is no meaningful way of aggregating the tuple of major, minor and patch version numbers to ultimately come up with a single number to represent the version delta. It can be said that $x > y > z$, but it is impossible generalizable the values of the variables as it is completely dependent on a dependency's individual release cycle.

2.4.4 Public API delta

To express the amount of change between two releases of a dependency, the source code of the two versions can be compared, as suggested by Raemaekers et al. [38]. In this paper, the authors use a tool called Clirr[2] to analyze changes in the dependencies API of Java libraries. While Clirr can only analyze changes in Java code, comparable tools could be written for other programming languages. These changes include the removal and addition of classes, methods, fields and more. More details are available in the referred paper.

The number of changes made between each separate version of a dependency can be accumulated to obtain the delta between releases. These changes can include properties like classes added, classes removed, methods changed and methods removed. An example can be found in Table 2.2. It is also possible to modify this measurement so that only the changes between two versions are computed, rather than accumulate the changes of all intermediate versions.

[2] http://clirr.sourceforge.net

	Version sequence number	Version release date	Version number delta	Public API delta
Indicator of recentness	•	+	•	−
Indicator of change	•	−	•	+
Ease of implementation	+	+	+	−
Sensitivity to outliers	High	High	Neutral	Low

Table 2.3: Comparison of the different version distance measurements. Criteria can have a positive relation (+), a negative relation (−) or a neutral relation (•)

Considerations

Analyzing the API of the dependency is a sensible method because the system relying on this dependency is a direct consumer of this API. A change to the API may thus result into an effort in order to reach compatibility again. However, not every change to the API will require such effort and the effort involved is determined by the level of code coupling [10] and exact usage of the API. Implementing this measurement is highly complex and not compatible across platforms.

2.4.5 Measurement overview

The public API delta has the most favorable rating when looking at Table 2.3. This measurement has a low sensitivity to outliers and is a good indicator for change. However, it is hard to implement, nor is the measurement as intuitive as the other measurements.

Because of this, the version release date and version sequence number measurements were selected as candidate measurements. Both these measurements are easy to implement and don't make any assumptions about versioning schemes of the software, unlike the version number delta.

2.5 Datasets

To see how the two selected measurements perform on real data we use a dataset of industry systems, enriched with data from a central software repository. This section describes the datasets and how they were used. Figure 2.3 gives an overview of how the different datasets were obtained.

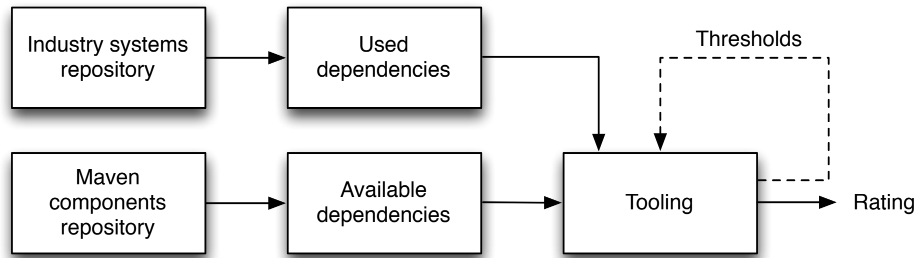


Figure 2.3: Overview of how the different datasets are used to compute a rating.

2.5.1 Dataset of industry systems

To perform our analysis a corpus of industry systems is needed. These systems were made available for study by SIG. Clients provide SIG with regular source code snapshot of their systems, allowing for this type of analysis.

A total of 75 systems from 30 different clients were identified as Java systems which manage their dependencies through Maven[3], a tool for building software projects and managing software dependencies. To perform the analysis dependencies have to be declared explicitly. Maven systems were selected as this was the most widely used dependency manager across all the systems which are analyzed by SIG, although a same type of analysis could be performed on systems using other package managers.

Extracting dependencies

In order to analyze the dependencies of a system, the exact dependencies and their versions have to be mined from the system's source code. Systems which use Maven for their dependency management specify these dependencies in a manifest file, located in the root directory of the system. Additional manifests files can be placed in subdirectories. This manifest file is called `pom.xml` by convention. An example of such a file can be found in Listing 2.1. Version numbers can also be defined through a `properties` element and referenced by the name of the property.

Mining a system's dependencies is a two-step process, resulting in a list of dependencies specified for a system:

1. Search the system directory for files called `pom.xml`.
2. Parse the XML files to retrieve the dependencies and remember the dependency if the dependency wasn't already specified for this system. Variable version numbers are replaced by the value found for the

[3] http://apache.maven.org

key in the <properties> element. Dependencies with undeclared version numbers are discarded. This was only the case for 34 unique dependencies in our dataset.

```
<project>
    <properties>
        <commons-io-version>2.4</commons-io-version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.8.1</version>
        </dependency>
        <dependency>
            <groupId>commons-io</groupId>
            <artifactId>commons-io</artifactId>
            <version>{$commons-io-version}</version>
        </dependency>
        ...
    </dependencies>
</project>
```

Listing 2.1: Example POM file showing the system's dependencies.

2.5.2 Dataset of dependency versions

The data extracted from the repository of industry systems is incomplete in two respects. First of all it cannot be expected that every single version of a dependency is included in this dataset. The intermediate releases between two versions of a dependency are needed to compute the version sequence number distance. Secondly, the mined manifest files only contain dependency names and version numbers, but not their release dates. This release date is needed in order to compute the version release date distance.

In order to complete the database of dependency versions the Maven.org repository is used. This repository is the most widely used repository of Java packages and used by the Maven package manager in its default configuration. For every unique dependency (combination of groupId and artifactId) found in our dataset of industry systems, a query was made to the Maven.org repository. If an exact match was found all versions of that unique dependency were added to our database, together with their respective release dates.

2.5.3 Descriptive statistics

A total of 3107 unique dependencies were retrieved from the dataset of industry system, consisting out of 8718 unique dependency versions. Of these dependency versions 5603 (64%) were classified as internal dependencies

which are not freely available online. This classification was done by comparing the `groupId` of a system with the `groupId` of a dependency. If these are the same it is assumed that the dependency was developed internally.

Querying the Maven.org repository resulted in 2326 unique dependency versions with release dates out of 3115, a hit percentage of 75%. A total of 23431 unique intermediate dependency versions were retrieved. These are the dependency versions which were not found in the dataset of industry systems, but are earlier or later versions of unique dependencies which were found in this dataset. These dependency versions are used to compute the release sequence number distance.

2.5.4 Limitations

The sample of systems is limited to systems which use Maven for its dependency management. This means that the dataset is biased towards systems where the importance of dependency management is understood.

Additionally, the systems are sourced from a repository which were collected by SIG to asses code quality. This raises the possibility that the analyzed systems are of a higher quality than the average software system. However, these systems were not rated on dependency freshness specifically. It could also be argued that the systems are of lower quality, because high quality systems would not require outside help.

Finally, due to the transitive nature of Maven manifests it might be possible that systems include more dependencies than the amount which was ultimately retrieved. The analysis is only performed on first-level dependencies, rather than dependencies of dependencies.

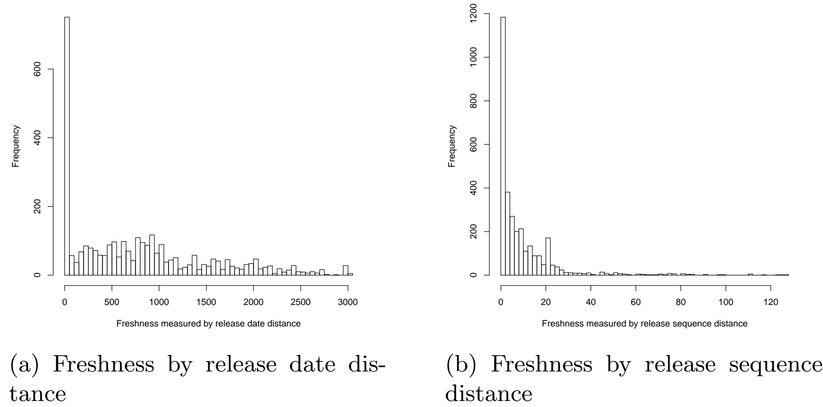
2.6 Dependency freshness in practice

As stated in chapter 1, it is assumed that a high level of freshness is desirable for a dependency. However, it is expected that distribution of freshness does not follow this ideal and presumably follows a fat-tailed power-law distribution, like many other software system properties do [2]. In order to asses this hypothesis, a histogram was plotted for the dependency freshness of all dependencies used in the analyzed systems, measured by the version release distance as well as the version sequence distance.

When analyzing Figure 2.4 (a) the length of the tail stands out, showing that some dependencies are over 3000 days old. Upon further investigation these cases often involve rather mature libraries such as those maintained by the Apache Commons project[4].

An example of this would be the release of the `commons-logging.commons-logging` package, version 1.1.1 on 2007-11-26. This release was

[4]http://commons.apache.org



(a) Freshness by release date distance

(b) Freshness by release sequence distance

Figure 2.4: Distribution of freshness across all dependencies.

followed by version 1.1.2 on 2013-03-16. Calculating the release date distance between the two versions would yield a number found in the tail of the overall distribution, while its perceived freshness might be a lot higher.

The release sequence histogram (Figure 2.4 (b)) does more accurately follow the anticipated power-law distribution. This measurement is more forgiving to mature dependencies which are updated after several years of inactivity, as described in the previous paragraph. However, it is more susceptible to dependencies which have a shorter release cycles. For instance, the `org.eclipse.jetty.jetty-client` package saw 128 releases in a timespan of less than 5 years.

When performing a Spearman correlation test a value of 0.637 is found, indicating a strong, significant ($P < 0.05$) correlation between the two measurements. A Spearman correlation test is used as a monotonic relationship between the two measurements is expected and the variables are not normally distributed. This result confirms that as the version release date distance increases, it it highly probable that the version sequence number distances increases, too (and vice versa).

In Figure 2.5 both the version release date distances and version sequence number distance are plotted. This plot illustrates the different types of release cycles; data points below the red trend line generally have long release cycles, while the data points below the diagonal follow a quick release cycle.

When looking at the overall state of dependency freshness using the version sequence number distance we can conclude that only 16.7% of the dependencies display no update lag at all; the most recent version of a dependency is used. Over 50% of the dependencies have an update lag of at least 5 versions, which is considerable.

The version release date distance paints an even worse picture. The large

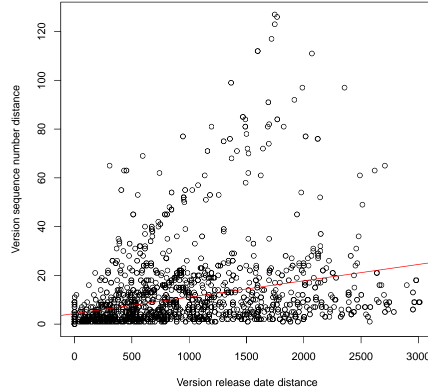


Figure 2.5: A plot showing the relation between the release date distance and version sequence number distance of a dependency.

majority (64.1%) of the dependencies has an update lag of over 365 days, with a tail up to 8 years. This indicates that dependency management is neglected across the analyzed dataset.

2.7 Discussion

To answer **RQ1**: How can we measure the dependency freshness of a single dependency?
We defined several measurements in order quantify the difference between two versions of a dependency. Ultimately we used both the version sequence number distance and version release date distance to analyze a set of industry systems.

While both measurements are sensitive to outliers, the version sequence number distance is selected as the measurement for the remainder of this research. The primary reason to choose this measurement is that it discounts dependencies on quick release cycles (which are thus more subject to change), rather than mature projects which only see minor updates.

In future work further refinements to these measurements could be made (for instance by normalizing the version release date distance using the version sequence number distance) to reduce outlier sensitivity.

Chapter 3

Dependency freshness at the system-level

In this chapter a metric is created that captures dependency freshness at the system-level. To do this, the version sequence number distance measurement that was defined in the previous chapter is used.

3.1 Software metrics

"Software metrics is a term that embraces many activities, all of which involve some degree of software measurement." [19]. Software metrics encompass the software system itself, as well as the process around the software, respectively measuring its internal and external attributes.

Metrics allow stakeholders, managers and engineers to understand what is happening during the development process by making certain aspects of the product or process measurable and thus more visible. This will allow for additional control over the project as well as acting on possible anomalies therefor improving the overall outcome of a project [18, 19].

In order to create a software metrics system, quality factors have to be identified which are then used to determine the quality of a piece of software. These factors are then assigned to internal or external attributes of the system. Next, measurements are established in order to quantify such an attribute and finally a metric aggregates these measurements to a single rating [3].

Software metrics must satisfy several properties in order to be considered valid. Meneely, Smith and Williams [31] conducted a literature study of 20 relevant papers, identifying 47 unique validation criteria. During the study they identified two different philosophies within the software metrics community, favoring different kind of validity criteria, namely a goal-driven-approach or theory-driven approach. The goal-driven approach prioritizes pragmatism over theoretical soundness as long as the metric allows for "as-

sessment, prediction and improvement" of a system or process. Internal validity of a metric plays a central role in the theory-driven approach as these metrics are often used to create a better understanding of a system. A more succinct set of criteria can be found in IEEE Standard 1061-1998 [3]

3.2 Normative versus descriptive metrics

In the previous section a metric is defined as a function which maps measurements to a quality aspect of a system. Creating a metric thus requires the understanding of what the optimum of a certain quality aspect actually is and how far a single measurement deviates from this optimum. This approach is a normative approach, as it compares the system to a norm; an ideal model on how a certain aspect of the system should behave.

The problem of this approach is coming up with these models through empirical research. Setting threshold values to classify measurements often relies on experience and results are often not reproducible nor generalizable [2]. For instance, the upper bound of the McCabe complexity metric was set at 10 in its original paper [30], but this number is only backed up by anecdotal evidence.

An alternative to a normative approach is a descriptive approach. In this approach measurements are performed on a repository of systems which are then used as a benchmark [26]. The threshold values are then determined through a statistical and reproducible process, as explained by Alves et al. [2]. By doing this, a relative metric is created which allows for ranking of systems amongst each other, rather than an absolute measure to describe its quality.

The downside of this approach is that such a metric could rate a system which is a far from perfect at the upper end of the scale, just because all systems across the dataset perform poorly. However, a descriptive approach likely results in a metric with a high discriminative power because it takes the distribution of the systems into account by definition. The metric will also be more actionable [40] as improvements made to the system are more clearly reflected in the metric rating. Ultimately, through continuous recalibration of the metric, a scenario can be approached in which the majority of systems are always fully up-to-date.

3.3 Benchmark-based aggregation metric

Because little research has been done on the desired freshness properties of dependencies a descriptive approach is taken to define the metric. This allows for a definition of a metric without relying on expert knowledge. The methodology that is used is benchmark-based and widely used at SIG [4].

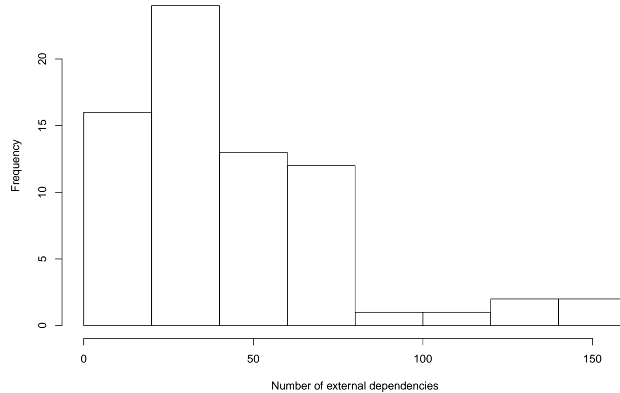


Figure 3.1: Distribution of the number of external dependencies per system.

To do this, the version distance of all dependencies in each system in the benchmark dataset is computed. The version distance distribution is used to assign risk categories to each dependency, using the thresholds obtained in the first-level calibration. Next, a mapping is created to rate a system based on the relative amount of dependencies in each risk category, using the second-level calibration.

3.3.1 Measurement aggregation

The first step to defining such a metric is by creating a risk profile for each separate system. A risk profile is the distribution of measurements across a system, classified into certain risk categories. This results in a risk profile like $\langle 27, 9, 5, 2 \rangle$ meaning that a system has 27 'low risk', 9 'moderate risk', 5 'high risk' and 2 'very high risk' dependencies.

The threshold values for these classification are determined by plotting a cumulative density function, containing all version sequence measurements across the dataset. Figure 3.2 shows the overall distribution of version distance as well as the distribution for each system. Three red lines are plotted to indicate the 70th, 80th and 90th percentile, which are used to derive the values of the thresholds for the risk categories. A wide variety of freshness can be observed between system, so no additional weight per dependency is required [2].

After establishing these thresholds, risk profiles for each separate system can be created as shown in Figure 3.3. Each bar represents a single system from the dataset and each color within the bar indicates the relative amount of dependencies in a certain risk category. The length of each differently colored bar is only an indicator of how many dependencies are present in a certain risk category. The system with a single red bar is a system with

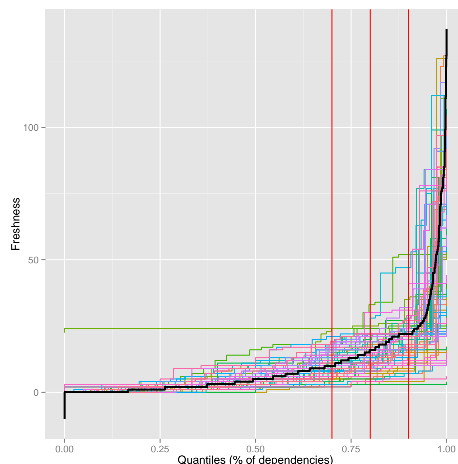


Figure 3.2: Dependency freshness per system, measured by release sequence distance. The black line indicates the average. Red vertical lines are placed on the 70th, 80th and 90th percentile.

Risk category	Low risk	Moderate risk	High risk	Very high risk
Interval	$[0, 10)$	$[10, 16)$	$[16, 22)$	$[22, \infty)$

Table 3.1: First-level calibration thresholds for dependency risk profiles.

only a few dependencies, that are all classified as 'very high risk'.

Note that the number of dependencies per system (Figure 3.1) varies widely; 24.5 at the first quantile, 37 at the median and 60.5 dependencies at the third quantile.

The exact numeric thresholds corresponding to the quantiles are listed in Table 3.1. The classification of a dependency is done by computing the version release sequence distance, after which the corresponding risk category is determined. For instance, a dependency with a version release sequence distance of 12 will be classified as a 'moderate risk' dependency. This process is repeated for each dependency with a known version release sequence distance.

3.3.2 Transformation to rating

While the risk profiles give a good indicator of the distribution of undesirable dependency versions, it is hard to compare the different sections of bars. Alves et al. [1] propose an algorithm for ranking these risk profiles as well as a way to derive a rating from such a profile. To accommodate this, a mapping has to be created that assigns a rating based on the relative size of the risk categories. SIG uses a star rating systems, ranging from one star

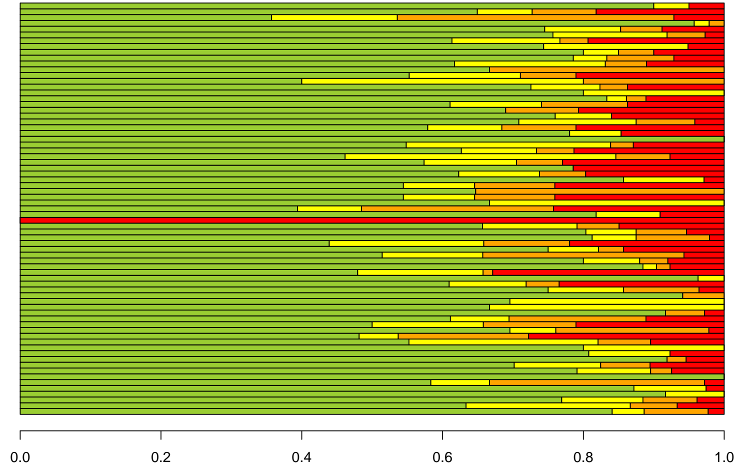


Figure 3.3: Dependency risk profiles for all systems in the dataset created using the first-level calibration thresholds.

Risk profiles	Low risk	Moderate risk	High risk	Very high risk
Non-cumulative	62.2%	20.7%	13.3%	3.8%
Cumulative	100%	37.8%	17.1%	3.8%

Table 3.2: Non-cumulative risk profiles can be transformed in cumulative risk profiles by including the dependencies of higher risk categories.

for low quality systems, to five stars for high quality systems.

The first step to aggregating these risk categories is transforming the risk profile into a cumulative risk profile. This is done by including the dependencies of higher risk categories into the original risk category. Intuitively this makes sense; dependencies which are of 'very high risk' are of at least 'high risk', too. An example of this transformation is shown in Table 3.2.

Once the risk profiles are transformed, exact threshold values for the categories have to be set. Using the set of systems in the benchmark and their associated risk profiles, thresholds are computed so that ultimately a distribution of $\langle 5\%, 30\%, 30\%, 30\%, 5\% \rangle$ of system ratings is reached.

The rating distribution is arbitrary as the methodology is generalizable to n partitions with their respective intervals. The reasoning behind this distribution is that the maximum and minimum ratings are quite rare, while the ratings closer to the median are more evenly distributed.

To assign a discrete rating to a risk profile the star rating has to be found

Rating	Moderate risk	High risk	Very high risk
⋆⋆⋆⋆⋆	8.3%	0%	0%
⋆⋆⋆⋆	30.4%	14.3%	7.7%
⋆⋆⋆	38.9%	30.6%	19.7%
⋆⋆	60.0%	46.3%	27.8%

Table 3.3: Second-level calibration thresholds for cumulative risk profiles.

for which no cumulative risk category from a risk profile is greater than the threshold. The risk profile shown in Table 3.2 would thus score three stars.

$$\begin{aligned} 37.8\% < 60.0\% \wedge 17.1 < 46.3 \wedge 27.8\% < 3.8\% &\Rightarrow \star\star \\ 37.8\% < 38.9\% \wedge 17.1 < 30.6 \wedge 27.8\% < 3.8\% &\Rightarrow \star\star\star \\ 37.8\% > 30.4\% & \end{aligned}$$

This risk profile satisfies all the threshold for a two and three star rating, but fails to get a four star rating when the risk profile is compared with the 'moderate risk' threshold. The threshold for this category is set at 30.4% while 37.8% of the dependencies was classified as such.

It is also possible to assign a more fine-grained rating to a risk profile by using linear interpolation. This is done by first computing the discrete rating of a system and subtracting the absolute volume of dependencies in this risk category, normalized by the length of the risk category's interval. The lowest value of this computation for each risk category is taken and 0.5 is added for ease of arithmetic rounding when the rating is translated to a star rating.

Applying this process for the risk profile in Table 3.2 is as follows.

$$\begin{aligned} moderateRiskRating &:= 3 + 0.5 - (37.8 - 30.4)\frac{1}{38.9 - 30.4} = 2.629 \\ highRiskRating &:= 3 + 0.5 - (17.1 - 14.3)\frac{1}{30.6 - 14.3} = 3.328 \\ veryHighRiskRating &:= 3 + 0.5 - (3.8 - 7.7)\frac{1}{19.7 - 7.7} = 3.825 \end{aligned}$$

Because this risk profile scored three stars in the discrete rating process, the continuous risk rating for each risk category is interpolated from the the upper and lower bounds of the categories belonging to this rating. Ultimately, this risk profile will be rated as 2.629, because it is the lowest rating of all three risk category ratings.

3.4 Discussion

To answer **RQ2**: How can we measure the dependency freshness of a system as a whole? A benchmark-based metric was defined as there is no normative data available on good dependency management. A two-step process was used to define this metric. First, different risk categories were set to classify dependencies by their version distance. Secondly, a mapping was created so that the risk profiles can be aggregated to ratings. This rating was calibrated so that the systems in the benchmark follow a $\langle 5\%, 30\%, 30\%, 30\%, 5\% \rangle$ distribution. The validation of this metric will be performed in the next chapter.

Chapter 4

Validation

To verify the metric developed in the chapter 3, three different types of validation studies are performed, namely:

Interviews A series of semi-structured interviews (section 4.1) are performed to check the usefulness of the metric in practice. The interviews are also used to check the discriminative power of the metric.

Reported vulnerabilities analysis A dataset of known vulnerabilities is matched with our dataset of used dependencies (section 4.2). This will allow us to investigate the relationship between the security aspect of a system and its dependency freshness rating.

Longitudinal analysis Ratings for systems with multiple versions will be compared to see whether the rating shows sufficient change through time (section 4.3).

The metric is checked for the criteria set by IEEE Standard 1061-1998 [3], as well as the general usefulness of the metric in practice. Finally the claim of a relationship between dependency freshness and security will be validated. The criteria in the IEEE Standard 1061-1998 [3] can be summarized as follows:

Correlation When the metric value changes, the value of the quality factor has to change sufficiently.

Tracking When the metric value changes, the value of the quality factor has to change in the same direction, unless the metric and quality factor are inversely related.

Consistency Metric values and their corresponding quality factor should have a strictly monotonic relation.

Predictability A change in a metric used in a quality factor has to sufficiently predict a change in a related quality factor.

Discriminative power Metric values have to clearly discriminate between high and low quality systems.

Reliability The metric displays the above criteria over a sufficiently large amount of systems.

4.1 Usefulness

To validate whether the proposed metric is considered useful, five semi-structured interviews are performed. In the introduction of this study we defined the goal of this research to create a metric which can quantify whether a system's dependencies are up-to-date or not. If this is considered useful by the interviewees and the metric is an accurate indicator for the attribute, than the metric must be useful.

During these interviews quantitive information about dependency management general is collected, as well the information about the application of the metric. We hope that the information about the dependency management process in general will give additional insight in why systems often have outdated dependencies.

The subjects of the interview are SIG technical consultants. Technical consultants support general consultants on client projects and develop the tooling to perform their analysis. Technical consultants are generally highly educated (Master degree or higher) and experienced software engineers.

Semi-structured interviews are used as this data collection method can be used for collecting quantitive, foreseeable information while still allowing the collection of new insights [24]. The technical consultants were asked about two types of systems, known and unknown systems (see below). By asking technical consultants about a system they are familiar with, we are able to ask about the system owner's motivation for (not) updating their dependencies.

Known system A system from a client that the technical consultant is assigned to. The exact system was selected by the author by picking a system assigned to the technical consultant. The system was discarded if the system had fewer than approximately 20 dependencies, after which another system was picked.

Unknown systems Five systems from the dataset of industry systems, with distinctive star ratings. Systems with an extreme high or low number of dependencies or comparable rating were discarded until 5 systems with distinct star ratings were obtained.

Printouts containing the dependencies and version distance of each dependency were brought to the interviews. An example of such a printout is

provided in Listing 4.1. Each line contains the `groupId` and `artifactId` of the dependency, together with the version number. This is followed by the version sequence number distances as computed using the dataset created in chapter 2. The last line shows the total amount of dependencies in the system, the amount of dependencies which were classified as internal and the number of dependencies with unknown histories. Dependencies with unknown histories are dependencies not found in the Maven.org repository.

```
 - commons-lang.commons-lang at 2.6.0 with distance 0
 - org.springframework.spring-web at 3.2.3 with distance 10
 - javax.mail.mail at 1.4.1 with distance 11
...
Found 58 dependencies, 15 with unknown histories, 10 internal-
    only.
```

Listing 4.1: Example output from the dependency freshness rating tool showing detected dependencies, their version and distance.

During the interviews the subjects were asked to estimate the rating of the known system, as well as rank each unknown system from a low-level to a high-level of dependency freshness. The interview guide below provides a more elaborate overview of how the interviews were structured.

Interviews were all conducted by the author and generally lasted for 30 minutes. All interviews were conducted at the SIG office. Each technical consultant was only interviewed once to avoid testing effect biases.

4.1.1 Interview guide

The interview is split in four different phases. After the interviewee has given a rating for the known system, the printouts of the unknown systems are provided. When the ranking of unknown systems is established, the printout of the known system is provided and discussed.

Introduction The interviewer gives a general introduction to the dependency freshness metric, the goal the metric serves and the type of dependencies that are used.

Awareness and issues Interviewees are asked about their experience with and attitude towards dependency management.

1. Are you aware of any issues – current or past – related to dependency management in the known system?
2. Is the development team responsible for the known system aware of issues arising from bad dependency management?

Metric evaluation The utility of the metric is assessed as well as several of its quality aspects.

3. Do you have an idea about the dependency freshness of the known system?
4. If you were to rate the known system – considering the $\langle 5\%, 30\%, 30\%, 30\%, 5\% \rangle$ distribution – what rating would you give?
5. If you were to rank the following unknown systems from a low to high dependency freshness rating, how would you rank them?

Metric application The rating of the known system is provided and the actionability of the metric is discussed.

6. Does the metric correspond with the perceived dependency freshness of the known system?
7. Do you think this is a valuable metric considering its impact on security, stability and flexibility of the known system?
8. Would this metric translate to an advice to the customer?
9. Would you take any specific action given the printout of the known system?

4.1.2 Results

The results of the interviews are summarized in table 4.1. The answers are formulated in such a way that interview questions are succinctly answered while still providing context. Results for questions four and six are reported as a single answer as questions six is a follow-up on question four. The same applies for question eight and nine.

1. Are you aware of any issues – current or past – related to dependency management in this specific system?	
1.	Yes, we are aware that some of the dependencies in the main system are not up-to-date.
2.	No, not as far as we are aware of.
3.	No, this system has no specific issues as far as I know of.
4.	Yes, this is largely due to strict contracts between client and supplier, and rigid internal processes.
5.	Yes. We've reported this to the client on multiple occasions.

2. Is the development team aware of issues arising from bad dependency management?	
1.	Yes. Someone on the team has a monthly reoccurring task on his calendar. I also keep an eye on this from time-to-time.
2.	I'm not sure, other than that I assume that they try to use the latest versions when they add a new dependency.
3.	No, but that might be only for this system, given its age and use.
4.	Yes, they are certainly aware of these issues, but the supplier is constrained by strict contracts and long change processes at the customer.
5.	Yes, they are aware, but there is a lot of process keeping the development team back.

3. Do you have an idea about the dependency freshness of the system?	
1.	Yes. I have a pretty good idea, mainly because I check this myself from time-to-time.
2.	No, I have no idea given it has not been an issue. But given this is a rather small and new system, I expect a high rating.
3.	The components in this system are widely used, but it's old and not too actively maintained.
4.	This is a reasonably critical system, running on a high-availability platform, so I expect it to be somewhat outdated.
5.	I assume this system scores pretty bad, although they might have improved over the past few years given our advice.

4. If you were to rate the system – considering the $\langle 5\%, 30\%, 30\%, 30\%, 5\% \rangle$ distribution – what rating would you give? and 6. Does the metric correspond with the perceived dependency freshness of the system?	
1.	Prediction: 4, actual: 4.

2.	Prediction: 4, actual: 4.
3.	Prediction: 3, actual: 3.
4.	*Invalid due to protocol error.*
5.	Prediction: 2, actual: 3.

7. Do you think this is a valuable metric considering its impact on security, stability and flexibility of the system?	
1.	I think it shows how fast a system is able to respond to change.
2.	The rating shows something about the professionality of the team. It shows whether they think about these things. It's a good indicator for all these attributes and maybe for performance, too.
3.	I think it's a good indicator for all three attributes.
4.	The metric tells something about how the organization and the development team works.
5.	It serves as a good indicator to quickly check whether something is going wrong with regards to dependency management, which might require further investigation.

8. Would this metric translate to an advice to the customer? and 9. Would you take any specific action given the following printout?	
1.	There are several dependencies we will look into. Mainly Spring dependencies seem to be quite outdated and there are also some that I don't recognize.
2.	I would look into the specific functionality of the different dependencies and see whether something is really important, for instance from a security perspective.
3.	This is certainly something that we could report back to clients. The rating becomes kind of a vehicle to start a conversation.
4.	I would be curious to hear whether this is the result of constraints imposed by the environment this has to run on, or laziness. This would certainly be something I'd mention to the client.
5.	For this system I won't be making any recommendations. This system is already being split up into smaller systems and seems to perform better than expected.

Table 4.1: Summarized answers collected during the interviews with SIG consultants.

Metric ranking

During the interview interviewees were presented with 5 printouts of systems and asked to rank these systems from a low-level to a high-level of dependency freshness. Printouts were fully anonymized; only a unique identifier was displayed on the paper. The results of this test are shown in Table 4.2. Numbers in bold indicate systems ranked differently than the metric's ranking.

System			**Subject**				
#	**Rating**	**Rank**	**1**	**2**	**3**	**4**	**5**
1108	5.053	5	5	5	5	5	5
1994	4.105	4	4	**2**	4	4	4
850	3.248	3	3	3	3	3	3
362	2.188	2	2	**4**	2	2	2
181	1.427	1	1	1	1	1	1

Table 4.2: Systems as ranked during the interviews and rank produced by the dependency freshness metric.

As shown in Table 4.2, all interviewees except for one ranked the systems identical to the metric. This means that the metric can be considered to be of good discriminative quality as the highest and lowest scoring systems are always ranked appropriately. The reason why subject two chose a different ranking could be attributed to an additional weighing factor the interviewee applied to certain dependencies during the ranking process.

4.1.3 Analysis

Several of the technical consultants consider dependency management an issue within the projects they evaluate. This is especially true for older systems with a relative large number of dependencies. Organizations with critical application and rigid processes often impose negative restrictions on dependencies on the (internal) development team, either at the library level or runtime environment.

Some consultants were not aware of any issues. They either attributed the absence of problems to the volume, maturity or environment of the system. Smaller and newer systems may have a higher dependency freshness because when dependencies are added to a system, the most recent version of the dependency is included. Internal systems often have lower security requirements and the system owner thus may find dependency management not too important from a security perspective.

The consultants were able to accurately rate the system, as shown in

question 4 and 6 – in the proximity of one star. This indicates that the rating is quite well aligned with the perceived dependency freshness of the system. It also shows that the version distance is a good mapping from individual dependencies to the system's dependency freshness, confirming the predictive quality of the metric.

Some of the consultants were able to reason about this using their knowledge of the system, for instance the system's age and owner. A few had actually inspected the system's dependencies earlier, by hand.

When the consultants where asked to rank the unknown systems, different issues and questions came up:

- Are dependencies that are used more often throughout the system weighed differently?
- Are dependencies of different types weighed differently? For instance Hibernate (a database abstraction layer) may be more crucial than JUnit (a testing framework).
- How are dependencies on fast release cycles weighed?
- Are transitive dependencies taken into account?

In spite of these concerns, the consultants were able to accurately rank the system, as shown in section 4.1.2.

All interviewees were favorable towards the usefulness of the metric. Some saw the metric as an indicator of the client's or the supplier's work processes, or the quality of their work processes. One consultant elaborated on how the metric served as an indicator of security, stability and flexibility, ultimately agreeing with all three aspects.

Throughout the interviews the importance of good dependency management was stressed, but every interviewee also acknowledged how difficult this really is. "This is a typical problem developers know about, but there is no clear overview of the problem and it only becomes apparent when a new feature has to be implemented and x has to be updated", was noted by one of the interviewees.

The importance of keeping up and having a high level of test coverage was also voiced by several interviewees. Once the freshness starts to slip, it takes quite the effort to get up-to-date again.

After applying the metric on internal systems at SIG, developers spent several days on updating dependencies, even though they were aware of these issues before the rating was computed. This shows that a metric makes these issues more visible. A high level of test coverage made it possible to update dependencies quickly, as failing tests report any issues arising from these updates.

4.1.4 Threats to validity

Determining whether the resulting metric is useful depends on context. This study tried to approach this concept from two different perspectives, namely whether the metric serves as an indicator of quality attributes of the system and whether the metric will translate into advice to the system owners. The author believe that these two perspectives serve the most prevailing use-cases in which this metric may be used.

There are also several confounding variables which may have influenced the results of the interviews. For instance, if technical consultants recently inspected the dependencies of a system manually as part of their regular work, they are better able to predict the dependency freshness of a system. Asking the consultants about different known systems also introduced another variable as systems vary widely in size, age and functionality. Additionally, a confirmation bias may have impacted the answers to the questions regarding the usefulness of the metric.

4.2 Vulnerabilities detection

In the introduction of this study in chapter 1, several system attributes were presented that suffer when a system has a low dependency freshness. Amongst these attributes is the system's security. In order to validate whether this statement is true the relationships between a system's dependency freshness and presence of vulnerabilities in the system is investigated.

In this validation a system security is operationalized by looking at the number of known security vulnerabilities in the system's dependencies. This approach directly measures the effects of having an outdated dependency from a security perspective and may serve as a convincing argument for stakeholders to keep a high level of dependency freshness.

When a vulnerability in a dependency is reported, a new version of the dependency is released fixing this vulnerability. Systems that immediately update the dependency (and thus have a higher dependency freshness) are no longer vulnerable from attacks using this vulnerability. Systems that do not update will remain exploitable and are thus less secure.

4.2.1 Methodology

Determining whether a certain dependency contains a reported security vulnerability is done through a Common Vulnerabilities and Exposures (CVE) system, which describes vulnerabilities reported in software systems [5]. Developers use these systems to spread reported vulnerability to the users of the software. Each vulnerability has a unique CVE-ID, title and description of the issue.

Because CVEs are mostly unstructured the data obtained from the CVE system has to be processed before it can be matched to a specific dependency. In this study data from Cadariu [14] is used in which dependencies found in the Maven.org ecosystem were matched with CVEs using a tool called DependencyCheck[1]. This data was formatted as shown in Listing 4.2 and will be used as a running example.

```
org.apache.wicketwicket6.5.05048478701057026157.txt (cpe:/
    a:apache:wicket:6.5.0) : CVE-2013-2055
```

Listing 4.2: Example of the data obtained from [14] to determine whether a dependency has reported vulnerabilities.

The first part of each line is the path to a temporary file, containing the `groupId`, `artifactId`, version number and random number. The token between parentheses is the name of the matches software package according to the CPE naming convention. The last token refers to the CVE-ID which was matched. It is clear that this data is not formatted in such a way that is can be incorporated in the dependency version dataset directly. The following steps were performed to recreate the fully qualified name of the dependencies in the dataset.

1. Find the longest repeating token in the file path. This would be "wicket" in the example.

2. Split the file path after this token, resulting in the dependency's `groupId` "org.apache.wicket" and `artifactId` "wicket6.5.05048478 701057026157.txt".

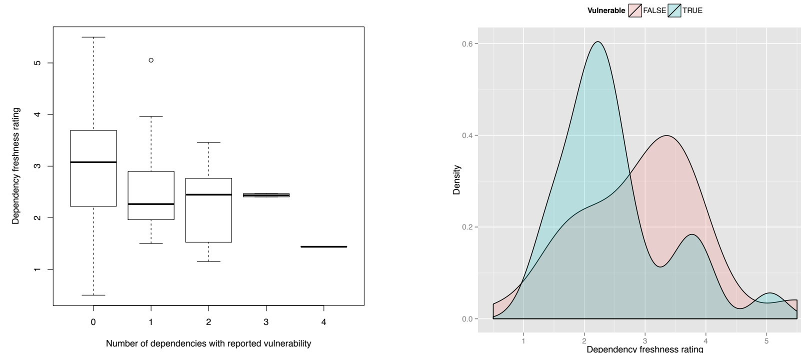
3. Find the version number either from the CPE definition, or perform a pattern match on the file path. In the example the version number can be taken from the CPE definition: "6.5.0".

4. Strip the version number from the `artifactId`, resulting in"wicket".

By using this algorithm 1642 out of 1683 dependency names were recovered. The remaining 41 records were discarded to keep the steps reproducible. Ultimately 339 dependency versions were marked as containing at least one reported security vulnerability. This low number is attributed to a high number of duplicate and mismatches in this used dataset as well as noise in the CVEs themselves. An example of such a mismatch is a non-Java project, matched to a Java project with comparable names.

Finally, the dependency freshness rating of each system in the dataset was calculated, together with the number of dependencies that had one or more reported vulnerabilities.

[1] https://github.com/jeremylong/DependencyCheck



(a) Dependency freshness rating by number of dependencies with reported vulnerabilities.

(b) Dependency freshness rating for systems with and without reported vulnerabilities.

Figure 4.1: Dependency freshness rating and dependency vulnerabilities

4.2.2 Results and analysis

A box plot was created to assess the number of dependencies with reported vulnerabilities per system. This plot is displayed in Figure 4.1 (a). This type of plot allows for analysis of the the distribution of the dependency freshness rating by the number of vulnerable dependencies.

The majority of the systems were found to have no reported vulnerabilities. The data becomes increasingly sparse as the number of vulnerable dependencies increases, which can be expected. Because the dependency freshness metric is calibrated to a $\langle 5\%, 30\%, 30\%, 30\%, 5\% \rangle$ distribution, the majority (90%) of the systems has a rating between 1.5 and 4.5, skewing the distribution, although this is not apparent in the plot.

A relationship between the dependency freshness rating and the number of vulnerable dependencies can be observed in the box plot, but due to the sparsity of the data it is impossible to make statistically significant claims. However, it does indicate that system with a low dependency freshness are likely to have have more reported vulnerabilities in their dependencies than systems with a high dependency freshness. In order to make significant claims about the dataset the results have to be aggregated further.

Figure 4.1 (b) shows the distribution of systems without vulnerable dependencies and with vulnerable dependencies. This plot clearly shows the shift in distribution, which is significantly different (Wilcoxon rank-sum test, $W = 25$, $n = 71$, $P < 0.05$ two-sided). Systems with vulnerable dependencies have a mode of 2.2 and systems without vulnerable dependencies have a mode of 3.3. The mode is used to describe the central tendency of the distributions as they are clearly skewed.

When the data is tabulated (Table 4.3) and grouped by the star rating

Rating	No vulnerable dependencies	Vulnerable dependencies
⋆⋆⋆⋆⋆	2	1
⋆⋆⋆⋆	11	3
⋆⋆⋆	18	4
⋆⋆	11	15
⋆	4	2

Table 4.3: Number of systems with vulnerabilities grouped by star rating.

a tipping point can be observed between two and three stars systems. The distinction that can be made here is that dependencies with three or more stars are considered fresh, while others are considered stale, from a security perspective. A change in non-vulnerable versus vulnerable ratio can also be seen for systems rating two stars, although this can not be observed for one star systems. This can be attributed to the small number of systems in this group.

After making this subdivision the effect size can be expressed through an odds ratio by simple cross-multiplication: $\frac{(2+11+18)/(1+3+4)}{(11+4)/(15+2)} = 4.3$. This means that systems which score less than three stars are more than four times as likely to have vulnerable dependencies.

Systems with vulnerable dependencies and a one or two star rating account for 24% of the systems in the dataset. This group contains 68% of the systems with vulnerable dependencies, again showing the relation between dependency freshness and the possibility of having vulnerable dependencies.

4.2.3 Threats to validity

In this study a correlation was found between the dependency freshness rating of a system and whether this system has a dependency with a security vulnerability. However, this does not imply causation as counter examples can be found in the dataset: systems with a higher dependency freshness can also have issues. This should not be surprising, as newer software can also contain reported security vulnerabilities.

Additionally, the data obtained for determining whether dependencies contained reported vulnerabilities was generated automatically, which has been shown to be difficult [14]. Because the actual names of the dependencies are matched to a known set of systems, there is a high level of confidence that the resulting data set is correct (no false positives), but not complete. For instance, popular Java projects like Spring and Hibernate are not included. It is expected that adding these projects reinforce the conclusions made in our research as they contain several vulnerabilities and are widely used.

4.3 Tracking

When the proposed metric is used to monitor dependency freshness through time it has to display sufficient change. This is called the tracking aspect of a metric, according to the IEEE Standard 1061-1998 [3]. There are two factors which can influence the dependency freshness rating of a system:

1. The dependencies of the system are modified (added, updated, downgraded or removed).
2. A new version of a dependency included in the system is released.

The main concern is that the metric flatlines, meaning that no or too little change in the rating is observed when dependencies are modified or new versions of the dependencies are released. To check whether this is the case a retroactive longitudinal analysis is performed on the dataset of industry systems.

4.3.1 Methodology

The dataset made available by SIG contains systems which are constantly monitored, meaning that system owners deliver updates of their source code at fixed intervals. A version of a system's source code is called a snapshot. The interval between snapshots can range from every week to several months.

For this analysis systems that have fewer than five snapshots are excluded. This reduced the dataset to 50 systems. No filter was imposed on the maximum interval between snapshots. The dependency freshness rating of each individual snapshot was then computed and plotted in Figure 4.2.

Although the snapshots of one system goes back as far as 2007, the line plot is limited to late 2009 when more systems were added to the dataset. Ratings that are clearly erroneous (caused by invalid or incomplete `pom.xml` files) compared to the other ratings of the same system are removed manually. Only 4 out of 1796 ratings were removed.

4.3.2 Results and analysis

To assess the change in rating, the variance of all the ratings belonging to a single system is calculated. These ratings are computed for every snapshot of a system. The variance of the rating for the different systems turns out to be quite low, only 0.04 in a range of 5. The median is chosen to describe the distribution of variance per system because of an apparent power-law distribution. Only nine systems have a variance of 0.2 or greater, meaning that the majority of the systems sees little changes to its dependency freshness rating. When excluding the systems without dependency churn the median variance improves only slightly, to 0.06.

However, sufficient change in ratings seems to be present in the dataset when looking at Figure 4.2. Snapshots that include dependency updates are clearly visible in the line plot, displayed as an ascending line. If a snapshot contains no updated dependencies the line is stable, or slightly declining if new versions of the dependencies used in the system were released.

When observing Figure 4.2, the following types of systems with regards to dependency freshness can be distinguished:

Stable Systems with a stable dependency freshness rating. The system's dependencies see little to no updates.

Improving Systems with an increasing dependency freshness rating. Dependencies are updated faster than they are released.

Declining Systems with an decreasing dependency freshness rating. Dependencies are updated slower than they are released.

This analysis shows an interesting concept, namely that a system's dependency freshness can be characterized using two dimensions: namely the speed at which the dependencies are updated and the speed at which the dependencies are released. In turn this shows that good dependency management (i.e. a high dependency freshness rating) can be achieved by making the correct combinations of these dimensions. Systems which use dependencies with a short release cycling thus require a high update speed in order to achieve a high rating, while systems with dependencies on a long update cycling can get away with a slower update speed. Using more mature (i.e. dependencies on a long update cycle) dependencies will thus require less effort to maintain a certain dependency freshness rating. On the other hand, systems with a dependency freshness rating that has been stable for an extensive period of time might also be an indicator of problems.

4.4 Threats to validity

4.4.1 Construct validity

Data quality

To conduct this study several data sources had to be mined and mixed. The repository of industry systems is curated by a select amount of people, but the systems are of course delivered by a third-party, which may led have led to inconsistencies in the data. An example of this would be a missing `pom.xml` file for a given system snapshot. These snapshot were simply discarded.

The quality of the Maven.org repository is another factor, especially the release date of the dependencies. Issues with the release date inaccuracies

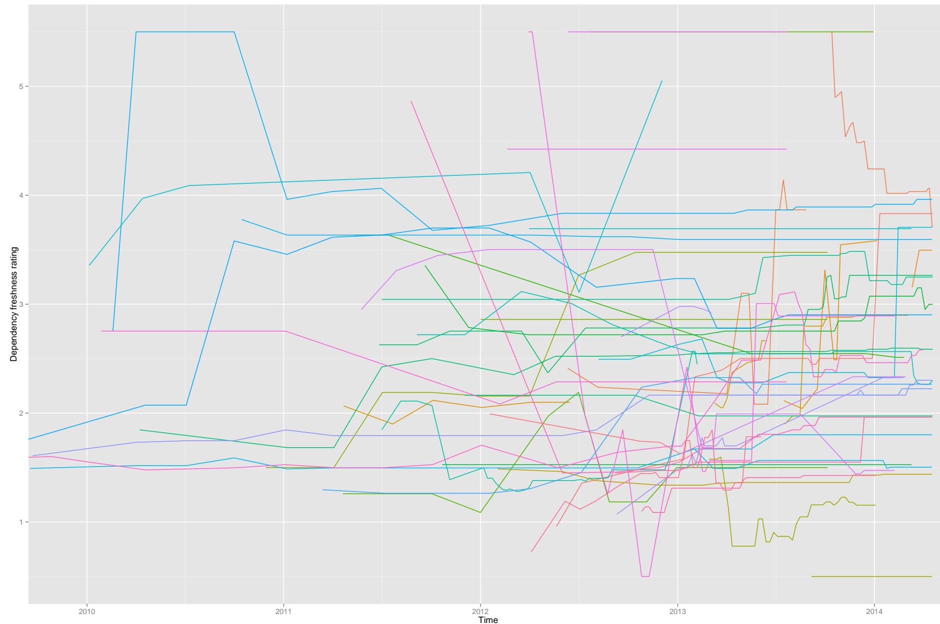


Figure 4.2: Dependency freshness rating per snapshot grouped by systems with more than 5 snapshots.

only arose for older dependency versions. Ultimately, the release date distance measurement was not used for the system-level metric, so no manual cleanup of this data was performed. The release date was used to order version numbers chronologically.

4.4.2 External validity

Generalizability to other systems and technologies

The industry systems used in this study are typically systems which are developed by or developed for larger businesses. This means that the dependency update behavior described in this study may be depicted different than it really is. However, this dataset has been used across a wide body of peer-reviewed research and is used to calibrate the SIG Maintainability Model.

The dataset of systems is also limited to a specific technology stack, namely JVM based systems which used the Maven dependency manager. There is no reason why the approach presented in this research could not be generalized to other version manager using other platforms. However, the requirement of using a dependency manager may impose a bias as well, as the need for dependency management is understood by the development team.

Repeatability

The data that is used to calibrate the metric is not public and obtaining a comparable set of industry systems is difficult. This makes it impossible to reproduce the exact thresholds that were found in this study. However, the methodology used to obtain these thresholds is clearly documented, allowing others to replicate this study using other datasets.

4.5 Discussion

Through the validation studies the quality of the metric can be evaluated using the criteria in section 3.1, based on IEEE Standard 1061-1998 [3]. Both the consistency and reliability criteria are satisfied through the used methodology; the benchmark-based approach assigns higher ratings to systems with a relative large amount of dependencies with a small version distance. The metric was applied on 71 systems, therefore satisfying the reliability criteria.

The discriminative and predictive quality of the metric were validated in the interviews. Technical consultants were able to accurately distinguish between system with a high level and low level of dependency freshness. All but one technical consultant managed to rank five random systems in the same way as the metric. The version distance showed to be good mapping from individual dependencies to the system's dependency freshness.

The tracking and consistency criteria were validated in the longitudinal analysis. In this analysis systems with varying amounts of rating variance were observed. Some systems showed a higher rating variance than others, but this is inherent to the type of dependencies used, as well as the amount of dependency updates performed.

The validation using reported security vulnerabilities showed that systems with a low dependency freshness score are more than four times as likely to contain security issues in these dependencies. This confirmed the relationship between the security quality factor of a system and its dependency freshness rating, although no claims of causation could be made.

All interviewees considered the system-level metric useful for various reasons, also serving as an indicator for several other properties of the system and it's development process.

Chapter 5

Related work

Little research has been done regarding dependency management in software systems according to the best of our knowledge. Recent research at Google acknowledges the importance of proper dependency handling, as it is the most common build problem [41]. This study on dependency freshness hopes to expand upon one of these problems.

The concept of update lag that served as the inspiration for the dependency freshness concept was first described by Raemaekers et al. [37] while researching API stability of popular open source libraries. In a later study it was concluded that this lag is slightly correlated with the amount of change introduced in a new version of a dependency [38].

However, software repository mining in general is an active field of research. It is not only concerned with source code repositories, but also other repositories related to and created by software or software development processes [22]. These include issue trackers as well as runtime logs. Some research tries to interlink these repositories in order to gain a more complete insight in the software development process [20].

An example of this is the work of Strohmaier et al. [43], who compared socially-inferred and technically-inferred networks arising from artifacts from the Eclipse project. This was done by mining both the Eclipse source, as well as the project's Bugzilla issue tracker, taking into account assigned users and those reporting the issues. Our study will also combine data from several repositories, namely a repository of industry systems, the Maven.org dataset and a CVE dataset.

A comprehensive survey of software repository mining research was performed by Kagdi et al. [27], in which a taxonomy is proposed to describe software repository mining research. It also gives an overview of the different types of research conducted, grouped by several categories. According to the categorization provided in the survey, this research is classified as a software metrics study

While the dataset used in this research consist of proprietary systems,

other studies often use open-source software as these are easily accessible and as feature-rich as proprietary systems, which are often harder to obtain access to [22]. The Maven and Java ecosystem has been used in several studies [29, 38, 37, 44], mainly because of its popularity, available tooling and widespread use in industry systems.

A longitudinal study of software evolution has been performed by Businge et al. [13], in which several quality aspects of Eclipse plugins where analyzed. This analysis also involved the changes made to the external dependencies of the plugins, through time. It is not clear whether these dependencies are limited to Eclipse components, or library dependencies in general, like our study on dependency freshness. The change in dependencies was expressed in a churn-like metric, counting the amount of dependencies added and removed.

Research has been done on the API usage of dependencies, for instance to automate the migration between versions of a dependency [29]. The migration patterns between dependencies which provide comparable functionality was also studied, showing how dependencies are swapped within a system when the used dependency is no longer to be found suitable [44]. While this study focusses on migrations, our study focusses on updates to a new version of the same dependency, rather than another dependency.

Mileva et al. [32] performed an analysis of different version of external dependencies so see which version was the most popular. The "wisdom of the crowd" is used to determine the "best" dependency version, categorizing some of the dependency users as earlier adopters or late followers. They also observed the processes of migrating back to an old version of a dependency in case of compatibility issues. As part of this research a tool was released to aid developers in selecting the most appropriate version of a dependency, based on popularity of the dependency in the Maven repository. Rather than using usage statistics, our study assumes that the latest version of a dependency is the most desirable.

The concept of software aging has been studied from several perspectives, most often from a maintenance point of view. The most popular metaphor is that of technical debt, as touched upon in the introduction of our research [15, 28]. An interesting analysis on the OpenBSD codebase showed a positive correlation between software maturity and a software quality aspect: more mature code had fewer reported vulnerabilities [33]. This analysis is comparable to the reported security vulnerability analysis that was performed in chapter 4.

Chapter 6

Conclusion

The main goal of this study is to create a metric to aid stakeholders in deciding on whether a system's dependencies should be updated. The contributions of this thesis are:

- The definition of several measurements to express the distance between two versions of a dependency, including their advantages and disadvantages.
- The analysis of the version distance of dependencies found in a set of industry systems was used to assess the current state of dependency management, using two of the presented measurements.
- The definition of a metric using a benchmark approach, so that a single rating can express the dependency freshness at the system-level. This metric can help stakeholders when making decisions about updating dependencies.
- The validation of this metric, showing the importance and usefulness of such a metric, as well as the relation between low dependency freshness and vulnerable dependencies. The metric also showed sufficient change through time, making it suitable for monitoring systems for a longer period. This validation was based on IEEE Standard 1061-1998 [3].

The main findings include the bad state of dependency management in general. Very few systems have been found to have up-to-date dependencies, which reinforces the need for a metric to describe dependency freshness. No normative data was found to describe what good dependency management should entail, which resulted into creating a benchmark-based metric.

Practitioners voiced the importance of having proper dependency management, but also the challenges for keeping up-to-date. During the validation of the metric the implications of outdated dependencies where presented, which serves as a convincing argument for organizations to maintain a high level of dependency freshness across their systems.

6.1 Application in practice

During this study several use cases for this metric were envisioned and some of these use cases were even brought up during the validation interviews. The different use cases on how this metric can be incorporated in the software engineering process are outlined below:

Inspection The metric is used for a one-time inspection of the system to see whether dependency management is an issue.

Monitoring The system is monitored throughout a period of time to see whether dependency freshness is improving or declining. This can be combined with a goal of keeping the dependency freshness of a system within a certain bandwidth.

Remediation If the dependency freshness of a system is deemed to be insufficient, this can be remedied by using the additional output as shown in section 4.1. The list of outdated dependencies then serves as a guide to improve the dependency freshness of the system.

The metric can thus be incorporated into several places within the software engineering process and can take a corrective and preventive role, either continuously or on a case-by-case basis.

6.2 Future work and opportunities

We have identify several areas of future work, namely:

Metric refinements There are several aspects to our study which can be improved upon, such as taking into account other attributes of the dependencies (section 6.2.1), refinements of the underlying measurements (section 6.2.2) and effort estimation (section 6.2.3).

Metric evaluation Assess whether applying the metric in the software development process improves the dependency freshness of the system (section 6.2.5).

Practical use To apply the current dependency freshness in practice we suggest additional improvements of the used datasets (section 6.2.4).

6.2.1 Dependency context

This thesis only considered the relationship between a dependency and a system as a binary one. The metric does not take into account how tightly a dependency is coupled to the application, or how often it is called. Because

tightly coupled dependencies are harder to update, they present a higher risk to the system than loosely coupled dependencies [35].

Neither does the metric look at properties of the dependency like size or popularity of the dependency across its ecosystem. While this kept the metric simple, applying the same weight to each dependency may be counter intuitive. During the interviews it became apparent that some dependencies are considered to be more important that others. Some dependencies impose a lot of structure on a project (such as frameworks like Spring) and are considered to be more crucial than others.

The same may apply to the systems themselves. Internal systems may be less subject to security threats, but might be more focussed on keeping technical debt low due to its legacy status. This would mean that several profiles for the metric could be created, emphasizing different quality aspects.

Additionally, the type of dependency may also influence the tolerable dependency freshness level. For instance, a unit-testing framework, like JUnit, is less critical to keep up-to-date than an database abstraction layer like Hibernate. The popularity of a dependency is also a factor to consider [36].

Other system attributes to consider when reasoning about dependencies is whether having more dependencies has a higher impact on system freshness than only having a few. It can be argued that having more dependencies introduces more complexity to the system and introduces problems like dependency compatibility. Furthermore, the functional size of a dependency can be factored in. Smaller, more focussed dependencies have a smaller attack vector compared to dependencies which provide a lot of functionality.

6.2.2 Version distance

In section 2.4 several version distance measurements are defined, although ultimately one of these measures is used in the system-level metric. The other measurements that are described may provided additional granularity in version distance, thus improving the overall metric. The version delta measurement becomes especially meaningful when a project adheres to the semantic versioning principle [38]. This measurement can then serve as a lightweight proxy for the public API delta measurement.

6.2.3 Update effort

While the metric gives a good indication of how a system is performing with regards to dependency freshness, it it still hard to put this number into context. As discussed in the introduction, the choice of updating dependencies is a balancing act between effort and risk. Yet, the metric does not give a clear indication what amount of effort has to be put into the system in order to counter the risk expressed in the dependency freshness rating.

This effort estimation is hard for several reasons. The first being that it is hard to assess the effort it takes to rewrite an implementation of an API to a new version of that API. While such a change may seem trivial in retrospect, the amount of effort to perform such a task depends on experience, documentation, test coverage and other variables.

Secondly, without knowing how a dependency is interconnected with the rest of the system, it is hard to asses the overall impact. This is closely related to the future work described in section 6.2.1.

6.2.4 Dataset cleanup

The dataset that was used to perform this research was not cleaned up manually, in order to enhance reproducibility. When applying the metric to real-life systems this might result in a skewed ratings in certain edge cases. The following actions could be taken to address this.

1. Add additional repositories other than the central, public Maven repositories, such as vendor-specific repositories (e.g. repositories by Orcale, IBM, Adobe, etc.).
2. Filter beta and pre-release versions (e.g. 1.2-b2, 3.2RC1).
3. Map dependencies with changed namespaces to the correct version (e.g.`org.apache.commons.lang` to `org.apache.commons.lang3`)
4. Adjust for dependencies which maintain several branches at the same time (e.g. a 2.4 stable branch and 2.5 beta branch).

These clean up actions would make the dataset more usable for day-to-day use.

6.2.5 Impact on software quality

Metrics are often used to monitor and improve the quality of software systems. Now that a metric is defined to monitor the dependency freshness of a project it would be interesting to see whether applying this metric has a positive impact on a system's dependency management.

In the study several preconditions are outlined in to enable a good dependency management process. These preconditions include a high-level of automated testing and agile processes. The relationship between these requirements and good dependency management seems worth investigating.

Bibliography

[1] Tiago L Alves, José Pedro Correia, and Joost Visser. Benchmark-based aggregation of metrics to ratings. In *Software Measurement, 2011 Joint Conference of the 21st International Workshop on and 6th Int'l Conference on Software Process and Product Measurement (IWSM-MENSURA)*, pages 20–29. IEEE, 2011.

[2] Tiago L. Alves, Christiaan Ypma, and Joost Visser. Deriving metric thresholds from benchmark data. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–10. IEEE, 2010.

[3] IEEE Standards Association et al. IEEE Std 1061–1998 IEEE Standard for a Software Quality Metrics Methodology, 1998.

[4] Robert Baggen, José Pedro Correia, Katrin Schill, and Joost Visser. Standardized code quality benchmarking for improving software maintainability. *Software Quality Journal*, 20(2):287–307, 2012.

[5] David W Baker, Steven M. Christey, William H Hill, and David E Mann. The development of a common enumeration of vulnerabilities and exposures. In *Recent Advances in Intrusion Detection*, volume 7, page 9, 1999.

[6] Victor R. Basili and Barry W. Boehm. COTS-based systems top 10 list. *Computer*, 34(5):91–95, May 2001.

[7] Victor R. Basili, Gianluigi Caldiera, and H Dieter Rombach. The goal question metric approach. *Encyclopedia of software engineering*, 2(1994):528–532, 1994.

[8] Meriem Belguidoum and Fabien Dagnat. Dependency management in software component deployment. *Electronic Notes in Theoretical Computer Science*, 182(0):17 – 32, 2007. Proceedings of the Third International Workshop on Formal Aspects of Component Software (FACS 2006).

[9] John Adrian Bondy and Uppaluri Siva Ramachandra Murty. *Graph theory with applications*, volume 6. Macmillan London, 1976.

[10] Eric Bouwers, Arie van Deursen, and Joost Visser. Dependency profiles for software architecture evaluations. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pages 540–543. IEEE, 2011.

[11] Eric Bouwers, Joost Visser, and Arie Van Deursen. Towards a catalog format for software metrics. In *Proceedings of the 5th International Workshop on Emerging Trends in Software Metrics*, pages 44–47. ACM, 2014.

[12] Daniel Burrows. Modelling and resolving software dependencies. In *Conferenza Italiana sul Software Libero (CONFSL 2010)*, 2005.

[13] John Businge, Alexander Serebrenik, and Mark van den Brand. An empirical study of the evolution of eclipse third-party plug-ins. In *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IW-PSE)*, IWPSE-EVOL '10, pages 63–72, New York, NY, USA, 2010. ACM.

[14] Mircea Cadariu. Tracking vulnerable components in software systems. Master's thesis, Delft University of Technology, 2014.

[15] Ward Cunningham. The WyCash portfolio management system. In *ACM SIGPLAN OOPS Messenger*, pages 29–30. ACM, 1992.

[16] Roberto Di Cosmo, Stefano Zacchiroli, and Paulo Trezentos. Package upgrades in foss distributions: Details and challenges. In *Proceedings of the 1st International Workshop on Hot Topics in Software Upgrades*, page 7. ACM, 2008.

[17] Justin R Erenkrantz. Release management within open source projects. *Proceedings of the 3rd Open Source Software Development Workshop*, pages 51–55, 2003.

[18] Norman E. Fenton and Martin Neil. Software metrics: successes, failures and new directions. *Journal of Systems and Software*, 47(2):149–157, 1999.

[19] Norman E. Fenton and Shari Lawrence Pfleeger. *Software metrics: a rigorous and practical approach.* PWS Publishing Co., 1998.

[20] Michael Fischer, Martin Pinzger, and Harald Gall. Populating a release history database from version control and bug tracking systems. In *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pages 23–32. IEEE, 2003.

[21] Rishab Aiyer Ghosh. Clustering and dependencies in free/open source software development: Methodology and tools. *First Monday*, 8(4), 2003.

[22] Ahmed E. Hassan. The road ahead for mining software repositories. In *Frontiers of Software Maintenance, 2008.*, pages 48–57. IEEE, 2008.

[23] Ilja Heitlager, Tobias Kuipers, and Joost Visser. A practical model for measuring maintainability. In *Quality of Information and Communications Technology, 2007. QUATIC 2007. 6th International Conference on the*, pages 30–39. IEEE, 2007.

[24] Siw Elisabeth Hove and Bente Anda. Experiences from conducting semi-structured interviews in empirical software engineering research. In *Software Metrics, 2005. 11th IEEE International Symposium*, pages 10–pp. IEEE, 2005.

[25] Jez Humble and David Farley. *Continuous delivery: reliable software releases through build, test, and deployment automation.* Pearson Education, 2010.

[26] Capers Jones. *Applied software measurement: global analysis of productivity and quality*, volume 3. McGraw-Hill New York, 2008.

[27] Huzefa Kagdi, Michael L. Collard, and Jonathan I Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, 19(2):77–131, 2007.

[28] Philippe Kruchten, Robert L Nord, and Ipek Ozkaya. Technical debt: From metaphor to theory and practice. *IEEE Software*, 29(6), 2012.

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

[30] Thomas J. McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, SE-2(4):308–320, Dec 1976.

[31] Andrew Meneely, Ben Smith, and Laurie Williams. Validating software metrics: A spectrum of philosophies. *ACM Transactions on Software Engineering and Methodology*, 21(4):24:1–24:28, February 2013.

[32] Yana Momchilova Mileva, Valentin Dallmeier, Martin Burger, and Andreas Zeller. Mining trends of library usage. In *Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*, pages 57–62. ACM, 2009.

[33] Andy Ozment and Stuart E Schechter. Milk or wine: does software security improve with age? In *Usenix Security*, 2006.

[34] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[35] Steven Raemaekers, Arie van Deursen, and Joost Visser. Exploring risks in the usage of third-party libraries. In *BENEVOL 2011 (10th BElgian-NEtherlands software eVOLution)*, 2011.

[36] Steven Raemaekers, Arie van Deursen, and Joost Visser. An analysis of dependence on third-party libraries in open source and proprietary systems. In *Sixth International Workshop on Software Quality and Maintainability, SQM*, volume 12, 2012.

[37] Steven Raemaekers, Arie van Deursen, and Joost Visser. Measuring software library stability through historical version analysis. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 378–387. IEEE, 2012.

[38] Steven Raemaekers, Arie van Deursen, and Joost Visser. Semantic versioning versus breaking changes: A study of the Maven repository. In *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*, 2014.

[39] Donald J. Reifer, Victor R. Basili, Barry W. Boehm, and Betsy Clark. Eight lessons learned during COTS-based systems maintenance. *IEEE Software*, 20(5):94–96, 2003.

[40] John M Roche. Software metrics and measurement principles. *ACM SIGSOFT Software Engineering Notes*, 19(1):77–85, 1994.

[41] Hyunmin Seo, Caitlin Sadowski, Sebastian G Elbaum, Edward Aftandilian, and Robert W Bowdidge. Programmers' build errors: a case study (at Google). In *International Conference on Software Engineering*, pages 724–734, 2014.

[42] Net Market Share. Desktop operating system market share. http://www.netmarketshare.com.

[43] Markus Strohmaier, Michel Wermelinger, and Yijun Yu. Using network properties to study congruence of software dependencies and maintenance activities in Eclipse. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.210.1888.

[44] Cédric Teyton, Jean-Rémy Falleri, Marc Palyart, and Xavier Blanc. A study of library migration in Java software. *arXiv preprint arXiv:1306.6262*, 2013.

Appendix A: Software metric catalog format

Name:	Version release date distance	VRDD	**Level:**	Base
Entity:	Software project		**Type:**	Internal
Attribute:	Age		**Range:**	$[0, \infty]$
Definition:	Count the days between two releases of a software project		**Expected value:**	> 0
Rationale (theoretical):			**Variability:**	
The amount of days between two releases expresses how many days a release is lagging on another release.			**Scale type:**	Ratio
			Related metrics:	
			Version sequence number distance	Correlated
Implications (practical):			Version number delta	Related
			Public API delta	Related
			Dependency freshness	Used by
			Validation:	
Applicable in context:			*Measuring dependency freshness in software systems*, Cox	Introduction, case study
When comparing two releases of software project with explicit, dated releases.				
Solution strategies:		**Solution type:**		

Name:	Version sequence number distance	VSND	**Level:**	Base
Entity:	Software project		**Type:**	Internal
Attribute:	Age		**Range:**	$[0, \infty]$
Definition:	Count the number of releases between two releases of a software project		**Expected value:**	> 0
Rationale (theoretical):			**Variability:**	
The number of releases between two releases expresses how many releases a release is lagging on another release.			**Scale type:**	Ratio
			Related metrics:	
			Version release date distance	Correlated
Implications (practical):			Version number delta	Related
			Public API delta	Related
			Validation:	
Applicable in context:			*Measuring dependency freshness in software systems*, Cox	Introduction, case study
When comparing two releases of software project with explicit releases.				
Solution strategies:		**Solution type:**		

Name:	Version number delta	VND	**Level:**	Base
Entity:	Software project		**Type:**	Internal
Attribute:	Age		**Range:**	$([0, \infty], [0, \infty], [0, \infty])$
Definition:	The cumulative difference of the most significant change version number between every successive release.		**Expected value:**	$(> 0, > 0, > 0)$
Rationale (theoretical):			**Variability:**	
The number of major, minor and patch releases between two releases expresses how many releases a release is lagging on another release.			**Scale type:**	Ratio
			Related metrics:	
			Version release date distance	Related
Implications (practical):			Version sequence number distance	Related
			Public API delta	Related
			Validation:	
Applicable in context:			*Measuring dependency freshness in software systems*, Cox	Introduction, case study
When comparing two releases of software project using extends numbers systems for version numbers.				
Solution strategies:		**Solution type:**		

Name:	Public API delta	PAD	**Level:**	Base
Entity:	Software project		**Type:**	Internal
Attribute:	Age		**Range:**	$[0, \infty]$
Definition:	Count the number of breaking changes to the public API between two releases.		**Expected value:**	
Rationale (theoretical):			**Variability:**	
The number of breaking changes between two releases is an indicator of the functional change.			**Scale type:**	Ratio
			Related metrics:	
			Version release date distance	Related
Implications (practical):			Version sequence number distance	Related
			Version number delta	Related
			Validation:	
Applicable in context:			*Measuring dependency freshness in software systems*, Cox	Case study
When comparing two releases of software project with explicitly declared public APIs.			*Semantic versioning in Practice*, Raemaekers et al.	Introduction, Empirical validation
Solution strategies:		**Solution type:**		
Add new interfaces rather than change existing ones.		Treating		

Name:	Dependency freshness	DF	**Level:**	Derived
Entity:	Software system		**Type:**	Internal
Attribute:	Dependency freshness		**Range:**	$[0.5, 5.5]$
Definition:	Assign components to risk categories based on their version sequence number distance. Risk profiles are then transformed to a rating based on a benchmark dataset.		**Expected value:**	$1.5 < x < 4.5$
Rationale (theoretical):			**Variability:**	
Having a large amount of software components with a low version sequence number distances is more desirable than a few software components with a very high version sequence number distance.			**Scale type:**	Interval
			Related metrics:	
			Version sequence number distance	Base metric
Implications (practical):				
It is undesirable to have a software system with outdated third-party components. Components should be up-to-date to attain a high level of dependency freshness.				
			Validation:	
Applicable in context:			*Measuring dependency freshness in software systems*, Cox	Introduction, empirical validation
Software projects using software components developed by a third-party.				
Solution strategies:		**Solution type:**		
Update third-party components to their latest versions		Solving		