Pontifícia Universidade Católica
do Rio de Janeiro

Rodrigo Nunes Laigner

Cataloging Dependency Injection Anti-Patterns in Software Systems

Dissertação de Mestrado

Dissertation presented to the Programa de Pós–graduação em Informática da PUC-Rio in partial fulfillment of the requirements for the degree of Mestre em Informática.

Advisor: Prof. Marcos Kalinowski

Rio de Janeiro
February 2020

Pontifícia Universidade Católica
do Rio de Janeiro

Rodrigo Nunes Laigner

Cataloging Dependency Injection Anti-Patterns in Software Systems

Dissertation presented to the Programa de Pós–graduação em Informática da PUC-Rio in partial fulfillment of the requirements for the degree of Mestre em Informática. Approved by the Examination Committee.

Prof. Marcos Kalinowski
Advisor
Departamento de Informática – PUC-Rio

Prof. Alessandro Fabricio Garcia
Departamento de Informática – PUC-Rio

Prof. Leonardo Gresta Paulino Murta
Instituto de Computação – UFF

Rio de Janeiro, February 10th, 2020

Rodrigo Nunes Laigner

The author received his Bachelor degree in Information Systems from the Instituto de Computação (IC) of Universidade Federal Fluminense (UFF) in 2017, in cooperation with University of Colorado Denver (UCD). Previously to graduate studies, he led several projects in industry related to software engineering and data management. During the course of the graduate studies, he participated in several projects in academic settings focused in the areas of software architecture and design. His main research interests are: Software Architecture, Software Design, and Big Data Software Engineering.

Bibliographic data

Laigner, Rodrigo Nunes

Cataloging Dependency Injection Anti-Patterns in Software Systems / Rodrigo Nunes Laigner; advisor: Marcos Kalinowski. – Rio de janeiro: PUC-Rio , Departamento de Infor-mática, 2020.

v., 140 f: il. color. ; 30 cm

Dissertação (mestrado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática.

Inclui bibliografia

1. Informática – Teses. 2. injeção de dependência;. 3. inversão de dependência;. 4. inversão de controle;. 5. anti-padão;. 6. java;. 7. catálogo;. 8. design;. 9. arquitetura;. 10. modularização;. 11. acoplamento;. 12. refatoração.. I. Kalinowski, Marcos. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Acknowledgments

First and foremost, I would like to thank God, the Father Almighty, Maker of heaven and earth and of all things visible and invisible.

I would also like to thank my family, specially my mom, my grand aunt, my sister, and my uncle. Without your support, this journey would not be possible. You are everything to me. I extend this gratitude to my two respected cousins, which I get really inspired, Brunno and Rondinelli.

A special thanks to my friends Julio, Gabriel, Max, Pedro, Rafael, Philipe, Isadora, Lauro, Asher, Vitor, and Arthur. Your friendship are very special to me.

A very special gratitude goes to my advisor Marcos Kalinowski. I would like to thank Marcos for guiding me into the marvelous academic path. Your wisdom, humbleness, hard work, and motivation are encouraging.

My sincere thanks also go to the members of my thesis defense, Alessandro Garcia and Leonardo Murta. Your commitment to software engineering research and education are inspiring. I would like to thank you a lot for the contributions on this thesis. I would also like to thank all professors from PUC-Rio for their contribution to my education. A special thanks to Sérgio Lifschitz, Noemi Rodriguez, Darlan Arruda, and Marco Vaz Salles, for their support throughout this journey.

I extend my gratitude to my colleagues and friends from PUC-Rio, Tecgraf and OPUS. In particular, I would like to thank Amadeu, Waldir, Rogério, Daniel, Gustavo, Julio, Angelo, Luiz, Diogo, Anderson, Hugo, Alan, Tassio, Marx, and Paulo for all giving support and friendship.

I am very grateful for the opportunity to work at Tecgraf/PUC-Rio in one of the most exciting projects of my life. A special thanks to Processamento de Eventos Complexos group. I extend my gratitude to Leonardo Barros for the support and patience throughout this journey.

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

For last, I gratefully acknowledge PUC–Rio for the financial support.

Abstract

Laigner, Rodrigo Nunes; Kalinowski, Marcos (Advisor). **Cataloging Dependency Injection Anti-Patterns in Software Systems**. Rio de Janeiro, 2020. 140p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Background 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, such as increasing coupling, hindering the achievement of DI's main goal. Even though the literature suggests the existence of DI anti-patterns, there is no detailed documentation of such bad practices. Moreover, there is no evidence on their occurrence and perceived usefulness from the developer's point of view. **Aims** 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 Java DI anti-patterns. **Method** We propose a catalog containing 12 Java DI anti-patterns. We selected 4 open-source and 2 closed-source software projects that adopt a DI framework and developed a tool to statically analyze the occurrence of the candidate 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. We extended the survey in order to gather the perception of a set of 15 expert and novice developers through an online questionnaire. **Results** At least 9 different DI anti-patterns appeared frequently in the analyzed projects. In addition, the feedback received from the developers confirmed the relevance of the catalog. Besides, the respondents expressed their willingness to refactor instances of anti-patterns from source code. **Conclusions** The 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.

Keywords

dependency injection; dependency inversion; inversion of control; anti-pattern; java; catalog; design; architecture; modularization; coupling; refactoring.

Resumo

Laigner, Rodrigo Nunes; Kalinowski, Marcos. **Catalogando Anti-Padrões de Injeção de Dependência em Sistemas de Software**. Rio de Janeiro, 2020. 140p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Contexto Injeção de Dependência (DI) é um mecanismo comumente aplicado para desacoplar classes de suas dependências com o objetivo de prover uma melhor modularização do software. No contexto de Java, a existência de uma especificação de DI e frameworks populares, como o Spring, facilitam o emprego de DI em projetos de software. Entretanto, más práticas de implementação de DI podem trazer más consequências, como maior acoplamento, dificultando alcançar o principal objetivo de DI. Apesar de a literatura sugerir a existência de anti-padrões de DI, não há uma documentação detalhada de tais más práticas. Em adição, não há evidência da ocorrência e da percepção de utilidade dos mesmos do ponto de vista de desenvovedores. **Objetivos** Nosso objetivo é revisar os anti-padrões de DI reportados com o objetivo de analisar sua completude e propor um novo catálogo de anti-padrões de DI para Java. **Método** Nós propomos um catálogo contendo 12 anti-padrões de DI para Java. Nós selecionamos 4 projetos open-source e 2 projetos closed-source que adotam um framework de DI e desenvolvemos uma ferramenta que analisa estaticamente a ocorrência dos anti-padrões de DI candidatos no código fonte das aplicações. Em adição, nós conduzimos uma pesquisa por meio de entrevistas face a face com três desenvolvedores experientes que regularmente aplicam DI em seus projetos. Nós estendemos a pesquisa com o objetivo de obter a percepção de um conjunto de 15 desenvolvedores experientes e novatos por meio de um questionário online **Resultados** Ao menos 9 anti-padrões de DI apareceram frequentemente nos projetos de software analisados. Em adição, a avaliação recebida dos desenvolvedores confirmaram a relevância do catálogo. Por fim, os respondentes expressaram o desejo de refatorar as instâncias de anti-padrões de DI propostas. **Conclusões** O catálogo contém anti-padrões de DI que ocorrem na prática e são úteis. Compartilhar com praticantes da indústria os permitirá evitar a introdução de anti-padrões em seus projetos de software.

Palavras-chave

injeção de dependência; inversão de dependência; inversão de controle; anti-padão; java; catálogo; design; arquitetura; modularização; acoplamento; refatoração.

Table of contents

List of figures

List of tables

List of Abreviations

AOP – Aspect-Oriented Programming
DI – Dependency Injection
IoC – Inversion of Control
DIP – Dependency Inversion Principle
OOP – Object-Oriented Programming
OOD – Object-Oriented Design
LOC – Lines of Code
AST – Abstract Syntax Tree
TAM – Technology Acceptance Model
API – Application Programming Interface
IDE – Integrated development environment

1
Introduction

In this chapter, we provide the context of the work presented in this dissertation, along with our motivation to tackle the problem herein described. Next, we present our problem statement and the methodology followed in order to answer our research questions.

1.1
Context and Motivation

Software engineering research have been investigating several approaches for decreasing coupling in software systems over time, such as design patterns [12] and aspect-oriented programming (AOP) [15]. Coupling is a quality attribute of a module in an application. As higher the level of coupling to another modules of the system, the likelihood of increased efforts when it comes to introduce change is expected in this module [12]. In other words, a software system that holds higher coupling levels will probably incur in an increased maintenance effort. A particular mechanism that has been explored in a lesser extent towards the same end, it is, decreasing coupling levels in an application, is Dependency injection (DI), a mechanism for improving software modularity.

DI enables less coupling among modules by refraining them from being aware of implementation details of each other [31]. DI is built upon two design principles: dependency inversion principle (DIP) and inversion of control (IoC). The first suggests a design oriented to abstractions, while IoC relies refraining the control of the application to a third-party module or framework.

DI has become a common practice in the software industry, as characterized by the existence of DI frameworks and industry-oriented publications [24] [32]. For instance, Spring [29], one of the most popular Java frameworks, and Google AdWords[1], 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.

DI frameworks are usually employed in software systems in order to support practitioners to achieve lower coupling levels among modules, by

[1]https://github.com/google/guice/wiki/AppsThatUseGuice

capturing the needed dependencies of a module in run-time and providing the module with concrete implementations that fulfill the given dependencies. In order to accomplish this task, DI frameworks often employ a component called DI container in order to autonomously handle dependency provision at run-time.

Despite the existence of well designed frameworks, such as Spring [29] and Guice [13], that provide programming mechanisms to support achieving the benefits DI is supposed to supply, such as code element annotations, 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 [32][36] conjectures the existence of DI anti-patterns, these propositions do not provide a comprehensive analysis over the state of the practice of such bad DI implementation characteristics in source code. For instance, it is unknown if these propositions are generic (most are presented in the context of a programming language or framework) enough to port its ideas to other contexts (if they are broad) and its practical relevance (occurring within a broad range of software projects). Most importantly, there is no evidence acceptance and perceived usefulness from developers' point of view.

It is worthy to mention the words of Jim Couplin [8] on anti-patterns:

> The study of AntiPatterns is an important research activity. The presence of 'good' patterns in a successful system is not enough; you also must show that those patterns are absent in unsuccessful systems. Likewise, it is useful to show the presence of certain patterns (AntiPatterns) in unsuccessful systems, and their absence in successful systems

Furthermore, as asserted by Brown et al. [2], anti-patterns decidedly lead to negative consequences. For instance, in the context of DI, consequences vary from high coupling to potential overuse of memory. Consequently, it is unknown if these proposals effectively support practitioners on their development activities that involves the employment of DI in industrial settings.

This overall scenario motivates further empirical evaluations to properly document a catalog of DI anti-patterns in software systems.

1.2 Problem Statement, Goal, and Research Questions

There is an assumption raised in industrial settings that as long as the source code implements DI through applying DI frameworks, the structural quality of the program is improved. However, this assumption might not always hold. By scattering bad implementation practices related to DI through source code, developers might unconsciously introduce design, architectural, performance, and standardization problems that make the source code more susceptible to contain bugs, depending on the complexity of the DI code element involved, and more difficult to change. For instance, not properly implementing DI interferes in quality attributes of the source code, such as coupling [25].

Unfortunately, there is limited guidelines about how to avoid bad implementation practices related to DI in software systems. Industry related literature focused on the .NET Framework (e.g., books written for industrial developers, such as [32], [36]) 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. While there are some mentions of problems and anti-patterns (e.g. [24] and [32]) related to DI implementation in industrial publications, current research is not properly addressing the topic. The lack of a comprehensive catalog of DI anti-patterns may also undermine this specific technique to be better applied together with other techniques that also aim at reducing coupling in a software system, such as AOP. Thus, the general problem is described as follows.

General Problem: While properly employing DI is an important step towards improving structural quality of software systems, there is a lack of guidance on how to effectively detect, analyze, and remove DI anti-patterns from elements of source code.

Based on this problem, using the design science [46] template, our research goal can be defined as follows:

Improve the structural quality of software systems that employ DI **by** proposing and evaluating a catalog of DI anti-patterns **that satisfies** providing comprehensive guidance on detecting, analyzing, and removing DI related problems in software systems **in order to** support practitioners in their development activities.

In our case, we had some idealized assumptions drawn up from our previous industrial experience and knowledge on design principles and patterns

in software engineering and its related literature. Our assumptions at this early time were: (1) bad implementation practices associated to DI usage form DI anti-patterns (2) DI anti-patterns are not directly associated with a given DI framework, but rather, concerns mainly the violations of the principles behind DI, DIP and IoC, and (3) each specific DI anti-pattern occurs several times throughout the source code.

To address our goal and also investigate whether our assumptions hold, we derived three Research Questions (RQs), which are detailed hereafter.

RQ1. Are there problem candidates associated with DI implementation that are not properly covered by the currently documented DI anti-patterns?

Although industry-oriented publications address DI anti-patterns [32] [36], little is known about their practical relevance, i.e., if the suggested anti-patterns occur in practice and if they are relevant by the point of view of developers. Moreover, investigating the completeness of documented DI anti-patterns will enable us to uncover the main gaps in current literature. Besides, such analysis will allow us to reason over characteristics of source-code elements that potentially lead to problems, such as architectural violations and code smells. Lastly, the product of this investigation will enable us to propose a catalog of DI anti-patterns.

RQ2. Do the proposed DI anti-patterns occur in practice?

By answering RQ2 we will be able to understand whether the proposed DI anti-patterns are relevant to the state of the practice. In other words, we seek to understand if the proposed catalog of DI anti-patterns represents problems that are introduced by developers in practice. Otherwise, the catalog could be considered useless. Lastly, RQ2 will allow us to advance to a more in-depth analysis of the conjectured catalog, this time by gathering the opinion of expert developers about the proposed DI anti-patterns catalog.

RQ3. What is the acceptance and perceived usefulness from the point of view of experienced developers?

Obtaining feedback from experienced developers about the proposed catalog is an important validation step before sharing it with the community. Answering RQ3 will allow us to conclude if the proposed catalog is comprehensive enough to support developers in their day by day development activities working with software systems that employ DI mechanisms.

1.3 Research Methodology

The research methodology was designed based on the research questions and involves uncovering gaps in current documentation of DI anti-patterns and proposing an initial catalog of DI anti-patterns (RQ1), investigating whether the instances of DI anti-patterns proposed catalog occur in practice (RQ2), and investigating the acceptance and perceived usefulness of the catalog from the point of view of developers (RQ3). The methodology employed for each of these steps is detailed hereafter.

1.3.1 Proposing an initial catalog of DI anti-patterns

First, we start by reviewing reported DI anti-patterns and other instances of problems related to DI employment in software systems. The objective is to analyze their completeness with the objective of uncovering gaps in current propositions in order to to come up with an initial catalog of DI anti-patterns. While industry-oriented publications mainly focus on .NET, we opted to target at the Java platform due to the following reasons: (a) the lack of documentation regarding bad DI implementation practices; (b) the existence of a myriad of DI frameworks (such as Guice [13] and Spring [29]); (c) large industrial adoption ; (d) the existence of a Java DI specification (JSR-330) [26]; (e) and the large number of open source software repositories written in Java.

We employ two methodological approaches to derive an initial proposition of DI anti-patterns, inductive and deductive. The inductive approach was primarily based on our experience in industrial settings. For the deductive approach, we conjectured on possible DI anti-patterns based on design principles, such as General Responsibility Assignment Software Patterns (GRASP) and SOLID. At the end of this process, RQ1 is answered.

1.3.2 Investigating practical occurrence of the proposed catalog

Next, after coming up with an initial catalog of DI anti-patterns, it is important to investigate the feasibility of our proposition. In other words, we seek to verify the practical relevance of the catalog by gathering the rate of occurrence of each anti-pattern instance in the context of both open and closed-source software projects. Therefore, we have developed a static analysis tool to automatically detect instances of the proposed anti-patterns from the source code of software systems. We have selected a set of open-source

software repositories from GitHub and two closed-source projects to perform our analysis on them. At the end of this process, RQ2 is answered.

1.3.3 Investigating acceptance and usefulness of the proposed catalog

Finally, we assess the acceptance and usefulness of our proposal from expert developers by designing an expert survey. Although investigating the rate of occurrence of such proposed anti-patterns may lead to practical relevance of the proposition, we still need to gather developers' perceptions over the proposed catalog. Specifically, we aim at gathering their perception over each proposed anti-pattern instance to verify if they can be characterized as anti-patterns. Besides, we assess whether the developers are willing to use our catalog to guide their development process. At the end of this process, RQ3 is answered.

1.4 Outline

The remainder of this work is organized as follows. Chapter 2 provides the background on principles behind DI, design principles, and structural problems in software engineering.

Chapter 3 briefly introduces related work about DI and DI anti-patterns.

Chapter 4 presents our initial effort on cataloging Java DI anti-patterns. We describe our method to come up with instances of anti-patterns and also explain their negative consequences and code transformations to remove such anti-patterns from source code.

Chapter 5 introduces our initial effort on identifying instances of DI anti-patterns in software systems through a manual approach. Then, it introduces a static analysis tool developed to automatically identify DI anti-patterns. Lastly, it presents the results of the occurrence of each DI anti-pattern in software projects.

Chapter 6 presents the investigation and results on the acceptance and perceived usefulness of the proposed catalog of DI anti-pattern instances from the point of view of developers.

Finally, Chapter 7 presents the concluding remarks of this Master's thesis and discusses future work.

2
Background

2.1
Introduction

DI has become a common practice in the software industry, characterized by the existence of a reference specification, such as Java JSR-330 [26], DI frameworks, such as Spring [29], and industry-oriented publications, such as [24]. However, the principles behind DI and the concept of DI are heterogeneously defined in the literature.

Thus, this section provides a comprehensive background on design principles, comprising the principles behind DI and general design principles used in this work.

Thereafter, we describe the DI concept and forms of using DI based on current programming languages and frameworks. Besides, a subtle analysis of best practices on applying DI based on the work of [24] is presented. Then, we introduce the Java specification for DI (JSR-330).

Finally, the terminology on code smells and anti-patterns are introduced to better position the reader on understanding related work.

2.2
Design Principles

2.2.1
Inversion of Control

In the eighties, object-oriented programming (OOP) was in the midst of an increasing trend of interest from the research community. At that point, the Smalltalk programming language emerged as the main object of attention from industry and academia, mainly due to the object-oriented paradigm. In this context, Johnson and Foote [14] work influenced software reuse in OOP, by introducing fundamental concepts to design reusable classes, such as Inversion of Control (IoC).

The term IoC was first introduced by Johnson and Foote [14] in the context of software frameworks. To make use of functionalities provided by a framework, it is usually required to extend a set of classes. Meanwhile, this

extension often refrains the system from controlling its execution, incurring in an inversion of control. This behavior is exemplified by Johnson and Foote [14] as follows.

> One important characteristic of a framework is that the methods defined by the user to tailor the framework will often be called from within the framework itself, rather than from the user's application code. The framework often plays the role of the main program in coordinating and sequencing application activity

Thus, by extending a set of classes of the framework, a framework oftentimes departs the system from controlling part of its execution life cycle. This concept is fundamental to understand the role of a DI framework plays in a software system.

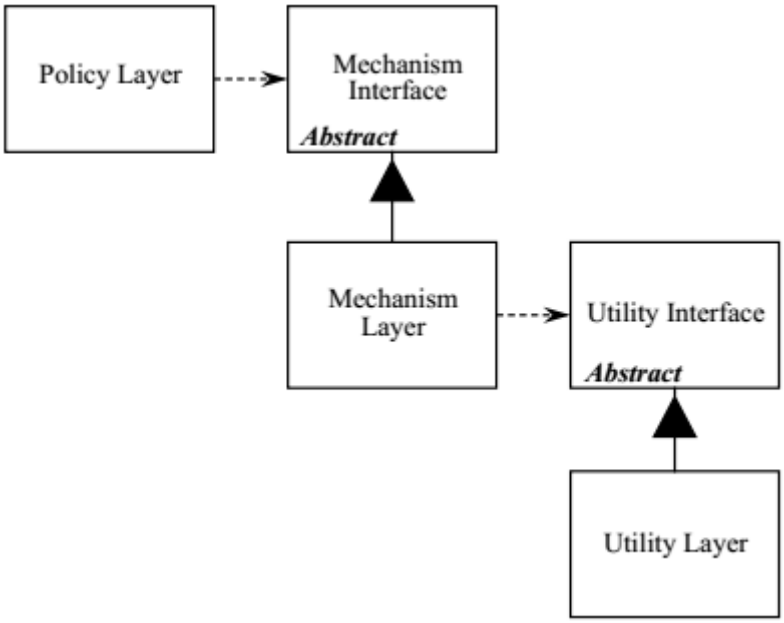
2.2.2 Dependency Inversion

The dependency inversion principle (DIP) was introduced by Martin [21] and is part of the SOLID [20], a set of module design principles. The DIP states that modules should not depend on the details of another module. Instead, abstractions should be used in place of direct dependencies to enable reuse [21]. Through Figure 2.1, Martin [21] exemplifies this concept as follows.

> Each of the lower level layers are represented by an abstract class. The actual layers are then derived from these abstract classes. Each of the higher level classes uses the next lowest layer through the abstract inter-face. Thus, none of the layers depends upon any of the other layers.

In other words, Martin [21] states that a module must not directly reference a module from another layer of an application. Rather, every module should be a realization of an abstraction. This way, modules from lower layers won't be able to be coupled to details of higher layers, thus abstracting implementation details and enforcing a transitive dependency chain. By enforcing this pattern, a system may be more susceptible to less efforts when it comes to changes over time. Lastly, based on Martin's [21] description, Yang et al. [38] assert that in Java, DIP means that "a class should depend on interface or abstract types, not on concrete types".

Figure 2.1: Abstract Layers (extracted from Martin [21])



2.2.3 Open-closed

Meyer [22] asserts that "a module should be open for extension but closed for modification," sentence which characterizes the Open-closed principle (OCP). By stating that a module should be open for extension, Meyer [22] means that in case of the necessity of different behaviors in a given module, a design that is oriented to extension must be enforced in this given module. In other words, different behaviors must take place when it comes to requirements change or meeting new needs of the application.

On the other side, by "closed for modification," [22] it is important to recognize that exposing internal information of a module to external modules may allow the modification of behavior that should not change, incurring in a risk to the application.

2.2.4 GRASP

General Responsibility Assignment Software Patterns, a.k.a. GRASP, are principles that govern the assignment of responsibility to classes in object-oriented design (OOD), which are also known as patterns of assigning responsibilities. According to Larman [17], responsibility in the context of OOD is the "obligations of an object in terms of its behavior." Responsibilities are divided in two types: (a) *knowing* and (b) *doing*.

Larman [17] asserts that *knowing* responsibilities concern knowing about private encapsulated data or related objects, as well "knowing about things it can derive or calculate." In other words, *knowing* is about being aware of information and computation that can be done by the object itself. Moreover, *doing* responsibilities concern "doing something itself" and "initiating action in other objects." In addition, *doing* is about "controlling and coordinating activities in other objects." In other words, *doing* is about holding information on who to call in order to receive the results of a computation. A subtle example of such division of responsibilities is provided by Larman [17] as follows.

> I may declare that "a Sale is responsible for creating SalesLineItems" (a doing), or "a Sale is responsible for knowing its total" (a knowing)

Although GRASP presents in total 9 patterns [17], we will present the ones that are related to this work.

Information Expert, also known as *Expert*, is centered on reasoning over the assignment of responsibility to the class that has the information to fulfill it. The information may be a computation composed of a set of fields (private encapsulated data) or the aggregated result of a set of actions initiated in other objects.

Creator primarily reasons on which class has the responsibility to create an instance of another class. Based on Larman's example [17], let's consider class B and A. Thus, if B aggregates, contains, has initializing data, or records instances of A, then B may be responsible for the creation of A instances.

Next, according to Larman [17], coupling "is a measure of how strongly one element is connected to, has knowledge of, or relies on other elements." Also, he adds that "a class with high [...] coupling relies on many other classes."

Thus, ***Low Coupling*** is a principle that aims at avoiding the consequences of high coupling, such as forcing changes in related classes due to local changes, complexity in the process of comprehending the behavior of a class, refrain the achievement of reuse. Indeed, Low Coupling principle is about assigning responsibilities effectively, "so that (unnecessary) coupling remains low."

Besides, Larman [17] asserts that "cohesion is a measure of how strongly related and focused the responsibilities of an element are." In other words, if several unrelated responsibilities are placed in a given class, we may consider this a class with low cohesion. It is possible to assert that low cohesion is related to the presence of high coupling, because the negative consequences are somewhat close, as stated by Larman [17], on which he argues that low cohesion

classes are "hard to comprehend," "hard to reuse," and "hard to maintain." Thus, the principle of ***High Cohesion*** is about assigning responsibilities so that cohesion remains high.

Pure Fabrication concerns assigning "a highly cohesive set of responsibilities to an artificial or convenience "behavior" class that does not represent a problem domain concept — something made up, in order to support high cohesion, low coupling, and reuse." There are cases where subjective objects, the ones that are not directly related to the problem domain of the application, must be created on run time. For instance, objects that support database-centric operations, such as obtaining a pool of connections or persisting a tuple in the database management system. In order to do not couple your domain and logic classes to database operations, abstracting these operations in fabricated objects may support the achievement of high cohesion, low coupling, and reuse in the application.

Indirection is related to the problem of "where to assign a responsibility to avoid direct coupling between two (or more) things" [17]. Larman [17] asserts that Indirection regards the assignment of "responsibility to an intermediate object to mediate between other components or services, so that they are not directly coupled."

2.3 Dependency Injection

2.3.1 Concept

IoC and DIP form a basis for comprehending DI. According to Crasso et al. [9], 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. [38], 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. [12]. In the context of DI, Crasso et al. [9] assert that the responsibility for component creation is given to a DI container, as depicted in 2.2. In particular, the DI container is usually employed in order to enable the IoC principle in DI. A DI container is the component of a DI framework responsible for dependency provision at run time, acting as a mediator in cases

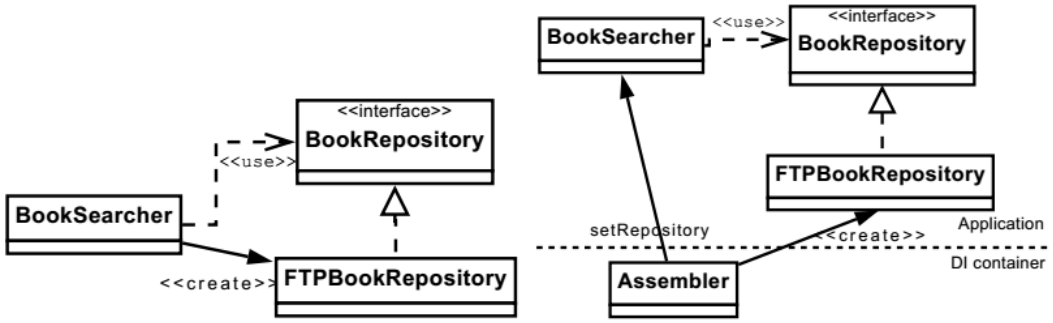
where a given dependence is needed by a class.

Furthermore, 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.

Figure 2.2 presents an example of the role of a DI container in a software system. On the left side (a) of Figure 2.2, the class *BookSearcher* is responsible for the creation of an instance of the *FTPBookRepository* class to make use of the *BookRepository* interface. This scheme introduces a coupling between *BookSearcher* and *FTPBookRepository*, a concrete implementation. It is worthy to mention that, in case of changes to *FTPBookRepository* constructor, for example, it will trigger changes to *BookSearcher*, incurring a greater maintenance effort.

Besides, on the right side (b) of Figure 2.2, instead of *BookSearcher*, the DI container (represented by the module *Assembler*) is the one responsible for acknowledging the coupling of *BookSearcher* to *BookRepository* interface, and providing a concrete implementation (*FTPRepository*) to *BookSearcher*.

Figure 2.2: Dependence provision without DI (a) and using DI (b) (extracted from Crasso et al. [9])



2.3.2 Forms of Dependency Injection

As mentioned earlier, the main component of a DI framework that handles dependency provision is the DI container. However, there are different ways to let the DI container know how to provide a given dependence. This acknowledgment is expressed through program elements, such as constructors and set methods. Fowler [11] describes that the characteristic on which a

dependency is provided as form of DI, although other author [24] describes the same as *Injection idioms*. Fowler [11] defines three main forms of dependency injection, depicted as follows.

Interface injection: Based on defining and implementing interfaces, analogous to Figure 2.2 (b) example.

Setter injection (or set method injection): Relies on *set* methods to receive and assign a concrete implementation to an attribute of the class.

Constructor injection: Analogous to setter injection, however, it receives concrete implementations on constructor method.

At first, Spring framework 1.0 version relied on XML files for configuring the concrete implementation classes to be instantiated on run time. This form of configuration was considered error-prone and with the introduction of annotations on Java platform, frameworks, such as Guice [13], turned to rely solely on annotations in order to configure dependence provision in an application. The annotations follow the JSR-330 specification [26] and introduced alternative forms of DI, as described below.

Attribute injection (or Field injection): Injection occurs directly on an attribute annotated with *@Inject* (as defined by JSR-330 [26]) or *@Autowired* (Spring specific annotation for injection). This pattern removes the need to create a set method for the attribute or inserting a parameter with same type (interface) in class constructor (for posterior assignment to a class variable).

Method injection: Although a setter and constructor are methods, this form of injection is explicit on JSR-330 [26], where an ordinary method annotated with *@Produces* (in Spring, the *@Bean* annotation is also used) let the DI container aware that some of its parameters need an injected instance. Lastly, it is important to mention that direct container calls can provide a concrete implementation (if correctly configured) at any point of the system.

Figure 2.3 depicts a simple injection occurring in an attribute of a class, where the attribute is annotated with *@Inject*. This injection is usually performed on class initialization, although some frameworks adopt lazy initialization (only when the attribute is actually referenced in run time) for injecting an instance.

Figure 2.3: Attribute injection

```
public class A {

   @Inject
   private ExampleType exampleTypeAttr;

   // code omitted for brevity
}
```

Figure 2.4 depicts an injection occurring through a *set* method for a given attribute of a class. The *set* method is annotated with *@Inject* to let the DI container aware the class needs a proper instance (of the type expressed in the parameter of the *set* method). This injection is usually performed on class initialization.

Figure 2.4: Set method injection

```
class B {

    private ExampleType exampleTypeAttr;

    @Inject
    public void setExampleTypeAttr(ExampleType exampleParam) {
        this.exampleTypeAttr = exampleParam;
    }

}
```

Figure 2.5 depicts an injection occurring in the constructor of a class. The constructor is annotated with *@Inject* to let the DI container aware that the parameters of the constructor should be injected. This injection is decidedly performed on class initialization.

Figure 2.5: Constructor injection

```
class C {

    private ExampleType exampleTypeAttr;

    @Inject
    public void C(ExampleType exampleParam) {
        this.exampleTypeAttr = exampleParam;
    }

}
```

Figure 2.6 depicts an injection occurring in the context of a method that provides an instance of a given class type. Thus, the method is annotated with *@Provides* to let the DI container aware that the parameters of the method should be injected. This injection is decidedly performed when an instance of *AnotherExampleType* is requested by another class instance.

Figure 2.6: Method injection

```
class D {

    @Provides
    public AnotherExampleType ordinaryMethod(ExampleType
        exampleParam) {
        // do something with exampleParam
        // returns an instance of AnotherExampleType
    }

}
```

Lastly, Figure 2.7 shows an excerpt of a direct call to DI container (in Spring, this is fulfilled through *ApplicationContext* class) with the objective to request a given dependence on run time. In this case, a concrete implementation of *IDataSource* is requested through method *getRepository*.

Table 2.1: 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
Resource	Same purpose of *@Inject* annotation. The Resource annotation marks a resource that is needed by the application.

Figure 2.7: Direct container call

```
public class E {

    @Inject
    private ApplicationContext context;

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

}
```

2.3.3 Java Dependency Injection Specification

With the widespread of DI frameworks for Java, such as Dagger[1], an effort to conceptualize a standardization gave birth to JSR-330 [26], a specification for DI implementation. The specification is adopted by Guice [13] and Spring [29], and defines a set of annotations, which are described in Table 2.1.

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

According to the specification [26], a *Singleton* "identifies a type that the injector only instantiated once. Not inherited." A type (i.e., a class) annotated with *@Singleton* will be instantiated once a time across all the execution run time of the application. It means that, every time an object needs an instance of a class annotated with *@Singleton*, excluding the first call for instantiation, the same instance of the class will be provided over time.

An example of a class that is a feasible *Singleton* is a class that provides logging capabilities. Once instantiated, it is possible to assert that more than one class in an application will require logging capabilities. This way, as a *singleton*, it avoids the replication of different instances of the same class over the run time.

Scopes were first defined by the JSR-299 specification [27] and are used in the context of web applications to support communication between client and server. Scope infers a level of restriction, so in the context of web applications, scopes are rules applied to the life cycle of a given object. Furthermore, the JSR-330 [26] specification details that

> By default, if no scope annotation is present, the injector creates an instance (by injecting the type's constructor), uses the instance for one injection, and then forgets it. If a scope annotation is present, the injector may retain the instance for possible reuse in a later injection."

In the context of DI, a *Qualifier* annotation is "applied to injection points to distinguish which implementation is required by the client" [38]. An example is the type of protocol used to send a message, such as UDP or TCP, which can be defined on run time by a qualifier annotation.

Named is a string based qualifier that can be used to provide more granularity on dependence provision process. It can be used in an analogous way to *Qualifier* annotation.

Lastly, *Inject* annotations defines an injection point for which a DI container must provide an object instance. It can identified in constructors, methods, and fields, as already addressed. The same applies for *Resource* annotation.

2.3.4 The Process of Injecting a Concrete Implementation

As mentioned in Section 2.3.1, when employing DI, even though a software project follows an interface-oriented design, there is still the task

to instantiate a concrete implementation. In general, DI frameworks delegate this responsibility to the DI container, as shown in Figure 2.2.

Most often, software systems design a specific concrete implementation and the work of the DI container is facilitated. However, it may occur that an interface is implemented by two or more concrete classes. In this case, the container must be advised about which concrete implementation to inject.

In this section we provide a brief explanation on how Java DI frameworks deal with the instantiation of concrete implementations. Also, we explain some language constructs that allow for defining which specific concrete implementation must be instantiated for a given injection point.

At first, DI frameworks, such as Spring, tied a concrete class to a given dependence via external XML file. A simplified example extracted from the project Broadleaf is shown in Figure 2.8, where the interface AdminPermission is binded to the concrete implementation named AdminPermissionImpl.

In Spring, the annotation *@Component* is employed in a class to let the container aware this class must be provisioned in injection points. However, by using XML injection definition, concrete implementations do not need to be annotated with *@Component*, since the injection is explicitly defined in the XML file.

Figure 2.8: XML injection definition

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <!-- Entity mappings - override for extensibility -->
    <bean
        id="org.broadleafcommerce.openadmin.server.security.domain.AdminPermission"
        class="org.broadleafcommerce.openadmin.server.security.domain.AdminPermissionImpl"
        scope="prototype"/>

</beans>
```

However, developers often claimed this approach was error-prone and led to rework in the case of software evolution. This context limited a broader adoption of Spring framework at the time. Pushed by the Java DI specification, explained in Section 2.3.3, frameworks (e.g., Google Guice) opted to allow the

definition of injection points through annotations, such as *@Inject* (see Section 2.3.2).

Although configuring the bindings to each interface through XML bring concerns, they offer a flexible constructs to allow binding different concrete implementations to an interface requested in different injection points. Based on the example retrieved from Broadleaf project (2.8), we extend this example to accommodate the exhibition of different concrete implementations (*AdminPermissionImpl* and *AdminPermissionImpl2*) that implement the same interface (*AdminPermission*) being injected into different injection points of the system (class *ExampleClassDefault* and *ExampleClass2*).

Figure 2.9: Example of different bindings to an interface

```
<bean
  id="implDefault"
  class="org.broadleafcommerce.openadmin.server.security.domain.
  AdminPermissionImpl" />

<bean
  id="impl2"
  class="org.broadleafcommerce.openadmin.server.security.domain.
  AdminPermissionImpl2" />

<bean
  id="exampleDefault"
  class="com.example.ExampleClassDefault" >
    <property name="service" ref="implDefault" />
</bean>

<bean
  id="example2"
  class="com.example.ExampleClass2" >
    <property name="service" ref="impl2" />
</bean>
```

Besides, through the *Qualifier* annotation (mentioned in Table 2.1) it is also possible to assign a given concrete implementation to a specific injection point. The semantics are similar to the one provided in Figure 2.9. However, some can contest this approach, once the annotation is an element of the source code, explicitly couples an element of an injection point to a concrete implementation.

Figure 2.11: Example of provider method

```
@Produces
public IDataSource provideDataSource(){
    IDataSource dataSource = null;
    // assign an instance of concrete implementation to dataSource
    return dataSource;
}
```

There may exist cases where the developer cannot bind a specific implementation to an injection point. For instance, an example case is where the instantiation to be provided by the DI container is directly dependent on a given state or business rules of the application. This context calls for a dynamic instantiation of concrete implementations.

Through XML-based configuration, a factory method can be employed to deal with such dynamic instantiation. As shown in Figure 2.10, the instance to be injected in the property *dataSource* in class *ExampleBean* is fulfilled by the method *getDataSource* in class *IDataSourceFactory*.

Figure 2.10: XML-based instantiation through factory method

```
public class IDataSourceFactory {
    public IDataSource getDataSource() {
        // code ommitted
    }
}
-----------------------------------------------------------------------
<bean id="dataSourceFactory"
  class="com.example.IDataSourceFactory" />
<bean id="dataSourceRef"
  class="com.example.IDataSourceFactory"
  factory-method="getDataSource" factory-bean="dataSourceFactory" />
</bean>
<bean id="exampleBean" class="com.example.ExampleBean">
    <property name="dataSource" ref="dataSourceRef" />
</bean>
```

In addition, through annotation-based configuration, the annotations *@Provides* (for Google Guice) and *@Bean* (for Spring framework) can also be used to provide dynamic instantiation of dependencies. These methods are often called provider methods. An excerpt of a provider method is shown in Figure 2.11, where an instance of *IDataSource* is returned.

Lastly, it is important to explain (i) how the DI container actually acknowledges the components to bind. In this manner, a brief description on how Spring framework manages the provision of dependencies is provided as follows. In case of XML-based configuration, the application must explicitly point out to Spring framework which set of files contain the configuration of the bindings to be realized by the DI container. Then, each XML file is parsed and the bindings retrieved from the XML file are consecutively processed and stored in memory, in data structures that provide fast lookup of components, such as Hash Tables. Next, the DI framework verifies, for each component scanned from the XML files, if the necessary dependencies can be provisioned based on the configuration provided.

Besides, there are two cases of bindings: (i) components that are lazy-loaded and (ii) components that are instantiated at the start of the application. The first regards bindings that are only consumed when the component requesting the instantiation is actually loaded by the application. The second class of binding is about loading every component (and their dependencies) that is not marked as lazy-loading at the start of application. In this cases, there are typically two types of component, singleton and prototype. The first type regards keeping the same instance of the component for each injection point for the whole application life cycle. The second regards creating a different instance of the component each time it is referred.

On the other side, on annotation-base configuration, instead of parsing XML files, at the start of the application, the source code is inspected, class by class, and each component information (its name and its respective dependencies) is stored in data structures in memory. Then, the same process observed in the XML-based configuration is also employed in this case. The DI framework verifies, for each component, if the necessary dependencies can be provided (if there is a concrete implementation that fulfills it). If at least one component does not have the necessary dependencies, an error is throw by the framework when the application starts.

In general, the DI container initializes at application startup all wired components, it is, components that are marked as 'managed' by the DI container. In case of lazy-loaded components, at runtime, when it comes to fulfill this type of dependence request, the DI container tries to instantiate it with its respective dependencies. If it is not possible to instantiate such lazy-loaded component, an exception is often raised by the framework.

2.3.5 Dependency Injection Best Practices

To the best of our knowledge, although there is not a catalog of DI patterns in literature, there are propositions of design considerations concerning DI adoption. Prassana [24] argues that the problem on choosing an injection idiom must take into consideration aspects about testing and maintainability. Also, he argues that "there are such important consequences to either choice [of injection idiom] that potentially lead to difficult, underperformant, and even broken applications." In this section, we assert about some considerations over choosing an injection idiom.

As mentioned earlier, the problem of breaking an application can take place in the context of state management of attributes in a class. For instance, favoring *set method injection* may incur in two classes of problem, as explained by Prassana [24]. First, by using setter injection, the attribute receiving the instance provided by the DI container cannot be declared as *final*. It opens up the possibility of a direct call to this same set method, possibly incurring in the overwrite of the instance provided by the DI container. Second, once setter method is preferred in a given application, usually other form of injection is not used. However, by not enforcing the assignment of value to an attribute on construction time, unit tests that make use of this specific attribute might be compromised.

Prassana [24] asserts that "[c]onstructor injection affords us the ability to create immutable dependencies by declaring fields as final". Indeed, constructor injection decidedly enable a design oriented to immutable fields, as opposed to setter injection. A simple tentative on changing the value of the assigned injected instance on source-code won't enable the compilation of the class. On the other hand, Prassana [24] argues that "a constructor-injected object cannot be created unless all dependencies are available," which does not enable the introduction of lazy injected instances in order to accelerate construction time. Moreover, Prassana [24] asserts that "the explicitness of setter injection can itself be an advantage, as [c]onstructors that take several arguments are difficult to read."

Thus, there is a not a perfect injection idiom, since each one of them provides singular benefits and drawbacks, as explained earlier. Moreover, we would like to assert that the injection idiom must be chosen per use case, taking into consideration aspects related to concurrency, maintenance, testing, and performance.

Moreover, Prassana [24] discuss that singleton-scoped objects are different from singleton objects (a.k.a singleton pattern). A singleton-scoped object

is instantiated by the DI container once (and is active throughout all application life cycle) for each injection point. For instance, let's consider the singleton-scoped class *B*. Also, let's consider class *A* with a constructor injection providing an instance of type *B* as the first parameter. If the second parameter of the constructor method is also from type *B*, a different instance is provided by the DI container.

Prassana [24] asserts that singleton-scoped objects has advantages over objects with no scope: (a) "objects can be created at startup [...], saving on construction and wiring overhead", and (b) "there is a single point of reference when stepping through a debugger." Moreover, Prassana [24] argues that "business objects are perfect candidates for singleton scoping. They hold no state but typically require data-access dependencies." In addition, he [24] asserts that "if an external resource is designed to be held open over a long period, then yes, it may warrant singleton scoping." Lastly, Prassana argues that some reasoning over the use of singleton-scoped objects must take place, since "object graphs that have several dependencies, which themselves have dependencies, and so on, are not necessarily expensive to construct and assemble."

On the other side, a singleton object (originated from singleton pattern), although its instance is also active through the entire application life cycle, the life of a singleton object is tied to the life of the application (and not to an injection point). Prassana [24] argues that singleton objects often brings difficulties in the testing process, where "any object created and maintained outside an injector [i.e. managed by a DI container] does not benefit from its other powerful features, particularly life cycle [management] and interception"

For instance, a singleton-scoped object often refrains the developer from binding different instances of a particular object in different test cases, as argued by Prassana: "[DI allows you to] quickly bind a key to a different scope simply by changing a line of configuration."

2.4 Structural Problems

2.4.1 Code Smells

Code smell is a suspect of high level about a problem found in one or more elements of the source code that usually leads into a structural problem in the system. Literature agrees that although a code smell does not decidedly yield a deeper structural problem, the odds on incurring in a problem are

generally higher than the opposite [4]. One of the most prominent code smells covered in literature is *God Class* [23]. According to Cedrim [4], a *God Class* is characterized by holding several responsibilities, introducing difficulties in the process of comprehending the class when it comes to read, modify, and evolve the same. In addition, according to Lanza and Marinescu [16], a God Class "refers to those classes that tend to centralize the intelligence of the system. An instance of a god class performs most of the work, delegating only minor details to a set of trivial classes and using the data from other classes".

Another code smell that is often found in software systems is *Long Method*. This smell concerns an increased complexity entailed in a given method. This complexity is often derivative from multiple lines of code and extensive use of foreign attributes.

2.4.2 Anti-Patterns

Patterns in software engineering have its origin in the work of of Jim Coplien [7]. According to Brown et al. [2], Coplien's paper titled "A generative development-process pattern language" set a startpoint "for the patterns movement to incorporate not just software design patterns, but analysis, organizational, instructional, and other issues as well."

Coplien's work has preceded the influential work on software design patterns, Design Patterns: Elements of Reusable Object-Oriented Software [12]. Brown et al. [2] asserts that a design pattern is a "common, practical software design constructs that could be easily applied to most software projects." Another important definition is given by Larman [17], where "a pattern is a named problem/solution pair that can be applied in new context, with advice on how to apply it in novel situations and discussion of its trade-offs." In other words, a design pattern is a solution construct to a recurring problem observed in software systems.

On the other hand, the antithesis of a pattern is often referred to an anti-pattern. 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." A more compelling description of an anti-pattern comes from the same work Brown et al. [2], as follows.

> AntiPatterns provide real-world experience in recognizing recurring problems in the software industry and provide a detailed remedy for the most common predicaments. AntiPatterns highlight the most common problems that face the software industry and provide the tools to enable you to recognize these problems and to determine

> their underlying causes. Furthermore, AntiPatterns present a detailed plan for reversing these underlying causes and implementing productive solutions.

An additional definition is provided by Arnaoudova et al. [1], where an anti-pattern is recognized as "the opposite to design patterns," identifying "poor solutions to recurring design problems." Moreover, Arnaoudova et al. [1] assert that anti-patterns "are generally introduced by developers not having sufficient knowledge and–or experience in solving a particular problem, or misusing good solutions, i.e., design patterns."

In line with these definitions, differently from a design pattern, anti-patterns are effective ways to communicate to practitioners how to avoid bad implementation practices in source code. Thus, 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.

2.5 Concluding Remarks

The concept of DI is not uniformly defined in the literature. This way, this chapter addresses the terminology adopted throughout this dissertation by presenting the design concepts behind DI and formalizing the concept of DI. This formalization is important so next chapters are motivated without ambiguity.

As explained, DI is a particular mechanism for applying inversion of control in an application, once an auxiliary element (DI container) takes on the responsibility for solving a module's dependencies on run time, which are concrete implementations.

In the context of DI, it is noteworthy to mention that the DI container is an instance of *Creator* pattern depicted in GRASP, since he is responsible for binding instances of dependencies to classes on run time, thus allowing lower coupling among modules, since modules do not need to acknowledge dependencies or additional information in order to create external dependencies.

Critics over the indirection of responsibility that the DI container brings may exist. For instance, the DI container removes from system modules their responsibility to provide the necessary dependencies its own dependencies required in order to be instantiated.

However, it is noteworthy to mention that DI container is further employed in the context of service classes and utilitary classes [24], such as

logging, database connectivity, and framework coupling in general, as can be identified ahead in some analyzed projects.

3
Related Work

3.1
Introduction

In this chapter, we present studies that are similar to our own. This effort enables the identification of gaps in the literature that we aim to fill by this dissertation. As discussed before, there is not a comprehensive guide to identify anti-patterns related to the use of DI in software projects. Although technical literature suggests the existence of DI anti-patterns, there is no validation over the documented instances. Consequently, practitioners have little or no support on how to effectively identify DI anti-patterns in source code.

As this work is carried out in the intersection of anti-patterns and DI, 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.

Thus, in this section, we present related work distributed on five topics: the forms to use DI, DI and web services, DI and maintenance, DI bad smells, and DI anti-patterns. Additionally, we also present related work on building catalogs of anti-patterns in software engineering.

3.2
Dependency Injection Forms

Yang et al. [38] 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.

3.3 Dependency Injection and Web Services

Crasso et al. [9] investigated the impact of DI on the development of web service applications in the context of DI4WS, a development model that allows for service discovery and consumption. Crasso et al. [9] found out that DI enables faster development, cleaner code, looser coupling and simplifies service discovery, even though the overhead on memory allocation is higher compared to other design alternatives.

3.4 Dependency Injection and Maintenance

Razina and Janzen [25] conducted a study to measure the effects of the use of DI on software maintainability. In particular, the authors [25] focused on investigating the coupling and cohesion level of modules that apply the DI mechanism. They [25] selected a set of 20 open source systems, where each set contains a pair of software systems. The pair is composed by a project that employs DI mechanism and one similar project that does not employ DI. The authors [25] relied on three metrics to uncover the maintainability level of the projects: coupling between objects [3], response for class [3], and lack of cohesion in metrics [5] .

The authors [25] found that "[t]here does not appear to be a trend in lower coupling or higher cohesion measures with or without the presence of dependency injection." However, "a trend of lower coupling in projects with higher dependency injection percentage (more than 10%) was evident."

Although the results exposed relevant findings, the validity of the work is questioned due to: (a) how the pairs of software systems were composed and (b) the characteristic (in terms of LOC and type of application) of the applications selected for the study.

3.5 Dependency Injection Bad Smells

Roubtsov et al. [31] 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. A summary of the proposition is shown in Table 3.1. The first column represents the

Table 3.1: Summary of DI bad smells from Roubtsov et al. [31]

Principle	**Annotations**
Configuration should be separated from functionality	Application server: *@Install* and *@Startup* Web server: *@Path* and *@RequestMapping* Database Server: *@Table* and *@Column*
Information should not be duplicated	Principle violated by interface annotations mentioning the interface implementation (@ImplementedBy, @ProvidedBy)
Information should not be duplicated	Annotations duplicating the database structure (@Id, @OneToOne, @OneToMany, @ManyToOne, and @ManyToMany)
Interfaces should not be explicit	Java interceptor annotations (@Interceptors, @AroundInvoke)

modularity principle behind the bad smell and the second column exhibits the annotations involved in the respective violation.

The authors [31] assert that annotation *@ImplementedBy* triggers a potential inconsistency due to maintenance. It is important to address that this impact is only achieved in case of the introduction of another implementation possibility. In addition, Roubtsov et al. [31] argue that circular dependency can be achieved between interface and its implementation. Although the authors [31] did not provide an example, design patterns, such as Factory, can be used as mechanisms for solving circular dependencies.

Regarding configuration annotations, such as *@Install* and *@Startup*, [26] address that separating application server configuration and build files is a feasible resolution. It is important to observe that configuration annotations are mechanisms by which frameworks can introduce its own behavior in the application on run time. Then, coupling in this context cannot be excluded. However, it is possible to provide an interface oriented design, isolating classes annotated with these configuration annotations, on which the binding of the instance is accomplished by a DI container. Isolating these classes in a different component is in charge of developer, being a pure architectural choice.

Over dependence on the web-server violation, on which the authors [31] claim that "redeployment of the software on a new server is hindered by presence of explicit dependencies on web-pages", web-deployment descriptors

(configuration files) is not the only solution. It is noteworthy that annotations mainly goal is to diminish the need for configuration files.

Lastly, about annotations concerning database structure, I would assert that due to Java Persistence API, a programming interface specification, Roubtsov et al. [31] argument over the impact of redeploying the Java system on a new server due to presence of explicit dependencies on the database tables, is not conceivable. As a pattern on the Java platform (Java persistence API), redeployment in a new server is unlikely to require the removal of persistence API annotations.

3.6 Dependency Injection Anti-Patterns

The only explicit proposal of DI anti-patterns is the one described by Seemann [32] and Deursen and Seemann [36], which contains a set of four DI anti-patterns, shown in Table 3.2.

Table 3.2: DI anti-patterns extracted from Seemann [32]

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

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 [11], is described by Seemann [32] 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.

3.7 Catalogs of Anti-Patterns in Software Engineering

As mentioned in Section 2.4.2, software engineering literature has explored catalogs with the objective to clearly communicate to practitioners best practices when it comes to the codification of software. Soon, software engineering researchers and practitioners realized that although there were clear guidelines on promoting good object-oriented design, applications still suffered from several problems, such as poor design, technical debts, and bugs.

In line with the words of Arnaoudova et al. [1], researchers became aware that, more than patterns, practitioners should be warned about what not to do or what practices to avoid when it comes to design software projects. Thus, catalogs of anti-patterns have been designed in order to overcome these aforementioned problems. A set of works that introduce anti-patterns in software engineering and were very influential to this work is presented afterward.

In "A New Family of Software Anti-Patterns: Linguistic Anti-Patterns," Arnaoudova et al. [1] introduce software linguistic anti-patterns, which represents "recurring, poor naming and commenting choices" on source code. For instance, some consequences entailed by linguistic anti-patterns are misunderstanding the main proposal of a method, confounding the return type of a method base on its name, and unexpected allocation of new objects.

Next, in "Characterizing and Detecting Anti-patterns in the Logging Code," Chen and Jiang [6] conduct a study to to address the problem of anti-patterns in the logging code, which they define as "recurrent mistakes which may hinder the understanding and maintainability of the logs." Although previous work focused on *where-to-log* and *what-to-log*, Chen and Jiang [6] tackled the problem from the perspective of *how-to-log*. According to the authors [6], this problem concerns the development and maintenance of high-quality logging code, specifically those related to performance issues. Chen and Jiang [6] asserts that "excessive logging may cause unexpected side-effects like performance slow-down or high disk I/O bandwidth."

Then, they [6] examined three open source software systems in order to

uncover bad practices related to logging code. Thus, they were able to find 64 representative instances of logging code anti-patterns and most of them were accepted by developers that maintained the aforementioned open source systems examined.

3.8 Concluding Remarks

Software engineering researchers have carried out many studies on improving structural quality of software systems [4]. However, we observe that existing empirical and mining studies on structural quality lack in-depth discussion over DI anti-patterns. The studies analyze aspects related to the use of forms of DI [38], and the effects of DI on development [9] and maintainability [25] of software systems. In this chapter, we presented studies that are similar to our own. This effort enables the identification of gaps in the literature that we aim to fill by this dissertation.

The results found by Yang et al. [38] stating that forms of DI are not commonly used in software systems make room for a systematic investigation of violations of principles behind DI in source code.

Regarding the work of Roubtsov et al. [31], even though the authors provide means for resolving each smell, they provide 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 smells heavily focused on annotations related to the J2EE persistence model, which are based upon Java Persistence API (JPA), a specification for persistence in Java.

The DI anti-patterns addressed by Seemann [32] and Deursen and Seeman [36] 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.

As discussed before, although technical literature suggests the existence of DI anti-patterns, there is no validation over the documented instances. Consequently, practitioners have little or no support on how to effectively identify DI anti-patterns 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. In this way, there is an opportunity to research in these topics. The next chapters present the efforts to fulfill these identified gaps.

4
Proposing a Catalog of Java Dependency Injection Anti-Patterns

4.1
Introduction

As mentioned in Chapter 3, 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 summary, reported DI anti-patterns and DI code smells fail to depict their application context scenario, and most importantly, fail to present evidence on their practical relevance.

In order to address such limitations, in this chapter we report on our efforts in documenting a candidate catalog of Java DI anti-patterns. As previously mentioned, in this proposal, we focused on the Java platform due to the following reasons: (a) the lack of documentation regarding this specific platform, (b) the existence of a myriad of DI frameworks (such as Guice [13] and Spring [29]), (c) industrial large adoption (i.e. it is easier to find developers to contribute with opinions over the catalog), (d) a specification aimed to DI (JSR-330) [26], (e) and the large number of open source software repositories written in Java available.

The anti-patterns were derived from two criteria: First, based on the observation of the recurrence of bad characteristics of DI code elements, such as the violation of DIP or IoC principles. These were observed in industry projects by the author while maintaining software in practice and evolved through discussions with researchers. Second, as DI is supposed to improve structural quality of object-oriented applications, we also explored a set of DI anti-patterns that could be present in source code, harming design principles, such as GRASP [17].

4.2 Method

From a methodological point of view, there are typically two approaches for coming up with new propositions: inductive and deductive [40].

The inductive approach relies on observation of a phenomena to uncover a pattern (or set of patterns) that might lead to a theory. According to Lodico et al. [18], "the researcher [, through inductive reasoning,] uses observations to build an abstraction or to describe a picture of the phenomenon that is being studied." In this context, we have observed the state of the practice while maintaining software projects that adopts DI framework in industrial settings.

On the other side, deductive approach concerns "developing a hypothesis [...] based on existing theory, and then designing a research strategy to test the hypothesis" [39]. It means that existing theory is used as a basis for establishing a proposal so that evidence can be gathered based on a strategy developed to evaluate the proposal. For this matter, we relied on the mapping of violations of design principles (theory) , i.e. GRASP and SOLID, in the context of the employment of DI in software projects. This mapping would enable us to hypothesize over possible anti-patterns (hypothesis).

Through maintaining software projects in industry, in efforts related to corrective and evolutionary software maintenance, the author was able to preliminarily identify patterns in elements of the source code that violated design principles behind DI, namely, IoC and DIP, and also some of the design principles presented in GRASP and SOLID. The experience maintaining software projects lasted 7 months, and 3 closed-source projects from an industrial partner were maintained in the period. All 3 projects were information systems developed in Java aimed at supporting business processes in different organizations. Due to the disclosure agreement on exposing information about the closed-source projects maintained at the time, we cannot described further details about code elements involved.

Rather than the inductive approach, deductive reasoning relies on theory to hypothesize about a phenomenon that might occur in real-world. In our context, we rely on the theory of GRASP and SOLID design principles to complement the proposition of the DI anti-patterns that might occur in practice. Some excerpts of reasoning over the existence of DI anti-patterns are provided hereafter.

In regard to GRASP, the *Creator* pattern advocates for reasoning upon which class (A) is responsible for instantiation of another class (B). A factor that drives the assignment of responsibility in the Creator pattern is the presence of dependencies in A that B needs in order to be instantiated by

A. Thus, the presence of a direct container call harms the Creator pattern, once the container is a generic class provided by the DI framework in order to support instantiation of objects in scenarios where injection of elements is not possible, such as testing integration with third party libraries.

Next, GRASP introduces the *Indirection* principle, which concerns the introduction of a mediator object in the context of two communicating objects. Indirection is achieved in DI by means of IoC and the employment of the DI container. It is worthy to note that any tentative to swipe the control of the framework, refraining the DI container from the responsibility to provide instances on run time, to mediate the communication between modules may be defined as an anti-pattern.

Besides, in the JSR-330 specification, the annotation *@Provides* is responsible for letting the DI container aware that a given dependency must be provided by the method annotated. Thus, it is inferred that these methods are very cohesive, it is, it should enforce the principle of Low Coupling.

In addition, by receiving an injected element (it does not matter in which form), opening this specific code element for modification entails in the violation of the OCP. The violation occurs because it is not guaranteed that the correctness of the program is maintained. Also, enforcing a design not oriented to abstractions causes the violation of the DIP. Lastly, fabricating instances of objects in domain classes may incur in the violation of the Pure Fabrication principle.

4.3 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. [12] 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

Table 4.1: Catalog of Java DI Anti-Patterns (Part 1)

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

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 Tables 4.1 and 4.2. In subsections ahead, we describe each DI anti-pattern, addressing its respective category, negative consequences, and suggested resolution.

4.3.1 Intransigent injection

Intransigent injection concerns dependencies that are not needed on construction time, however, they are decidedly provided by the DI container on construction time. This scheme introduces additional workload and memory consumption on construction time. In other words, overuse of object allocation

Table 4.2: Catalog of Java DI Anti-Patterns (Part 2)

Identifier	Name	Description	Category
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

in memory during construction time is entailed. It is worse scenario is observed if it's not a lightweight object, impacting on performance. Thus, this anti-pattern is categorized as a performance problem. Figure 4.1 presents the structure of occurrence followed by an example solution, separated by a dashed line. In the occurrence example, the injected attribute *example1* is not used in construction time, thus, the process of injecting a given instance in this attribute might require additional workload to DI container. The example resolution provided concerns relying on a *Provider*, an interface type defined by JSR-330 that is responsible for providing a given instance when it is requested. Thus, in the example resolution, an instance for the injected attribute *example1* is only provided when its use is required.

4.3.2 Concrete class injection

Concrete class injection concerns a dependence requested via dependency injection on which the element type of the dependence 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 dependence acknowledges an implementation detail, i.e. the con-

Figure 4.1: Intransigent injection

```
public class A {

   @Inject
   private IExampleInterface0 example0;
   @Inject
   private IExampleInterface1 example1;

   public A() {
       example0.doSomething();
   }

   public void foo() { /* omitted code */ }

   public void bar() {
       example1.doSomething();
       /* omitted code */
   }
}
-----------------------------------------------------------------------
public class A_Without_Intransigent_Injection {

   @Inject
   private IExampleInterface0 example0;
   @Inject
   private Provider<IExampleInterface1> example1Provider;

   public A() {
       example0.doSomething();
   }

   public void foo() { /* omitted code */ }

   public void bar() {
       IExampleInterface1 example1 = example1Provider.get();
       example1.doSomething();
       /* omitted code */
   }

}
```

crete class; second, this solution introduces less flexibility on testing, once a mock object would need to be an inherited class of the given concrete class in order to modify desired behavior; finally, according to Gamma et al. [12], coupling to a concrete class can increase maintenance efforts.

Gamma et al. [12] advocates for programming to an interface, which is a natural solution to this anti-pattern. Figure 4.2 presents the structure of occurrence together with a solution, separated by a dashed line. The example solution concerns following an interface oriented design when it comes to request a dependence. Further, the resolution example depicts a code transformation, on which an interface (see *IExampleInterface*) is created so that the class *ConcreteExample* implements it. Then, rather than relying on a concrete class injection (which configures a high coupling to class *ConcreteExample*), the class *B_Without_Concrete* now follows dependency inversion principle, once it depends on an interface (*IExampleInterface*).

4.3.3 Long producer method

Long producer method concerns a method that performs activities that are out of the scope of providing a dependency, which must be its main objective. This context defines this anti-pattern as a design problem. A negative consequence entailed is undermining the ability of the software to adapt to change when requirements change. Figure 4.3 depicts an example of long producer method occurrence along with an example of solution, separated by a dashed line.

The example problem shows a high complex method that should be simple, once the main concern of a Producer method (see *@Produces* annotation) is to a provide a given dependency. The DI container, when it identifies the existence of a Producer method for a given type, transfer the responsibility for dependence provision to the Provider method.

On the other hand, in solution part, in case where business logic is necessary in order to obtain a dependence, rather than relying on a Producer method, a business method is desirable. In other words, the *@Provision* annotation is removed, so the dependence provision process of the class holding the old Provision method is shortened. Also, refactoring the method in order to decrease cyclomatic complexity is another important step. In addition, a suitable code transformation is employing aspect-oriented programming in order to trigger important tasks based on the life-cycle of the Provider method. For instance, an example code transformation is defining a pointcut on the Provider method, so the Provider method is intercepted and the logic is

Figure 4.2: Concrete class injection

```
public class B {

   @Inject
   private ConcreteClassExample example;

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


executed prior or after the dependence provision. Particularly, we aimed to provide an excerpt of a *Producer* method without high cyclomatic complexity and fewer responsibilities.

4.3.4 God DI class

As mentioned in Chapter 2, a God class often embrace a multiple set of responsibilities that would be better handled if distributed properly. A signal of a God DI Class may be in the form of a class with poor modularity, possibly indicating a deeper problem.

This pattern in source code can also occurs in the context of dependencies provided by a DI container. Therefore, this anti-pattern concerns the injection of a substantial number of dependencies in a class. This anti-pattern is

Figure 4.3: Long producer method

```
public class C {
    // omitted code
    @Produces
    public ProducedBean generateReport(){
        Set<Integer> selectedBacklogIds = this.getSelectedBacklogs();
        if(selectedBacklogIds == null) {
            Collection<Product> products = new ArrayList<Product>();
            productBusiness.storeAllTimeSheets(products);
            for (Product product: products) {
                selectedBacklogIds.add(product.getId());
            }
            return Action.PROCESS;
        }
        // omitted code
        Workbook wb = this.timesheetExportBusiness.
            generateTimesheet(this, selectedBacklogIds, startDate,
                endDate, timeZone, userIds);
        this.exportableReport = new ByteArrayOutputStream();
        try {
            wb.write(this.exportableReport);
        } catch (IOException e) {
            return Action.ERROR;
        }
        return Action.SUCCESS;
    }
}
-----------------------------------------------------
public class C_Without_Long_Producer {
    // omitted code
    @Produces
    public ProducedBean generateReport(){
        if(selectedBacklogIds == null) {
            processSelectedBacklogs();
            return Action.PROCESS;
        }
        if (selectedBacklogIds.contains(0)) {
            processSelectedBacklogIds();
        }
        writeToLog();
        return Action.SUCCESS;
    }
}
```

primarily concerned over injected instances that are often inconsequentially introduced by developers without reasoning over the increased dependence of the class with other components.

Negative consequences entailed by this anti-pattern concerns a possible increased effort on maintenance tasks in the class. As derived from a design smell, it is configured as a design problem.

Figure 4.4 depicts the pattern of occurrence of a God DI class. The example depicts an excerpt of a class with high level of complexity, in terms of number of injected element instances, and number of methods.

Also, below the dashed line, Figure 4.4 exhibits a suggestion of a refactoring that removes the anti-pattern, dividing dependencies and behavior into different classes. The resolution example depicts a code transformation applied to previous class D, on which a refactoring type called "Extract Class" [4] was employed three times in order to reduce the complexity of class D.

4.3.5 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 4.5 presents the structure of occurrence together with an example solution, separated by a dashed line. The example shows a class (E) with an injected instance that is not used though any method of the class. Next, the solution concerns removing the non used injection element.

4.3.6 Static dependence provider

Static dependence providers are related to Fabrics and Service Locators. The first refers to a class that has the objective to provide a given concrete implementation, not being a *Provider* class. On the other side, Service Locator pattern also applies to this context, since it is a class that has the responsibility for serving all dependencies that might be required on run time.

Negative consequences entailed by this anti-pattern are high dependence on fabric in source code, configuring a high coupling to a fabric class. In case of Service Locator, the dependency on this pattern is even worse due to its widespread usage in the project. Indeed, inversion of control is not achieved

Figure 4.4: God DI class

```
public class D {
   @Inject private IExample1 one;
   @Inject private IExample2 two;
   @Inject private IExample3 three;
   @Inject private IExample4 four;
   @Inject private IExample5 five;
   // other several dependencies injected
   @Inject private IExampleN n;

   void methodOne() { /* reference to several dependencies */ }
   void methodTwo() { /* reference to several dependencies */ }
   // other several methods
   void methodThree() { /* reference to several dependencies */ }
}
```

```
public class D_Part_1 {
   @Inject private IExample1 one;
   @Inject private IExample2 two;
   @Inject private IExample3 three;

   void methodOne() { /* code omitted */ }
}
public class D_Part_2 {
   @Inject private D_Part_1 dPartOne;
   @Inject private IExample4 four;
   @Inject private IExample5 five;
   @Inject private IExample6 six;

   void methodTwo() { /* code omitted */ }
}
public class D_Part_3 {
   @Inject private D_Part_2 dPartTwo;
   @Inject private IExample7 seven;
   @Inject private IExample8 eight;
   @Inject private IExample9 nine;
   @Inject private IExampleN n;

   void methodThree() { /* code omitted */ }
}
```

Figure 4.5: 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
   }
}
```

in both cases. Both classes of problem concerns architectural problems, since both violate DIP and IoC principle.

Figure 4.6 depicts an example of Service Locator occurrence and an example resolution, separated by a dashed line. The occurrence example exhibits the class (E) with a dependence provision made by a service locator. In other words, rather than relying on the DI container for injecting an instance of *IDataSource* type on *dataSource* attribute, the code relies on a service locator.

On the other side, the example resolution on Figure 4.6 enforces the use of DI container for dependency injection at run time by relying on a *Producer* method in order to provide an instance of *IDataSource*. Particularly, the resolution example above shows a code transformation, in which the logic for creating an instance of *IDataSource* is modularized within a Producer method. This way, the class *E_Without_Service_Locator* is not coupled to a service locator class anymore.

4.3.7 Direct container call

Direct container calls can provide a concrete implementation at any point of the system. 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,

Figure 4.6: Static dependence provider

```
public class E {

    @Inject
    private Parser parser;

    public void execute(List<String> files) throws Exception {
      IDataSourcedataSource dataSource = (IDataSource)
          ServiceLocator.getInstance()
          .getBeanInstance("IDataSource");

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

   @Bean
   public IDataSource provideDataSource(){
      // logic for creating an instance of IDataSource
   }
}

public class E_Without_Service_Locator {
   @Inject
   private Parser parser;
   @Inject
   private IDataSource dataSource;

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

employing container call for dependence resolution conveys an architectural violation. A suggested solution relies on applying DI to occurrences of container calls aimed at providing a dependence.

The Figure 4.7 shows an example of container call in the Spring framework and a suggested solution, separated by a dotted line. The occurrence example shows the class (E) with a dependence provision made by a direct container call. In other words, rather than relying on the DI container for injecting an instance of *IDataSource* type on *dataSource* attribute, the code relies on a direct container call. Next, the suggested resolution concerns removing the element that performs a container call and enforcing the use of a DI container for dependence provision.

4.3.8 Open window injection

This anti-pattern is applied when an injected instance is not used, but passed as parameter to another class method or opened for external accessing (e.g. by get method or public/protected access modifier). Two negative consequences are observed. In the first case, it adds a useless intermediary element between the class that needs a given concrete implementation and the DI container. On the second case, it opens a door for external modification, which could possibly yield the introduction of bugs. A suggested resolution concerns the following actions: (a) on the given class, remove the method that provides the injected dependence to external classes; (b) In addition, remove the injected dependence from the parameter list of the external method; (c) For last, in the external class, add the dependency injection request as parameter list earlier.

Figure 4.8 show an example of open window injection occurrence, on which the *parser* object is passed as parameter to another method. Following to that, the resolution example depicts a code transformation where the injected element *parser* is not passed as parameter to method *doSomething* of the interface *IExampleInterface* anymore. The concrete implementation of *IExampleInterface* is now responsible for defining its dependence on an instance of *Parser* type.

4.3.9 Framework coupling

It refers to elements on source code that are dependent on a given framework implementation. As the name of the anti-pattern expose, it can be represented as annotations or method calls to framework configuration classes

Figure 4.7: 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
    private Parser parser;

    @Inject
    private IDataSource dataSource;

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

Figure 4.8: Open window injection

```
public class F {

    @Inject
    private Parser parser;
    @Inject
    private IExampleInterface one;

    public Parser getParser() {
        return parser;
    }
    public void execute(List<String> files) throws Exception {
         for(String file : files){
                Object parsedObject = parser.parse(file);
                one.doSomethingWithParsed(
                    parser, parsedObject);
         }
    }
}
-----------------------------------------------------------------------
public class F_Without_Passing {

   @Inject
   private Parser parser;
   @Inject
   private IExampleInterface one;

    public void execute(List<String> files) throws Exception {
      for(String file : files){
            Object parsedObject = parser.parse(file);
            one.doSomethingWithParsed(parsedObject);
      }
    }
}

public class ConcreteExample
    implements IExampleInterface {

   @Inject
   private Parser parser;

   @Override
   public void doSomethingWithParsed(Object parsedObject) {
      // omitted code
   }
}
```

Figure 4.9: Framework coupling

```
public class J {

    @Autowired
    private Parser parser;

    @Autowired
    private 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
    private Parser parser;

    @Inject
    private IDataSource dataSource;

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

along the source code. 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 4.9 depicts a class that employs Spring framework *@Autowired* annotation and, below the dashed line, the same class, now employing JSR-330 *@Inject* annotation.

Figure 4.10: Open door injection

```
public class H {
    @Inject
    private Parser parser;

   public void setParser(Object parser) {
      this.parser = parser;
   }

    // code omitted for brevity

}
-----------------------------------------------------------------
public class H_Without_Anti_Pattern {
    @Inject
    private Parser parser;

    // code omitted for brevity
}
```


4.3.10 Open door injection

This anti-pattern is applied when an inject request is provided by a DI container, however, the instance requested is open for modification by an external element. It usually happens when the developer lacks sufficient knowledge about DI. Open door injection can configure a hard to follow traceability, hindering program comprehension. Also, bugs are another possibility, since concrete implementation is open to chance by an external class.

Figure 4.10 depicts the presence of a public set method that allows changing of the injected instance of *parser* in runtime. In details, the example depicts a public set method ("setParser"), which allows for modification of the instance of an injected element ("parser") by an external class. The resolution shown below the dashed line is the removal of the element on source code (e.g. public set method) that enables changing injected element.

4.3.11 Multiple assigned injection

This anti-pattern occurs when the reference to an injected instance is spread among multiple attributes. This anti-pattern is correlated to *Open door injection*, since it opens a gap for an undesirable modification of the injected object at run time. Figure 4.11 provides an example of occurrence of such anti-pattern. The example depicts the assignment of an injected instance of

ExampleDAO to an attribute of a parent class ("GenericBusinessImpl").

In the case of injection instance being assigned to an attribute of superclass, a better approach would be overriding an abstract method. This way, the overridden abstract method would provide the instance injected, not incurring on reference duplication. Thus, the resolution example shown below the dashed line on Figure 4.11 depicts a code transformation that removes the assignment of an injected instance to an attribute presented in a parent class. The removal makes room for an abstract method in the parent class, which still allow the reference to the original injected instance.

4.3.12 Multiple forms of injection

This anti-pattern refers to the use of multiple forms of injection to a given element. It leads to misunderstanding of injection process for less experienced developers. Figure 4.12 provides an excerpt of the occurrence of this anti-pattern, where there are two forms of injection for the same element ("exampleDAO"). The first is an attribute injection. The second is a constructor injection. Then, the example resolution depicts only one form of injection (constructor) for the element *exampleDAO*.

4.4 Concluding Remarks

Documented DI anti-patterns do not directly consider the design principles behind DI, namely, IoC and DIP, and the existence of design principles that guide good object-oriented design, such as GRASP and SOLID. In addition, even with such importance in industrial settings, existing catalogs do not focus on Java platform.

Considering this scenario, we address our RQ1 by applying two methodological approaches to derive an initial catalog of Java DI anti-patterns. First, based on observations of bad characteristics in source code, i.e., characteristics of implementation in source code that violates design principles, such as IoC and DIP. Second, as a deductive approach, we have listed a set of anti-patterns such that instances could appear in software systems that adopt DI as a mechanism to decrease coupling.

An initial set of 12 anti-patterns related to the employment of DI in software systems are derived from such process. Each one of them provides a name, a description, negative consequences and a suggestion of resolution in order to guide practitioners' in their day to day development activities.

Figure 4.11: Multiple assigned injection

```
class ExampleBusiness
           extends GenericBusinessImpl{

    private IDAOexampleDAO exampleDAO;

    @Inject
    public void setExampleDAO(ExampleDAO exampleDAO) {
        this.genericDAO = exampleDAO;
        this.exampleDAO = exampleDAO;
    }

}
-----------------------------------------------------------------------
abstract class GenericBusinessImpl {

    abstract IDAO getGenericDAO();

}

class ExampleBusiness
           extends GenericBusinessImpl{

    private IDAOexampleDAO exampleDAO;

    @Inject
    public void setExampleDAO(ExampleDAO exampleDAO) {
        this.exampleDAO = exampleDAO;
    }

    @Override
    protected IDAO getGenericDAO() {
        return this.exampleDAO;
    }

}
```

Figure 4.12: Multiple forms of injection

```
class ExampleBusiness
          extends GenericBusinessImpl {

    @Inject
    private IDAOexampleDAO exampleDAO;

    @Inject
    public void setExampleDAO(ExampleDAO exampleDAO) {
        this.exampleDAO = exampleDAO;
    }
}
-----------------------------------------------------------------------
class ExampleBusiness
          extends GenericBusinessImpl{

    private IDAOexampleDAO exampleDAO;

    @Inject
    public void setExampleDAO(ExampleDAO exampleDAO) {
        this.exampleDAO = exampleDAO;
    }

}
```

5
Assessing Practical Occurrence of the Proposed Catalog

5.1
Introduction

Although existing studies have suggested DI anti-patterns, none of them have explored the practical occurrence of the anti-pattern instances. In other words, it is unknown if such instances occur within software systems. Without this knowledge, it is not possible to effectively assess the relevance of anti-patterns in practice. Besides, under the light of the principles behind DI, namely, IoC and DIP, we found that existing studies fail to present anti-patterns targeted at the DI principles. For instance, most suggested anti-patterns may also be applied in a context where a DI framework and DI principles are not employed, such as *Control Freak* [32].

Thus, in Section 4, we conjectured a set of DI anti-patterns aimed at the Java platform. Even though the proposed DI anti-patterns have focused on properly addressing violations of the principles behind DI and patterns present in GRASP, they might not matter in practice, incurring the same problem of existing documentation. Hence, it is important to understand if the proposed DI anti-patterns represent problems that are introduced by developers in practice. Otherwise, the catalog would be considered useless. To address RQ2, we selected a set of open and closed-source software projects that adopt a DI framework. Next, we developed a tool to statically analyze the occurrence of the DI anti-patterns within their source code. By answering this question, we will be able to reveal the degree of occurrence of the candidate catalog of DI anti-patterns in software projects, thus, opening a window for further efforts towards validation.

Thus, Section 5.2 introduces our initial efforts towards validating the proposed catalog of DI anti-patterns, describing the steps taken to develop a static analysis tool to automatically detect instances of DI anti-patterns. In addition, Section 5.2 describes the results of the detection of DI anti-patterns in several software systems. Section 5.5 provides the threats of validity incurred by this method. Lastly, we explain the results from this study in Section 5.4.

5.2 Developing an automatic detection tool

At the time we were investigating feasible approaches to enable a fast process of identification of the proposed DI anti-pattern instances on source code, we did not find any tool that would easily allow us expressing the rules that flag elements of code as positive or negative regarding being an instance of anti-pattern.

Although no tool was able to automatize all steps per se, (namely, querying a repository of software project through a query language, cloning the repositories queried, submitting rules to identify anti-patterns in a project's source code, outputting the results), we found that *Repodriller* [30], a framework for mining software repositories, would enable the identification of annotations in source code. Thus, we started with a manual analysis aimed at providing initial evidence that the candidate DI anti-patterns have instances on real open source projects. *Repodriller* [30] was used to filter the occurrence of DI injection point annotations (such as *@Inject* and *@Autowired*) and DI container references (e.g. direct container calls) in the preliminary selected projects. After filtering, classes and its associations were manually analyzed in order to verify if DI elements on source code incurred in an anti-pattern instance.

Although we were able to identify some instances of anti-patterns in the source code of randomly selected projects, we realized that the manual procedure was error-prone and not cost-effective in terms of time. Thus, in order to support the automatic detection of each proposed DI anti-pattern in source code, a software tool called DIAnalyzer was developed. The source code of DIAnalyzer is available on GitHub [1].

5.2.1 Designing DI anti-patterns detection tool

Based on the proposed catalog of DI anti-patterns presented in Section 4, this section aims at describing the development of a software system called DIAnalyzer that automatically identifies every anti-pattern proposed. The tool is a static code analyzer implemented using the JavaParser [33] library, which relies on an Abstract Syntax Trees (AST) in order to flag elements of code that represent DI anti-patterns candidates. The requirements of the tool are explained hereafter.

Functional Requirements

[1] https://github.com/rnlaigner/dianalyzer

Table 5.1: Rules for anti-patterns detection

Identifier	Rule
AP1	AP1 is applied when an injected attribute is not referenced in all methods of a class
AP2	AP2 is applied when an attribute that receives an injection is a concrete implementation
AP3	AP3 is applied when the sum of the cyclomatic complexity of the Producer method is greater than 8
AP4	AP4 is applied when the sum of the cyclomatic complexity of all methods of the class being inspected is greater than 46 and the number of attributes injected in class being inspected is greater or equal 5
AP5	AP5 is applied in an attribute if this attribute is an injected attribute, however, the same is not used in the class
AP6	A heuristic was used to detect these instances, as follows: An attribute instance is obtained by calling a dependence on which its name or class name contains fabric or factory
AP7	AP7 is applied when an instance of *ApplicationContext* class calls the method *getBean*
AP8	AP8 is applied when an injected instance is passed as parameter to another class method or opened for external accessing by a method
AP9	AP9 is applied when the annotation *@Autowired* is employed in order to inject dependence instances
AP10	AP10 is applied when an injected instance is allowed to be changed on a public set method
AP11	AP11 is applied when more than one class attribute receives exact same injected instance
AP12	AP12 is applied when a class attribute is registered to receive an injected instance by more than one form of injection (e.g. constructor and attribute)

- **FR1.** The tool must identify instances of anti-patterns given a set of rules (the rules are shown in Table 5.1)
- **FR2.** The user can provide the project to be analyzed
- **FR3.** The user can provide the output path of the results provided by the tool
- **FR4.** The tool must provide as output an spreadsheet with the following information: For each DI anti-pattern detected, the class, element, and DI anti-pattern must be shown in each line
- **FR5.** The user must be able to download open source projects from GitHub given an query provided as input

Non Functional Requirements

- **NF1.** An analysis process cannot take more than 10 minutes
- **NF2.** The system must be able to execute on Windows, MacOS and Linux distributions - Portability
- **NF3.** The user must be able to start an analysis in less than 1 minute - Usability

Figure 5.1 shows a schematic overview of DIAnalyzer. The design of the project followed an orientation to abstraction, as suggested by Martin [21]. Thus, the system architecture is decomposed in a set of subsystems. The subsystems are: repository extractor, data model extractor, analysis, and report. The description of each subsystem is provided as follows.

The Repository Extractor subsystem is responsible for submitting a request (in form of a query) to GitHub API in order to clone a set of open source projects into user's file system.

As mentioned earlier, to support the identification of the DI anti-patterns, the *JavaParser* framework was employed. The Data Model Extractor subsystem converts each file of the project under analysis into a model that can be manipulated by the system. The Data Model Extractor Subsystem is built as a layer above the *JavaParser* framework, abstracting its internals in order to ease reuse and diminish coupling to *JavaParser* from other subsystems of the architecture. *JavaParser* relies on constructing an AST for a given compilation unit (i.e. Java class) and represents object oriented elements, such as methods, attributes, and classes in form of a vertex in a tree.

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 anti-pattern are found on Table 5.1. It is worth of mention that the detection strategies applied to AP3 and AP4 were based on Lanza and Marinescu [16].

The Analysis subsystem is the core part of the architecture, since it contains the logic that verifies if an anti-pattern is applied in the context of a class. Basically, each anti-pattern is modeled as a class in the Analysis subsystem. An anti-pattern class is composed by a set of rules. Each rule is also modeled as a class, being responsible for identification of injected elements

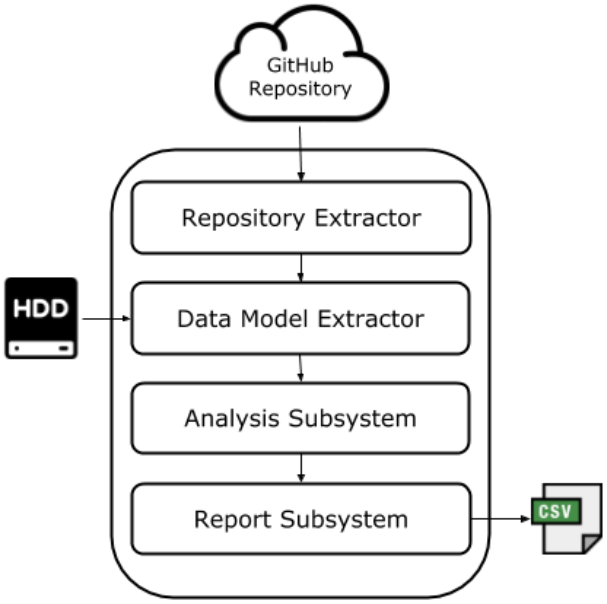


Figure 5.1: Schematic overview of DIAnalyzer

that obey the characteristics of given rule. Some rules also have as dependence a data source, which is implemented as a class that provide additional source code information on run time.

Lastly, the Report subsystem is responsible for handling requests related to convert the results of the Analysis Subsystem into a report.

5.2.2 Evaluating DI anti-patterns detection tool

Although building a static analysis tool would allow us to efficiently mine software repositories, threats of validity could be risen against the effectiveness of the tool. Works that present propositions in form of catalogs (such as Chen and Jiang [6]) usually rely on an oracle data-set in order to conduct an evaluation of tools that are built with the objective to flag instances present in the catalog.

However, as there is no available oracle data-set 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 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, our tool 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 5.2 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.

Table 5.2: Precision results of DIAnalyzer

DI	Project	
Anti-Pattern	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)

5.3 Detecting DI anti-patterns

With a satisfactory result in the precision and recall evaluation carried out in DIAnalyzer, we are confident that DIAnalyzer can effectively flag instances of DI anti-patterns from source code. Then, we have divided the process of detecting instances of candidate anti-patterns in software projects in two steps: open-source and closed-source applications.

5.3.1 Open-source software systems

Mining open-source software repositories constitute a common research practice in the software engineering field. For instance, studies on code smells [23] and refactoring [4] often rely on source code repositories to support their analysis. In line with this method, we found worthy to start the analysis of DI anti-pattern instances with open-source repositories.

Since we have a tool that automatically identifies anti-pattern instances on source code, no manual analysis is required. 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 5.3, presenting the (i) name, (ii) Java lines of code, and (iii) number of commits for each project.

Table 5.3: 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

We applied DIAnalyzer on the latest releases of the four selected projects. The detection results are depicted in Table 5.4. 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.* Chapter 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 5.4: Occurrence of DI Anti-Patterns in open-source projects

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

5.3.2 Closed-source software systems

An important step in the software engineering field is making sure propositions reflect on the practice of software engineering. Without such validation, it is unknown whether the candidate anti-patterns are relevant in industrial settings. However, researchers often rely on mining open-source repositories due to the complexities involved in obtaining closed-source software repositories. Legal issues concerning strategic processes a software supports are an example of such impedance.

At this point, we were already aware that the proposed instances of DI anti-patterns occur within popular open-source software systems. However, we would like to investigate whether the proposed DI anti-patterns also occur in industrial settings. Besides, in the case of verified occurrences, we would like to comprehend if the same trends found in open-source repositories are also found in closed-source repositories.

Thus, intending to obtain the source code of industrial software systems, we identified two candidate companies that the author has had previous working experience in software development and maintenance. We designed a

consent term safeguarding the companies against any misuses of the source code provided. We sent them the term along May and June, 2019. The companies responded positively and expressed their willingness to support the research being conducted. Both firms provided one software for analysis. It is important to mention that the projects obtained from the firms were not developed or maintained by the author of this work.

In total, 2 projects from two different industry partners were obtained. Table 5.5 shows the characteristics of both projects. In this work, we cannot expose details about the software due to non-disclosure agreement restrictions, so we cannot give full details about the closed-source projects under analysis. In short, both software projects are web-based systems written in Java and present approximately 30K LOCs.

The first (CS1) adopts Spring [29], a framework that already provides DI capabilities. CS1 is a shorter (in LOC) Java project compared to open-source projects we previously analyzed. By analyzing the project, we observed and confirmed with the firm that the project was mainly developed and maintained by a senior developer, who was already experienced in development with the Java platform.

The second closed-source project (CS2) employs Guice framework to support DI capabilities. Again, CS2 is shorter in LOC compared to open-source projects. As confirmed with the firm, CS2 was also mainly developed by a senior developer.

Due to the use of Guice framework on CS2, we have adapted DIAnalyzer to support annotations present in Guice (e.g. *@Produces*) and Guice direct container calls. In addition, we have fixed the bug reported in Section 5.2.2 about the malformed output of the tool in AP7. Then, we have applied DI Analyzer to automatically detect instances of candidate anti-patterns in the closed-source projects obtained. The results are shown in Table 5.6.

Table 5.5: Selected projects

Index	**LOC**	**Framework**
CS1	29.405	Spring
CS2	32.204	Guice

Table 5.6: Occurrence of DI Anti-Patterns in closed-source projects

DI Anti-Pattern	Project CS1	Project CS2
AP1	68	265
AP2	2	12
AP3	0	0
AP4	1	4
AP5	15	86
AP6	1	3
AP7	0	0
AP8	64	35
AP9	68	0
AP10	3	0
AP11	0	0
AP12	0	0

Overall, the patterns observed in closed-source repositories are closely related to the findings of the previously analyzed open-source projects. For instance, AP1, AP2, AP4, AP5, and AP8 have instances in both projects. Regarding AP3, no instances were verified again. In line with the open-source projects, both closed-source projects present anti-pattern instances for each DI anti-pattern category (*cf.* Chapter 4).

Anti-patterns AP1, AP2, AP4, AP5, and AP8 have instances in both closed-source projects. Hence, although expert developers tend to follow an interface-oriented design and avoid classes that centralize the intelligence of the system, they may introduce some anti-pattern instances in source code. This can be explained by fast prototyping sprints in the software life cycle and lack of attention (in case of AP5). Instances of AP9 solely found in CS1 is explained by a wide adoption of the Spring annotation *@Autowired*. Since CS2 employs Guice, a framework that follows JSR-330 convention of *@Inject* annotation, no instances of AP9 are verified in CS2.

Besides, it is observed that, in opposition to the findings in open-source repositories, AP7, AP11, and AP12 have no instances in CS1 and CS2. In addition, AP10 are verified only three times (CS1). We believe these results

are related to three main factors: (i) the LOC of the closed-source repositories, which are smaller compared to the analyzed open-source projects, (ii) the number of developers involved in development activities, which is also smaller compared to the open-source projects analyzed, and (iii) the expertise of the developers involved in the development of the closed-source projects.

In other words, we suggest that the expert developers involved in development activities of the closed-source projects analyzed are aware of the risks entailed to the software architecture by: the introduction of direct container calls (AP7); opening injected fields to external modification (AP10); the assignment of instances provided by the DI container to several fields (AP11); the introduction of multiple forms of injection for a given element (AP12).

5.4 Results

The characteristics of the open-source and closed-source projects analyzed differ profoundly. For instance, the number of lines of code and the number of developers involved in the development process vary greatly. In open-source projects, the smallest project (P1) has 58.171 LOCs and a high number of developers are involved, as investigated in the commit history. In opposite, in the closed-source projects, we were not able to gather and analyze projects with similar characteristics to the open-source ones. Both closed-source projects analyzed are small (in terms of LOC) in comparison with the open-source projects and were mainly developed and maintained by one expert developer each, which we believe is the reason why some anti-patterns occurrences were not observed.

Some DI anti-patterns are prominent in both open and closed-source, such as AP1, AP2, AP4, AP5, AP6, AP8, and AP10. We believe this pattern occurs due to fast development sprints (AP4, AP6), lack of knowledge about principles behind DI (AP2, AP6, AP7, AP10), and misuse of DI framework and lack of attention (AP1, AP5). In addition, AP7, AP10, AP11, and AP12 are mostly observed in open-source projects. We believe that lack of proper knowledge of DI hinders avoiding these instances in source code. As expected, since CS2 employs Guice, AP9 did not appear in its source code. Guice framework relies solely on JSR-330 specification to define injection points and does not have annotations different from the definition as Spring does. Lastly, in closed-source projects, AP3 again does not have any instance as found in open-source projects. Also, AP7, AP11, and AP12 does not occur in closed-source projects. As mentioned earlier, we believe this trend is influenced by three factors: (i) smaller size and (ii) smaller number of developers involved,

and (iii) expertise of the main developers involved.

By reviewing the historical commits of open-source projects, we verified that anti-patterns progressively scatter around source code during the development process. One of the possible reasons is that novice developers tend to base their code on previously committed code. Another characteristic found in projects that have a significant number of anti-patterns instances is the number of developers that actively contributed to the code base. In other words, projects with multiple developers tend to show more anti-patterns instances compared to those with less contributing developers.

In the closed-source analysis, we observed that the presence of a expert developers being mainly responsible for the project code base highly affected the number of anti-pattern occurrences.

Although we cannot suggest that expert developers tend to avoid bad DI implementation practices, once the open-source projects analyzed may also present commits of expert developers, we are confident that DIAnalyzer can support developers in the process of identifying bad implementation practices related to DI in software projects.

5.5 Threats to Validity

Internal Validity. The tool built to flag instances of anti-patterns may miss some instances 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 anti-patterns in source code of the analyzed software projects. In addition, we double-checked the findings with the support of an independent researcher.

External Validity. Albeit selecting projects with different number of LOC and commits, most of them are implemented using the Spring framework. However, the risk of leaning the findings towards a specific framework is mitigated because one of the closed-source projects analyzed employs Guice as DI framework. In addition, our findings were verified in open-source and closed-source systems from industrial settings, which strengthen the practical relevance of the catalog of DI anti-patterns.

Construct Validity and Reliability. Since there is no benchmarking dataset for DI anti-patterns, we built an oracle data-set by ourselves in order to evaluate DIAnalyzer. The data-set was built and verified by two independent researchers. The process for building the oracle is similar to Chen and Jiang [6] work. Lastly, the oracle does not contain instances of AP3. We believe this does

not undermine our findings, since this specific anti-pattern did not appeared in any analyzed project.

5.6 Concluding Remarks

In this chapter we have provided a comprehensive evaluation of the practical occurrence of our candidate catalog of Java DI anti-patterns.

We have started by stating the difficulties found in the process of manually detecting instances of DI anti-patterns in source code. Then, we proceed to the description of the development of a static analysis tool to support the process of identification of anti-patterns in source code. We described the specification and the architecture of the solution. Next, we explained the process of validating the tool in regard to recall and precision. The tool showed an average precision of 97.78% in Agilefant and 89.80% in Libreplan. In addition, a relative recall analysis demonstrated that 92.19% of the instances were identified by the tool.

With the confidence brought by the developed static analysis tool, we divided the process of identifying instances of anti-patterns in two parts: open-source and closed-source projects. The detection results suggests that the anti-patterns occur frequently in software systems.

6
Investigating Perceived Usefulness of Proposed Catalog

6.1
Introduction

In the last Section (cf. 5), we verified that the conjectured anti-patterns occur within software systems. Although instances of our candidate anti-patterns are retrieved from both closed and open-source software systems, an important step towards strengthening the validation of the proposed catalog of anti-patterns is gathering the perception of experienced practitioners. In other words, we aim to understand if industry practitioners consider the catalog useful and are willing to apply our catalog in their working environment.

Thus, in this chapter, we document our efforts to investigate the acceptance of our catalog among industry practitioners by the application of the Technology Acceptance Model (TAM) [10] in its three dimensions: ease of use, usefulness, and intention of use. Hence, besides investigating the occurrence of each DI anti-pattern in software projects as explored in the previous chapter, we have designed and conducted an interview-administered survey and an online survey to assess the acceptance and perception of usefulness from expert developers regarding the proposed catalog.

Obtaining the perception of experienced developers over the candidate catalog proposed is an important validation step prior to sharing it with the software engineering community. Using the GQM (Goal Question Metric) definition template described by Wohlin et al. [37], 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.

As TAM is employed to assess a given technology (in our case, the candidate catalog), we want to assess the willingness of the expert developers to adopt the catalog as a tool in their development activities. Also, we aim to gather a preliminary assessment of difficulties found by developers on understanding our catalog. The rate of answers a certain facet might indicate opportunities for improving the catalog.

To achieve the goals, this study designed two classes of surveys to gather the opinion of developers over the candidate catalog. As mentioned at the beginning of this chapter, we first designed an interview-based questionnaire to allow an in-depth analysis of the instrumentation and each candidate anti-pattern proposed. Then, with the lessons learned interview-based and corrections leveraged by opinions of expert developers, an online survey was designed to obtain a wider range of views regarding the proposed catalog. Lastly, the results, lessons learned, and threats of validity are explained.

6.2 Interview-Based Survey

This section presents the details of an interview-based survey conducted in order to obtain preliminary results about the usefulness of our proposed catalog.

6.2.1 Design

Towards achieving our goals, we first designed a descriptive survey. According to Linaker et al. [19], a descriptive survey supports claiming or assertions about a particular subject. Thus, as we are claiming that our candidate catalog provides a comprehensive set of DI anti-patterns, a descriptive survey meets our goal. Regarding the target population, for this preliminary study, we followed the recommendation to select developers that are most appropriate for our goal in order to provide accurate answers, rather than expecting that a random target population would allow an effective analysis of our subject [34]. Indeed, we targeted at a population of practitioners with large expertise on applying design principles, frameworks, and dependency injection in software systems.

An interviewer-administrated questionnaire was designed in order to avoid threats of validity, such as doubts that could arise during the process, then leading to a wrong answer by the respondent. Linaker et al. [19] assert that employing thus questionnaire type enables clarifying ambiguous questions. Thus, we aimed to further support the interviewees in comprehending the context, purpose, and consequences of each candidate anti-pattern proposed during the interview.

The questionnaire is divided in three parts, as explained as follows. The first part concerned gathering information about the respondents' academic background and industrial experience. For instance, questions included years of experience developing software and current position in industry. Regarding

technical skills, we inquiry about object-oriented analysis and design, design principles and patterns, anti-patterns, source code inspection, dependency injection, and Java programming language. In addition, we collected information about English reading and comprehension skills.

The second part of the questionnaire consisted of questions regarding the candidate catalog. For each DI anti-pattern proposed, following the pattern structure mentioned in chapter 4, the information provided concerns: (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 the information provided, the interviewees were inquired to answer the following question: "Can the proposed DI anti-pattern actually be characterized as an anti-pattern?" The question is responded based on a five-point Likert scale (1- Agree, 2- Partially Agree, 3- Neutral, 4- Partially Disagree, and 5-Disagree). For this specific design survey, the interviewees were also invited to include comments over the general structure of the analyzed DI anti-pattern and possible disagreements with the anti-pattern or even about the resolution example provided.

The final part of the questionnaire concerned the application of the Technology Acceptance Model (TAM) [10]. According to Turner et al. [35], TAM is a suitable tool to capture the user's acceptance of a given technology. The technology, in our case, is the candidate catalog and the anti-patterns proposed within it. TAM questions aim 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[1].

6.2.2 Execution

The execution of the survey followed the strategy of identifying a sample of the population according to the survey design. Thus, we prioritized experts in software development, those with extensive experience in industry on developing and maintaining software systems, particularly with source code inspection, design patterns, and software design and architecture skills.

Then, we identified three interviewees from three different organizational units of two different industrial partners. The identified interviewees are further described in Table 6.1. It is observed that the interviewees have long-lasting experience in industry and a strong background in software development, being

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

Table 6.1: Background of respondents

Information	Respondent		
	I1	**I2**	**I3**
Academic background	Master	Bachelor	PhD
English reading and comprehension skills	Advanced	Advanced	Advanced
Experience developing software	> 10 years	> 10 years	> 10 years
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

a reliable source when it comes to evaluating the candidate catalog. It is noteworthy to mention that we intentionally opted for selecting a small sample of experts to conduct in-depth interviews, allowing qualitative discussions about our initial DI anti-patterns catalog.

Table 6.1 presents background information of the interviewees, in which it is possible to observe that they indeed have a strong background in object-oriented analysis and design, being a suitable source when it comes to evaluating the proposed catalog. Although I3 does not possess a strong experience in DI, I3 was able to assess the proposed anti-patterns due to having strong software design skills. Due to the format of the questions and the questionnaire type, the survey was provided through a printed document. The interviewees were informed that there was no limit of time. The interviews took place in April 2019, and lasted 80, 85, and 100 minutes, respectively for interviewees I1, I2, and I3.

6.2.3 Results

The results of the survey are presented in Tables 6.2 and 6.4.

The results of the perception on the proposed DI anti-patterns are shown in Table 6.2. 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 catalog.

The adapted TAM questions and their results are shown in Tables Table 6.3 and 6.4. Due to space restrictions, the index of each question in depicted in Table 6.3. 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 catalog is helpful and that developers would show willingness to apply it.

Table 6.2: Perception over the DI anti-patterns in interview-based survey

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

6.3 Online Survey

With the lessons learned and the preliminary evidence collected from the application of an interview-based survey described in Section 6.2, we decided to pursue further evidence on the usefulness of the proposed catalog by collecting the opinions of a wider range of developers. Although the interview-based survey provided an in-depth assessment of every aspect of the catalog, the process is not time-effective, taking on average two hours of interview to collect the necessary evidence. Thus, to decrease the time spent on collecting evidence about the usefulness, ease of use, and acceptance of the catalog, but also supporting the respondents with a suitable questionnaire-based survey, a Google forms [2] survey was used to implement an online questionnaire.

6.3.1 Design

We basically converted the survey provided in a form of document to respondents to an online version. Again, the first part of the questionnaire is responsible to gather background information of the respondent. As the

[2] www.google.com/forms/about

Table 6.3: TAM questions index

Dimension	Index	Question
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)
	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)
	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)
	T4	I would find the proposed catalog of DI anti-patterns useful in my job
Ease of use	T5	Learning to use the proposed catalog of DI anti-patterns would be easy for me
	T6	I would find it easy to use the proposed catalog of DI anti-patterns to prevent DI related problems in software systems
	T7	It would be easy for me to become aware of the proposed catalog of DI anti-patterns
	T8	I would find the proposed catalog of DI anti-patterns easy to apply
Intention to use	T9	I intend to apply the proposed catalog of DI anti-patterns regularly at work

Table 6.4: Respondents perception over the catalog of DI anti-patterns

Index	Respondent		
	I1	I2	I3
T1	Strongly Agree	Strongly Agree	Strongly Agree
T2	Agree	Strongly Agree	Neutral
T3	Strongly Agree	Strongly Agree	Strongly Agree
T4	Strongly Agree	Strongly Agree	Strongly Agree
T5	Strongly Agree	Agree	Agree
T6	Strongly Agree	Strongly Agree	Neutral
T7	Strongly Agree	Strongly Agree	Agree
T8	Strongly Agree	Strongly Agree	Agree
T9	Strongly Agree	Strongly Agree	Neutral

Table 6.5: Background information required for online survey

Information	Identifier
Academic background	B1
Experience developing software	B2
Current position	B3
Dependency injection employment	B4
Experience with dependency injection	B5

respondent has no support from researchers, i.e., it is self-administered, we have decreased the number of information required by summarizing specific questions in a unique one. For instance, instead of inquiring about object-oriented technical skills, we require information about experience in software development. Also, we inquire about expertise in dependency injection in terms of years and level of experience, particularly in the industry. As we aimed to secure that respondents complete the questionnaire, these changes diminish the burden of responding to such a long questionnaire. The background information for the online surveys conducted ahead are shown in Table 6.5.

For the second part of the questionnaire, we have maintained the same structure. However, differently from the previous survey described in Section 6.2, we have included the following information for each anti-pattern:

- A text describing the example of occurrence of the anti-pattern, including the main elements involved, such as attributes and methods
- A text describing the suggested resolution, including information about the code transformations performed

The descriptions about the examples of source code provided were important to allow a further and faster understanding of the code snippets. In the earlier survey (described in Section 6.2), we have provided these descriptions as requested by the interviewees. Furthermore, as we also aim to understand the applicability of a refactoring process in each anti-pattern, a new question was added, which is *"Would you fix the anti-pattern on source code? Why?"*

The new question added aims to collect the willingness of the developer to perform an anti-pattern fix on source code. We particularly introduced this question in order to investigate the prospects of future work towards patterns for refactoring DI anti-patterns in source code. Lastly, in the final part of the questionnaire, TAM questions were maintained exactly as the first survey.

Again, the final part of the questionnaire concerned the application of the TAM. The complete instrumentation employed in the online survey can be accessed online[3].

6.3.2 Execution

In order to verify the applicability of the new designed survey to be openly available online, we have run a pilot study. The pilot study aimed at comprehending if the form was sufficient to be openly available without guidance and support from researchers conducting the study.

Again, we have relied on the strategy of identifying a sample of the population that would meet our survey design. Extensive experience in industry and software engineering were pre-requisites we addressed. The preliminary online survey was openly available from July, 10th until October, 29th. In this preliminary assessment, a total of 11 respondents were contacted and 6 completed the online questionnaire. We informally contacted the respondents to gather feedback. Most of them described the online survey as "easy to understand and fill", however, filling the survey was described as "time-consuming" due to the extension of the questionnaire. The background of the respondents of the preliminary online survey is shown in Table 6.6. As can be observed, the respondents have strong expertise in software development and applying dependency injection. I3 does not posses a strong experience in DI, however, we consider I3 a valuable respondent due to his extensive experience in industry.

Due to the perception that the preliminary survey was large in extension, in order to increase the likelihood of having complete answers on the survey in an openly available online version, we divided the questionnaire into two parts. Both parts provide distinct anti-patterns and contain the same amount of anti-patterns per category. Table 6.7 shows the distribution of anti-patterns in two distinct online surveys. After the pilot study and these adjustments, we were more confident that the online survey could be made available to any respondent.

The online survey was openly available from October, 20th until November, 15th. A total of 9 respondents have provided their opinion on the catalog. The background of the respondents of the openly available online survey is shown in Table 6.8. It is possible to observe that the respondents of this survey are less experienced. This is important because this difference in expertise may provide us different insights on how developers with shorter experience evaluate the catalog.

[3]https://zenodo.org/record/3610177

Table 6.6: Background of respondents of the preliminary online survey

Information	**Respondent**		
	I1	**I2**	**I3**
B1	Master	Master	Master
B2	16 years	20 years	23 years
B3	Systems Analyst	Professor	Systems Analyst
B4	Several projects in industry	Several projects in industry	A project in industry
B5	12 years	15 years	1 year
	Respondent		
	I4	**I5**	**I6**
B1	PhD	Master	Master
B2	20 years	19 years	10 years
B3	Professor	Project Leader	Team Leader
B4	Several projects in industry	Several projects in industry	Several projects in industry
B5	9 years	7 years	5 years

Table 6.7: DI Anti-patterns distribution over two online surveys

Category	**Open Online Survey**	
	1	**2**
Performance	AP1	AP5
Design	AP2 AP3 AP4	AP8 AP10 AP11
Architecture	AP6	AP7
Standardization	AP9	AP12

Table 6.8: Background of respondents of the openly online survey

Information	Respondent		
	I7	**I8**	**I9**
B1	Master	Bachelor	Master
B2	7 years	11 years	10 years
B3	PhD Student	Developer	Project Manager
B4	For my own use	Several projects in industry	Several projects in industry
B5	2 years	3 years	3 years
	Respondent		
	I10	**I11**	**I12**
B1	Bachelor	Bachelor	Bachelor
B2	10 years	3 years	3 years
B3	Developer	Developer	Developer
B4	Several projects in industry	Several projects in industry	Several projects in industry
B5	3 years	2 years	3 years
	Respondent		
	I13	**I14**	**I15**
B1	Bachelor	Bachelor	Bachelor
B2	4 years	5 years	3 years
B3	Developer	Developer	Master's Student
B4	Several projects in industry	Several projects in industry	Several projects in industry
B5	4 years	1 year	1 year

6.3.3 Results

6.3.3.1 Preliminary Online Survey

The results of the first six respondents, which were previously selected to participate in the online survey, on the proposed DI anti-patterns, are exhibited

in Table 6.9.

Table 6.9: Perception over the DI anti-patterns in preliminary online survey

DI	**Respondent**					
Anti-Pattern	**I1**	**I2**	**I3**	**I4**	**I5**	**I6**
AP1	Agree	Partially disagree	Partially agree	Agree	Partially agree	Partially disagree
AP2	Agree	Agree	Partially agree	Agree	Agree	Partially agree
AP3	Agree	Neutral	Agree	Agree	Agree	Agree
AP4	Partially agree	Neutral	Agree	Agree	Agree	Agree
AP5	Agree	Agree	Agree	Agree	Agree	Agree
AP6	Partially disagree	Agree	Agree	Agree	Agree	Agree
AP7	Partially disagree	Partially agree	Partially agree	Agree	Agree	Agree
AP8	Partially agree	Partially agree	Agree	Agree	Agree	Agree
AP9	Partially agree	Partially agree	Disagree	Agree	Agree	Agree
AP10	Partially agree	Agree	Agree	Agree	Agree	Agree
AP11	Agree	Agree	Agree	Agree	Agree	Agree
AP12	Agree	Agree	Agree	Agree	Agree	Agree

It is important to highlight that only 4 anti-patterns (AP1, AP6, AP7, and AP9) present disagreements. However, AP6 and AP7 present disagreements from a respondent with the highest rate of disagreements (I1). By conducting a qualitative analysis on his responses, we have observed that he considers the *Service Locator* pattern a good practice when it comes to integrate several frameworks in a software system, although he considers the strong dependence on these types of classes a drawback (AP6). We argue that there are better approaches to refrain the system to be coupled to a singleton object, such as the use of *Provider* classes and *Producer* methods, which were

explained along the instrumentation. Also, he asserts that the use of direct container calls does not yield drawbacks to the application, since he "never experienced the change of the framework of a project in production." We argue that the proposal on addressing direct container call as an anti-pattern is related to the increased effort in future maintenance activities on the application. Besides, the disagreements on AP1 and AP9 are results that we already expected, since the interview-based survey provided the same insights.

On the other side, we observed a positive perception over the DI anti-pattern instances. From 72 enquiries, only 4 disagreements and only 2 neutral responses are observed. It is noteworthy to mention that some neutral and disagreements responses do not come from the anti-pattern instance conjectured, but rather the resolution provided. The qualitative analysis as follows will go over these instances.

Regarding the answers on the willingness of the respondents on fixing the candidate DI anti-pattern, because of the format of open-ended question, a qualitative analysis was conducted over each response. The overall perceptions are described hereafter. Some respondents (**I1** and **I5**) have provided their opinion in Portuguese, so in order to explain the results, we translated some sentences to formal English. The complete responses can be found in Appendices 7.5.

AP1 have shown a result that is very close to the result found in the interview-based survey. In this preliminary online survey, 2 responses for each "Agree", "Partially agree", and "Partially disagree" option were collected. There is a pattern observed in the responses over fixing AP1 that suggests this anti-pattern should only be enforced in a specific case. For instance, performance is the main subject of attention, exemplified by the following opinions:

- **I3**: "I would only fix it if [the instance to be injected] is too heavy to be loaded eagerly"
- **I4**: "[T]he injection of unnecessary dependencies demands computing time"
- **I5**: "I would agree if the given instance is characterized as a bottleneck in the performance of the system"
- **I6**: "Probably yes, [however, if] I could measure the performance impacts first"

Besides, I1 and I2 expressed concerns over the risks entailed by a refactoring process of such instance in a stable source code and unresolved instances at construction time, respectively.

Next, as in the previous survey, AP2 is largely understood as a problem, as 4 respondents agree and 2 partially agree with this anti-pattern. In regard to fixing AP2, respondents primarily agreed that this instance hurt DIP, which is exemplified by the following quotes:

- **I2**: "Yes, the injection being based on interfaces allows for dynamic proxing, and automation of transactional control and similar transversal requirements."
- **I6**: "Yes. Because of DIP (Dependency Inversion Principle)."

However, I3 would only remove this instance if strictly necessary, stating that he "would only fix it if there is a real need for interface decoupling." Also, I1 states that he would only fix if there is the need to introduce a new feature in the given class.

Next, although AP3 did not show any instances in both open and closed-source repositories, the respondents were unanimous in agreeing AP3 is an anti-pattern.

Regarding AP4, a high rate of agreements were observed again. The respondents stated the following opinions:

- **I3**: "Yes. The code should be simpler and modularized."
- **I4**: "yes, as it does not follow the low coupling principle"
- **I5**: "God classes tend to hide business concepts that should be better explicit in properly different classes."
- **I6**: "Yes, it seems [God DI class] is doing to much and it will be hard to maintain."

On the other side, I1 and I2 expressed concerns over this anti-pattern instance. Although I1 generally agrees, he argues that in some circumstances AP4 may be necessary. For instance, I1 argues that in cases where a system must support calls from external system to an internal Application Programming Interface (API), it is a better approach to maintain a unique point of integration for the sake of avoiding a complex documentation for external development teams. In addition, I2 argues that this problem is not specific to DI, but rather "an architectural design flaw, already addressed by basic object orientation practices and most common pattern catalogs."

In regard to AP5, the respondents were unanimous in agreeing this entails an anti-pattern.

Next, AP6 also demonstrates a high rate of agreement. The opinions of the expert developers are expressed as follows:

- **I2**: "The fix on AP6 shows the most decoupled organization of code."
- **I3**: "Yes. If one uses DI, one should rely on it."
- **I4**: "yes, as the resulting code becomes more flexible and generic"
- **I6**: "Yes. Would fix it. But, it may be very hard on existing code of medium to large projects."

Again, I1 expressed concerns over decidedly defining AP6 as an DI anti-pattern. I1 asserts that there are cases where it is necessary to integrate several frameworks in a software project. Then, he asserts that although a strong dependence on a singleton which provides dependencies on run time is a bad characteristic in the source code, he often introduces ways to customize it and change the singleton behavior in an unit test, for example.

Although the negative consequences of AP7 is very related to AP6, AP7 demonstrates a lower acceptance in comparison with AP6. Most concerns are related to the need to resolving dependencies on run time that would not be possible through the DI container. For example, I2 asserts that "there are cases [...] in which dependencies may be dynamically resolved at run time, and even if a provider is created to proxy the resolution, somewhere, someone is going to have direct access to the container." In addition, I3 argues that "there are cases where you can only do what you need if you access the bean context." These comments are important because they explicitly tackles at limitations of current solutions in dependency injection. We assert that these gaps should be properly handled by current DI frameworks.

In addition, other respondents positively argued about fixing AP7, as shown by the following quotes:

- **I4**: "yes, as it violates the low coupling principle"
- **I5**: "Yes, [...] because of changing specific solutions in source code, the strong dependence of the application code brought difficulties to the process."
- **I6**: "Yes, the main goal of using DI is not depending directly on the injected objects. Also, depending on a framework would create the same kinds of problems."

Next, AP8 also have shown a high rate of positive responses. The main reasons for fixing this anti-pattern are explicit by the following quotes:

- **I2**: "I would fix it because it doesn't make sense to pass injections along as parameters. Unless, of course, you are supplying the injection to objects that live outside the container."

- **I3**: "Yes. Again one should rely on DI container when appropriate."
- **I4**: "yes, as it eliminates unnecessary code"
- **I6**: "Yes. No need for a class to receive one object just to pass it forward."

I1 argues that he agrees with the anti-pattern, however, in a case where the application relies on a library that does not provide mechanisms for injecting instances in its internal code, this anti-pattern actually becomes a solution. We do agree, but we suggest a better caveat is modularizing the dependence on this library, once in a newer version, this library could also enable injecting mechanisms in its internal modules.

Regarding AP9, although respondents agreed that enforcing the specification is a better choice, half of them expressed concerns over this anti-pattern applicability. The following sentences summarizes the concerns.

- **I2**: "I would fix whatever can be generic. However some frameworks have more resources than the oracle api. That said, I have been in projects that having access to specific resources of the framework was a greater benefit than maintaining the Api in specification level."
- **I3**: "No. If you choose a DI container implementation, and if it is a well informed and discussed choice, there is no reason not to go all the way through. One always gains and looses by making such a choice, but that's life."
- **I4**: "yes, as it makes the code less dependant of technology specifics"
- **I6**: "Yes, for me it makes no sense to depend on the DI framework. We use DI to reduce coupling so depending on a framework all around the code does not make sense."

AP10 is also unanimously agreed. The respondents expressed different reasons regarding the motivation for removing this anti-pattern from source code, as shown hereafter:

- **I1**: "Normally, I would not change. However, I can see cases where it is useful to change a default implementation to another one."
- **I2**: "I would fix AP10 to protect the injection and the whole architectural integrity"
- **I3**: "Yes. Rely on the DI container whenever it's possible, simpler and addresses the issue."
- **I4**: "yes, as it would prevent negative effects to ripple to other parts of the system."

- **I5**: "Yes, [...] it requires attention because it can be hard to detect such instances. It would be great if a static analysis tool could indicate such instances"
- **I6**: "Yes, exactly the same reason written in the negative consequence section."

AP11 is also unanimously agreed. The main reasons mentioned are: "it simplifies the code" (I4) and it "avoid[s] unnecessary duplicity."

Lastly, AP12 is also unanimously agreed. The main reason regards code comprehension. Opinions are summarized as follows:

- **I2**: "I would fix ap12 to maintain overall convention and organization of the project."
- **I3**: "Yes. There is no reason to have two inject annotations."
- **I4**: "yes, as it makes the code easier to understand and maintain."
- **I6**: "Yes. Very confusing having two ways of doing the same thing. I would prefer the chosen standard way of doing and follow it everywhere."

Although several responses showed a partial agreement, by verifying their responses on the willingness of fixing the anti-pattern, the respondents expressed an agreement with the anti-pattern, only expressing concerns over a conjectured scenario. Thus, the qualitative analysis was very important to uncover this kind of pattern in the responses. In addition, adding the question over fixing the anti-patterns also allowed the collection of responses of this nature. The results strengthen our confidence that the proposed catalog is important for development activities involving the application of DI.

After responding on the DI anti-patterns, the respondents were enquired about the utility of the catalog based on TAM facets. The adapted TAM questionnaire and the results are exhibited in the Table 6.10. These were ordered and positioned according to the results previously depicted. The Index column of both tables address the same questions depicted in Table 6.3.

In the preliminary survey, a strong positive perception is observed. 31 of 36 yield an agreement response, on which 3 of the 5 neutral responses comes from a specific respondent.

Table 6.10: Respondents perception over the catalog of DI anti-patterns (I1-I6)

Index	**Respondent**		
	I1	**I2**	**I3**
T1	Strongly Agree	Agree	Agree
T2	Strongly Agree	Agree	Agree
T3	Strongly Agree	Agree	Agree
T4	Strongly Agree	Neutral	Agree
T5	Strongly Agree	Agree	Agree
T6	Strongly Agree	Neutral	Agree
T7	Strongly Agree	Agree	Agree
T8	Strongly Agree	Agree	Agree
T9	Neutral	Neutral	Neutral
	I4	**I5**	**I6**
T1	Strongly Agree	Agree	Agree
T2	Strongly Agree	Agree	Strongly Agree
T3	Strongly Agree	Agree	Strongly Agree
T4	Strongly Agree	Agree	Strongly Agree
T5	Strongly Agree	Agree	Strongly Agree
T6	Strongly Agree	Agree	Agree
T7	Strongly Agree	Agree	Neutral
T8	Strongly Agree	Agree	Agree
T9	Strongly Agree	Agree	Agree

6.3.3.2 Openly Available Online Survey

The results of the openly available online survey are exhibited in Tables 6.11 and 6.12. We preferred to separate the exhibition of these results because the previous surveys targeted respondents previously selected with strong expertise in software development and object-oriented programming. In the openly available survey, any respondent over the internet could introduce their perceptions on the catalog.

Table 6.11: Perception over the DI anti-patterns in openly online survey 1

DI Anti-Pattern	Respondent			
	I7	**I8**	**I9**	**I10**
AP1	Agree	Partially disagree	Partially agree	Agree
AP2	Agree	Agree	Partially disagree	Agree
AP3	Neutral	Partially agree	Agree	Partially agree
AP4	Partially disagree	Agree	Agree	Partially agree
AP6	Agree	Agree	Partially agree	Neutral
AP9	Agree	Agree	Agree	Neutral

For the first openly online survey, which covers anti-patterns AP1-AP4, AP6, and AP9, only 3 (partial) disagreements are observed. In general, from 24 enquiries, 18 responses provided agreements over the instances. Again, we observed that some partial disagreements aimed at the resolution provided and not the instance of anti-pattern presented.

Due to the lack of expertise of the respondents, as expected, the comments regarding the anti-patterns were not as substantial as the ones provided by the expert developers in the previous survey. Besides, we assert that it is worthy to consider their points to strengthen our evidence on the validity of the proposed DI anti-pattern instances.

Although I8 partially disagrees with AP1, he contradicts himself by stating that "it's better to use the provider to lazy load the injected objects." In addition, he argues that the resolution provided is "a rather improvement than an anti pattern [fix]." In other words, it is inferred that he agrees with the anti-pattern but he does not agree with the provided resolution. In line with the answers from expert developers, I9 does not agree with the AP1 resolution because "the overload is small and should not be a problem." Again, impact on performance is cited by I7 and I10 on AP1. I7 asserts that he would fix it "due to the possible negative impact on the performance." I10 argues that "if it is a heavy object I would use AP1 solution."

Regarding AP2, I9 asserts that "there is no need to make an interface

for small systems." I9 concludes with: "If it is needed you can generate the interface afterwards upon needing." It is worthy to mention that the author of this work agrees with this assertion, however, testability of the class must be used as a parameter for avoiding an interface-oriented design. In other words, the testability of the class should be taken into consideration when defining an interface-oriented design or not. For instance, if an instance of this class is necessary in a unit test that primarily deals with a class that is coupled to it, a mock object could be used in order to avoid instantiation of all dependence chain of the first. The other respondents agreed that refraining modules to know concrete implementations is a design to target.

I10 suggested a different solution to AP3, on which he argues that a "point that would make me fix AP3 is that calling Provider methods instead of just have all complexity in his life-cycle make me feel I have more control of my code." I9 argues that AP3 "can generate performance issues in the system."

Regarding AP4, although I7 partially disagrees, he assert that "it is clear that a god class can produce negative consequences in maintenance," suggesting that he may agree that AP4 is an anti-pattern. In addition, I10 agrees with the resolution, however he warns about "creat[ing] so many classes with the same responsibilities." I8 and I9 agree with the resolution provided, arguing over "dividing responsibilities" and "improv[ing] maintainability", respectively.

On AP6, I10 argues that "you don't need to use the ServiceLocator to get the IDataSource each time you have to use it. You can create a global object and you can control it to call just once. In this case, DI doesn't bring such a huge difference for me." We argue that enforcing a global object won't enable the management of a scope different from a singleton. In addition, a global object may incur in the same drawbacks entailed by AP8. I9 argues that he would not fix it, "because ... [it is] not a big problem."

Lastly, in AP9, I10 argues that "in this specific case, we are talking about a huge framework that brings lots of benefits to the application (Spring). So, for me it worth even if it brings higher coupling to framework specifics." In opposition, I7 and I8 argues over "keep[ing] the code complied to JSR-300 and not to a specific framework" and "avoid[ing] hard code framework into abstract code," respectively.

Table 6.12: Perception over the DI anti-patterns in openly online survey 2

DI	Respondent				
Anti-Pattern	**I11**	**I12**	**I13**	**I14**	**I15**
AP5	Partially disagree	Agree	Agree	Agree	Agree
AP7	Agree	Partially agree	Disagree	Neutral	Partially agree
AP8	Agree	Agree	Disagree	Agree	Partially agree
AP10	Partially agree	Agree	Agree	Agree	Agree
AP11	Agree	Neutral	Agree	Agree	Agree
AP12	Agree	Disagree	Agree	Agree	Agree

For the second openly online survey, which covers anti-patterns AP5, AP7, AP8, and AP10-AP12, again, only 3 disagreements are observed. From 30 enquiries, 25 responses provided agreements over the instances.

Again, AP5 shows a high rate of agreement. Although I11 partially disagrees, he asserts that he would "apply the suggested solution."

Next, AP7 responses vary drastically. Although most responses (4) range from agree to partially agree, a disagreement appears. I13 argues that he would stick with the direct container call: "This may be used to improve performance, only loading some dependency when it is absolutely needed. The extra provider will add unnecessary complexity, only hiding the original intention." Also, I12 is not sure about the resolution, by stating: "Maybe. In this case I would've used a service locator, or at least encapsulate the dependency in a method of the provider object." It is important to state that we suggested in the instrumentation that encapsulation over a provider method is a feasible solution too.

Regarding AP8, the only disagreement is expressed by I13, which states that "this situation is necessary when integrating third-party code that will not have access to the DI container. It also allows a function or method to receive different implementations of a parameter." We agree with the first statement, however, the second statement incur in a violation of DIP and IoC principle.

AP10 and AP11 do not present any disagreement. For instance, for AP10, I15 states that "it is quite obvious that is wrong. In addition, frameworks

documentation do not provides this example."

Lastly, through analyzing I12 response on AP12, we found that he actually agrees with the anti-pattern. I12 disagreement on AP12 comes from the fact that he does not know that set method is a form of injection, as stated as follows: "I know that having two ways of injecting an object is wrong, however, this set method is wrong too. If you wanna create an object I would to this with spring injection or constructor injection, so it would only be created at the start of the object, and not with a set method."

Regarding the openly online survey, the adapted TAM questionnaires and their results are exhibited in the Tables 6.13 and 6.14. Again, these were ordered and positioned according to previous results and the Index column of both tables address the same questions depicted in Table 6.3. From 81 TAM questions, only 17 does not yield an agreement response. We consider the results positive and strengthen our confidence on the usefulness, ease of use and intention to use of the proposed catalog. Most importantly, the results suggests that the catalog is helpful and that developers show willingness to apply it.

Table 6.13: Respondents perception over the catalog of DI anti-patterns (I7-I10)

Index	**Respondent**			
	I7	**I8**	**I9**	**I10**
T1	Strongly Agree	Strongly Agree	Agree	Strongly Agree
T2	Strongly Agree	Strongly Agree	Agree	Strongly Agree
T3	Strongly Agree	Strongly Agree	Agree	Strongly Agree
T4	Agree	Strongly Agree	Strongly Agree	Strongly Agree
T5	Disagree	Neutral	Strongly Agree	Strongly Agree
T6	Disagree	Neutral	Strongly Agree	Strongly Agree
T7	Strongly Agree	Neutral	Agree	Strongly Agree
T8	Disagree	Agree	Strongly Agree	Strongly Agree
T9	Neutral	Agree	Agree	Strongly Agree

Table 6.14: Respondents perception over the catalog of DI anti-patterns (I11-I15)

Index	Respondent				
	I11	**I12**	**I13**	**I14**	**I15**
T1	Agree	Agree	Strongly Agree	Agree	Strongly Agree
T2	Agree	Agree	Strongly Agree	Agree	Strongly Agree
T3	Agree	Agree	Strongly Agree	Agree	Agree
T4	Strongly Agree	Neutral	Neutral	Neutral	Strongly Agree
T5	Agree	Neutral	Neutral	Agree	Agree
T6	Agree	Agree	Agree	Agree	Agree
T7	Agree	Agree	Neutral	Agree	Agree
T8	Strongly Agree	Disagree	Neutral	Agree	Agree
T9	Strongly Agree	Neutral	Disagree	Neutral	Agree

6.4 Reflection

Based on the opinion of the respondents collected throughout the application of the three surveys (interview-based expert survey on Section 6.2, online expert survey on Section 6.3.3.1, and openly available online survey on Section 6.3.3.2) covered in this chapter, along with comments gathered from the evaluation members, we realized the catalog needed a new version. Thus, the new version of the catalog of DI anti-patterns must embrace the reflections made from the point of view of developers over the catalog. Therefore, the updated version of the catalog is found in Appendix B. The main change points are summarized in Tables 6.15 and 6.16.

6.5 Threats to Validity

The threats of validity are organized according to Wohlin et al. [37] proposition.

Internal Validity. Regarding the first 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. It served as a preliminary assessment of the structure of the catalog and the instrumentation employed. Regarding the online survey, we provided more information regarding the source code examples to assist the respondents towards fulfilling the online questionnaire.

Construct Validity. Our qualitative analysis relies simple Likert scale agreements on the anti-patterns and the TAM statements. We reinforce the interpretation of the results providing argumentation based on open text answers. TAM is a widely employed tool to measure the perceived acceptance of technology propositions.

External Validity. Regarding the interview-based survey, as we planned to conduct a limited amount of interviews with a limited amount of subjects, the instrumentation is available for external replications. Regarding subject representatives, we selected experienced developers from three different organizational units. Regarding the instrumentation, we peer-reviewed the material before presenting it to the subjects. Regarding the preliminary online survey, although representatives that answered our survey are from the same institution, we assert that this does not yield a threat since they are lecturers and do not work closely. For the online instrumentation, we also used a peer-review process to employ the division of the survey into two distinct surveys.

6.6 Concluding Remarks

In this chapter, we described our performed strategy to evaluate our candidate catalog of DI anti-patterns. We designed three different surveys to gather the perception over the proposed catalog from industry practitioners and experts in object-oriented design and software development. In summary, the proposed anti-patterns were broadly accepted by the respondents. However, we observed that the context on which they are applied contributes to their acceptance. Besides, most of the disagreements did not come from the anti-pattern per se, but rather from the provided resolution example. Thus, introducing the question over the willingness of removing the anti-pattern from source code gave us important insights from the respondents. Nevertheless, we found that our candidate catalog yields instances of anti-patterns related to the employment of DI in software systems, opening a window for investigating feasible code transformations to remove these instances from source code.

Table 6.15: Summary of updates in the catalog (Part 1)

Source	**Problem/Proposition**	**Argument/Solution**
AP1 - Intransigent injection	Whether AP1 is dependent of the problem being solved	Reinforce the problem context where this anti-pattern is applied.
AP2 - Concrete class injection	In cases of small systems, an interface can be created afterwards upon needing	Smalls systems might not benefit from an interface-oriented design in some cases. The updated catalog reinforces this aspect.
AP3 - Long Producer method	Whether AP3 is just an instance of long method bad smell	Long method bad smell is applied to any generic method. AP3, however, is applied only when the provider method performs activities outside the its core scope, which is providing a dependence instance. The updated catalog reinforces this aspect.
AP4 - God DI class	Whether AP4 is just an instance of God class bad smell	AP4 concerns the injection of a substantial number of dependencies in a class. This anti-pattern is primarily concerned over injected instances that are often inconsequentially introduced by developers without reasoning over the increased dependence of the class with other components. The updated catalog reinforces this aspect.
AP5 - Non used injection	Whether AP5 is an anomaly in regard to DI specifically	The problem is more related to current integrated development environments (IDEs) which, once annotated, even the though the attributed is not used, do not warn the developer about the issue. The updated catalog reinforces this aspect.

Table 6.16: Summary of updates in the catalog (Part 2)

Source	Problem/Proposition	Argument/Solution
AP6 - Static dependence provider	In cases where different frameworks must be integrated in a single project, a "true" singleton can be used to provide dependencies, even though incurring in strong coupling observed in classes of the project	The anti-pattern does not assert about these type of situations. The updated catalog reinforces this aspect.
AP7 - Direct container call	There are cases that dependencies may be dynamically resolved at runtime with the support of the DI container	In these cases, the catalog advocates to wrap the container call in a provider class
AP8 - Open window injection	In case where an injected object needs to be passed to a component (or third-party solution) that lives outside the container, the anti-pattern is a solution	Reinforce the problem context.
AP9 - Framework coupling	The likelihood of changing a previously defined framework in a project is low, customers may not be willing to cover the costs of such change	Reinforce the decision of relying on the specification is better applied in the context of new software projects
Structural definition	The lack of explicitly arguing about the context of an anti-pattern made respondents skeptical about its applicability	Introduce the context element in the structural definition
Access modifiers on code snippets	The lack of private access modifier lead to open door injection	Perform the changes on snippets

7
Concluding Remarks

7.1
Introduction

The goal of this dissertation concerns addressing the lack of guidance on how to effectively detect, analyze, and remove DI anti-patterns from elements of source code.

Academic literature does not properly cover DI anti-patterns in source code. Industry-oriented publications, on the other hand, fail to provide empirical evidence on the practical relevance of their propositions. Therefore, we present a novel catalog of Java DI anti-patterns built upon object-oriented design principles. Furthermore, we provide empirical evidence on their practical occurrence, and usefulness and acceptance by enquiring practitioners from industry. In the following, we summarize our conclusions and discuss practical relevance of our research.

7.2
Summary of Conclusions

First, we investigated the literature on DI with the aim of identifying existing documentation on DI anti-patterns. Our focus was on understanding the completeness of propositions of DI anti-patterns and smells. Then, based on the gaps uncovered, such as lack of empirical evidence over the practical relevance on existing propositions, we applied two methodological approaches to derive an initial catalog of Java DI anti-patterns. Based on observations of bad implementation practices related to the employment of DI in closed-source projects in past work experiences (inductive approach) and conjecturing over a set of instances that harm the principles behind DI (deductive approach), namely, DIP and IoC, and object-oriented design principles, such as GRASP and SOLID, an initial effort towards documenting a catalog of Java DI anti-patterns was taken.

Second, motivated by our proposition over a candidate set of DI anti-patterns, we designed a static analysis tool called DIAnalyzer to automatically flag and report instances of anti-patterns from source code. An evaluation

carried out on DIAnalyzer revealed that the tool is reliable and can effectively retrieve instances of DI anti-patterns from the source code. Then, we applied the tool to a set of software systems, both open and closed-source. The investigation revealed that the DI anti-patterns are general and occur within different projects.

Lastly, we designed a study to analyze the acceptance and usefulness of the catalog from the point of view of expert developers. In order to allow an initial in-depth evaluation of our proposed catalog, an interview-based survey was undertaken to mitigate risks related to the design of the instrumentation and description of each anti-pattern. Feedback gathered from the expert developers regarding the instrumentation was used as input for the next step. Then, in order to scale our experiments, we have designed an online version of our survey. Before making it openly available, we have made it available to a set of selected representatives of experienced developers in order to understand possible gaps related to the instrumentation and the size of the survey. After minor adaptations, we have made it openly available online. After gathering the perception of developers on the proposed catalog, the results indicate that the catalog is perceived as relevant and useful. Lastly, after gathering the feedback of the evaluation members of this work, along with the collected point of view of developers over the instances of anti-patterns presented in this dissertation, we built an updated catalog of DI anti-patterns. The updated version of the catalog can be found in the Appendix B.

7.3 Limitations

Although we aimed to cover a wide range of anti-patterns distributed in different characteristics, namely, performance, design, architecture, and standardization, our catalog may miss some other DI related problems. Therefore, different researchers 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. This is due to the fact that every software system has its own characteristics, even though is written in a specific programming language and making use of a given framework.

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#).

Lastly, although we invested efforts to appropriately select representative software projects that employ DI, we are aware that including a wider range of software projects would improve external validity of our results.

7.4 Future Work

Although existing studies reported the existence of DI smells and DI anti-patterns, there is no study that investigates its completeness and how to effectively remove such bad implementation practices from source code. By tackling the gathered perceptions from expert developers described in this dissertation, we are able to better support developers on removing bad DI anti-patterns from source code. Furthermore, we consider important to explore a catalog of refactorings to support practitioners to effectively remove DI anti-patterns from source code. The catalog would map patterns of refactoring and offer a set of guidelines (i.e. code transformations) that developers can follow in order to improve the structural quality of Java applications that employ DI.

7.5 Research Publications

Table 7.1 lists the accepted and submitted publications derived from this dissertation.

Table 7.1: Publications derived from this Master's thesis

Paper Title	**Venue**	**Status**
Towards a Catalog of Java Dependency Injection Anti-Patterns [41]	SBES 2019	Accepted (3^{rd} best paper)
Unveiling Java Dependency Injection Anti-Patterns	JSERD (invited paper)	Submitted

Table 7.2 lists other accepted and submitted publications derived throughout the master's period.

Table 7.2: Other publications derived throughout the master's period

Paper Title	**Venue**	**Status**
A Systematic Mapping of Software Engineering Approaches to Develop Big Data Systems [43]	SEAA 2018	Accepted
Desmistificando Blockchain: Conceitos e Aplicações [44]	Computação e Sociedade (SBC)	Accepted
Towards a Technique for Extracting Relational Actors from Monolithic Applications [45]	SBBD 2019	Accepted
From a Monolithic Big Data System to a Microservices Event-Driven Architecture	SEAA 2020	To submit

Bibliography

[1] ARNAOUDOVA, V.; DI PENTA, M.; ANTONIOL, G. ; GUEHENEUC, Y.-G.. **A new family of software anti-patterns: Linguistic anti-patterns**. In: PROCEEDINGS OF THE 2013 17TH EUROPEAN CONFERENCE ON SOFTWARE MAINTENANCE AND REENGINEERING, CSMR '13, p. 187–196, Washington, DC, USA, 2013. IEEE Computer Society. https://doi.org/10.1109/CSMR.2013.28.

[2] BROWN, W. J.; MALVEAU, R. C.; III, H. W. M. ; MOWBRAY, T. J.. **AntiPatterns Refactoring Software, Architectures, and Projects in Crisis**. John Wiley & Sons, 1998.

[3] L. BRIAND, J. D.; WUST., J.. **A unified framework for coupling measurement in object-oriented systems**. IEEE Transactions on Software Engineering, 24(1):91–121, 1999.

[4] CEDRIM, D.. **Understanding and Improving Batch Refactoring in Software Systems**. PhD thesis, PUC-Rio, 2018.

[5] CHIDAMBER, S. R.; KEMERER., C. F.. **A metrics suite for object oriented design**. IEEE Transactions on Software Engineering, 20(6):476–493, 1994.

[6] CHEN, B.; JIANG, Z. M. J.. **Characterizing and detecting anti-patterns in the logging code**. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, September 2017.

[7] COPLIEN, J.. **A generative development-process pattern language**. ACM Press/Addison-Wesley Publishing Co. New York, NY, USA, 1995.

[8] COPLIEN, J.; SCHMIDT, D.. **Pattern Languages of Program Design**. Leanpub, 1995.

[9] CRASSO, M.; MATEOS, C.; ZUNINO, A. ; CAMPO, M.. **Empirically assessing the impact of di on the development of web service applications**. Journal of Web Engineering, 9:66–94, 2010.

[10] DAVIS, F. D.. **Perceived usefulness, perceived ease of use, and user acceptance of information technology**. JSTOR, 13(3):319–340, 1989.

[11] FOWLER, M.. **Inversion of control containers and the dependency injection pattern**, 2004. `http://martinfowler.com/articles/injection.html`. Last accessed 2004-01-23.

[12] GAMMA, E.; HELM, R.; JOHNSON, R. ; VLISSIDES, J.. **Design patterns: elements of reusable object-oriented software**. Addison-Wesley Longman Publishing Co., 1995.

[13] GOOGLE; GUICE. **Guice framework**, 2019. `https://github.com/google/guice`. Last accessed 2019-11-01.

[14] JOHNSON, R. E.; FOOTE, B.. **Designing reusable classes**. Journal of Object-Oriented Programming, 1(2):22–35, 1988.

[15] KICZALES, G.; LAMPING, J.; MENDHEKAR, A.; MAEDA, C.; LOPES, C.; LOINGTIER, J.-M. ; IRWIN, J.. **Aspect-oriented programming**. In: ECOOP'97 — OBJECT-ORIENTED PROGRAMMING, p. 220–242, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[16] LANZA, M.; MARINESCU, R.. **Object-Oriented Metrics in Practice**. Springer, 2006.

[17] LARMAN, C.. **Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)**. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[18] LODICO, M.G., S. D. V. K.. **Methods in Educational Research: From Theory to Practice**. John Wiley & Sons, 2010.

[19] LINAKER, J.; SULAMAN, S. M.; MAIANI DE MELLO, R. ; HÖST, M.. **Guidelines for conducting surveys in software engineering**. Technical report, Lund University, 2015. [Publisher Information Missing].

[20] MARTIN, R. C.. **Design principles and design patterns**, 2000. `https://web.archive.org/web/20150906155800/http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf`. Last accessed in 2009-01-01.

[21] MARTIN, R. C.. **The dependency inversion principle**. Report., 8(6):61–66, 1996.

[22] MEYER, B.. **Object-Oriented Software Construction**. Prentice Hall, 1988.

[23] OLBRICH, S. M.; CRUZES, D. ; SJØBERG, D. I. K.. **Are all code smells harmful? A study of god classes and brain classes in the evolution of three open source systems**. In: 26TH IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE (ICSM 2010), SEPTEMBER 12-18, 2010, TIMISOARA, ROMANIA, p. 1–10, 2010.

[24] PRASANNA, D. R.. **Dependency Injection**. Manning Publications Co., Greenwich, CT, USA, 2009.

[25] RAZINA, E.; JANZEN, D.. **Effects of dependency injection on maintainability**. In: INTERNATIONAL CONFERENCE SOFTWARE ENGINEERING AND APPLICATIONS, November 2007.

[26] REQUESTS, J. J. S.. **Jsr 330: Dependency injection for java**, 2015. `https://www.jcp.org/en/jsr/detail?id=330`. Last accessed 2004-01-23.

[27] REQUESTS, J. J. S.. **Jsr-299: Contexts and dependency injection for the java ee platform**, 2004. `http://docs.jboss.org/cdi/spec/1.0/html`. Last accessed 2004-01-23.

[29] PIVOTAL. **Spring framework**, 2019. `https://spring.io/`. Last accessed 2019-11-01.

[30] ANICHE, M.. **Repodriller framework**, 2019. `https://github.com/mauricioaniche/repodriller`. Last accessed 2019-11-01.

[31] ROUBTSOV, S.; SEREBRENIK, A. ; VAN DEN BRAND, M.. **Detecting modularity smells in dependencies injected with java annotations**. In: SOFTWARE MAINTENANCE AND REENGINEERING EUROPEAN CONFERENCE, p. 244–247, March 2010.

[32] SEEMANN, M.. **Dependency Injection in .NET**. Manning Publications Co., Shelter Island, NY, 2012.

[33] SMITH, N.; VAN BRUGGEN, D. ; TOMASSETTI, F.. **JavaParser: Visited**. Leanpub, 2018. `https://leanpub.com/javaparservisited`.

[34] TORCHIANO, M.; FERNÁNDEZ, D. M.; TRAVASSOS, G. H. ; DE MELLO, R. M.. **Lessons learnt in conducting survey research**. In: PROCEEDINGS OF THE 5TH INTERNATIONAL WORKSHOP ON CONDUCTING EMPIRICAL STUDIES IN INDUSTRY, p. 33–39. IEEE Press, 2017.

[35] TURNER, M.; KITCHENHAM, B.; BRERETON, P.; CHARTERS, S. ; BUDGEN, D.. **Does the technology acceptance model predict actual use? a systematic literature review**. Information and Software Technology, 52:463–479, 2010.

[36] VAN DEURSEN, S.; SEEMANN, M.. **Dependency Injection Principles, Practices, Patterns**. Manning Publications Co., Shelter Island, NY, 2018.

[37] WOHLIN, C.; RUNESON, P.; HÖST, M.; OHLSSON, M.; REGNELL, B. ; WESSLÉN, A.. **Experimentation in software engineering**. Springer, 2012.

[38] YANG, H. Y.; TEMPERO, E. ; MELTON, H.. **An empirical study into use of dependency injection in java**. In: AUSTRALIAN CONFERENCE ON SOFTWARE ENGINEERING, March 2008.

[39] WILSON, J.. **Essentials of Business Research: A Guide to Doing Your Research Project**. SAGE Publications, 2010.

[40] SJØBERG D.I.K., DYBÅ T., A. B. H. J.. **Building Theories in Software Engineering, In: Shull F., Singer J., Sjøberg D.I.K. (eds) Guide to Advanced Empirical Software Engineering**. Springer, 2008.

[41] LAIGNER, R.; KALINOWSKI, M.; CARVALHO, L.; MENDONÇA, D. S. ; GARCIA, A.. **Towards a catalog of java dependency injection anti-patterns**. In: PROCEEDINGS OF THE XXXIII BRAZILIAN SYMPOSIUM ON SOFTWARE ENGINEERING, SBES 2019, SALVADOR, BRAZIL, SEPTEMBER 23-27, p. 104–113, 2019.

[43] LAIGNER, R.; KALINOWSKI, M.; LIFSCHITZ, S.; MONTEIRO, R. S. ; DE OLIVEIRA, D.. **A systematic mapping of software engineering approaches to develop big data systems**. In: 2018 44TH EUROMICRO CONFERENCE ON SOFTWARE ENGINEERING AND ADVANCED APPLICATIONS (SEAA), p. 446–453. IEEE, 2018.

[44] ALVES, P.; LAIGNER, R.; NASSER, R.; ROBICHEZ, G.; LOPES, H. ; KALINOWSKI, M.. **Desmistificando blockchain: Conceitos e aplicações**. es). Computação e Sociedade. Rio de Janeiro: Sociedade Brasileira de Computação, p. 1–24, 2018.

[45] LAIGNER, R.; LIFSCHITZ, S.; KALINOWSKI, M.; POGGI, M. ; SALLES, M. A. V.. **Towards a technique for extracting relational actors from monolithic applications**. In: ANAIS DO XXXIV SIMPÓSIO BRASILEIRO DE BANCO DE DADOS, p. 133–144. SBC, 2019.

[46] WIERINGA, R. J.. **Design Science Methodology for Information Systems and Software Engineering**. Springer, 2014.

Appendices

A
Responses over fixing DI anti-patterns

Table A.1: I1 Perception over fixing DI anti-patterns in preliminary survey (part 1)

DI Anti-Pattern	**Respondent Perception**
AP1	"Nao. Apesar das consequencias negativas, dependendo do tamanho do projeto e das classes envolvidas o impacto nao chega a ser tao grande e nao vale o esforco de tempo e o risco de adicao de bug que o refactory ira trazer ."
AP2	"Depende. Se eu for na classe que possua esse antes pattern para adicionar uma nova feature ou funcionalidade, sim. Se apenas tiver usando essa classe e me deparar com ela nao."
AP3	"Nao. Opiniao muito particular, nao acredito que pegar um metodo grande e separar em metodos pequenos (famoso diminuir a complexidade ciclomatica) seja algo que diminua a complexidade de entendimento do codigo. Acho que a correcao disso vem de uma refactory mais profunda que nao eh simples de ser feita e cara quando se fala de um sistema legado. Deixaria como esta. Faria esta refactory se fosse extremamente necessario, como necessidade de correcao de um bug, por exemplo."
AP4	"De modo geral, concordo que seja um anti-pattern, mas consigo ver casos onde isso e necessario. Quando trabalhamos com componentizacao e modularizacao, principalmente com chamadas de sistemas extermos para API interna, as vezes eh melhor pagar um preco de um facade como esse grande e ter um ponto focal unico de integracao externa (simplicidade de explicacao para uma equipe de fora, simplicidade de documentacao, por exemplo)."
AP5	"Sim. Mesmo se nao estiver fazendo nada na classe costumo remover. E importante so observar as classes que possuem atributos publicos ou package, pois podem estar sendo usados por uma classe filha."

Table A.2: I1 Perception over fixing DI anti-patterns in preliminary survey (part 2)

DI Anti-Pattern	**Respondent Perception**
AP6	"Em geral costumo nao usar esse tipo de service locator, porem existem casos, onde e necessario integrar diversos frameworks num mesmo projeto (ja passei por isso) em que ele se torna necessario. Esse tipo de singleton considero como um "singleton verdadeiro", pois apesar de ter uma dependencia forte com ele, eh customizado e possivel de ter comportamento alterado durante um teste unitario por exemplo."
AP7	"Por coerencia com a respostas de cima, coloquei como discordo parcialmente. Alem disso, nunca vi na pratica trocar o framework de um projeto em producao. Eh lenda urbana isso."
AP8	"Concordo, mas no caso de ter uma biblioteca que vc nao tem como injetar dentro dela, nao eh um problema e a solucao."
AP9	"Concordo que se tiver como usar facilmente a especificacao, nao precisamos usar o framework. mas acredite (como disse em cima) ninguem vai trocar o framework. O cliente nao vai pagar por isso e vc nao vai querer tirar do bolso da sua empresa essa mudanca."
AP10	"Nao faria comumente, mas consigo ver casos onde possa ser util dar a possibilidade de trocar a implementacao default injetada por uma outra."
AP11	"Sim, mas apenas quando estou adicionando uma nova funcionalidade a esta classe ou alterando .Se eu apenas olhar o codigo mas nao tiver q mexer nela, eu nao altero."
AP12	"Sim. Seguindo a mesma linha de so altrar quando for criar uma funcionallidade ou modificar essa classe."

Table A.3: I2 Perception over fixing DI anti-patterns in preliminary survey (part 1)

DI Anti-Pattern	**Respondent Perception**
AP1	"No. I think the principle of depency injection is to wire the architecture. To create a resolver seems to me like just injecting a factory, and then the whole point of inversion of control is lost. To me this is a bad pratice, as the application can start with unresolved injections which may be potentially broken, leveraging the problem to runtime instead of startup time."
AP2	"Yes, the injection being based on interfaces allows for dinamyc proxing, and automation of transactional control and similar transversal requirements."
AP3	"The problem decipted on ap3 seems to me to be caused by bad architetural design, and not a bad pratice of dependency injection itself. Dependency injection and Inversion of Control have been created just to avoid this cause of programming. Doing something like that seems more like a bad design or a misunderstanding of dependency injection itself, than a common bad pattern."
AP4	"In the same manner as ap3, this doesnt seem to me as a dependency injection problem, but really an architectural design flaw, already adressed by basic object orientation practices and most common pattern catalogs."
AP5	"I would fix ap5 because it burdens the container in providing an unused dependency, and also because it prejudices the understanding of the code."
AP6	"The fix on ap6 shows the most decoupled organization of code."
AP7	"In the case depicted it makes sense to apply the fix. There are cases, although, in which dependencies may be dinamycally resolved at run time, and even if a provider is created to proxy the resolution, somewhere, someone is going to have direct access to the container."
AP8	"I would fix it because it doesnt make sense to pass injections along as parameters. Unless, of course, you are supplying the injection to objects that live outside the container."

Table A.4: I2 Perception over fixing DI anti-patterns in preliminary survey (part 2)

DI Anti-Pattern	**Respondent Perception**
AP9	"I would fix whatever can be generic. However some frameworks have more resources than the oracle api. That said, I have been in projects that having access to specific resources of the framework was a greater benefit than maintaining the Api in specification level."
AP10	"I would fix ap10 to protect the injection and the whole architectural integrity"
AP11	"I would fix it to avoid unecessary duplicity."
AP12	"I would fix ap12 to maintain overall convention and organization of the project."

Table A.5: I3 Perception over fixing DI anti-patterns in preliminary survey

DI Anti-Pattern	**Respondent Perception**
AP1	"I would only fix it if example1 is too heavy to be loaded eagerly."
AP2	"I would only fix it if there is a real need for interface decoupling. I see no point in having a lot of interfaces and only one concrete class for each one of them."
AP3	"Yes. It makes sense and simplifies the code."
AP4	"Yes. The code should be simpler and modularized."
AP5	"Yes. A good well configured IDE can do this or warn me to do it."
AP6	"Yes. If one uses DI, one should rely on it."
AP7	"There are cases where you can only do what you need if you access the bean context. Specially if you use spring boot, which hides a lot of the framework complexities and does not necessarily covers all issues. For instance: centralized error handling with async methods. These cases should be, in anyway, exceptions."
AP8	"Yes. Again one should rely on DI container when appropriate."
AP9	"No. If you choose a DI container implementation, and if it is a well informed and discussed choice, there is no reason not to go all the way through. One always gains and looses by making such a choice, but that's life."
AP10	"Yes. Rely on the DI container whenever it's possible, simpler and addresses the issue."
AP11	"Yes. It's more adequate."
AP12	"Yes. There is no reason to have two inject annotations."

Table A.6: I4 Perception over fixing DI anti-patterns in preliminary survey

DI Anti-Pattern	**Respondent Perception**
AP1	"yes, as the injection of unnecessary dependencies demands computing time"
AP2	"yes"
AP3	"yes, as it does not follow the low cohesion principle"
AP4	"yes, as it does not follow the low coupling principle"
AP5	"yes, because the resulting code if easier do understand and more performatic"
AP6	"yes, as the resulting code becomes more flexible and generic"
AP7	"yes, as it violates the low coupling principle"
AP8	"yes, as it eliminates unnecessary code"
AP9	"yes, as it makes the code less dependant of technology specifics"
AP10	"yes, as it would prevent negative effects to ripple to other parts of the system"
AP11	"yes, as it simplifies the code without loosing flexibility"
AP12	"yes, as it makes the code easier to understand and maintain"

Table A.7: I5 Perception over fixing DI anti-patterns in preliminary survey

DI Anti-Pattern	**Respondent Perception**
AP1	"Sim, se fosse caracterizado como causa de um gargalo no desempenho do sistema."
AP2	"Fico na dúvida se eu refatoraria. Teria uma prioridade baixa."
AP3	"Sim, se fosse identificado como um ponto de gargalo do desempenho"
AP4	"Sim, sem dúvida. God classes tendem a esconder conceitos do negócio que poderiam estar mais explícitos em classes próprias."
AP5	"Sim, para evitar que os desenvolvedores fiquem confusos (por que isto está aqui se não está sendo usado?) e, com isso, demorem mais para implementar novas features ou corrigir bugs."
AP6	"Não. Como não uso um framework/lib de DI atualmente, teria dificuldade para ver esta falta de padronização na solução para obter a dependência por duas formas diferentes."
AP7	"Sim, trabalho em um projeto com mais de 10 anos e já tivemos que mudar soluções específicas uma ou duas vezes e tivemos dificuldade de fazê-lo pela grande dependência do código da aplicação com o código da solução a ser trocada"
AP8	"Sim, mas teria baixa prioridade."
AP9	"Sim, pelo menos motivo de AP7"
AP10	"Sim, mas poderia ser difícil identificar. Requer uma atenção e conhecimento de DI diferenciados. Seria ótima se uma ferramenta de análise estática indicasse isso."
AP11	"Sim, porque essa situação pode levar a erros confusos para debugar."
AP12	"Uma implementação dessas se não tiver um efeito colateral perceptível é possível que passe batido. Se fosse indicado por uma ferramenta de análise, sim, corrigiria."

Table A.8: I6 Perception over fixing DI anti-patterns in preliminary survey (part 1)

DI Anti-Pattern	**Respondent Perception**
AP1	"Probably yes, if using Java, but: 1- I could measure the performance impacts first. 2- If not using a native solution (or not a native solution in a different language), would we still be able to use a Provider? wouldn't it be like a service locator? 3- If we are using the provider, why not using it all the time, even on the constructor, and maybe that being the default? 4- If not using Java, the constructor signature would have the parameters and would save the objects so they could be used by all instance methods. Even though the constructor is not actually using them, it does not seem that strange that they receive the objects to construct the class."
AP2	"Yes. Because of DIP (Dependency Inversion Principle). But with one caveat: Dependency Injection became a complex subject but if we are talking about manual dependency injection and the most simple definition, one object providing dependencies to another object, one could inject a concrete object that does not need an interface into another object. Like passing an entity to a DAO as in Clean Architecture. It would depend of the different policy levels in the architecture. So I think of it as an anti-pattern in most cases, where it violates DIP."
AP3	Yes. It seems the code below @Produce is doing more than expected so the programmers need to do more investigation than needed to understand it.
AP4	Yes, it seems class D is doing to much and it will be hard to maintain.
AP5	I usually set the compiler/IDE to treat warnings as errors and that would be one. Even without a performance problem, for cleaner code.
AP6	Yes. Would fix it. But, it may be very hard on existing code of medium to large projects.

Table A.9: I6 Perception over fixing DI anti-patterns in preliminary survey (part 2)

DI Anti-Pattern	**Respondent Perception**
AP7	Yes, the main goal of using DI is not depending directly on the injected objects. Also, depending on a framework would create the same kinds of problems.
AP8	"Yes. No need for a class to receive one object just to pass it forward. But, the AP8 definition is saying ""Anti-pattern is applied when an injected instance is not used..."", even though parser is being used in the first example. So, there are worse cases when the first class does not even use the injected object."
AP9	Yes, for me it makes no sense to depend on the DI framework. We use DI to reduce coupling so depending on a framework all around the code does not make sense.
AP10	Yes, exactly the same reason written in the negative consequence section.
AP11	Yes, same as AP10.
AP12	Yes. Very confusing having two ways of doing the same thing. I would prefer the chosen standard way of doing and follow it everywhere.

Table A.10: I7 Perception over fixing DI anti-patterns in preliminary survey

DI Anti-Pattern	**Respondent Perception**
AP1	"Yes, due to the possible negative impact on the performance."
AP2	"Yes, in order to fix a misuse of the Dependency Injection."
AP3	"Yes, but mainly in order to reduce the cyclomatic complexity."
AP4	"Yes. It is clear that a god class can produce negative consequences in maintenance."
AP6	"Yes, in order to remove the direct dependency from E to ServiceLocator."
AP9	"Yes, in order to keep the code complied to JSR-300 and not to a specific framework."

Table A.11: I8 Perception over fixing DI anti-patterns in preliminary survey

DI Anti-Pattern	**Respondent Perception**
AP1	"It's better to use the provider to lazy load the injected objects. However, I see this a reather improvement than an anti pattern"
AP2	"Yes, creating interfaces to avoid coupling of the implementation."
AP3	"Yes, but it can be solved in a simpler way in my view. By creating small methods with single responsibility on appropriate classes"
AP4	"Yes, dividing responsibilities "
AP6	"Yes, because the ap6 doesn't mention that this is the only place that a implementation is being chosen like the suggested fix (projectconfigbeans). However, I'm not sure If the method getbeaninstance is controlled by the developer or it's a from a internal library - I don't develop in Java. If it is controlled by the debelopers, then it's not that bad - because the control over the factory looks similar to the fix for me."
AP9	"Yes, avoid hardcode framework into abstract code"

Table A.12: I9 Perception over fixing DI anti-patterns in preliminary survey

DI Anti-Pattern	**Respondent Perception**
AP1	"No, the overload is small and should not be a problem"
AP2	"No, there is no need to make an interface for small systems. If it is needed you can generate the interface afterwards upon needing"
AP3	"Yes, this can generate perfomance issues in the system"
AP4	"Yes, to improve maintainability"
AP6	"No, because its not a big problem"
AP9	"No, it is not a big problem for small projects where it highly improbable changing the farework"

Table A.13: I10 Perception over fixing DI anti-patterns in preliminary survey

DI Anti-Pattern	**Respondent Perception**
AP1	"It depends on how complex is the example1 object. If it is a heavy object I would use AP1 solution. Other point that would influence in my decision is how often the example1 object is needed in my class code."
AP2	"Yes. Because it makes codification more formal avoiding refactoring. The concrete class has to obey the interface structure, so my B_Without_Concrete class doesn't need to know about concrete implementation."
AP3	"Yes, but for me it is not just a DI problem. It is more about methods with so many responsibility that makes the code more complex and unreadable. Other point that would make me fix AP3 is that calling Provider methods instead of just have all complexity in his life-cycle make me feel I have more control of my code."
AP4	"Yes, but it depends on how can you group the responsibility of injected classes. Worse than have so many injection in one single class is to create so many classes with the same responsibilities. I agree that it should be implemented in most of cases but It can be weird somethings."
AP6	"You don't need to use the ServiceLocator to get the IDataSource each time you have to use it. You can create a global object and you can control it to call just once. In this case, DI doesn't bring such a huge difference for me."
AP9	"In this specific case, we are talking about a huge framework that brings lots of benefits to the application (Spring). So, for me it worth even if it brings higher coupling to framework specifics."

Table A.14: I11 Perception over fixing DI anti-patterns in preliminary survey

DI Anti-Pattern	**Respondent Perception**
AP5	"Yes. I will apply the suggested solution"
AP7	"Yes. I will apply the suggested solution"
AP8	"Yes. I will apply the suggested solution"
AP10	"No. This kind of occurrence may be a problem or not, can vary according with arctheture"
AP11	"Yes. I will apply the suggested solution"
AP12	"Yes. I will apply the suggested solution"

Table A.15: I12 Perception over fixing DI anti-patterns in preliminary survey

DI Anti-Pattern	**Respondent Perception**
AP5	"Yes, because if its not used we dont need to maintain."
AP7	"Maybe. In this case i would've used a service locator, or at least incapsulate the dependency in a method of the provider object."
AP8	"I would fix this, but I don't think that the suggested solution is right, because it depends in many factors. We have to know why the developer thinks that the dependency is needed by other objects to provide a good solution for him."
AP10	"Y"
AP11	"I don't understood why the solution is right, both of them look wrong."
AP12	"I know that having two ways of injecting an object is wrong, however, this set method is wrong too. If you wannna create an object I would to this with spring injection or constructor injection, so it would only be created at the start of the object, and not with a set method."

Table A.16: I13 Perception over fixing DI anti-patterns in preliminary survey

DI Anti-Pattern	**Respondent Perception**
AP5	"Yes"
AP7	"No. This may be used to improve performance, only loading some dependency when it is absolutely needed. The extra provider will add unnecessary complexity, only hiding the original intention."
AP8	"No. This situation is necessary when integrating third-party code that will not have access to the DI container. It also allows a function or method to receive different implementations of a parameter."
AP10	"Yes"
AP11	"Yes"
AP12	"Yes"

Table A.17: I14 Perception over fixing DI anti-patterns in preliminary survey

DI Anti-Pattern	**Respondent Perception**
AP5	"Yes, the class should be as clean as possible."
AP7	"Yes, as we are using DI this can be handled better."
AP8	"Yes, the class who has the dependency is not the one being injected."
AP10	"Yes, this increases complexity with no benefit. Probably this results from bad architecture."
AP11	"Yes, looks like bad architecture."
AP12	"Yes, because it increases complexity "

Table A.18: I15 Perception over fixing DI anti-patterns in preliminary survey

DI Anti-Pattern	**Respondent Perception**
AP5	"Yes, because the IDE probably will provide some warnings about the unused injection."
AP7	"I believe that I wouldn't write the EP7 code since it would give to the framework (e.g., spring) to manage it."
AP8	"Yes, refactoring decreases the coupling."
AP10	"Because it is quite obvious that is wrong. In addition, frameworks documentation do not provides this example."
AP11	"Because it violates one of the principles of OOP"
AP12	"For me, no make sense repeat the same injection twice."

B
Updated catalog of DI anti-patterns

Catalog of Dependency Injection Anti-Patterns

Rodrigo Laigner

Coupling is a quality attribute of a module in an application. As higher the level of coupling to another modules of the system, the likelihood of increased efforts when it comes to introduce change is expected in this module [1]. A particular programming mechanism that is employed to decrease coupling levels in an application is Dependency injection (DI), a mechanism for improving software modularity.

Jim Coplin [2] asserts that the study of anti-patterns is an important research activity, on which only showing the presence of 'good' patterns in a successful system is not enough. Thus, this material shows the result of 2 years research [3] [4] on cataloging instances of dependency injection anti-patterns. The instances of the catalog were observed and conjectured based on the experience of the author while maintaining software in industry settings. Also, the catalog evolved through discussions with researchers.

Each anti-pattern provides the following structural elements: a name, description, context where it is applied, drawbacks observed, pattern of occurrence, and resolution. Furthermore, we classify the 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 addition, each anti-pattern shows the pattern of occurrence and a resolution separated by a dashed line.

Intransigent injection

Description: Intransigent injection concerns dependencies that are not needed on construction time, however, they are decidedly provided by the DI container on construction time.

Figure 1: Intransigent injection

```
public class A {
   @Inject
   private IExampleInterface0 example0;
   @Inject
   private IExampleInterface1 example1;
   public A() {
       example0.doSomething();
   }
   public void foo() { /* omitted code */ }
   public void bar() {
       example1.doSomething();
       /* omitted code */
   }
}
-----------------------------------------------------
public class A_Without_Intransigent_Injection {
   @Inject
   private IExampleInterface0 example0;
   @Inject
   private Provider<IExampleInterface1>
       example1Provider;
   public A() {
       example0.doSomething();
   }
   public void foo() { /* omitted code */ }
   public void foo() {
       IExampleInterface1 example1 =
           example1Provider.get();
       example1.doSomething();
       /* omitted code */
   }
}
```

Figure 2: Concrete class injection

```
public class B {
   @Inject
   private ConcreteClassExample example;
   private void foo(){
      example.doSomething();
      // code omitted for brevity
   }
}
-------------------------------------------------
public class ConcreteClassExample
    implements IExampleInterface {
   @Override
   public void doSomething() {
      // code omitted for brevity
   }
}

public class B_Without_Concrete {
   @Inject
   private IExampleInterface example;
   private void foo(){
      example.doSomething();
      // code omitted for brevity
   }
}
```

Context: Objects that do not need to be instantiated on construction time and might not require additional effort or latency impediments if instantiated in a later time.

Drawbacks: This scheme introduces additional workload and memory consumption on construction time. It is a worse scenario if the instance provided is not a lightweight object, impacting on performance. Thus, this anti-pattern is categorized as a performance problem.

Pattern of occurrence: Figure 1 shows that the injected attribute *example1* is not used in construction time, thus, the process of injecting a given instance in this attribute might require additional workload to DI container.

Resolution: The resolution concerns the use of a *Provider*, an interface type defined by JSR-330 that allows for obtaining a given dependence when necessary. Thus, the resolution provided in Figure 1 concerns relying on a *Provider* to obtain from the DI container the given instance when its use is requested (see *example1*).

Concrete class injection

Description: Concrete class injection concerns a dependence requested via dependency injection on which the element type of the dependence is a concrete class.

Context: This anti-pattern is not applied on small systems (e.g., $<$ 50K). Once small systems present a smaller number of components in comparison with larger systems (e.g., $>$ 100K), they often do not benefit from an interface-oriented design.

Drawbacks: As a design problem, this anti-pattern yields the following negative consequences: (i) violation of IoC principle, once the class requesting its dependence acknowledges an implementation detail (i.e., the concrete class); (ii) Less flexibility on testing, once a mock object would need to be an inherited class of the given concrete class in order to modify desired behavior; (iii) According to Gamma et al. [1], coupling to a concrete class can increase maintenance efforts.

Pattern of occurrence: A class relies on a concrete class injection to request a dependence. Figure 2 exhibits the class *B* depending on the concrete class *ConcreteExample*, configuring a high coupling.

Resolution: Gamma et al. [1] advocates for programming to an interface, which is a natural solution to this anti-pattern. The resolution shows a code transformation, on which an interface (see *IExampleInterface*) is created so that the class *ConcreteExample* implements it.

Long producer method

Description: Long producer method concerns a method that performs activities that are out of the scope of providing a dependence, which must be its main objective. This context defines this anti-pattern as a design problem.

Context: This anti-pattern is only applied to *Producer* methods that do tasks outside of the scope of providing a dependence.

Drawbacks: This anti-pattern undermines the ability of the software to adapt to change when requirements change.

Pattern of occurrence: Figure 3 shows a high complex method that should be simple, once the main concern of a Producer method (see *@Produces* annotation) is to a provide a given dependency. The DI container, when it identifies the existence of a Producer method for a given type, transfer the responsibility for dependence provision to the Provider method.

Resolution: The code for providing the dependence and for doing tasks related to business logic should be departed. We provide an excerpt of a *Producer* method without high cyclomatic complexity and fewer responsibilities.

Figure 3: Long producer method

```
public class C {
    // omitted code
    @Produces
    public ProducedBean generateReport(){
        Set<Integer> selectedBacklogIds =
            this.getSelectedBacklogs();
        if(selectedBacklogIds == null) {
            Collection<Product> products = new
                ArrayList<Product>();
            productBusiness.
                storeAllTimeSheets(products);
            for (Product product: products) {
                selectedBacklogIds.
                    add(product.getId());
            }
            return Action.PROCESS;
        }
        // omitted code
        Workbook wb = this.timesheetExportBusiness.
            generateTimesheet(this,
                selectedBacklogIds, startDate,
                endDate, timeZone, userIds);
        this.exportableReport = new
            ByteArrayOutputStream();
        try {
            wb.write(this.exportableReport);
        } catch (IOException e) {
            return Action.ERROR;
        }
        return Action.SUCCESS;
    }
}
```

```
public class C_Without_Long_Producer {
    // omitted code
    @Produces
    public ProducedBean generateReport(){
        Set<Integer> selectedBacklogIds =
            this.getSelectedBacklogs();
        if(selectedBacklogIds == null) {
            return Action.PROCESS;
        } else
            if(this.exportableReportIsWritten()){
            return Action.ERROR;
        }
        return Action.SUCCESS;
    }
}
```

Figure 4: God DI class

```
public class D {
   @Inject private IExample1 one;
   @Inject private IExample2 two;
   @Inject private IExample3 three;
   @Inject private IExample4 four;
   @Inject private IExample5 five;
   // other several dependencies injected
   @Inject private IExampleN n;
   void methodOne() { /* reference to several
       dependencies */ }
   void methodTwo() { /* reference to several
       dependencies */ }
   // other several methods
   void methodThree() { /* reference to several
       dependencies */ }
}
```

```
public class D_Part_1 {
   @Inject private IExample1 one;
   @Inject private IExample2 two;
   @Inject private IExample3 three;
   void methodOne() { /* code omitted */ }
}
public class D_Part_2 {
   @Inject private D_Part_1 dPartOne;
   @Inject private IExample4 four;
   @Inject private IExample5 five;
   @Inject private IExample6 six;
   void methodTwo() { /* code omitted */ }
}
public class D_Part_3 {
   @Inject private D_Part_2 dPartTwo;
   @Inject private IExample7 seven;
   @Inject private IExample8 eight;
   @Inject private IExample9 nine;
   @Inject private IExampleN n;
   void methodThree() { /* code omitted */ }
}
```

Figure 5: Unused 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_Unused {
   public void foo() {
      // code omitted for brevity
   }
   public void bar() {
      // code omitted for brevity
   }
}
```

God DI class

Description: This anti-pattern concerns the injection of a substantial number of dependencies in a class. This anti-pattern is primarily concerned over injected instances that are often inconsequentially introduced by developers without reasoning over the increased dependence of the class with other components. Thus, it is configured as a design problem.
Context: Along with a substantial number of injected dependencies, the class also shows a multiple set of responsibilities that would be better handled if distributed properly.
Drawbacks: Increased efforts on maintenance tasks operated in the class.
Pattern of occurrence: Figure 4 depicts an excerpt of a class with high level of complexity, in terms of number of injected element instances, and number of methods.
Resolution: Figure 4 depicts a refactoring that removes the anti-pattern, dividing dependencies and behavior into different classes. The resolution example depicts a code transformation applied to previous class *D*, on which a refactoring type called *Extract Class* [5] was employed three times in order to reduce the complexity of class *D*.

Unused injection

Description: This anti-pattern regards a dependency requested via dependency injection that is actually not used in the class. This way, unused injection is categorized as a performance problem.
Context: Only applied if the unused injection is not used by any inherited class. The problem is more related to current integrated development environments (IDEs) which, once annotated, even the though the attributed is not used, do not warn the developer about the issue.

Drawbacks: 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.

Pattern of occurrence: The Figure 5 shows a class (E) with an injected instance that is not used though any method of the class.

Resolution: The solution concerns removing the unused injection element.

Figure 6: Static dependence provider

```
// ServiceLocator import ommitted
public class E {
    @Inject private Parser parser;
    public void execute(List<String> files) throws
        Exception {
      IDataSourcedataSource dataSource =
          (IDataSource)
          ServiceLocator.getInstance()
              .getBeanInstance("IDataSource");

      for(String file : files){
           Object parsedObject =
               parser.parse(file);
           dataSource.insert( key, parsedObject );
      }
    }
}
-----------------------------------------------------
public class ProjectConfigBeans {
   @Bean
   public IDataSource provideDataSource(){
      // logic for creating an instance of
          IDataSource
   }
}

public class E_Without_Service_Locator {
    @Inject private Parser parser;
    @Inject private IDataSource dataSource;
    public void execute(List<String> files) throws
        Exception {
      for(String file : files){
           Object parsedObject =
               parser.parse(file);
           dataSource.insert( key, parsedObject );
      }
    }
}
```

Static dependence provider

Description: Static dependence providers are related to Fabrics and Service Locators. The first refers to a class that has the objective to provide a given concrete implementation, not being a *Provider* class. On the other side, Service Locator pattern also applies to this context, since it is a class that has the responsibility for serving all dependencies that might be required on run time. Both classes of problem concerns architectural problems, since both violate DIP and IoC principle.
Context: The use of static dependence providers in projects that employ a DI framework where the dependence provision is not related to fulfill a third-party component or a class outside the container.
Drawbacks: It incurs a high coupling to a fabric class in the source code. In case of Service Locator, the dependence on this pattern is even worse due to its widespread usage in the project. Indeed, inversion of control is not achieved in both cases.
Pattern of occurrence: Figure 6 exhibits the class (E) with a dependence provision made by a service locator. In other words, rather than relying on the DI container for injecting an instance of *IDataSource* type on *dataSource* attribute, the code relies on a service locator.
Resolution: Figure 6 enforces the use of DI container for dependency injection at run time by relying on a *Producer* method in order to provide an instance of *IDataSource*. Particularly, the resolution example above shows a code transformation, in which the logic for creating an instance of *IDataSource* is modularized within a Producer method. This way, the class *E_ Without_ Service_ Locator* is not coupled to a service locator class anymore.

Direct container call

Description: Direct container calls can provide a concrete implementation at any point of the system. The nature of this anti-pattern is similar to using a static fabric or a Service Locator. Once DI is chosen as an architectural standard for the project, employing container call for dependence resolution conveys an architectural violation.
Context: The use of static dependence providers in projects that employ a DI framework where the dependence provision is not related to fulfill a third-party component or a class outside the container. In cases where dependencies must be dynamically resolved at runtime with the support of the DI container, this anti-pattern is not applied.
Drawbacks: High coupling to framework specifics, since it relies directly on the framework to provide the depen-

Figure 7: 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 private Parser parser;
    @Inject private IDataSource dataSource;
    public void execute(List<String> files) {
      for(String file : files){
            Object parsedObject =
                parser.parse(file);
            dataSource.insert( key, parsedObject );
      }
    }
}
```

Figure 8: Open window injection

```
public class F {
    @Inject private Parser parser;
    @Inject private IExampleInterface one;
    public Parser getParser() {
        return parser;
    }
    public void execute(List<String> files) throws
        Exception {
      for(String file : files){
            Object parsedObject =
                parser.parse(file);
            one.doSomethingWithParsed(
                parser, parsedObject);
      }
    }
}
-----------------------------------------------------
public class F_Without_Passing {
    @Inject private Parser parser;
    @Inject private IExampleInterface one;
    public void execute(List<String> files) throws
        Exception {
      for(String file : files){
            Object parsedObject =
                parser.parse(file);
            one.doSomethingWithParsed(parsedObject);
      }
    }
}

public class ConcreteExample
    implements IExampleInterface {
   @Inject private Parser parser;
   @Override
   public void doSomethingWithParsed(Object
       parsedObject) {
      // omitted code
   }
}
```

Figure 9: Framework coupling

```
public class J {
    @Autowired private Parser parser;
    @Autowired private 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 private Parser parser;
    @Inject private IDataSource dataSource;
    public void execute(List<String> files) {
      for(String file : files){
            Object parsedObject =
                parser.parse(file);
            dataSource.insert( key, parsedObject );
      }
    }
}
```

dency. In addition, inversion of control principle is not achieved in this context.

Pattern of occurrence: The Figure 7 shows the class (*E*) with a dependence provision made by a direct container call. In other words, rather than relying on the DI container for injecting an instance of *IDataSource* type on *dataSource* attribute, the code relies on a direct container call.

Resolution: The resolution concerns removing the element that performs a container call and enforcing the use of a DI container for dependence provision.

Open window injection

Description: This anti-pattern is applied when an injected instance is not used, but passed as parameter to another class method or opened for external accessing (e.g. by get method or public/protected access modifier).

Context: In case where an injected object needs to be passed to a component (or third-party solution) that lives outside the container, the anti-pattern is a solution. Thus, the anti-pattern does not apply to this context.

Drawbacks: Two negative consequences are observed. In the first case, it adds a useless intermediary element between the class that needs a given concrete implementation and the DI container. On the second case, it opens a door for external modification, which could possibly yield the introduction of bugs.

Pattern of occurrence: Figure 8 show an example of occurrence, on which the inject element *parser* is passed as parameter to another method.

Resolution: The resolution depicts a code transformation where the injected element *parser* is not passed as parameter to method *doSomething* of the interface *IExampleInterface* anymore. The concrete implementation of *IExampleInterface* is now responsible for defining its dependence on an instance of *Parser* type.

Figure 10: Open door injection

```
public class H {
   @Inject private Parser parser;
   public void setParser(Object parser) {
      this.parser = parser;
   }
    // code omitted for brevity
}
-----------------------------------------------------
public class H_Without_Anti_Pattern {
    @Inject private Parser parser;
    // code omitted for brevity
}
```

Framework coupling

Description: It refers to elements on source code that are dependent on a given framework implementation. As the name of the anti-pattern expose, it can be represented as annotations or method calls to framework configuration classes along the source code. We categorize this anti-pattern as part of standardization category.
Context: Avoiding this anti-pattern is better applied in the context of new software projects. In legacy systems, the complexity entailed by removing this anti-pattern may not be worthy the effort.
Drawbacks: In the context of Java, which presents a specification for DI, a framework specific annotation, for example, incurs in high coupling to the framework. 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.
Pattern of occurrence: Figure 9 depicts a class that employs Spring framework *@Autowired* annotation.
Resolution: A suitable option for removing coupling from a given DI framework is relying on the adoption of annotations presented in the specification.

Open door injection

Description: This anti-pattern is applied when an inject request is provided by a DI container, however, the instance requested is open for modification by an external element. It usually happens when the developer lacks appropriate knowledge about DI.
Context: Any context.
Drawbacks: Open door injection can configure a hard to follow traceability, hindering program comprehension. Also, bugs are another possibility, since concrete implementation is open to chance by an external class.

Figure 11: Multiple assigned injection

```
class ExampleBusiness
          extends GenericBusinessImpl{
    private IDAOexampleDAO exampleDAO;
    @Inject
    public void setExampleDAO(ExampleDAO
        exampleDAO) {
        this.genericDAO = exampleDAO;
        this.exampleDAO = exampleDAO;
    }
}
-----------------------------------------------------
abstract class GenericBusinessImpl {
    abstract IDAO getGenericDAO();
}

class ExampleBusiness
          extends GenericBusinessImpl{
    private IDAOexampleDAO exampleDAO;
    @Inject
    public void setExampleDAO(ExampleDAO
        exampleDAO){
        this.exampleDAO = exampleDAO;
    }
    @Override
    protected IDAO getGenericDAO() {
        return this.exampleDAO;
    }
}
```

Pattern of occurrence: Figure 10 depicts the presence of a public set method that allows changing of the injected instance of *parser* in runtime. In details, the example depicts a public set method (*setParser*), which allows for modification of the instance of an injected element (*parser*) by an external class.
Resolution: The resolution concerns the removal of the element on source code (e.g., public set method) that enables changing injected element.

Multiple assigned injection

Description: This anti-pattern occurs when the reference to an injected instance is spread among multiple attributes.
Context: This anti-pattern is not applied in cases where the class receiving an injection also assigns it to an element of its superclass (which is not managed by the DI container), thus, forcing the assignment to multiple attributes.
Drawbacks: This anti-pattern is correlated to *Open door injection*, since it opens a gap for an undesirable modification of the injected object at run time.
Pattern of occurrence: Figure 11 provides an example

Figure 12: Multiple forms of injection

```
class ExampleBusiness
          extends GenericBusinessImpl {
    @Inject private IDAOexampleDAO exampleDAO;
    @Inject
    public void setExampleDAO(ExampleDAO
        exampleDAO) {
        this.exampleDAO = exampleDAO;
    }
}
-----------------------------------------------------
class ExampleBusiness
          extends GenericBusinessImpl{
    private IDAOexampleDAO exampleDAO;
    @Inject
    public void setExampleDAO(ExampleDAO
        exampleDAO) {
        this.exampleDAO = exampleDAO;
    }
}
```

of occurrence, where the assignment of an injected instance of *ExampleDAO* to an attribute of a parent class (*GenericBusinessImpl*).

Resolution: In the case of injection instance being assigned to an attribute of super-class, a better approach would be overriding an abstract method. This way, the overridden abstract method would provide the instance injected, not incurring on reference duplication. Thus, the resolution shown in Figure 11 depicts a code transformation that removes the assignment of an injected instance to an attribute presented in a parent class. The removal makes room for an abstract method in the parent class, which still allow the reference to the original injected instance.

Multiple forms of injection

Description: This anti-pattern refers to the use of multiple forms of injection to a given element (e.g., set method and by constructor).

Context:

Drawbacks: It leads to misunderstanding of injection process for less experienced developers. In some frameworks, it can lead to duplicate injection work.

Pattern of occurrence: Figure 12 provides an excerpt of the occurrence of this anti-pattern, where there are two forms of injection for the same element (*exampleDAO*). The first is an attribute injection and the second is a constructor injection.

Resolution: The resolution depicts only one form of injection (constructor) for the element *exampleDAO*.

References

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

[2] James Coplien and Douglas Schmidt. *Pattern Languages of Program Design*. Leanpub, 1995.

[3] Rodrigo Laigner, Marcos Kalinowski, Luiz Carvalho, Diogo S. Mendonça, and Alessandro Garcia. Towards a catalog of java dependency injection anti-patterns. In *Proceedings of the XXXIII Brazilian Symposium on Software Engineering, SBES 2019, Salvador, Brazil, September 23-27*, pages 104–113, 2019. doi: 10.1145/3350768.3350771.

[4] Rodrigo Laigner. Cataloging dependency injection anti-patterns in software systems, 2020. PUC-Rio. Master's dissertation.

[5] Diego Cedrim. *Understanding and Improving Batch Refactoring in Software Systems*. PhD thesis, PUC-Rio, 2018.

This document was created on and last updated on February 19, 2020. Source files are at https://www.overleaf.com/read/xmyythnqhymz.