Towards a Catalog of Java Dependency Injection Anti-Patterns

Rodrigo Laigner
PUC-Rio, Brazil
rlaigner@inf.puc-rio.br

Marcos Kalinowski
PUC-Rio, Brazil
kalinowski@inf.puc-rio.br

Luiz Carvalho
PUC-Rio, Brazil
lmcarvalho@inf.puc-rio.br

Diogo Mendonça
CEFET/RJ, Brazil
diogo.mendonca@cefet-rj.br

Alessandro Garcia
PUC-Rio, Brazil
afgarcia@inf.puc-rio.br

ABSTRACT

[Context] Dependency Injection (DI) is a commonly applied mechanism to decouple classes from their dependencies in order to provide better modularization of software. In the context of Java, the availability of a DI specification and popular frameworks, such as Spring, facilitate DI usage in software projects. However, bad DI implementation practices can have negative consequences. Even though the literature suggests the existence of DI anti-patterns, there is no detailed catalog of such bad practices. Moreover, there is no evidence on their occurrence and perceived usefulness from the developer's point of view. [Goal] Our goal is to review the reported DI anti-patterns in order to analyze their completeness and to propose and evaluate a novel catalog of DI anti-patterns in the context of Java. [Method] We propose an initial catalog containing twelve Java DI anti-patterns. We selected four open source software projects that adopt a DI framework and developed a tool to statically analyze the occurrence of the DI anti-patterns within their source code. Also, we conducted a survey through face to face interviews with three experienced developers that regularly apply DI. [Results] At least nine different DI anti-patterns appeared in each analyzed project. In addition, the feedback received from the developers confirmed their relevance and the importance of investing further effort towards a catalog. [Conclusion] The results indicate that the initial catalog contains Java DI anti-patterns that occur in practice and are useful. Sharing it with practitioners may help them to avoid such anti-patterns. Sharing it with the research community will enable further improving the catalog.

CCS CONCEPTS

• **Software and its engineering → Object oriented development**; *Software design engineering*.

KEYWORDS

dependency injection, anti-pattern, java, pattern, framework

ACM Reference Format:
Rodrigo Laigner, Marcos Kalinowski, Luiz Carvalho, Diogo Mendonça, and Alessandro Garcia. 2019. Towards a Catalog of Java Dependency Injection Anti-Patterns. In *SBES '19: Brazilian Symposium on Software Engineering, September 23–27, 2019, Salvador, BA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Dependency injection (DI) is a mechanism for improving software modularity. DI enables less coupling among modules by refraining them from being aware of implementation details of each other [15]. DI has become a common practice in the software industry, as characterized by the existence of DI frameworks and industry-oriented publications [12] [16]. For instance, Spring[1], one of the most popular Java frameworks, and Google AdWords[2], a large-scale web application, have its components interconnected through DI. Furthermore, Java defines a specification targeted at DI and Microsoft .NET Core provides native DI capabilities.

Despite the existence of well designed frameworks, such as Spring and Guice[3], the implementation of DI is not trivial and demands in-depth knowledge on object-oriented design. Hence, while DI might support providing a better modularization, once system modules do not need to concern about dependence resolution, improper DI usage may hinder the effective achievement of this goal.

Although the technical literature [16][20] suggests the existence of DI anti-patterns, there is no evidence about their occurrence, acceptance and perceived usefulness from developers' point of view. Furthermore, such anti-patterns can lead to negative consequences, such as high coupling and potential overuse of memory. Therefore, it is necessary to properly document DI anti-patterns. Considering this scenario, our study aims at reviewing the completeness of the DI anti-patterns reported in the literature. We also propose a catalog of DI anti-patterns in the context of Java, and investigate their occurrence by mining open source repositories. Additionally, this study seeks to gather the perception of developers through an expert survey in order to validate the proposed catalog.

The contributions of this paper are: (i) an initial effort towards documenting a catalog of Java DI anti-patterns; (ii) an investigation on the occurrence of the proposed anti-patterns, indicating that they are general and occur within different projects; and (iii) gathering the perception of developers on the proposed catalog by an expert

[1]https://spring.io
[2]https://github.com/google/guice/wiki/AppsThatUseGuice
[3]https://github.com/google/guice

survey, indicating that the catalog is perceived as relevant and useful.

The remainder of this paper is organized as follows. In Section 2 we provide the background on dependency injection and DI anti-patterns and discuss related work. In Section 3, we describe our goal and research questions. Next, the proposed catalog of DI anti-patterns is introduced in Section 4. An evaluation of the occurrence of the cataloged DI anti-patterns is presented on Section 5. Section 6 presents the expert survey on the acceptance and perceived usefulness of the catalog. Section 7 discusses the threats of validity. Finally, Section 8 presents the concluding remarks.

2 BACKGROUND AND RELATED WORK

This section provides a brief background on the principles behind DI and its main concepts. Also, previously reported DI anti-patterns are reviewed and related work is presented.

2.1 Dependency Injection

The term *Inversion of Control* (IoC) was first introduced by Johnson and Foote [8] in the context of software frameworks. In order to make use of functionalities provided by a framework, it is usually needed to extend a set of classes. Meanwhile, this extension often refrains the system from controlling its own execution, incurring in an inversion of control. Later, the dependency inversion principle (DIP) was introduced by Martin [11]. The DIP states that modules should not depend on details of another module. Instead, abstractions should be used in place of direct dependencies in order to enable reuse [11]. Based on Martin's [11] description, Yang et al. [22] assert that in Java, DIP means that "a class should depend on interface or abstract types, not on concrete types". These concepts form a basis for comprehending DI.

According to Crasso et al. [4], DI is a programming mechanism that "builds on the decoupling given by isolating components behind interfaces, and focuses on delegating the responsibility for component [or module] creation and binding to a DI container". As noted by Yang et al. [22], DI is a specific structural form of DIP. Indeed, DI implements the DIP principle, once components are decoupled through an interface oriented design.

However, although we achieve better modularity through abstractions, we still have to deal with instantiation of concrete classes, as noted by the Gamma et al. [7]. In the context of DI, Crasso et al. [4] assert that the responsibility for component creation is given to a DI container. In particular, the DI container is usually employed in order to enable the IoC principle in DI. In the context of DI, IoC is about delegating responsibilities that are outside of the scope of a given class to other classes of the system, the ones that can actually be accountable for these given responsibilities, such as instance provision. Although it is possible to achieve IoC in DI without a DI container, such container is typically employed by DI frameworks to ease the instantiation of classes.

With the widespread of DI frameworks for Java, such as Dagger[4], an effort to conceptualize a standardization gave birth to JSR-330 [14], a specification for DI implementation. The specification is adopted by Guice and Spring, and defines a set of annotations, which are described in Table 1.

[4]https://google.github.io/dagger/

Table 1: Annotations defined in JSR-330

Annotation	Description
Singleton	A type annotated with @Singleton will be instantiated only once across the run-time of the application
Scope	Scopes are rules applied to the life-cycle of a given object. Depending on the framework, the default implementation varies from retaining the instance for reuse or providing a new instance every time a dependence is requested
Qualifier	The value associated with a Qualifier annotation is used to distinguish which concrete implementation should be provided
Named	Used to identify an object instance. It aims at assigning a specific name to a dependence
Inject	Defines an injection point for which a DI container must provide an object instance. It can be placed in constructors, methods, and fields

Table 2: DI anti-patterns extracted from [16]

Name	Description
Control Freak	Dependencies are controlled directly, as opposed to IoC
Bastard Injection	Foreign defaults are used as default values for dependencies
Constrained Construction	Constructors are assumed to have a particular signature
Service Locator	An implicit service can serve dependencies to consumers but isn't guaranteed to do so

2.2 DI Anti-Patterns

According to Brown et al. [2], an anti-pattern is a "description of a commonly occurring solution to a problem that generates decidedly negative consequences". In line with this definition, we can understand a DI anti-pattern as a recurring DI usage pattern in source code that degrades aspects that DI is supposed to improve, such as coupling, or other quality aspects, such as performance. The only explicit proposal of DI anti-patterns is the one described by Seemann [16] and Deursen and Seemann [20], which contains a set of four DI anti-patterns, shown in Table 2.

In *Control Freak* anti-pattern, the principle of IoC is not achieved, since a dependency is obtained through directly creating an instance of a concrete implementation. This behavior introduces high coupling into the system through modules that make use of direct creation of instances.

Bastard Injection concerns specifying a default constructor with the objective of creating a default dependence instance. Usually implemented aiming at supporting unit testing, the class with a bastard injection incurs in high coupling with the default dependence created by its default constructor.

Constrained Construction regards introducing an implicit constraint on a dependency, i.e., a constructor with a particular signature. This behavior represents a problem when late binding is needed due to application requirements.

Finally, *Service Locator*, a design pattern introduced by Martin Fowler [6], is described by Seemann [16] as an anti-pattern in the context of DI applications, since it implies in widespread coupling to a static factory (in this case, the Service Locator class) throughout source code.

The DI anti-patterns addressed by Seemann [16] correspond to generic rules of thumb when it comes to DI adoption in software projects. For instance, regarding *Control Freak*, the author suggests that the presence of the *new* keyword and static factories as indicative of high level of coupling in source code. However, *Control Freak* can also be considered an anti-pattern in projects that do not implement DI. The DI anti-patterns proposed in this paper address more specific DI related problems in Java source code, such as misuse of specification annotations.

2.3 Related Work

In order to analyze which studies have approached DI anti-patterns, we complemented informal searches by submitting a generic search string ("dependency injection"), applied to title, abstract, and keywords, to Scopus digital library. Most of the 110 retrieved studies do not focus on DI or DI anti-patterns. Only two academic studies were somehow related to our work. Their description is presented in the following paragraphs. We also applied backward and forward snowballing on these two studies, but identified no other studies with this focus.

Roubtsov et al. [15] claim that overuse of annotations can potentially lead to violations of modularity principles. They propose a catalog of "bad smells" over dependencies injected in the context of Java annotations. Even though their work provides means for resolving each smell, it provides no comprehensive discussion concerning the validity of the proposed "code smells". It is important to note that, from thirteen annotations analyzed, only two are related to DI (*@ImplementedBy*, *@ProvidedBy*), which are annotations introduced by the Guice framework. Moreover, they recognize that the cataloged concerns heavily focus on annotations related to the J2EE persistence model, which are based upon Java Persistence API (JPA), a specification for persistence in Java.

Yang et al. [22] conducted an empirical study concerning the use of DI in Java applications. This study was focused on analyzing projects that do not rely on a specific DI framework. In order to measure the use of DI, the authors defined four forms of employing DI: constructor and method dependency injection with and without default implementation. They employed a static analysis tool for finding these forms of DI in 34 open source projects. The results show no evidence on the use of investigated forms of DI and indicate that, instead, other mechanisms, such as service locators, were employed by the developers of the analyzed projects.

The results found by Yang et al. [22] make room for a systematic investigation of violations of principles behind DI in source code. Therefore, it is important to characterize elements of source code that are hindering the proper employment of DI in software projects. This work, through a proposal of a catalog of Java DI anti-patterns and subsequent investigation of its occurrence, intends to fill this gap.

3 RESEARCH QUESTIONS AND GOALS

Not properly implementing DI interferes in quality attributes of the source code, such as coupling [13]. However, while there are some mentions of problems and anti-patterns related to DI implementation in industrial publications (e.g., [16] [12]), current research is not properly addressing the topic. Given the benefits DI can bring to software projects, our goal is to review the reported DI anti-patterns in order to analyze their completeness and to propose and evaluate a novel catalog of DI anti-patterns in the context of Java. Such catalog should capture recurring DI related Java source code problems related to bad implementation practices, such as the violation of IoC and DIP principles. To address this goal, we derived three more specific Research Questions (RQs), which are detailed hereafter.

RQ1. Are there additional problem candidates associated with DI implementation that are not properly covered by the currently documented DI anti-patterns? Industry related literature focused on the .NET Framework (e.g., books written for industrial developers, such as [16] [20]) suggests some DI anti-patterns. However, they do not include any kind of evaluation of their suggested anti-patterns. Hence, there is no evidence on their relevance (e.g., the rate of occurrence and the negative effects on source code are not evaluated). Furthermore, reports on DI anti-patterns in the context of Java are lacking. Based on the observation of the recurrence of bad characteristics of DI code elements, such the violation of IoC or DIP principles, we have catalogued a candidate set of Java DI anti-patterns. The resulting Java DI anti-pattern catalog is presented in Section 4.

RQ2. Do the proposed DI anti-patterns occur in practice? It is also important to understand if the proposed DI anti-patterns occur in practice, i.e., it represents problems that are introduced by developers in practice. Otherwise the catalog could be considered useless. To address this research question, we selected four open source software projects that adopt a DI framework and developed a tool to statically analyze the occurrence of the DI anti-patterns within their source code. This empirical evaluation and its results are described in Section 5.

RQ3. What is the acceptance and perceived usefulness from the point of view of experienced developers? Obtaining feedback from experienced developers about the candidate catalogue is an important validation step prior to sharing it with the community. Hence, besides investigating the occurrence of each DI anti-pattern, we designed and conducted an interview-administered survey to assess the acceptance and perception of usefulness from the point of view of expert developers regarding the proposed catalogue. The survey and its results are described in Section 6.

4 A CATALOG OF DI ANTI-PATTERNS

As mentioned in Section 2.2, previously reported DI anti-patterns aim at generic problems. For instance, *Control Freak* can also be found in other contexts where IoC is adopted without DI. In addition, we were not able to identify existing literature regarding DI anti-patterns in the context of Java.

In this section, we describe a candidate catalog of Java DI anti-patterns. Brown et al. [2] advocates for a structural definition of a pattern through a template, because it "assures that important questions are answered about each pattern". Thus, similarly to Arnaoudova et al. [1], we describe each of the candidate DI anti-patterns with the following elements: name, description, negative consequences, pattern of occurrence, and solution. As Gamma et al. [7] argues, a name "is a handle we can use to describe a design problem, its solutions, and consequences in a word or two". The description defines the problem and the context on which it is applied. The description also depicts the structure of the anti-pattern in form of source code. Negative consequences concern the observed drawbacks. Pattern of occurrence depicts a representation of the anti-pattern in source code. The solution describes the means on which the anti-pattern is removed and also presents a snippet that illustrates the source code without the anti-pattern.

Furthermore, we classify the proposed DI anti-patterns into four different classes of problems: *Architecture*, *Design*, *Performance*, and *Standardization*. *Architecture* concerns architectural violation, such as the the violation of IoC and DIP principles. *Design* problems are related to the presence of design issues, such as design smells. *Performance* problem concerns impact on memory usage or response time, such as useless dependency provision. Finally, *Standardization* is related to sticking to a DI coding style, such as following the specification (JSR-330).

In total, our candidate catalog contains twelve proposed DI anti-patterns, which are summarized in Table 3. In subsections ahead, we describe one representative DI anti-pattern from each category. The complete catalog documentation can be found online [5].

4.1 Non used injection

This anti-pattern regards a dependency requested via dependency injection that is actually not used in the class. It overloads the DI container with the incumbency to provide the non used dependency on run time. Worst case scenario if it is not a lightweight object, or if it is not a singleton scope, impacting on performance. This way, non used injection is categorized as a performance problem. Figure 1 presents the structure of occurrence together with an example solution, separated by a dashed line.

Figure 1: Non used injection

```
public class E {

   @Inject
   private ExampleType one;

   public void foo() { /* no reference to one */ }
   public void bar() { /* no reference to one */ }
}
-------------------------------------------------------------
public class E_Without_Non_Used {
   public void foo() { /* code omitted for brevity */ }
   public void bar() { /* code omitted for brevity */ }
}
```

4.2 Concrete class injection

Concrete class injection concerns a dependency requested via dependency injection on which the element type of the dependency is a concrete class. As a design problem, this anti-pattern produces the following negative consequences: first, this solution yields a violation of IoC principle, once the class requesting its dependency acknowledges an implementation detail, i.e. the concrete class; second, this solution introduces less flexibility on testing, once a mock object cannot substitute the dependency on a concrete class; finally, coupling to a concrete class can increase maintenance efforts [7]. Gamma et al. [7] advocates for programming to a interface, not an implementation, which is a natural solution to this anti-pattern. Figure 2 presents the structure of occurrence together with a solution, separated by a dashed line.

Figure 2: Concrete class injection

```
public class B {

   @Inject
   ConcreteExample example;

   private void foo(){
      example.doSomething();
      // code omitted for brevity
   }
}
-------------------------------------------------------------
public class ConcreteExample
    implements IExampleInterface {

   @Override
   public void doSomething() {
      // code omitted for brevity
   }
}

public class B_Without_Concrete {

   @Inject
   IExampleInterface example;

   private void foo(){
      example.doSomething();
      // code omitted for brevity
   }
}
```

[5]https://zenodo.org/record/3066339

Table 3: Catalog of Java DI Anti-Patterns

Identifier	Name	Description	Category
AP1	Intransigent injection	Dependencies that are not needed on construction time, however, are provided by the DI container, introducing additional workload and memory consumption	Performance
AP2	Concrete class injection	Reference on concrete class for injection	Design
AP3	Long Producer method	Method that performs activities that are out of the scope of providing a dependence, which is its main objective	Design
AP4	God DI class	Related to code smell God Class, however, applied to dependencies provided by a DI container	Design
AP5	Non used injection	Dependency requested via DI that is not used	Performance
AP6	Static dependence provider	Usage of static fabrics or Service Locator to obtain a dependence	Architecture
AP7	Direct container call	Relying on DI container in order to obtain a dependence	Architecture
AP8	Open window injection	An injected instance is passed as parameter to another class method or opened for external accessing (e.g. get method)	Design
AP9	Framework coupling	Elements on source code that are dependent on a given DI framework implementation	Standardization
AP10	Open door injection	An injection request is fulfilled by a DI container, however, the instance is opened for modification by an external element (e.g. set method)	Design
AP11	Multiple assigned injection	An injected instance is assigned to multiple attributes (may include external attributes)	Design
AP12	Multiple forms of injection	Refers to the use of multiple forms of injection to a given element, such as attribute and constructor	Standardization

4.3 Direct container call

Direct container calls can provide a concrete implementation at any point of the system. As mentioned in Section 2.2, the nature of this anti-pattern is similar to using a static fabric or a Service Locator. Indeed, negative consequences include high coupling to framework specifics, since it relies directly on the framework to provide the dependency. In addition, again, inversion of control principle is not achieved in this context. Once DI is chosen as an architectural standard for the project, employing container call for dependency resolution conveys an architectural violation. Naturally, a suggested solution relies on applying dependency injection to occurrences of container calls aimed at providing a dependency, as depicted in Figure 3, which shows an example of container call in the Spring framework and a suggested solution, separated by a dotted line.

4.4 Framework coupling

In the context of Java, which presents a specification for DI, a framework specific annotation, for example, incurs in high coupling to the framework. This way, we categorize this anti-pattern as part of standardization category. In addition, in case where compatibility is a requirement, this anti-pattern can lead to greater effort in maintenance activities, framework change or framework version update. A suitable option for removing coupling from a given DI framework is relying on the adoption of annotations presented in the specification. Figure 4 depicts a class that employs Spring framework *@Autowired* annotation and, below the dashed line, the same class, now employing JSR-330 *@Injection* annotation.

5 DI ANTI-PATTERNS OCCURRENCE

This section presents the evaluation of the occurrence of Java DI anti-patterns in the source code of software projects. First, we present DIAnalyzer, a static analysis tool developed with the objective of automatically identifying the occurrence of the proposed DI anti-patterns in software source code. Next, the performance of DIAnalyzer is evaluated through a precision and recall analysis. Finally, based on a set of selected projects, the results of the tool are presented along with the threats.

5.1 DIAnalyzer

In order to support the detection of each proposed DI anti-pattern in source code, a software tool called DIAnalyzer was developed. The tool is a static code analyzer implemented using the JavaParser

Figure 3: Direct container call

```
public class F {

   @Inject
   private Parser parser;

   @Inject
   private ApplicationContext context;

   protected IDataSource getRepository() {
      return (IDataSource)
           context.getBean("ftpDataSource");
   }

   public void execute(List<String> files) {

      IDataSource dataSource = getRepository();

      for(String file : files){
         Object parsedObject = parser.parse(file);
         dataSource.insert( key, parsedObject );
      }
   }
}
------------------------------------------------------------
public class F_Without_Container_Call {

   @Inject
   Parser parser;

   @Inject
   IDataSource dataSource;

   public void execute(List<String> files) {
     for(String file : files){
         Object parsedObject = parser.parse(file);
         dataSource.insert( key, parsedObject );
     }
   }
}
```

[17] library, which relies on Abstract Syntax Trees (ASTs) in order to flag elements of code that represent DI anti-patterns candidates. A rule-based strategy approach was employed to identify the DI anti-patterns. For example, to check whether a class contains the AP7 anti-pattern, in the case of a project employing the Spring framework, we first identify the presence of a coupling to *ApplicationContext* class. Then, based on an attribute declaration of type *ApplicationContext*, we identify method calls to *getBean* from this attribute, passing a string as a parameter. This string identifies either a desirable concrete class or an interface. If there are at least one method invocation of this nature, this code snippet is flagged as containing AP7. The rules applied in order to detect each DI

Figure 4: Framework coupling

```
public class J {

   @Autowired
   Parser parser;

   @Autowired
   IDataSource dataSource;

   public void execute(List<String> files) {
     for(String file : files){
         Object parsedObject = parser.parse(file);
         dataSource.insert( key, parsedObject );
     }
   }
}
------------------------------------------------------------
public class J_Without_Framework_Coupling {

   @Inject
   Parser parser;

   @Inject
   IDataSource dataSource;

   public void execute(List<String> files) {
     for(String file : files){
         Object parsedObject = parser.parse(file);
         dataSource.insert( key, parsedObject );
     }
   }
}
```

anti-pattern are found online[6]. It is worth of mention that the detection strategies applied to AP3 and AP4 were based on Lanza & Marinescu [9].

5.2 DIAnalyzer Evaluation

As there is no available oracle dataset which contains the verified instances of the DI anti-patterns we propose in this work, in order to evaluate DIAnalyzer, we built an oracle by ourselves. The first author of this paper randomly selected a set of classes from latest releases of two projects (Agilefant and Libreplan). These projects were randomly selected among projects with representative usage of JSR-330 annotations. Then, the first author manually identified 141 occurrences of DI anti-patterns (89 from Agilefant and 52 from Libreplan) related to 83 different classes (43 from Agilefant and 40 in Libreplan), concerning eight different DI anti-patterns. Thereafter, the instances identified were handed over to a second researcher that performed a double check on the manually detected instances. Then, the second researcher randomly selected a set of instances for each DI anti-pattern in both projects. In total, 43 manually detected instances were reviewed, confirming them as correctly identified anti-patterns. Nevertheless, we are aware that this activity

[6]https://zenodo.org/record/3066339

Table 4: Precision results of DIAnalyzer

DI Anti-Pattern	**Project**	
	Agilefant	**Libreplan**
AP1	100% (152/152)	100% (145/145)
AP2	- (0/0)	100% (24/24)
AP4	100% (7/7)	80% (4/5)
AP5	100% (19/19)	90% (18/20)
AP6	- (0/0)	100% (5/5)
AP7	100% (2/2)	40% (2/5)
AP8	80% (4/5)	88% (30/34)
AP9	100% (152/152)	100% (144/144)
AP10	100% (84/84)	100% (2/2)
AP11	100% (25/25)	- (0/0)
AP12	100% (1/1)	100% (4/4)

is naturally error-prone and that our oracle may still miss some instances.

We have conducted a relative recall analysis of DIAnalyzer considering the manually generated oracle. During this analysis, DIAnalyzer was able to retrieve 130 out of the 141 manually identified instances, including instances of all eight DI anti-patterns contained in the oracle, resulting in a relative recall of 92.19%. Hence, we were confident that the tool can effectively detect anti-pattern instances. A more detailed analysis on the precision identifying each DI anti-pattern follows.

In order to calculate the precision, we have manually examined every DI anti-pattern detected by DIAnalyzer in a randomly selected scope of classes (43 from Agilefant and 39 from Libreplan). Table 4 shows the precision results. Each row corresponds to an anti-pattern and each column refers to the precision. DIAnalyzer detected 835 instances of DI anti-patterns, with precision between 80 to 100% for AP1, AP2, AP4, AP5, AP6, AP8, AP9, AP10, AP11, and AP12. The reason for the 40% precision on Libreplan regarding AP7 is due to a malformed output of the tool, which duplicates the instance found. As a consequence, several DI anti-patterns were informed more than once, harming the precision results. We have also calculated the average precision of DIAnalyzer per project. The average precision for Agilefant was 97.78% and the average precision for Libreplan was 89.80%. We considered these precision results to be sufficient for our purpose of evaluating the occurrence of the DI anti-patterns in Java projects.

5.3 Detecting DI Anti-Patterns

We apply DIAnalyzer on different software projects with the goal of checking the occurrence of our proposed catalog of DI anti-patterns.

Therefore, the first step is to choose a suitable set of software projects. GitHub was chosen as the repository source of software projects. Our study selected four GitHub projects that meet the following quality criteria: (i) Dependency injection usage within the project, i.e., employing a DI framework, such as the one provided by Spring; (ii) historical developer engagement with several commits; (iii) source code repository mainly written in Java. The list of selected projects is in Table 5, presenting the (i) name, (ii) Java lines of code, and (iii) number of commits for each project.

We applied DIAnalyzer on the latest releases of the four selected projects. The detection results are depicted in Table 6. It is possible to observe that AP1, AP2, AP4, AP5, AP7, AP8, AP9, and AP10 have instances in all four projects. Additionally, all four projects present anti-pattern instances for each DI anti-pattern category (*cf.* Section 4).

The large number of instances for AP1 and AP9 for almost all of the analyzed projects (except for the occurrences of AP9 in project P4) is noteworthy. We believe that the large number of AP1 occurrences is due to the lack of judgment by developers over the need of introducing extra injections in a class. Regarding the large number of AP9 occurrences, it can be explained by a wide adoption of a Spring specific annotation *@Autowired.* On the other hand, AP11 only had instances in P1, suggesting a design choice that led to this anti-pattern in this specific project. AP3 was not found in any project, suggesting that developers of the analyzed systems are aware that dependency provision methods must be highly cohesive and present low complexity.

Table 5: Selected projects

Index	Name	LOC	Commits
P1	Agilefant	58.171	5.166
P2	BroadleafCommerce	327.058	9.146
P3	Libreplan	284.090	9.659
P4	Shopizer	109.792	305

Table 6: Occurrence of the catalog of DI Anti-Patterns

DI Anti-Pattern	**Project**			
	P1	**P2**	**P3**	**P4**
AP1	366	1127	1149	854
AP2	3	277	52	185
AP3	0	0	0	0
AP4	11	20	22	22
AP5	41	215	101	161
AP6	6	21	35	0
AP7	4	45	20	3
AP8	13	122	167	110
AP9	367	152	1102	3
AP10	114	90	2	15
AP11	37	0	0	0
AP12	1	0	5	1

6 PERCEIVED USEFULNESS

In this section we describe an expert survey designed with the main goal of gathering initial results on the perceived usefulness of the proposed catalog of DI anti-patterns. Using the GQM (Goal Question Metric) definition template [21] our goal can be further defined as: *Analyze* the proposed catalog of DI anti-patterns *for the purpose of* characterization *with respect to* the acceptance and perceived usefulness *from the point of view of* software developers with large industrial experience applying DI *in the context of* Java software projects that use DI.

6.1 Survey Design

In order to achieve our goal, a descriptive survey was designed. As stated by Linaker et al. [10], a descriptive survey provides the support to make claims or assertions about a subject. Indeed, once we are claiming that a proposed catalog of DI anti-patterns can actually be characterized as anti-patterns, a descriptive survey is suitable for our need. In order to decide upon the target population, we followed the advice of checking whether the selected developers are the most appropriate to provide accurate answers instead of focusing on hopes to get high response rates [18]. Thus, our target population is comprised of practitioners with large expertise on applying DI principles and frameworks in software projects.

Regarding the questionnaire type, an interviewer-administrated questionnaire was employed in order to avoid threats of validity. According to Linaker et al. [10], using this kind of questionnaire the interviewer can help to clarify ambiguous questions. This way, we were able to support the interviewee in completely understanding the purpose of each proposed DI anti-pattern along the interview. The questionnaire guiding the interview consisted of three parts.

The first part aimed at gathering information on the interviewees academic background and industrial experience. The second part consisted of questions regarding the proposed catalogue of DI anti-patterns. For each anti-pattern, the following information is provided: (i) the name of the DI anti-pattern, (ii) a short description of the DI anti-pattern, (iii) a characterization of occurrence in form of source code, (iv) the negative consequences, (v) a description of a possible resolution, and (vi) a characterization of resolution in form of source code. Based on this information, the interviewees were asked to answer the following question for each anti-pattern: "Can the proposed DI anti-pattern actually be characterized as an anti-pattern?". The answer is provided based on a five-point Likert scale (1- Agree, 2- Partially Agree, 3- Neutral, 4- Partially Disagree, and 5-Disagree). In addition, the interviewees were invited to add comments over the general structure of the analyzed DI anti-pattern and possible disagreements.

According to Turner et al. [19], the Technology Acceptance Model (TAM) [5] is suitable to capture the user's acceptance of a given technology. Having this in mind, the final part of the questionnaire involved gathering feedback based on TAM questions to assess three acceptance model constructs: usefulness, ease of use, and intention to use. The complete instrumentation employed in our survey can be accessed online[7].

[7]https://zenodo.org/record/3066339

6.2 Survey Results

The execution strategy consisted of identifying a sample of the population according to the survey design. Prioritizing experts with relevant industrial experience applying DI, we identified three interviewees from three different units of two different companies. The results of the survey are also shown in Tables 7, 8, and 9.

Based on Table 7, it is possible to observe that respondents have strong skills in object-oriented analysis and design. I3 does not posses a strong experience in DI, however, I3 was able to assess the proposed anti-patterns due to having strong software design skills.

The results of the perception on the proposed DI anti-patterns are shown in Table 8. It is noteworthy to mention that from 39 inquiries over DI anti-patterns, we observed only 2 (partial) disagreements (AP1 and AP9). AP1 is the only anti-pattern proposed that does not have any full agreement response. I1 mentions that "AP1 does not yield an anti-pattern when it comes to lightweight objects". In addition, I2 asserts that "dependencies that are not needed on construction time should be moved to another class in order to save resources". For AP9, I2 argues that "most projects do not change the chosen DI framework" and I3 argues that "the anti-pattern applies only when compatibility is defined as a requirement".

On the other hand, 37 responses concerned "Agree" (31) and "Partially Agree" (6) responses, which yields 94.8% of the total answers. In addition, from the 13 proposed anti-patterns, 11 contain at least two full agreements. Most of these are from the architecture and design problems categories. Regarding the partially agree responses, in AP3, I1 agreed the occurrence is bad, but argued that "it is not directly related to DI". Also, in relation to AP11, I1 did not agree with the example solution provided, arguing that "the problem exposed in the structure of occurrence is a poorly implemented refactoring". The comments provided by the respondents suggest that the partial agreements regard context-based situations (e.g., situations in which the code structure represents a problem depending on the requirements). We believe that this result reflects the fact the complete context information of the anti-patterns was not included in the survey. Overall, given the experience of the respondents on design principles and patterns, these observations provide a positive perception of the catalogue.

The adapted TAM questions and their results are shown in Table 9. It is possible to observe a strong positive perception, once 25 from 27 questions yield an agreement response. Only T2 and T6 present neutral responses. The positive results on perceived usefulness, ease of use and intention to use indicate that our proposed catalogue is helpful and that developers would show willingness to apply it.

7 THREATS TO VALIDITY

Internal Validity. Regarding the proposal of the catalog, although we covered a set of 12 anti-patterns distributed in four categories, our catalog may miss some other DI related problems. Therefore, all authors have reviewed the composition and completeness of the catalog. In addition, our approach may miss some instances of DI anti-patterns in the source code, once every software project may show different implementation characteristics. To mitigate this threat we evaluated the DIAnalyzer tool regarding relative recall and precision. We believe we have identified most of the DI

Table 7: Background of respondents

Information	Respondent		
	I1	I2	I3
Academic background	Master	Bachelor	PhD
English reading and comprehension skills	Advanced	Advanced	Advanced
Experience developing software	>10y	>10y	>10y
Current position	Project Manager	Tech Leader	Tech Leader
Object-oriented analysis and design	Several projects in industry	Several projects in industry	Several projects in industry
Design principles and patterns	Several projects in industry	Several projects in industry	Several projects in industry
Anti-patterns	Several projects in industry	A project in industry	A project in industry
Source code inspection	Several projects in industry	Several projects in industry	Several projects in industry
Dependency injection	Several projects in industry	Several projects in industry	A project in industry
Java	Several projects in industry	A project in industry	Several projects in industry

Table 8: Perception over the proposed DI anti-patterns

DI Anti-Pattern	Respondent		
	I1	I2	I3
AP1	Partially disagree	Partially agree	Partially agree
AP2	Agree	Agree	Agree
AP3	Partially agree	Agree	Agree
AP4	Agree	Agree	Agree
AP5	Agree	Agree	Agree
AP6	Agree	Agree	Agree
AP7	Agree	Agree	Agree
AP8	Agree	Agree	Agree
AP9	Agree	Partially disagree	Partially agree
AP10	Agree	Agree	Agree
AP11	Partially agree	Agree	Agree
AP12	Agree	Agree	Agree

anti-patterns in source code of the analyzed software projects. Finally, regarding the survey, the interview-based approach was used specifically to clarify any doubts regarding the questions and the proposed catalog, not biasing respondents towards their agreement.

External Validity. We targeted Java software projects for characterizing and detecting DI anti-patterns. Some of the DI anti-patterns proposed may not be directly applicable to projects implemented in other programming languages. However, although we focused on Java-based systems, we believe the catalog is generic enough to port its ideas to another object-oriented programming language (e.g. C#). Additionally, albeit selecting projects with different number of LOC and commits, all of them are implemented using the Spring framework. We plan to address this problem in the future by including closed source projects that are implemented using other Java DI frameworks. Regarding the survey, we selected representatives of experienced developers from three different organizational units.

Construct Validity and Reliability. Since there is no benchmarking dataset for DI anti-patterns in source code, we built an oracle dataset by ourselves in order to evaluate DIAnalyzer. The dataset was built and verified by two independent researchers. The process for building the oracle is similar to Chen & Jiang [3] work. Lastly, the oracle dataset does not contain instances of AP3. We believe this does not undermine our findings, since this anti-pattern did not appeared in any analyzed project. Additionally, our survey was peer reviewed by two independent researchers and intentionally conducted with a small sample, to allow in-depth qualitative discussions on our initial proposal.

8 CONCLUDING REMARKS

DI is adopted in software projects in order to improve quality attributes, such as coupling. Although Java presents well-designed DI frameworks, such as Spring and Guice, there is no comprehensive guideline on how to avoid DI anti-patterns. This work introduced an initial catalog of DI anti-patterns targeting Java-based systems.

In order to assess the practical relevance of our catalog (i.e., the occurrence of the problems), we developed DIAnalyzer, which statically identifies DI anti-pattern occurrences in Java source code. The detection results suggests that the DI anti-patterns occur frequently in software projects.

Additionally, we designed an expert survey in order to gather the perception of developers over the catalog. The survey results indicate that our proposed catalog concerns relevant DI problems. Also, the usefulness of the catalog was observed by a large agreement in the TAM questionnaire.

Table 9: Respondents perception over the proposed catalogue of DI anti-patterns

Dimension	Index	Question	Respondent		
			I1	I2	I3
Usefulness	T1	Being aware of the proposed DI anti-patterns would improve my performance in preventing DI related problems in software systems (i.e. preventing faster)	Strongly Agree	Strongly Agree	Strongly Agree
	T2	Being aware of the proposed DI anti-patterns would improve my productivity in preventing DI related problems in software systems (i.e. preventing more and faster)	Agree	Strongly Agree	Neutral
	T3	Being aware of the proposed DI anti-patterns would enhance my effectiveness in preventing DI related problems in software systems (i.e. preventing more)	Strongly Agree	Strongly Agree	Strongly Agree
	T4	I would find the proposed catalog of DI anti-patterns useful in my job	Strongly Agree	Strongly Agree	Strongly Agree
Ease of use	T5	Learning to use the proposed catalog of DI anti-patterns would be easy for me	Strongly Agree	Agree	Agree
	T6	I would find it easy to use the proposed catalog of DI anti-patterns to prevent DI related problems in software systems	Strongly Agree	Strongly Agree	Neutral
	T7	It would be easy for me to become aware of the proposed catalog of DI anti-patterns	Strongly Agree	Strongly Agree	Agree
	T8	I would find the proposed catalog of DI anti-patterns easy to apply	Strongly Agree	Strongly Agree	Agree
Intention to use	T9	I intend to apply the proposed catalog of DI anti-patterns regularly at work	Strongly Agree	Strongly Agree	Neutral

In the future, we want to extend the number of analyzed software projects, including closed source projects from industry and software projects that adopt different DI frameworks. Also, after our initial qualitative evaluation, we aim at adapting our survey to replicate it with more developers.

REFERENCES

[1] Venera Arnaoudova, Massimiliano Di Penta, Giuliano Antoniol, and Yann-Gael Gueheneuc. 2013. A New Family of Software Anti-Patterns: Linguistic Anti-Patterns. In *European Conference on Software Maintenance and Reengineering*.

[2] William J. Brown, Raphael C. Malveau, Hays W. McCormick III, and Thomas J. Mowbray. 1998. *AntiPatterns Refactoring Software, Architectures, and Projects in Crisis.* John Wiley & Sons.

[3] Boyuan Chen and Zhen Ming (Jack) Jiang. 2017. Characterizing and Detecting Anti-patterns in the Logging Code. In *International Conference on Software Engineering*.

[4] M. Crasso, C. Mateos, A. Zunino, and M. Campo. 2010. Empirically Assessing the Impact of DI on the Development of Web Service Applications. *Journal of Web Engineering* 9 (2010), 66–94.

[5] F. Davis. 1989. Perceived usefulness, perceived ease of use, and user acceptance of information technology. *MIS Quarterly* 13 (3) (1989), 319–340.

[6] Martin Fowler. 2004. Inversion of Control Containers and the Dependency Injection pattern. http://martinfowler.com/articles/injection.html

[7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Longman Publishing Co.

[8] Ralph E. Johnson and Brian Foote. 1988. Designing Reusable Classes. *Journal of Object-Oriented Programming* 1, 2 (1988), 22–35.

[9] M. Lanza and R. Marinescu. 2006. *Object-Oriented Metrics in Practice.* Springer.

[10] J. Linaker, S. M. Sulaman, R. Maiani de Mello, and M. Höst. 2015. *Guidelines for Conducting Surveys in Software Engineering.* Technical Report. Lund University. [Publisher Information Missing].

[11] R. C. Martin. 1996. The Dependency Inversion Principle. *Report.* 8(6) (1996), 61–66.

[12] Dhanji R. Prasanna. 2009. *Dependency Injection.* Manning Publications Co., Greenwich, CT, USA.

[13] E. Razina and D. Janzen. 2007. Effects of Dependency Injection on Maintainability. In *International Conference Software Engineering and Applications.*

[14] JSRs: Java Specification Requests. 2015. JSR 330: Dependency Injection for Java. https://www.jcp.org/en/jsr/detail?id=330

[15] Serguei Roubtsov, Alexander Serebrenik, and Mark van den Brand. 2010. Detecting Modularity Smells in Dependencies Injected with Java Annotations. In *Software Maintenance and Reengineering European Conference.* 244–247.

[16] Mark Seemann. 2012. *Dependency Injection in .NET.* Manning Publications Co., Shelter Island, NY.

[17] Nicholas Smith, Danny van Bruggen, and Federico Tomassetti. 2018. *JavaParser: Visited.* Leanpub. https://leanpub.com/javaparservisited

[18] Marco Torchiano, Daniel Méndez Fernández, Guilherme Horta Travassos, and Rafael Maiani de Mello. 2017. Lessons learnt in conducting survey research. In *Proceedings of the 5th International Workshop on Conducting Empirical Studies in Industry.* IEEE Press, 33–39.

[19] Mark Turner, Barbara Kitchenham, Pearl Brereton, Stuart Charters, and David Budgen. 2010. Does the technology acceptance model predict actual use? A systematic literature review. *Information and Software Technology* 52 (2010), 463–479.

[20] Steven van Deursen and Mark Seemann. 2018. *Dependency Injection Principles, Practices, Patterns.* Manning Publications Co., Shelter Island, NY.

[21] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, and A. Wesslén. 2012. *Experimentation in software engineering.* Springer.

[22] Hong Yul Yang, Ewan Tempero, and Hayden Melton. 2008. An Empirical Study into Use of Dependency Injection in Java. In *Australian Conference on Software Engineering.*