An Empirical Evaluation of OSGi Dependencies Best Practices in the Eclipse IDE

Lina Ochoa
Centrum Wiskunde & Informatica
Amsterdam, Netherlands
lina.ochoa@cwi.nl

Thomas Degueule
Centrum Wiskunde & Informatica
Amsterdam, Netherlands
thomas.degueule@cwi.nl

Jurgen Vinju
Centrum Wiskunde & Informatica
Amsterdam, Netherlands
Eindhoven University of Technology
Eindhoven, Netherlands
jurgen.vinju@cwi.nl

ABSTRACT

OSGi is a module system and service framework that aims to fill Java's lack of support for modular development. Using OSGi, developers divide software into multiple *bundles* that declare constrained dependencies towards other bundles. However, there are various ways of declaring and managing such dependencies, and it can be confusing for developers to choose one over another. Over the course of time, experts and practitioners have defined "best practices" related to dependency management in OSGi. The underlying assumptions are that these best practices (i) are indeed relevant and (ii) help to keep OSGi systems manageable and efficient. In this paper, we investigate these assumptions by first conducting a systematic review of the best practices related to dependency management issued by the OSGi Alliance and OSGi-endorsed organizations. Using a large corpus of OSGi bundles (1,124 core plug-ins of the Eclipse IDE), we then analyze the use and impact of 6 selected best practices. Our results show that the selected best practices are not widely followed in practice. Besides, we observe that following them strictly reduces classpath size of individual bundles by up to 23% and results in up to ±13% impact on performance at bundle resolution time. In summary, this paper contributes an initial empirical validation of industry-standard OSGi best practices. Our results should influence practitioners especially, by providing evidence of the impact of these best practices in real-world systems.

CCS CONCEPTS

• **Software and its engineering** → *Software configuration management and version control systems*; *Software libraries and repositories*;

ACM Reference Format:
Lina Ochoa, Thomas Degueule, and Jurgen Vinju. 2018. An Empirical Evaluation of OSGi Dependencies Best Practices in the Eclipse IDE. In *Proceedings of MSR '18: 15th International Conference on Mining Software Repositories (MSR '18).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3196398.3196416

MSR '18, May 28–29, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5716-6/18/05…$15.00
https://doi.org/10.1145/3196398.3196416

1 INTRODUCTION

The time-honored principle of separation of concerns entails splitting the development of complex systems into multiple components interacting through well-defined interfaces. This way, the development of a system can be broken down into multiple, smaller parts that can be implemented and tested independently. This also fosters reuse by allowing software components to be reused from one system to the other, or even to be substituted by one another provided that they satisfy the appropriate interface expected by a client. Three crucial aspects [33] of successful separation of concerns are module interfaces, module dependencies, and information hiding—a module's interface hides any number of different functionalities, possibly depending on other modules transitively.

Historically, the Java programming language did not offer any built-in support for the definition of versioned modules with explicit dependency management [41]. This led to the emergence of OSGi, a module system and service framework for Java standardized by the OSGi Alliance organization [42]. Initially, one of the primary goals of OSGi was to fill the lack of proper support for modular development in the Java ecosystem (popularly known as the "JAR hell"). OSGi rapidly gained popularity and, as of today, numerous popular software of the Java ecosystem, including IDEs (e.g., Eclipse, IntelliJ), application servers (e.g., JBoss, GlassFish), and application frameworks (e.g., Spring) rely internally on the modularity capabilities provided by OSGi.

Just like any other technology, it may be hard for newcomers to grasp the complexity of OSGi. The OSGi specification describes several distinct mechanisms to declare dependencies, each with different resolution and wiring policies. Should dependencies be declared at the package level or the component level? Can the content of a package be split amongst several components or should it be localized in a single one? These are questions that naturally arise when attempting to modularize Java applications with OSGi. There is little tool support to help writing the meta-data files that wire the components together, and so modularity design decisions are mostly made by the developers themselves. The quality of this meta-data influences the modularity aspects of OSGi systems. The reason is that OSGi's configurable semantics directly influences all the aforementioned key aspects of modularity: the definition of module interfaces, what a dependency means (wiring), and information hiding (e.g., transitive dependencies). A conventional approach to try and avoid such issues is the application of so-called "best practices" advised by experts in the field. To the best of our knowledge, the assumptions underlying this advice have not been

investigated before: are they indeed relevant and do they have a positive effect on OSGi-based systems? Our research questions are:

- **Q1** What OSGi best practices are advised?
- **Q2** Are OSGi best practices being followed?
- **Q3** Does each OSGi best practice have an observable effect on the relevant qualitative properties of an OSGi bundle?

To begin answering these questions, this paper reports on the following contributions:

- A systematic review of best practices for dependency management in OSGi emerging from either the OSGi Alliance itself or OSGi-endorsed partners; we identify 11 best practices and detail the rationale behind them (**Q1**);
- An analysis of the bytecode and meta-data of a representative corpus of OSGi bundles (1,124 core plug-ins of the Eclipse IDE) to determine whether best practices are being followed (**Q2**), and what is their impact (**Q3**).

Our results show that:

- *Best practices are not widely followed in practice.* For instance, half of the bundles we analyze specify dependencies at the bundle level rather than at the package level—despite the fact that best practices encourage to declare dependencies at the package level;
- *The lack of consideration for best practices does not significantly impact the performance of OSGi-based systems.* Strictly following the suggested best practices reduces classpath size of individual bundles by up to 23% and results in up to ±13% impact on performance at bundle resolution time.

The remainder of this paper is structured as follows. In Section 2, we introduce background notions on OSGi itself. In Section 3, we detail the methodology of the systematic review from which we extract a set of best practices related to dependencies management. In Section 4, we evaluate whether best practices are being followed on a representative corpus of OSGi bundles extracted from the Eclipse IDE. We discuss related work in Section 5 and conclude in Section 6.

2 BACKGROUND: THE OSGI FRAMEWORK

OSGi is a module system and service framework for the Java programming language standardized by the OSGi Alliance organization [42], which aims at filling the lack of support for modular development with explicit dependencies in the Java ecosystem (aka. the "JAR hell"). Some of the ideas that emerged in OSGi were later incorporated in the Java standard itself, e.g., as part of the module system released with Java 9. In OSGi, the primary unit of modularization is a *bundle*. A bundle is a cohesive set of Java packages and classes (and possibly other arbitrary resources) that together provide some meaningful functionality to other bundles. A bundle is typically deployed in the form of a Java archive file (JAR) that embeds a *Manifest file* describing its content, its meta-data (e.g., version, platform requirements, execution environment), and its dependencies towards other bundles. The OSGi framework itself is responsible for managing the life cycle of bundles (e.g., installation, startup, pausing). As of today, several certified implementations of the OSGi specification have been defined, including Eclipse Equinox[1] and Apache Felix[2] to name but a few. OSGi is a mature framework that comprises many aspects ranging from module definition and service discovery to life cycle and security management. In this paper, we focus specifically on its support for dependencies management.

Listing 1: An idiomatic `MANIFEST.MF` file

```
Bundle-ManifestVersion: 2
Bundle-Name: Dummy
Bundle-SymbolicName: a.dummy
Bundle-Version: 0.2.1.build-21
Bundle-RequiredExecutionEnvironment: JavaSE-1.8
Export-Package: a.dummy.p1,
  a.dummy.p2;version="0.2.0"
Import-Package: b.p1;version="[1.11,1.13]",
  c.p1
Require-Bundle: d.bundle;bundle-version:="3.4.1",
  e.bundle;resolution:=optional
```

2.1 The Manifest File

Every bundle contains a meta-data file located in META-INF/-MANIFEST.MF. This file contains a list of standardized key-value pairs (known as *headers*) that are interpreted by the framework to ensure all requirements of the bundle are met. Listing 1 depicts an idiomatic Manifest file for an imaginary bundle named `Dummy`.

In this simple example, the Manifest file declares the bundle `a.Dummy` in its version `0.2.1.build-21`. It requires the execution environment `JavaSE-1.8`. The main purpose of this header is to announce what should be available to the bundle in the standard `java.*` namespace, as the exact content may vary according to the version and the implementer of the Java virtual machine on which the framework runs. The Manifest file specifies that the bundle exports the `a.dummy.p1` package, and the `a.dummy.p2` package in version `0.2.0`. These packages form the public interface of the bundle—its API. Next, the Manifest file specifies that the bundle requires the package `b.p1` in version `1.11` to `1.13` (inclusive) and the package `c.p1`. Finally, the Manifest declares a dependency towards the bundle `d.bundle` in version `3.4.1` and an optional dependency towards the bundle `e.bundle`. We dive into greater details of the semantics of these headers and attributes in the next section.

It is important to note that the Manifest file is typically written by the bundle's developer herself, and has to co-evolve with its implementation. Therefore, discrepancies between what is declared in the Manifest and what is actually required by the bundle at the source or bytecode level may arise. Although some tools provide assistance to the developers (for instance using bytecode analysis techniques on bundles to automatically infer the appropriate dependencies), getting the Manifest right remains a tedious and error-prone task.

2.2 OSGi Dependencies Management

The OSGi specification declares 28 Manifest headers that relate to versioning, i18n, dependencies, capabilities, etc. Amongst them, six are of particular interest regarding dependencies management: `Bundle-SymbolicName` which "*together with a version must identify a unique bundle*", `Bundle-Version` which "*specifies the version of this*

[1] https://www.eclipse.org/equinox/

[2] https://felix.apache.org/

bundle", `DynamicImport-Package` which "*contains a comma-separated list of package names that should be dynamically imported when needed*", `Export-Package` which "*contains a declaration of exported packages*", `Import-Package` which "*declares the imported packages for this bundle*", and `Require-Bundle` which "*specifies that all exported packages from another bundle must be imported, effectively requiring the public interface of another bundle*" [42]. The OSGi specification prescribes two distinct mechanisms for declaring dependencies: at the package level, or at the bundle level. In the former case, it is the responsibility of the framework to figure out which bundle provides the required package—multiple bundles can export the same package in the same version. Conversely, the latter explicitly creates a strong dependency link between the two bundles.

The `Import-Package` header consists of a list of comma-separated packages the declaring bundle depends on. Each package in the list accepts an optional list of attributes that affects the way packages are resolved. The `resolution` attribute accepts the values `mandatory` (default) and `optional`, which indicate, respectively, that the package must be resolved for the bundle to load, or that the package is optional and will not affect the resolution of the requiring bundle. The `version` attribute restricts the resolution on a given version range, as shown in Listing 1.

When it requires another bundle through the `Require-Bundle` header, a bundle imports not only a single package but the whole public interface of another bundle, i.e., the set of its exported packages. As the `Require-Bundle` header requires to declare the symbolic name of another bundle explicitly, this creates a strong dependency link between both. Thus, not only does this header operate on a coarse-grained unit of modularization, but it also tightly couples the components together.

For a bundle to be successfully resolved, all the packages it imports must be exported (`Export-Package`) by some other bundle known to the framework, with their versions matching. Similarly, all the bundles it requires must be known to the framework, with their versions matching. This wiring process is carried out automatically by the framework as the bundles are loaded.

3 OSGI BEST PRACTICES

The OSGi specification covers numerous topics in depth and it can be hard for developers to infer idiomatic uses and good practices. Should dependencies be declared at the package or the bundle level? Can the content of a package be split amongst several bundles or should it be localized in a single one? These are questions that naturally arise when attempting to modularize Java applications with OSGi. Although all usages are valid according to the specification, OSGi experts tend to recommend or discourage some of them. In this section, we intend to identify a set of best practices in the use of OSGi. In particular, we look for best practices related to the specification of dependencies between bundles, thus answering our first research question:

Q1 What OSGi best practices are advised?

3.1 Systematic Review Methodology

To perform the identification of best practices related to OSGi dependencies management, we follow the guidelines specified by Kitchenham et al. [26], which include the definition of the research question, search process, study selection, data extraction, and search results. In this regards, **Q1** is selected as the *research question* of the systematic review.

3.1.1 Search process. Given the absence of peer-reviewed research tackling OSGi best practices (cf. Section 5), we select as primary data sources web resources of the OSGi Alliance and OSGi-endorsed products. The complete list of certified products[3] corresponds to *Knopflerfish*, *ProSyst Software*, *SuperJ Engine*, *Apache Felix*, *Eclipse Equinox*, *Samsung OSGi*, and *KT OSGi Service Platform (KOSP)*. With the aim to identify best practices, we define a *search string* that targets a set of standard *best practices* synonyms, and their corresponding antonyms:

```
((good OR bad OR best) AND (practices OR design)) OR smell
```

Some of the official web pages of the selected organizations provide their own search functionality. However, we seek to minimize the heterogeneous conditions of the searching environment and only use *Google Search* to explore the set of web resources. We use *JSoup*, an HTML parser for Java, to execute the search queries and to scrap the results. We compute all possible keyword combinations from the original search string and execute one query per combination and organization domain. For instance, to search for the `best AND practices` keywords in English-written resources on the OSGi Alliance domain, we define the following Google Search query: `http://www.google.com/search?q=best+practices+site:www.osgi.org&domains:www.osgi.org&hl=en`. We retrieved the resources in January 2018.

3.1.2 Study selection. Figure 1 details the resource selection process we follow in this study. First, we only include web resources written in English in the review. As shown before in the Google Search query, this language restriction is included as a filtering option in all searches: `hl=en`. In the end, the search engine returns a total of 268 resources.[4] Second, selected documents should describe best practices related to the management of dependencies in OSGi. To this aim, we conduct a two-task selection where we first consider the occurrences of keywords in the candidate resources, and then we perform a manual selection of relevant documents. On the one hand, we count the occurrences of the searched keywords in each web resource (including HTML, XML, PDF, and PPT files). If one of the keywords is missing in the resource, we automatically discard it. Using this criterion, we reduce the set to 156 resources, and finally 87 after removing duplicates. On the other hand, we manually review the resulting set, looking for documents that address the research question. In particular, if a resource points to another document (through an HTML link) that is not part of the original set of candidates, it is also analyzed and, if it is relevant to the study, it is included as part of our data sources. This task is performed by two reviewers to minimize selection bias. In the end, we select 21 web resources to derive the list of best practices related to OSGi dependencies specification. Some of the OSGi-endorsed organizations do not provide relevant information for the study.

3.1.3 Data extraction. During the data extraction phase, we consider the organization that owns the resource (e.g., OSGi Alliance),

[3] https://www.osgi.org/osgi-compliance/osgi-certification/osgi-certified-products/.
[4] https://github.com/msr18-osgi-artifacts/msr18-osgi-artifacts

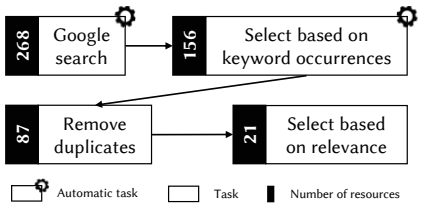

Figure 1: Resources selection of the systematic review.

its title, year of publication, authors, and the targeted best practices. To have a common set of best practices, one reviewer reads the selected resources and groups the obtained results in 11 best practices. Afterwards, two reviewers check which best practices are suggested per web resource. Table 1 presents the results of the review. The best practices labels in the table correspond to the best practices presented in Section 3.2.

3.2 Dependencies Specification Best Practices

In this section, we review the best practices identified and summarized in Table 1. We elaborate on the rationale behind each best practice using peer-reviewed research articles and the *OSGi Core Specification Release 6* [42].

3.2.1 Prefer package-level dependencies [B1]. Dependencies should be declared using the `Import-Package` header instead of using the `Require-Bundle` header. The latter creates a tight coupling between the requiring bundle and the required bundle, which is an implicit dependency towards an implementation rather than an interface. Thus, it impacts the flexibility of dependency resolution, as the resolver has only one source to provide the dependency (i.e., the required bundle itself). This also naturally complicates refactoring activities: moving a package from one bundle to the other requires to patch all bundles depending on it to point to the new bundle. In contrast, the `Import-Package` header only relies on an interface and various bundles may offer the corresponding package. Finally, `Require-Bundle` automatically imports all the exported packages of the required bundle, which may introduce unnecessary dependencies. This can get worse in some cases, since package shadowing can be introduced unwittingly [42].

3.2.2 Use versions when possible [B2]. Versions should be set when requiring bundles, or when importing or exporting packages. When a bundle requires another bundle or imports a package, a version range or a minimum required version can be defined. Versions must be consciously used to control the dependencies of a bundle, avoiding the acceptance of new versions that might break the component. Version ranges are preferred over minimum versions, because both upper and lower bounds, as well as all in between versions, are supposed to be tested and considered by bundle developers [9]. In addition, with version ranges the dependency resolver has fewer alternatives to resolve the given requirements, allegedly speeding up the process.

3.2.3 Export only needed packages [B3]. Only the packages that may be required by other bundles should be exported. Internal and implementation packages should be kept hidden. Because the set of exported packages forms the public API of a bundle, changes in these packages should be accounted for by the clients [8]. Consequently, the more packages are exported, the more effort is required to maintain and evolve the corresponding API.

3.2.4 Minimize dependencies [B4]. Unnecessary dependencies should be avoided, given their known impact on failure-proneness [6] and performance of the resolution process. In the case of OSGi framework and the employment of the `Require-Bundle` header, a required bundle might depend on other bundles. If these transitive dependencies are not considered in the OSGi environment, then the requiring bundle may not be resolved [42]. Moreover, dependencies specification in `Require-Bundle` and `Import-Package` headers may impact performance during the resolution process of the OSGi environment. A bundle is resolved if all its dependencies are available [42]. Presumably, the more dependencies are added to the Manifest file, the longer the framework will take to start and resolve the bundle assuming that all dependencies are included in the environment.

3.2.5 Import all needed packages [B5]. All the external packages required by a bundle must be specified in the `Import-Package` header. If this is not the case, a `ClassNotFoundException` may be thrown when there is a reference to a class of an unimported package [42]. This also applies to dynamic dependencies, e.g., classes that are dynamically loaded using the reflective API of Java. The only packages that are automatically available to any bundle are the ones defined in the namespace `java.*`, which are offered by the selected execution environment. However, this environment can offer other packages included in other namespaces. Thus, if these packages are not explicitly imported and the execution environment is modified, they will become unavailable and the bundle will not get resolved.

3.2.6 Avoid `DynamicImport-Package` [B6]. This header lists a set of packages that may be imported at runtime after the bundle has reached a resolved state. In this case, dependency resolution failures may appear in later stages in the life cycle of the system and are harder to diagnose. This effectively hurts the *fail fast* idiom adopted by the OSGi framework [40]. Also, the `DynamicImport-Package` creates an overhead due to the need to dynamically resolve packages every time a dynamic class is used [42].

3.2.7 Separate implementation, API, and OSGi-specific packages [B7]. It is highly recommended to separate API packages from both implementation and OSGi-specific packages. Therefore, many implementation bundles can be provided for a given API, favoring system modularity. The OSGi service registry is offered to select an implementation once a bundle is requiring and using the associated API packages. With this approach, API packages can be easily exported in isolation from implementation packages, allowing a change of implementation if needed. Moreover, implementation changes that result in breaking changes for clients bundles are avoided. The abovementioned APIs are known as *clean APIs*, i.e., exported packages that do not use OSGi, internal, or implementation packages in a given bundle [42].

3.2.8 Use semantic versioning [B8]. Semantic versioning[5] is a version naming scheme that aims at reducing risks when upgrading dependencies. This goal is achieved by providing concrete

[5] http://semver.org/

Table 1: Systematic review of OSGi dependencies specification best practices.

	Resource	Year	Author(s)	Best practices										
				B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	B11
OSGi Alliance	Automatically managing service dependencies in OSGi [31]	2005	M. Offermans	○	○	○	○	○	●	○	○	○	○	○
	OSGi best practices! [17]	2007	B.J. Hargrave et al.	●	●	○	●	●	○	●	○	○	○	○
	Very important bundles [36]	2009	R. Roelofsen	●	○	○	○	○	●	○	○	○	●	○
	OSGi: the best tool in your embedded systems toolbox [16]	2009	B. Hackleman et al.	●	○	○	○	○	○	●	○	○	○	○
	bndtools: mostly painless tools for OSGi [5]	2010	N. Bartlett et al.	○	○	●	○	●	○	○	●	○	○	○
	Developing OSGi enterprise applications [4]	2010	R. Barci et al.	○	○	○	○	○	○	●	○	○	○	○
	Experiences with OSGi in industrial applications [10]	2010	B. Dorninger	○	○	○	○	○	○	●	○	○	○	○
	Migration from Java EE application server to server-side OSGi for process management and event handling [23]	2010	G. Kachel et al.	○	○	●	○	○	○	○	○	○	●	○
	10 Things to know you are doing OSGi in the wrong way [30]	2011	J. Moliere	●	●	○	○	○	○	●	●	○	○	○
	Structuring software systems with OSGi [13]	2011	U. Fildebrandt	●	●	○	○	○	○	●	○	○	○	○
	Best practices for (enterprise) OSGi applications [44]	2012	T. Ward	●	●	●	●	○	○	○	●	●	○	○
	Building a modular server platform with OSGi [19]	2012	D. Jayakody	●	●	●	○	○	○	●	●	○	○	○
	OSGi application best practices [22]	2012	E. Jiang	●	●	○	○	○	○	●	●	●	○	○
	TRESOR: the modular cloud - Building a domain specific cloud platform with OSGi [15]	2013	A. Grzesik	○	○	○	○	○	○	●	○	○	○	○
	Guidelines [2]	n.d.	OSGi Alliance	○	○	○	○	○	○	●	○	○	○	○
	OSGi developer certification - Professional [3]	n.d.	OSGi Alliance	○	○	○	○	○	●	○	○	○	○	○
Felix	Using Apache Felix: OSGi best practices [32]	2006	M. Offermans	●	○	○	●	○	○	●	○	○	○	○
	OSGi frequently asked questions [11]	2013	Apache Felix	○	○	○	○	○	○	●	○	○	○	●
	Dependency manager - Background [12]	2015	Apache Felix	●	○	○	○	○	○	●	○	○	○	○
Equinox	Best practices for programming Eclipse and OSGi [18]	2006	B.J. Hargrave et al.	●	○	○	○	○	○	○	○	○	○	○
	OSGi component programming [45]	2006	T. Watson et al.	●	○	○	○	○	○	●	○	○	○	○

rules and conventions to label breaking and non-breaking software changes [35]. Following these rules, a version number should be defined as `major.minor.micro`. In some cases, the version number is extended with one more alphanumerical slot known as `qualifier`. The `major` number is used when incompatible changes are introduced to the system, while the other three components represent backward-compatible changes related to functionality, bugs fixing, and system identification, respectively. The use of semantic versioning supposedly communicate more information and reduces the chance of potential failures.

3.2.9 Avoid splitting packages [B9]. A split package is a package whose content is spread in two or more required bundles [42]. The main pitfalls related to the use of split packages consist on the mandatory use of the **`Require-Bundle`** header, which is labeled as a bad practice, and the following set of drawbacks mentioned in the *OSGi Core Specification* [42]: (i) *completeness*, which means that there is no guarantee to obtain all classes of a split package; (ii) *ordering*, an issue that arises when a class is included in different bundles; (iii) *performance*, an overhead is introduced given the need to search for a class in all bundle providers; and (iv) *mutable exports*, if a requiring bundle *`visibility`* directive is set to `reexport`, its value may suddenly change depending on the *`visibility`* value of the required bundle.

3.2.10 Declare dependencies that do not add value to the final user in the `Bundle-Classpath` header [B10]. If a non-OSGi dependency is used to support the internal functionality of a bundle, it should be specified in the `Bundle-Classpath` header. These dependencies are known as *containers* composed by a set of *entries*, which are then grouped under the *resources* namespace. They are resolved when no package or bundle offers the required functionality [42]. Given that a subset of these resources is meant to support private packages functionality, they should be kept as private packages and defined only in the classpath of the bundle.

3.2.11 Import exported API packages [B11]. All the packages that are exported and used by a given bundle should also be imported. This may seem counter-intuitive, as exported packages are locally contained in a bundle and can thus be used without being imported explicitly. Nevertheless, it is a best practice to import these packages explicitly, so that the OSGi framework can select an already-active version of the required package. Be aware that this best practice is only applicable to clean API packages [42].

4 OSGI CORPUS ANALYSIS

The best practices we identify in Section 3 emerge from experts of the OSGi ecosystem. The goal of the following two research questions is to assess their relevance and impact critically:

Q2 Are OSGi best practices being followed?
Q3 Does each OSGi best practice have an observable effect on the relevant qualitative properties of an OSGi bundle?

Specifically, because beyond their qualitative aspect they are meant to improve performance, we study their impact on the classpath size and resolution time of individual bundles. We first discuss the initial setup and method of our evaluation, and then go through all the selected best practices, aiming at answering our research questions for each of them. After some concluding remarks, we discuss the threats to validity. A complete description of all the artifacts discussed in this section (corpora, transformations, results), along with their source code, is available on the companion webpage.[6]

4.1 Studied Corpus

We use an initial corpus consisting of 1,124 OSGi bundles (cf. Table 2) corresponding to the set of core plug-ins of the Eclipse IDE 4.6 (Neon.1). This corpus emerges from the specific needs of a partner in the collaborative project CROSSMINER in which the authors are involved. The Eclipse IDE consists of a base platform that can be extended and customized through plug-ins that can be remotely installed from so-called *update sites*. Both the base platform and the set of plug-ins are designed around OSGi, which enables this dynamic architecture. The Eclipse IDE relies on its own OSGi-certified implementation of the specification: Eclipse Equinox. Because the Eclipse IDE is a mature and widely-used platform, its bundles are supposed of high quality. As they all contribute to the same system, they are also highly interconnected: the combination of `Import-Package`, `Require-Bundle`, and `DynamicImport-Package` dependencies results in a total of 2,751 dependency links. As a preliminary step, we clean the corpus to eliminate duplicate bundles and bundles that deviate from the very nature of Eclipse plug-ins. This includes:

- *Bundles with multiple versions*. We only retain the most recent version for each bundle to avoid a statistical bias towards bundles which (accidentally) occur multiple times for different versions.
- *Documentation bundles* that neither contain any code nor any dependency towards other bundles are considered as outliers to be ignored. The best practices are specifically about actual code bundles so these documentation bundles would introduce arbitrary noise.
- *Source bundles* that only contain the source code of another binary bundle are ignored since they are a (technical) accident not pertaining to the best practices either.
- Similarly, *test bundles* which do not provide any functionality to the outside would influence our statistical observations without relating to the studied best practices.

We identify and remove these bundles from the corpus according to their names. The (strong) convention in this Eclipse corpus is that these, respectively, end with a `.doc`, `.source`, or `.tests` suffix. The remaining bundles constitute our control corpus C_0.

[6] https://github.com/msr18-osgi-artifacts/msr18-osgi-artifacts

Table 2: Characteristics of the Eclipse 4.6 OSGi Corpus

Attribute	Value
Initial corpus size	1,124
Number of documentation bundles	17
Number of source bundles	446
Number of test bundles	97
Number of duplicate bundles	192
Studied corpus (C_0)	372
Total size of C_0 (MB)	163.76
Number of dependencies declared in C_0	2,751

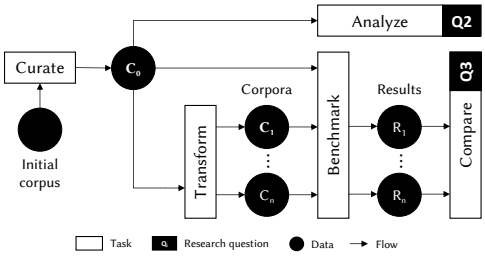

Figure 2: Analysis process.

4.2 Method

The overall analysis process we follow is depicted in Figure 2 and detailed below.

4.2.1 Selected best practices. For the current analysis, we focus on a subset of the best practices ([B1–B6]) elicited in Section 3, which can be studied using a common research method. The other best practices are interesting as future work: [B7, B10, B11] require distinguishing between implementation and API packages, [B8] requires distinguishing between breaking and non-breaking software changes, and [B9] requires refactoring the source code organization of the bundles in addition to their meta-data.

*4.2.2 Are OSGi best practices being followed? (**Q2**).* To answer this research question, we develop an analysis tool, written in Rascal [27], that computes a set of metrics on the control corpus C_0. Specifically, the tool analyses the meta-data (the Manifest files) and bytecode of each bundle to record in which way dependencies and versions are declared, which packages are actually used in the bytecode compared to what is declared in their meta-data, etc. Based on this information, we then count per best practice how many bundles (or bundle dependencies) satisfy it in the corpus. Using descriptive statistics we then analyze the support for the best practice in the corpus to answer **Q2**. For each best practice, based on the maturity of the Eclipse corpus the hypothesis is that they are being followed ($H2.i$).

*4.2.3 Does each OSGi best practice have an observable effect on the relevant qualitative properties of an OSGi bundle? (**Q3**).* To answer this research question, we hypothesize that each best practice would indeed have an observable impact on the size of dynamically computed classpaths ($H3.1.i$) and on the time it takes to resolve and load the bundles ($H3.2.i$). If either hypothesis is true, then there is indeed evidence of observable impact of the best practice

of some kind, if not then deeper analysis based on hypothesizing other forms of impact would be motivated. We are also interested to find out if there exists a correlation between classpath size and related resolution time ($H3.3$). Since the latter requires an accurate time measurement setup, while the former can be computed from meta-data, it would come in handy for IDE tool builders (recommendations, smell detectors, and quick fix) if classpath size would be an accurate proxy for bundle resolution time.

Figure 2 depicts how we compare the original corpus C_0 to alternative corpora C_i in which each best practice B_i has been simulated. For each B_i, a specialized transformation $T(B_i)$ takes as input the control corpus C_0 and turns it into a new corpus $C_0 \xrightarrow{T(B_i)} C_i$ where bundles are automatically transformed to satisfy the best practice B_i. For all transformations $T(B_i)$, we ensure that for all bundles that can be resolved in the original corpus, the corresponding bundle in the transformed corpus can also be resolved. For instance, the transformation $T(B_1)$ transforms every `Require-Bundle` header to a set of corresponding `Import-Package` headers, according to what is actually used in the bundle's bytecode. Note that bundles using extension points declared by other bundles must use the `Require-Bundle` header and therefore cannot be replaced with the corresponding `Import-Package` headers. Below, we discuss such detailed considerations with the result of each transformation. Then, we load every corpus C_i in a bare Equinox OSGi console and compute, for every bundle, (i) the size of its classpath, including the classes defined locally and the classes that are accessible through wiring links according to the semantics of OSGi, and (ii) measure the exact time it takes to resolve it. Resolution time of a bundle is measured as the delta between the time it enters the `INSTALLED` state ("*The bundle has been successfully installed*") and the time it enters the `RESOLVED` state ("*All Java classes that the bundle needs are available*"), according to the state diagram given in the OSGi specification [42, p. 107]. To report a change in terms of classpath size or performance, we also compute the relative change between observations in C_i and observations in C_0 as $d_{ij} = \frac{v_{0j} - v_{ij}}{v_{0j}} \times 100\%$, where d_{ij} is the relative change between the j^{th} observation of C_0 (i.e., v_{0j}) and the corresponding observation in C_i (i.e., v_{ij}). The median ($\widetilde{x}$) value of the set of relative changes is used as a comparison measure.[7] All performance measurements are conducted on a macOS Sierra version 10.12.6 with an Intel Core i5 processor 2GHz, and 16GB of memory running OSGi version 3.11.3 and JVM version 1.8. Measurements are executed 10 times each after discarding the 2 initial warm-up observations [7].

4.3 Results

To evaluate $H3.3$ we use both scatter plots and correlations (per corpus) that show the relation between our two studied variables, classpath size and resolution time. Figure 3 shows the graph to identify the hypothesized correlation in C_0. Given that there is no linear relation between the variables, we compute the non-parametric Spearman's rank correlation coefficient $\rho_0 = -0.17$, resulting in a weak negative relation. Similar results are observed on all corpora C_i: $\rho_1 = -0.06$, $\rho_2 = -0.17$, $\rho_3 = -0.11$, $\rho_4 = -0.13$, $\rho_5 = -0.17$, and $\rho_6 = -0.17$. According to both visual and statistical analysis, we can reject hypothesis $H3.3$. Therefore, it remains interesting to study these variables independently. The benchmark results regarding classpath size and resolution time for every corpus C_i, compared to the control corpus C_0, are given in Figures 4 and 5.

[7] We use $\widetilde{x_c}$ and $\widetilde{x_p}$, respectively, for classpath size and performance comparisons.

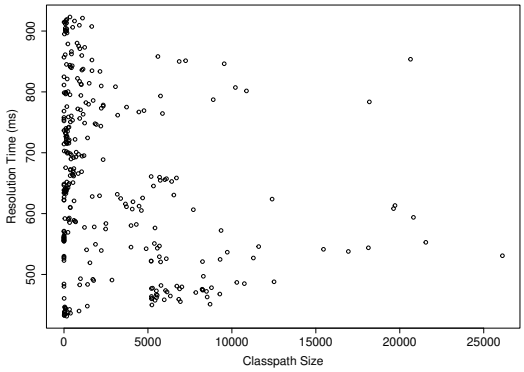

Figure 3: Classpath size is a poor indicator for resolution time (ms) in C_0 ($\rho_0 = -0.17$).

4.3.1 *Prefer package-level dependencies [B1].*

$H2.1$. To test this hypothesis, we count the number of bundles using the `Require-Bundle` and `Import-Package` headers. We cross-analyze these results by computing the number of extension plug-ins and the number of bundles declaring split packages, which may impact the use of the `Require-Bundle` header. The bundles declare 1,283 `Require-Bundle` dependencies and 1,459 `Import-Package` dependencies. 57.79% of the bundles use the `Require-Bundle` header, 50.00% use the `Import-Package` header, and 34.95% use both. These results suggest that this best practice tends not to be widely followed by Eclipse plug-ins developers. The declaration of extension points and extension bundles, as well as the use of split packages, contribute to these results. In fact, 38.98% of the bundles in C_0 are extension bundles that require a dependency on the bundle declaring the appropriate extension point to provide the expected functionality. Two of the bundles in C_0 use a `Require-Bundle` dependency to cope with the requirements of split packages.

$H3.1.1$ *and* $H3.2.1$. Transforming bundle-level dependencies to package-level dependencies reduces the classpath size of bundles by $\widetilde{x_c} = 15.40\%$. This is because, in the case of `Require-Bundle`, every exported package in a required bundle is visible to the requiring bundle, whereas the more fine-grained `Import-Package` only imports the packages that are effectively used in the bundle's code. We observe a gain of $\widetilde{x_p} = 7.11\%$ regarding performance (Figure 5).

4.3.2 *Use versions when possible [B2].*

$H2.2$. To tackle this question, we compute the number of versioned `Require-Bundle`, `Import-Package`, and `Export-Package` relations and the proportion of those that specify a version range. 84.80% of the `Require-Bundle` dependencies are versioned, of which 71.97% (i.e., 783) use a version range. In the case of `Import-Package`, 59.22% of the dependencies are versioned, of which 45.14% use a version range.

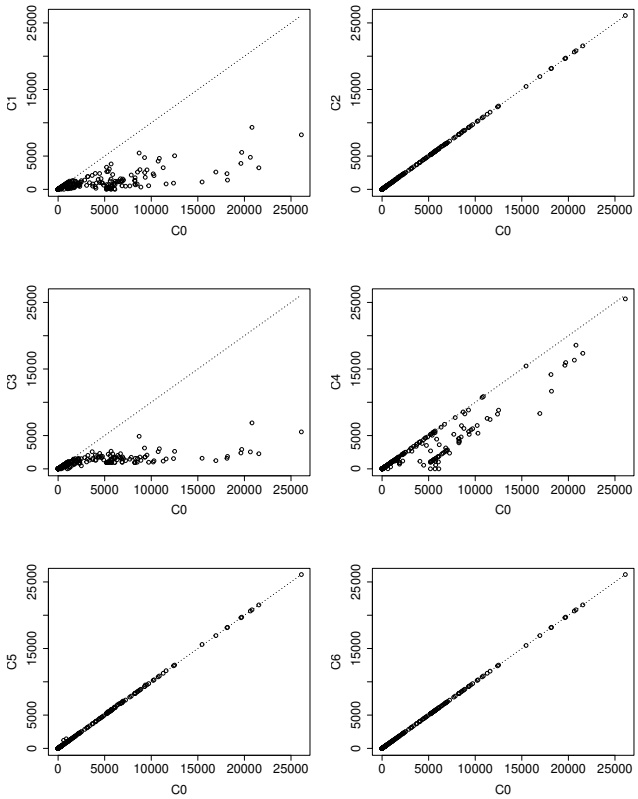

Figure 4: Comparing classpath size of corpora C_i (with best practices B_i applied) to the original corpus C_0.

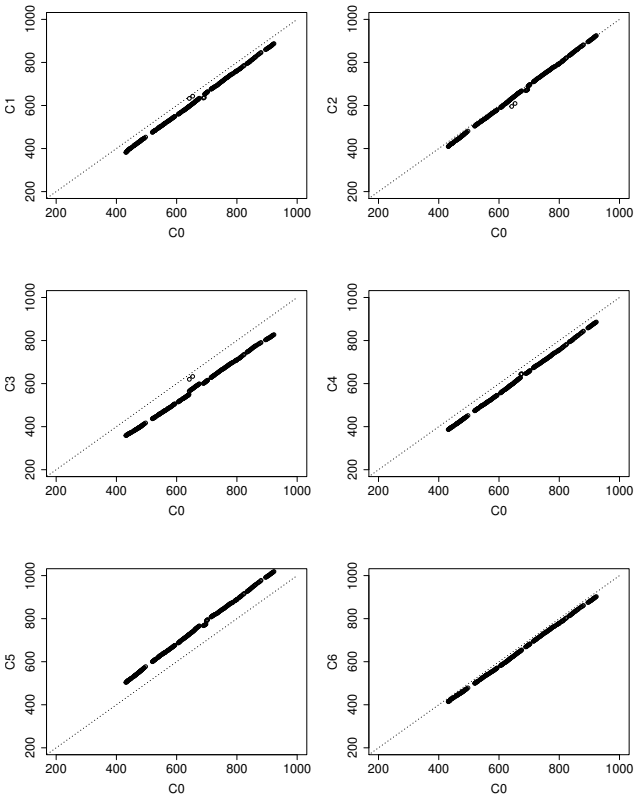

Figure 5: Comparing resolution time (ms) of corpora C_i (with best practices B_i applied) to the original corpus C_0.

Finally, 24.88% of the 2,620 exported packages tuples are explicitly versioned. The remaining tuples get a value of `0.0.0` according to the OSGi specification. We observe a tendency to use versions when defining **`Require-Bundle`** relations, which is highly advised given the need to maintain a tight coupling with a specific bundle. Nonetheless, the frequency of version specifications decreases when using **`Import-Package`** and even more so with **`Export-Package`**.

*H*3.1.2 *and H*3.2.2. The transformation $T(B_2)$ takes all unversioned **`Import-Package`** and **`Require-Bundle`** headers in C_0 and assign a strict version range of the form $[V, V]$ to them, where V is the highest version number of the bundle or package found in the corpus. In the resulting corpus C_2, we observe that this best practice has no impact on classpath size ($\widetilde{x_c} = 0\%$), and close to zero impact on resolution time ($\widetilde{x_p} = 1.56\%$) of individual bundles.

4.3.3 *Export only needed packages [B3].*

*H*2.3. In this case, we investigate how many of the exported packages in the corpus are imported by other bundles, using either **`Import-Package`** or **`Require-Bundle`**, taking versions into account. If an exported package is never imported, this may indicate that this package is an internal or implementation package that should not be exposed to the outside. There may, however, be a fair amount of false positives: some of the exported packages may actually be part of a legitimate API but are just not used by other bundles yet. From the whole set of *exported package* tuples, 14.62% are explicitly imported by other bundles. This suggests that a large portion of the packages that are exported are never used by other bundles. Nevertheless, if we also consider packages imported through the **`Require-Bundle`** header, at least 80.34% of the total tuples are imported by other bundles. The question that arises is: is this situation intended, or is it a collateral effect of the use of **`Require-Bundle`**?

*H*3.1.3 *and H*3.2.3. [B3] has an impact on classpath size in the transformed corpus C_3: exporting only the needed packages results in a $\widetilde{x_c} = 23.27\%$ gain sizewise. We also observe an improvement of $\widetilde{x_p} = 12.83\%$ in terms of resolution time for individual bundles.

4.3.4 *Minimize dependencies [B4].*

*H*2.4. To investigate whether bundles declare unnecessary dependencies, we cross-check the meta-data declared in the Manifest files with bytecode analysis. We deem any package that is required but never used in the bytecode as superfluous. In the corpus, 19.25% of the **`Require-Bundle`** dependencies are never used locally, i.e., none of the packages of the required bundle are used in the requiring bundle's code. Regarding **`Import-Package`** dependencies, 13.78% of the explicitly-imported packages are not used in the bytecode. Digging deeper into the relations, we find that the **`Require-Bundle`** declarations are implicitly importing 15,399 packages that have been exported by the corresponding required bundles. From this set, only 16.50% are actually used in the requiring bundle bytecode. These results suggest that developers tend not to use all the dependencies

they declare and that these could be minimized. The situation is much worse in the case of implicitly imported packages through the `Require-Bundle` header, which backs the arguments of [B1].

*H*3.1.4 *and H*3.2.4. [B4] has a close to zero impact on classpath size in the transformed corpus C_4 ($\widetilde{x_c} = 0.14\%$). However, the improvement is higher with regards to resolution time ($\widetilde{x_p} = 7.24\%$).

4.3.5 *Import all needed packages [B5].*

*H*2.5. We compute the number of packages that are used in the code but are never explicitly imported in the Manifest file by analyzing the bundles meta-data and bytecode. Our analysis identifies 2,194 packages (269 unique) that are never explicitly imported. Overall, 45.70% of the bundles in C_0 use a package that they do not explicitly import (excluding `java.*` packages).

*H*3.1.5 *and H*3.2.5. For every package that can be found somewhere in the corpus but is missing in the `Import-Package` list of a given bundle, the transformation $T(B_5)$ creates a new `Import-Package` statement pointing to it. The resulting corpus C_5 does not differ from C_0 in terms of classpath size, but appears to be slower in terms of resolution time ($\widetilde{x_p} = -13.35\%$). By creating new explicit dependencies to be resolved, this best practice adds to the dependency resolution process, which in turn may explain this difference.

4.3.6 *Avoid `DynamicImport-Package` [B6].*

*H*2.6. In the corpus, only 7 bundles declare `DynamicImport-Package` dependencies, for a total of 9 dynamic relations declared in C_0. 4 of these dynamically imported packages are not exported by any bundle. This may result in runtime exceptions after the resolution of the involved bundles. While there are some occurrences in the corpus of this not-advisable type of dependency, results suggest that developers tend to avoid using the `DynamicImport-Package` header and thus generally follow this best practice.

*H*3.1.6 *and H*3.2.6. We do not observe any impact in terms of classpath size, and in terms of performance we observe a gain of $\widetilde{x_p} = 3.47\%$. As our benchmark stops at resolution time and [B6] only has an impact after resolution time, this is unsurprising.

4.4 Analysis of the results

Figure 6 summarizes the overall results regarding relative change of our analysis for classpath size and resolution time.

4.4.1 *Are OSGi best practices being followed? (**Q2**).*

Overall, we observe that most of the best practices we identify are not widely followed in the corpus. This is for instance the case with [B1], despite being the most-widely advocated best practice among the ones we select (cf. Table 1).

> **Q2**: *OSGi best practices related to dependency management are not widely followed within the Eclipse ecosystem.*

4.4.2 *Does each OSGi best practice have an observable effect on the relevant qualitative properties of an OSGi bundle? (**Q3**).*

[B1] and [B3] appear to have a positive impact on classpath size (15.40% and 23.27%, respectively), whereas we observe a close to zero impact for [B2], [B4], [B5], and [B6]. Moreover, five of the selected best practices (i.e., [B1], [B2], [B3], [B4], and [B6]) show an improvement

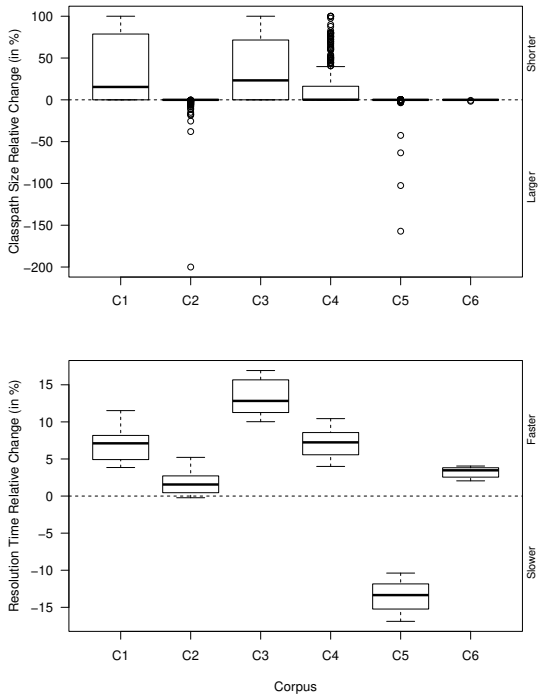

Figure 6: Relative change in classpath size and resolution time between the control (C_0) and transformed corpora (C_i).

on performance that oscillates between 1.55% and 12.83%. [B5] shows a negative impact of 13.35% relative change for the same variable. The absence of larger gains may be explained by the fact that the time required to build the classpath is negligible compared to the other phases involved in bundle resolution (e.g., solving dependencies constraints, as can be observed for [B5]).

> **Q3**: *Only one third of the OSGi best practices we analyze have a positive impact (of up to ~23% change) on the classpath size of individual bundles. Either way, impact on resolution times does not exceed ±13% relative change for all practices.*

4.5 Threats to Validity

In principle, the construct of measuring classpath size and resolution time for OSGi bundles can show the presence of a specific kind of impact of a best practice, but not the absence of any other kind of impact. Hence, for where we observed no impact, future analysis of possible other dependent quality factors (e.g. coupling metrics) is duly motivated. However, since the prime goal of OSGi is configuring which bundles to dynamical load into the classpath, any change to OSGi configuration must also be reflected in the classpath. Therefore, in theory, we would not expect any other unforeseen effects when a classpath does not change much. Although relevant, our research methods did not focus on the downstream effects of OSGi best practices on system architecture or object-oriented design quality in source code. However, minor changes to a classpath may have large impact on those aspects, in particular class visibility may impact software evolution aspects such as design erosion and code cloning. In IDEs specifically, performance is not always a

key consideration and other aspects of dependency management remain to be studied as future work. With respect to internal validity of the research methods, we calculated classpath size using the OSGi classloader and wiring APIs. Internally, for every bundle, OSGi creates a Java classloader that holds every class local to the bundle, plus all the classes of the bundles it depends on, regardless of the granularity of the dependencies, their visibilities, etc. The OSGi classloaders, on the other hand, *hide* classes from the required bundles when necessary, e.g., when a bundle only requires a few packages from another one using the `Import-Package` header. We aimed to calculate classpath size as seen by the OSGi framework itself, but results may vary if we look at Java classloaders instead. Besides, our analysis and transformation tools may be incorrect in some way. We tried to mitigate this pitfall by having our code written and reviewed by several developers, as well as by writing a set of sanity tests that would catch the most obvious bugs. The corpus we use for the analysis may also greatly influence the results we obtain for **Q2**—our conclusions only hold for the Eclipse IDE. Nonetheless, we tried to mitigate this effect as much as possible, for instance by taking into account the specificities of the extensions and extension points mechanism within Eclipse which influences our conclusions for [B1]. Similarly, for **Q3**, a different implementation of the OSGi specification may influence the benchmark results.

5 RELATED WORK

Seider et al. explore modularization of OSGi-based components using an interactive visualization tool [37]. They extract information from meta-data files and organize it at different abstraction levels (e.g., package and service). Forster et al. perform static analyses of software dependencies in the OSGi and Qt frameworks [14]. They identify runtime connections using source code analysis. Management of false positives is still a challenge in their research. Both approaches study dependencies in the OSGi framework but are more focused on the current state of the framework rather than potential dependencies specification pitfalls or smells.

With regards to smell detection in configuration management frameworks, Sharma et al. present a catalog of configuration management code smells for 4K Puppet repositories on GitHub [38]. Smell distributions and co-occurrences are also analyzed. Jha et al. provide a static analysis tool that aims at detecting common errors made in Manifest files of 13K Android apps [21]. Common mistakes are classified as: misplaced elements and attributes, incorrect attribute values, and incorrect dependency. Karakoidas et al. [24] and Mitropoulos et al. [29] describe a subset of Java projects hosted in the Maven Central Repository. They use static analysis to compute metrics related to object-oriented design, program size, and package design [24]. FindBugs tool is also used to detect a set of bugs present in the selected projects. They discover that bad practices are the main mistakes made by developers, but do not detail the kinds of recurrent smells. Finally, Raemaekers et al. analyze a set of projects hosted in the Maven Central Repository to check whether they adhere to the semantic versioning scheme [35]. They find that developers tend to introduce breaking changes even if they are related to a minor change.

With regards to dependency modeling approaches, Shatnawi et al. aim at identifying dependency call graphs of legacy Java EE applications to ease their migration to loosely coupled architectures [39]. Kula et al. also study JVM-based projects to support migration to more recent versions of a given open source library [28]. Nevertheless, manifold dependencies are not only specified in source code but also in configuration files. To face this challenge, both approaches parse dependencies from these sources and extract information in their own models: the Knowledge Discovery Meta-model (KDM) [34, 39] and the Software Universe Graph (SUG) [28]. Jezek et al. statically extract dependencies and potential smells from the source code and bytecode of applications [20]. Similarly, Abate et al. dig into *Debian*, *OPAM*, and *Drupal* repositories to identify failing dependencies and to present actionable information to the final user [1]. To this aim, dependencies information are gathered from components' meta-data files, which is then represented in the Common Upgradability Description Format (CUDF) model.

Considering software repositories and dependencies description, Decan et al. conduct an empirical analysis of the evolution of *npm*, *RubyGems*, and *CRAN* repositories [9]. Kikas et al. [25] also consider the first two mentioned repositories and the *Crates* ecosystem. In both cases, dependencies are identified and modeled in dependency graphs or networks to analyze the evolution of the repositories and the ecosystem resilience. Tufano et al. study the evolution of 100 Java projects, finding that 96% of them contain broken snapshots, mostly due to unresolved dependencies [43]. Finally, Williams et al. study more than 500K open source projects taken from *Eclipse*, *SourceForge*, and *GitHub*. They cross-check users needs against projects properties, by means of computing and analyzing metrics provided at forge-specific and forge-agnostic levels [46].

6 CONCLUSIONS & FUTURE WORK

In this paper, we first conducted a systematic review of OSGi best practices to formally document a set of 11 known best practices related to dependency management. We then focused on 6 of them and, using a corpus of OSGi bundles from the Eclipse IDE, we studied whether these best practices are being followed by developers and what their impact is on the classpath size and bundle resolution times. On the one hand, the results show that many best practices tend not to be widely followed in practice. We also observed a positive impact of applying two of the best practices (artificially) to classpath sizes (i.e., [B1] and [B3]), from which we can not conclude that the respective best practices are irrelevant. Based on this we conjecture most of the identified advice is indeed relevant. Deeper qualitative analysis is required to validate this. On the other hand, the performance results show that OSGi users can expect a performance improvement of up to ±13% when applying certain best practices (e.g., [B3]). For future work, building on this initial study, we plan to scale up our analysis on other OSGi-certified implementations (e.g., Apache Felix) and other corpora of bundles (e.g., bundles extracted from JIRA or Github), and to cross-reference relevant quality attributes on the system architecture and object-oriented design levels with the current OSGi meta-data and bytecode analyses.

ACKNOWLEDGMENTS

This work is partially supported by the EU's Horizon 2020 Project No. 732223 CROSSMINER.

REFERENCES

[1] Pietro Abate, Roberto Di Cosmo, Louis Gesbert, Fabrice Le Fessant, Ralf Treinen, and Stefano Zacchiroli. 2015. Mining component repositories for installability issues. In *12th Working Conference on Mining Software Repositories*. IEEE, Piscataway, 24–33.

[2] OSGi Alliance. n.d.. Guidelines. https://goo.gl/kT4FU6. (n.d.).

[3] OSGi Alliance. n.d.. OSGi developer certification – Professional. https://goo.gl/TftHFF. (n.d.).

[4] Roland Barcia, Tim deBoer, Jeremy Hughes, and Alasdair Nottingham. 2010. Developing OSGi enterprise applications. https://goo.gl/5fbom7. (2010).

[5] Neil Bartlett and Peter Kriens. 2010. bndtools: mostly painless tools for OSGi. https://goo.gl/FpmhJR. (2010).

[6] Veronika Bauer and Lars Heinemann. 2012. Understanding API usage to support informed decision making in software maintenance. In *16th European Conference on Software Maintenance and Reengineering*. IEEE, Washington, 435–440.

[7] Stephen M Blackburn, Kathryn S McKinley, Robin Garner, Chris Hoffmann, Asjad M Khan, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, et al. 2008. Wake up and smell the coffee: evaluation methodology for the 21st century. *Commun. ACM* 51, 8 (2008), 83–89.

[8] Bradley E. Cossette and Robert J. Walker. 2012. Seeking the ground truth: A retroactive study on the evolution and migration of software libraries. In *20th International Symposium on the Foundations of Software Engineering*. ACM, New York, 55:1–55:11.

[9] A. Decan, T. Mens, and M. Claes. 2017. An empirical comparison of dependency issues in OSS packaging ecosystems. In *24th International Conference on Software Analysis, Evolution and Reengineering*. IEEE, Piscataway, 2–12.

[10] Bernhard Dorninger. 2010. Experiences with OSGi in industrial applications. https://goo.gl/hPokNX. (2010).

[11] Apache Felix. 2013. OSGi frequently asked questions. https://goo.gl/g2Luqy. (2013).

[12] Apache Felix. 2015. Dependency manager – Background. https://goo.gl/758FPJ. (2015).

[13] Ulf Fildebrandt. 2011. Structuring software systems with OSGi. https://goo.gl/T9ywmA. (2011).

[14] Thomas Forster, Thorsten Keuler, Jens Knodel, and Michael-Christian Becker. 2013. Recovering component dependencies hidden by frameworks – Experiences from analyzing OSGi and Qt. In *17th European Conference on Software Maintenance and Reengineering*. IEEE, Washington, 295–304.

[15] Alexander Grzesik. 2013. TRESOR: the modular cloud – Building a domain specific cloud platform with OSGi. https://goo.gl/dfaV6m. (2013).

[16] Brett Hackleman and James Branigan. 2009. OSGi: the best tool in your embedded systems toolbox. https://goo.gl/CDNuWo. (2009).

[17] B. J Hargrave and Peter Kriens. 2007. OSGi best practices! https://goo.gl/pFQjeV. (2007).

[18] B. J. Hargrave and Jeff McAffer. 2006. Best practices for programming Eclipse and OSGi. https://goo.gl/Wp6KfW. (2006).

[19] Dileepa Jayakody. 2012. Building a modular server platform with OSGi. https://goo.gl/9tg1ok. (2012).

[20] K. Jezek, L. Holy, A. Slezacek, and P. Brada. 2013. Software components compatibility verification based on static byte-code analysis. In *39th Euromicro Conference on Software Engineering and Advanced Applications*. IEEE, Piscataway, 145–152.

[21] Ajay Kumar Jha, Sunghee Lee, and Woo Jin Lee. 2017. Developer mistakes in writing Android manifests: An empirical study of configuration errors. In *14th International Conference on Mining Software Repositories*. IEEE, Piscataway, 25–36.

[22] Emily Jiang. 2012. OSGi application best practices. https://goo.gl/qTDxqM. (2012).

[23] Gerd Kachel, Stefan Kachel, and Ksenija Nitsche-Brodnjan. 2010. Migration from Java EE application server to server-side OSGi for process management and event handling. https://goo.gl/9zKb2y. (2010).

[24] Vassilios Karakoidas, Dimitris Mitropoulos, Panos Louridas, Georgios Gousios, and Diomidis Spinellis. 2015. Generating the blueprints of the Java ecosystem. In *12th Working Conference on Mining Software Repositories*. IEEE, Piscataway, 510–513.

[25] Riivo Kikas, Georgios Gousios, Marlon Dumas, and Dietmar Pfahl. 2017. Structure and evolution of package dependency networks. In *14th International Conference on Mining Software Repositories*. IEEE, Piscataway, 102–112.

[26] Barbara Kitchenham, O. Pearl Brereton, David Budgen, Mark Turner, John Bailey, and Stephen Linkman. 2009. Systematic literature reviews in software engineering - A systematic literature review. *Inf. Softw. Technol.* 51, 1 (2009), 7–15.

[27] Paul Klint, Tijs van der Storm, and Jurgen Vinju. 2011. EASY meta-programming with Rascal. In *3rd International Summer School Conference on Generative and Transformational Techniques in Software Engineering III*. Springer, Berlin, Heidelberg, 222–289.

[28] Raula Gaikovina Kula, Coen De Roover, Daniel M German, Takashi Ishio, and Katsuro Inoue. 2017. Modeling Library Dependencies and Updates in Large Software Repository Universes. *arXiv preprint arXiv:1709.04626* (2017).

[29] Dimitris Mitropoulos, Vassilios Karakoidas, Panos Louridas, Georgios Gousios, and Diomidis Spinellis. 2014. The bug catalog of the Maven ecosystem. In *11th Working Conference on Mining Software Repositories*. ACM, New York, 372–375.

[30] Jerome Moliere. 2011. 10 Things to know you are doing OSGi in the wrong way. https://goo.gl/TdRh2e. (2011).

[31] Marcel Offermans. 2005. Automatically managing service dependencies in OSGi. https://goo.gl/BFT8x2. (2005).

[32] Marcel Offermans. 2006. Using Apache Felix: OSGi best practices. https://goo.gl/jmZsYD. (2006).

[33] D. L. Parnas. 1972. On the criteria to be used in decomposing systems into modules. *Commun. ACM* 15, 12 (1972), 1053–1058.

[34] Ricardo Pérez-Castillo, Ignacio García-Rodríguez de Guzmán, and Mario Piattini. 2011. Knowledge discovery metamodel-ISO/IEC 19506: A standard to modernize legacy systems. *Comput. Stand. Interfaces* 33, 6 (2011), 519–532.

[35] Steven Raemaekers, Arie van Deursen, and Joost Visser. 2014. Semantic versioning versus breaking changes: A study of the Maven repository. In *14th International Working Conference on Source Code Analysis and Manipulation*. IEEE, Washington, 215–224.

[36] Roman Roelofsen. 2009. Very important bundles. https://goo.gl/jznk62. (2009).

[37] Doreen Seider, Andreas Schreiber, Tobias Marquardt, and Marlene Brüggemann. 2016. Visualizing modules and dependencies of OSGi-based applications. In *IEEE Working Conference on Software Visualization*. IEEE, Piscataway, 96–100.

[38] Tushar Sharma, Marios Fragkoulis, and Diomidis Spinellis. 2016. Does Your Configuration Code Smell?. In *13th International Conference on Mining Software Repositories*. ACM, New York, 189–200.

[39] Anas Shatnawi, Hafedh Mili, Ghizlane El Boussaidi, Anis Boubaker, Yann-Gaël Guéhéneuc, Naouel Moha, Jean Privat, and Manel Abdellatif. 2017. Analyzing Program Dependencies in Java EE Applications. In *14th International Conference on Mining Software Repositories*. IEEE, Piscataway, 64–74.

[40] Jim Shore. 2004. Fail fast [software debugging]. *IEEE Software* 21, 5 (2004), 21–25.

[41] Rok Strniša, Peter Sewell, and Matthew Parkinson. 2007. The Java module system: core design and semantic definition. In *22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*. ACM, New York, 499–514.

[42] The OSGi Alliance. 2014. OSGi Core Release 6 Specification. (Jun 2014).

[43] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. 2017. There and back again: Can you compile that snapshot? *Journal of Software: Evolution and Process* 29, 4 (2017).

[44] Tim Ward. 2012. Best practices for (enterprise) OSGi applications. https://goo.gl/wmmNaR. (2012).

[45] Thomas Watson and Peter Kriens. 2006. OSGi component programming. https://goo.gl/eR5Tmo. (2006).

[46] James R. Williams, Davide Di Ruscio, Nicholas Matragkas, Juri Di Rocco, and Dimitris S. Kolovos. 2014. Models of OSS project meta-information: A dataset of three forges. In *11th Working Conference on Mining Software Repositories*. ACM, New York, 408–411.