computers

Article

Measuring Impact of Dependency Injection on Software Maintainability

Peter Sun, Dae-Kyoo Kim *, Hua Ming and Lunjin Lu

Department of Computer Science and Engineering, Oakland University, Rochester, MI 48309, USA
* Correspondence: kim2@oakland.edu; Tel.: +1-248-370-2863

Abstract: Dependency injection (DI) is generally known to improve maintainability by keeping application classes separate from the library. Particularly within the Java environment, there are many applications using the principles of DI with the aim to improve maintainability. There exists some work that provides an inference on the impact of DI on maintainability, but no conclusive evidence is provided. The fact that there are no publicly available tools for quantifying DI makes such evidence more difficult to be produced. In this paper, we propose two novel metrics, dependency injection-weighted afferent couplings (DCE) and dependency injection-weighted coupling between objects (DCBO), to measure the proportion of DI in a project based on weighted couplings. We describe how DCBO can serve as a more meaningful metric in assessing maintainability when DI is also considered. The metric is implemented in the CKJM-Analyzer, an extension of the CKJM tool to perform static analysis on DI detection. We discuss the algorithmic approach behind the static analysis and prove the soundness of the tool using a set of open-source Java projects.

Keywords: dependency injection; coupling; maintainability

check for updates

Citation: Sun, P.; Kim, D.-K.; Ming, H.; Lu, L. Measuring Impact of Dependency Injection on Software Maintainability. *Computers* **2022**, *11*, 141. https://doi.org/10.3390/computers11090141

Academic Editor: Paolo Bellavista

Received: 4 August 2022
Accepted: 14 September 2022
Published: 18 September 2022

1. Introduction

Software development has grown increasingly dependent on external libraries that are built by other companies. While it provides convenience in development, it significantly increases the cost of software maintenance, especially for the changes that involve the dependency with libraries [1]. The use of external libraries also requires significant overheads, such as extra code to be imported and compiled, resulting in performance bottlenecks. Additionally, external libraries can introduce security vulnerabilities unknown to the developers using those libraries. These issues are exacerbated if the external libraries are open source and maintained by the community, resulting in inconsistent updates and lack of maintenance from the original developers.

Component frameworks (e.g., Spring [2]) help mitigate development cost. A key feature of component frameworks for object-oriented programming (OOP) is dependency injection (DI). DI is a pattern of sending ("injecting") necessary fields ("dependencies") into an object, instead of requiring the object to initialize those fields itself. Existing literature [3–6] suggests that the use of DI can help improve the maintainability of software systems. On the other hand, there are also warnings against using DI due to possible negative effects [7,8].

A software quality metric often used to measure maintainability is coupling between objects (CBO) [9]. CBO is the total number of couplings present within the software system, or the sum of the system's afferent couplings (CA) and efferent couplings (CE). CA counts how many other classes use the class being analyzed, while CE counts how many classes the class being analyzed uses. Therefore, when a coupling exists between two objects, the object being depended on will increase its CA value by 1, and the object depending on the other object will increase its CE value by 1, generating an overall CBO value of 2. Generally, higher CBO yields lower maintainability because of the increased complexity of the system.

In this paper, we present two novel metrics, dependency injection-weighted afferent couplings (DCE) and dependency injection-weighted coupling between objects (DCBO), to analyze DI and assess the impact of DI on software maintainability and its tool support, CKJM-Analyzer [10], which is an extension of the CKJM tool [9]. DCE weighs each efferent coupling depending on whether it is soft-coupled (e.g., with DI) or hard-coupled (e.g., with the *new* keyword, or with using an object generator that requires parameter information from the user). DCBO utilizes DCE in place of CE as a weighted metric of overall coupling. CKJM-Analyzer is a cross-platform command line interface (CLI) with two primary goals—(i) develop a standard operating procedure to iteratively analyze Java projects for CKJM metrics and (ii) count the instances of the DI pattern in Java projects to determine the DI proportion. We validate the metric and tool with a set of open-source Java projects.

The remainder of the paper is organized as follows. Section 2 presents a background on DI. Section 3 describes DCBO and its algorithmic approach implemented in the CKJM-Analyzer tool. Section 4 describes the evaluation of CKJM-Analyzer on experimentally generated projects and open-source projects. Section 5 discusses the results with regards to the impact of DI in maintainability, the effect of DCBO on coupling analysis, the limitations and potential future work. Section 6 gives an overview of the related work on the effect of DI in software systems, as well as work in measuring coupling weight. Section 7 concludes the paper with discussion on future research work.

2. Dependency Injection

DI is a specific form of the *dependency inversion principle* [4], which is a pattern that suggests that higher-level objects should dictate most of the complex logic in the system and also create dependencies for lower-level objects to use. DI is a subset of this principle because it highlights how lower-level objects should rely on higher-level objects for their dependencies. DI is a design pattern to improve the maintainability of software systems by reducing developer effort in adding coupling through injecting dependencies in classes using an external injector which is a class object or file (e.g, an XML-based configuration file in Spring Framework [2]). As coupling is reduced, consequently the complexity of classes is also diminished. DI also makes it easier to pinpoint dependency-related errors as dependency injection is localized in one place (viz. the injector).

A dependency is typically injected in four ways [4]: (i) via constructor parameters, which is known as *constructor no default* (CND); (ii) via method parameters, which is known as *method no default* (MND); (iii) via constructor parameters or a default object (using the `new` command), which is known as *constructor with default* (CWD); and (iv) via method parameters or via a default object, which is known as *method with default* (MWD). Consider the code snippets below. The `Dog` class has no dependency injection, the `DogPenCND` class implements CND, the `DogPenMND` class implements MND, the `DogPenCWD` class implements CWD, and the `DogPenMWD` class implements MWD. Note that CWD extends on CND functionality, and MWD extends on MND functionality (not shown in the code snippet for brevity).

```
public class Dog {
Dog() {}
}
```

```
public class DogPenCND {
Dog dog;
DogPenCND(Dog dog) {
this.dog = dog;
}
}
```

```
public class DogPenMND {
Dog dog;
void AddDog(Dog dog) {
this.dog = dog;
}
}
```

```
public class DogPenCWD {
Dog dog;
```

```
public class DogPenMWD {
Dog dog;
DogPenCWD() {                     void AddDog() {
this.dog = new Dog();             this.dog = new Dog();
}                                 }
}                                 }
```

Typically, the implementation of services is specified in the injector, which is often used as a clue for the use of DI. When a change needs to be made in the service, it is done through the injector without changing the client. In this way, the client remains unchanged, and thus, the code becomes more flexible and reusable. Consider the code snippet below. The `DogPenGeneratorCND` class acts as the injector, injecting the `Dog` class into each `DogPenCND` object. In this way, each different `DogPenCND` object does not need to generate its own `Dog` dependency. This is particularly helpful when the same dependency is injected in multiple different objects. The CND and MND structures follow this benefit.

```
public class DogPenGeneratorCND() {
Dog dog = new Dog("Dog1");
DogPenCND pen1 = new DogPenCND(dog);
DogPenCND pen2 = new DogPenCND(dog);
DogPenCND pen3 = new DogPenCND(dog);
}
```

In contrast, the CWD and MWD structures do not entirely follow the injector benefit. Consider the code snippet below. If developers use the "default" constructor or method available in the CWD/MWD structures, it requires the object itself to generate its own dependency object.

```
public class DogPenGeneratorCWD() {
DogPenCWD pen1 = new DogPenCWD();
DogPenCWD pen2 = new DogPenCWD();
DogPenCWD pen3 = new DogPenCWD();
}
```

In our work, we also analyze a fifth way of injecting dependencies using beans provided by Spring Framework [2]. Beans represent concrete classes implementing an interface and are configured in an XML file. Any class can inject those beans directly into its constructor or setter functions. Consider the example XML configuration snippet below, where a concrete class "ConcreteClass" implements "AbstractClass" is wired up to an identifier "object".

```
<?xml version="1.0" encoding="UTF-8"?>
<!--spring.xml file-->
<beans ...>
<bean id="object" class="path.to.ConcreteClass"/>
</beans>
```

Consider the code snippet below. Any application can inject that bean using the identifier "object", simplifying object injection. Additionally, developers can create other concrete classes implementing the "AbstractClass" interface and switch out the original "ConcreteClass" through the XML file alone, avoiding unnecessary compilation and code changes.

```
ApplicationContext appContext = new ClassPathXmlApplicationContext
("spring.xml");
AbstractClass obj = (AbstractClass) appContext.getBean("object");
```

DI comes with a few technical hindrances. Firstly, it requires all the dependencies to be resolved before compilation if the compiler is not configured to recognize injected dependencies. That is, the compiler cannot recognize the presence of injected dependencies unless it is configured. Secondly, the frameworks built upon DI are often implemented with reflection or dynamic programming, which can hinder the use of IDE automation, such as reference finding, call hierarchy displaying, and safe refactoring [11].

3. Measuring the Impact of DI on Maintainability

In this section, we describe the proposed approach for measuring the impact of DI on software maintainability. The following are the research questions we seek to answer in this work.

- *Does change in DI proportion impact CBO?* We believe this is an important question to address because previous research has argued that DI should reduce coupling, and therefore reduce CBO [3]. It should follow that an inverse relationship between DI and CBO (increase in DI will decrease CBO) can be quantified. We use experimental analysis to determine whether that inverse relationship exists.
- *How effective is CBO in quantifying maintainability?* In our paper, we utilize CBO to assess maintainability. We also use the normalization techniques proposed by Okike and Osofisan [12] to allow for more informative analysis. We critique whether normalized CBO is useful in analyzing maintainability in the context of DI.
- *How should DI inform coupling in the software system?* This is the motivation behind our proposal for the new metrics, DCBO and DCE. DCBO and DCE consider the additional developer "effort" required when DI is not used in the software system, and we argue that it would decrease overall maintainability. We developed the open-source CKJM-Analyzer [10] tool that utilizes DCBO and DCE to determine DI proportion. We perform an analysis using the tool and discuss the results.

In particular, we hope to expand on the work conducted by Razina and Janzen [3], who focused on Spring Framework [2] projects to assess whether DI improves maintainability. We aim to provide an open-source software solution in counting instances of the DI pattern within projects, and propose a new metric in accordance with an updated DI definition.

3.1. Dependency Injection Weighted-Coupling between Objects

Yang et al. [4] acknowledged a reduction in reusability with respect to the CWD and MWD patterns because of the default object, but did not believe that was significant enough to remove the CWD and MWD definitions. We argue that only CND and MND conform to the DI pattern (cf. Section 2) and with respect to CWD and MWD, the default object significantly hinders maintainability if the dependency requires any constructor parameters. In other words, DI via parameter injection (i.e., CND/MND) requires only one change where the dependency is generated and injected into various classes, whereas DI via default object injection (i.e., CWD/MWD) requires N number of changes, where N is the number of classes that depend on that object. We deem constructor/method XML Java bean injection to follow the CND and MND patterns because the bean will construct the concrete class.

Using CND, MND, and XML DI patterns, we propose dependency injection-weighted afferent couplings (DCE) defined in Equation (1), where CE is the total efferent couplings of the class, and CDI is the total number of efferent couplings injected via DI.

$$DCE = CE - CDI \tag{1}$$

We subtract the number of efferent couplings injected via DI from the total efferent couplings because we are aiming to quantify the different in developer effort when using DI compared to other injection means. Utilizing CND, MND and XML injection patterns, proper dependency injection will avoid extensive file changes. Any number of concrete classes that implement the same interface can be easily swapped by only changing the XML configuration file. Additionally, DI can instantiate classes that have constructors requiring parameters without having the class utilizing the injected class to know what kind of parameters to generate. These advantages become more apparent with projects of a larger size or with a bigger inheritance tree since the same dependency can be injected to any depth so long as the same functionality is expected across the tree.

Next, we propose dependency injection-weighted coupling between objects ($DCBO$) defined in Equation (2), where CA is the afferent couplings of the class and DCE is Equation (1).

$$DCBO = CA + DCE \tag{2}$$

We specifically quantify DI by counting instances of the DI pattern (CND, MND, and XML). We utilized the abstract syntax tree (AST) created by the CKJM Extended tool [9] and created a fork version, CKJM-DI [13], to be used in the overall DI detection algorithm within CKJM-Analyzer [10]. CKJM-Analyzer was developed to address the need for an open-source solution that can quantify DI in Java projects. At a high level, it is a .NET command line interface (CLI) [10] that wraps the CKJM-DI [13] tool and provides additional functionality to count instances of the DI pattern and quantify DCBO. In the CKJM-Analyzer tool, we quantify DCBO for every class Equation (2), and finally return the mean for every project in the final report. We acknowledge that while the weights assigned in DCBO should sufficiently cover the CND/MND/XML DI definitions, it does not account for whether that injected object will enforce further coupling. For example, the injected object could call an internal function that requires multiple parameters, forcing the user to perform changes wherever that function is called, even though that object is injected using DI.

3.2. Dependency Injection Detection Algorithm

In this subsection, we discuss the specific algorithm implemented to detect DI. It identifies DI patterns analogous to CND, MND, and XML patterns, but does not consider the default object injection as a DI pattern (viz. CWD and MWD). Algorithm 1 shows the pseudocode for the algorithmic implementation to detect DI. $GetClassNames()$ returns a list of all class names in the project. $GetXmlInterfaces$ returns all Java beans wired in the XML configuration file. The number of DI parameters $diParams$ is quantified by finding the intersection of $paramTypes$ and $classNames$, which generates a list of distinct, non-primitive types (class files injected as parameters), and then finding the union of that intersection with $xmlParams$, a list of all classes injected via XML injection using the $IntersectIfUsingSpring()$ function. DI is evaluated as a proportion of efferent couplings injected via DI over total efferent couplings. We deem the lowest DI proportion to be 0 (0%) and the highest DI proportion to be 1 (100%).

3.3. Normalized CBO and Maintainability

Maintainability is defined as *"the ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment"—IEEE.* There have been many models proposed to quantify maintainability (e.g., [14–16]), but difficulties remain. For example, El Emam et al. [17] found that controlling for class size invalidated previously significant studies using CKJM metrics as a means to identify fault proneness, with many of those quality metrics also used to quantify maintainability. In our study, we seek to mitigate these issues by controlling for size, and then normalizing CBO and DCBO. We normalize CBO as NCBO using the formula proposed by Okike and Osofisan [12], $CM = 1 - \frac{1}{1+IS}$, where CM is the module complexity (normalized CBO), and IS is the coupling complexity (CBO). We normalize DCBO as NDCBO using the same formula. We use the normalized CBO and DCBO metrics as an indicator of the system's overall maintainability.

Algorithm 1 Detecting instances of DI patterns.

```
$project \leftarrow GetProjectPath()$
$classNames \leftarrow GetClassNames(project)$
$xmlInterfaces \leftarrow GetXmlInterfaces(project)$
$totalDICount \leftarrow 0$
$totalCe \leftarrow 0$
for each $classFile \in project$ do
    $paramTypes \leftarrow GetParamTypes(classFile)$
    $diParams \leftarrow Intersect(paramTypes, classNames)$        ▷ CND/MND pattern injection
    $ca \leftarrow GetAfferentCouplings(classFile)$
    $ce \leftarrow GetEfferentCouplings(classFile)$
    $xmlParams \leftarrow IntersectIfUsingSpring(ce, xmlInterfaces)$
    $diParams \leftarrow Union(diParams, xmlParams)$        ▷ CND/MND/XML pattern injection
    $dce \leftarrow ce - diParams.Count()$        ▷ Equation (1)
    $dcbo \leftarrow ca + dce$        ▷ Equation (2)
    $totalDICount \leftarrow totalDICount + diParams.Count()$
    $totalCe \leftarrow totalCe + ce$
end for
$di \leftarrow totalDICount/totalCe$
```

4. Evaluation

In this section, we use CKJM-Analyzer to measure the impact of DI patterns using experimentally generated Java projects, which are for controlled experiments, and open-source projects, which are for practical experiments.

4.1. Experimental Projects

We first evaluate CKJM-Analyzer, DCBO and DCE using experimentally generated Java projects for controlled experiments. We created three projects [18], the first without DI, the second with parameter-injected DI, and the third with XML-injected DI. Each project has an *App* class, a *Pen* interface, a concrete class *AnimalPen* implementing *Pen*, an *Animal* interface, and concrete classes *Dog* and *Cat* implementing *Animal*. The instantiation and injection differences are detailed below.

- *No DI*—The *App* class instantiates an *AnimalPen* class, and the *AnimalPen* class instantiates a *Dog* class.
- *Param DI*—The *App* class instantiates a *Dog* class, and then instantiates an *AnimalPen* class while injecting the *Dog* class.
- *XML DI*—The XML file injects an *AnimalPen* class into the *App* class, and the XML file also injects a *Dog* class into the *AnimalPen* class.

Table 1 shows the CSV results from CKJM-Analyzer on the experimental projects. These results show a current limitation of the CBO metric, where it is unable to differentiate efferent couplings based on whether it is initialized within the class or injected via DI. While introducing DI increased CBO because of the Spring Framework [2] class couplings, it allowed for the system to fully utilize interfaces and easily swap out between *Dog* and *Cat* classes within the XML configuration file without having to change code. The impact of DI is also more pronounced when XML injection is used, because all classes can be injected via the configuration, whereas the parameter injection still requires at least one class to initialize various other classes (e.g., *App*).

Table 1. CKJM-Analyzer experimental analysis results.

Project	DI	LOC	CBO	NCBO	DCBO	NDCBO	CA	CE	DCE
No DI	0.00	28	1.00	0.50	1.00	0.50	0.50	0.50	0.50
Param DI	0.25	31	1.33	0.57	1.17	0.54	0.67	0.67	0.50
XML DI	0.33	42	1.33	0.57	1.00	0.50	0.33	1.00	0.67

4.2. Open-Source Projects

We evaluate CKJM-Analyzer using open-source projects compiled from the works of Yang et al. [4] and Tempero et al. [19]. We found the projects in online repositories and converted them all to directories containing class files.

Table 2 shows the CSV results from CKJM-Analyzer on the open-source projects, augmented with the version number of the projects, in alphabetical order. In subsequent analyses, *jchempaint* was removed as an outlier because the project size was nearly 8 times larger than the next largest project.

Table 2. CKJM-Analyzer open-source analysis results.

Project	Version	DI	LOC	CBO	NCBO	DCBO	NDCBO	CA	CE	DCE
advanced-gwt	2.0.8	0.21	33,199	9.17	0.90	7.92	0.89	3.33	5.83	4.59
ant	1.4.1	0.25	54,268	8.99	0.90	7.87	0.89	4.46	4.53	3.41
colt	1.2.0	0.40	101,730	7.73	0.89	6.11	0.86	3.64	4.08	2.47
ine fitjava	1.1	0.21	5178	4.56	0.82	4.07	0.80	2.22	2.34	1.85
ganttproject	1.11.1	0.29	66,120	6.95	0.87	5.94	0.86	3.43	3.51	2.51
hsqldb	1.8.0.2	0.25	145,646	10.96	0.92	9.57	0.91	5.48	5.48	4.09
ireport	0.5.2	0.16	21,378	5.09	0.84	4.68	0.82	2.54	2.54	2.14
jchempaint	2.0.12	0.31	880,275	8.99	0.90	7.58	0.88	4.45	4.54	3.13
jfreechart	1.0.0	0.32	160,004	10.17	0.91	8.34	0.89	4.53	5.64	3.81
jgap	3.6.3	0.34	68,071	8.40	0.89	6.88	0.87	3.97	4.43	2.92
JHotDraw	7.0.9	0.36	19,070	6.71	0.87	5.49	0.85	3.31	3.40	2.18
jhotdraw	6.0.1	0.25	66,843	9.04	0.90	7.89	0.89	4.36	4.67	3.53
log4j	1.2.15	0.35	37,925	6.83	0.87	5.63	0.85	3.41	3.42	2.22
Mars	4.5	0.30	108,126	8.89	0.90	7.57	0.88	4.44	4.44	3.13
pdfbox	2.0.1	0.30	158,833	11.64	0.92	9.81	0.91	5.54	6.09	4.26
picocontainer	1.3	0.33	8931	8.16	0.89	6.83	0.87	4.08	4.08	2.75
poi	2.5.1	0.16	124,773	8.10	0.89	7.44	0.88	4.05	4.05	3.39
wro4j-core	1.5.0	0.18	29,059	7.45	0.88	6.65	0.87	3.11	4.34	3.54

To control for LOC, we visualize the impact of LOC on CBO in Figure 1. The trend line (denoted as a solid line) shows a general increase in CBO as LOC increases. To determine the significance of the increase, we visualize the impact of LOC on NCBO in Figure 2. The trend line shows a general increase in NCBO as LOC increases.

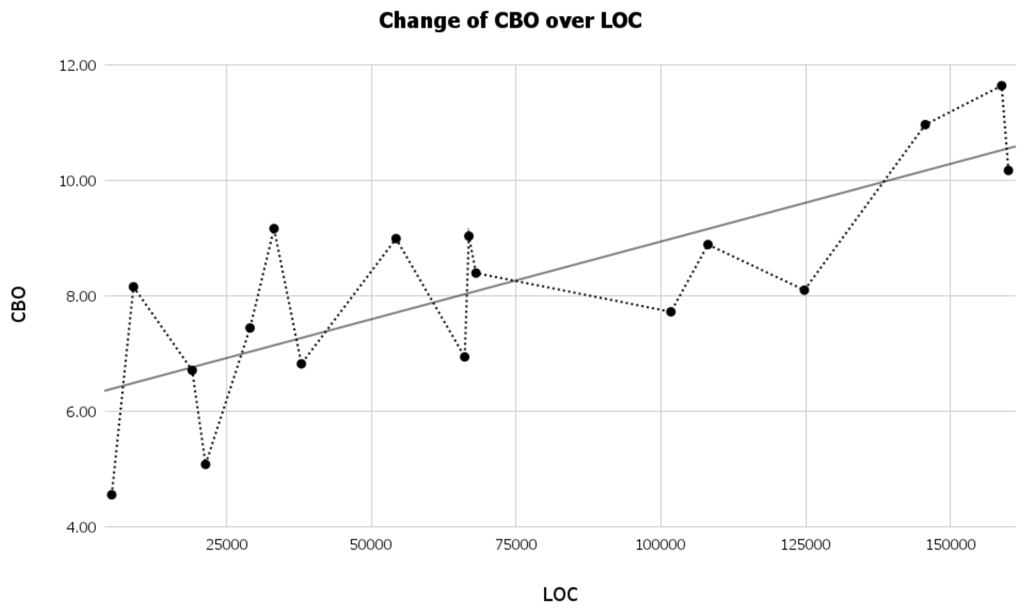

Figure 1. Impact of LOC on CBO.

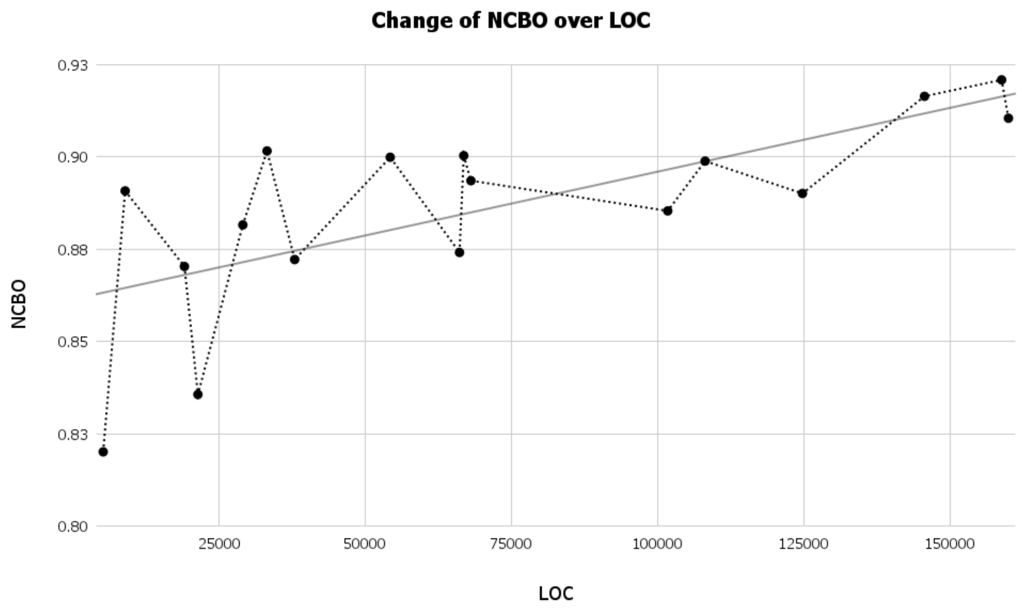

Figure 2. Impact of LOC on NCBO.

We perform Friedman statistical analysis [20] by separating the data set into two equal sets, with the first set containing half the data with lower LOC, and the second set containing half the data with higher LOC. Table 3 shows that comparing a dataset containing lower LOC with a dataset containing higher LOC yields $p \approx 0.01$, a significant change in NCBO. This shows that LOC is a confounding variable, as predicted by El Emam et al. [17].

Table 3. Holm/Shaffer table for $\alpha = 0.05$ on NCBO/LOC full dataset.

Algorithms	p
Low NCBO vs. High NCBO	0.01

In an effort to remove LOC as a confounding variable, we take the four projects with the lowest DI proportion and the four projects with the highest DI proportion and perform

Friedman statistical analysis [20]. Table 4 yields $p = 1.0$, indicating that there is no longer a significant change in NCBO as LOC increases in this subset of data. We use this subset of data to perform additional analysis on DI, CBO and DCBO.

Table 4. Holm/Shaffer table for $\alpha = 0.05$ on NCBO/LOC data subset.

Algorithms	p
Low DI NCBO vs. High DI NCBO	1.0

Table 5 shows the subset of data, sorted by increasing DI.

Table 5. CKJM-Analyzer open-source analysis data subset results.

Project	Version	DI	LOC	CBO	NCBO	DCBO	NDCBO	CA	CE	DCE
ireport	0.5.2	0.16	21,378	5.09	0.84	4.68	0.82	2.54	2.54	2.14
poi	2.5.1	0.16	124,773	8.10	0.89	7.44	0.88	4.05	4.05	3.39
wro4j-core	1.5.0	0.18	29,059	7.45	0.88	6.65	0.87	3.11	4.34	3.54
fitjava	1.1	0.21	5178	4.56	0.82	4.07	0.80	2.22	2.34	1.85
jgap	3.6.3	0.34	68,071	8.40	0.89	6.88	0.87	3.97	4.43	2.92
log4j	1.2.15	0.35	37,925	6.83	0.87	5.63	0.85	3.41	3.42	2.22
JHotDraw	7.0.9	0.36	19,070	6.71	0.87	5.49	0.85	3.31	3.40	2.18
colt	1.2.0	0.40	101,730	7.73	0.89	6.11	0.86	3.64	4.08	2.47

We visualize the impact of DI on CBO and DCBO via Figures 3 and 4. Increasing trend lines are observed for both CBO and DCBO, with a smaller increasing slope observed for the DCBO metric.

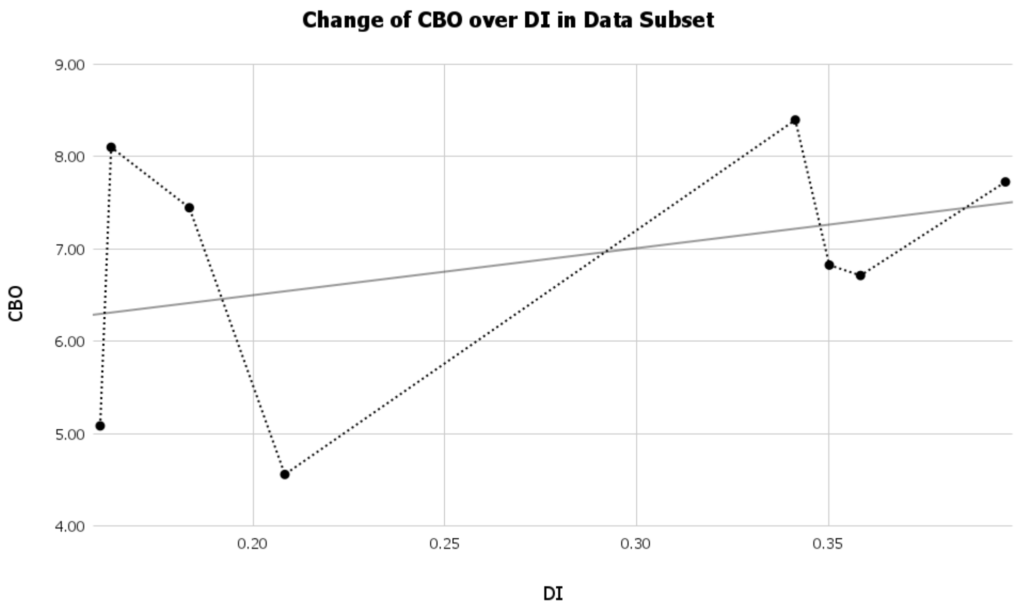

Figure 3. Impact of DI on CBO in data subset.

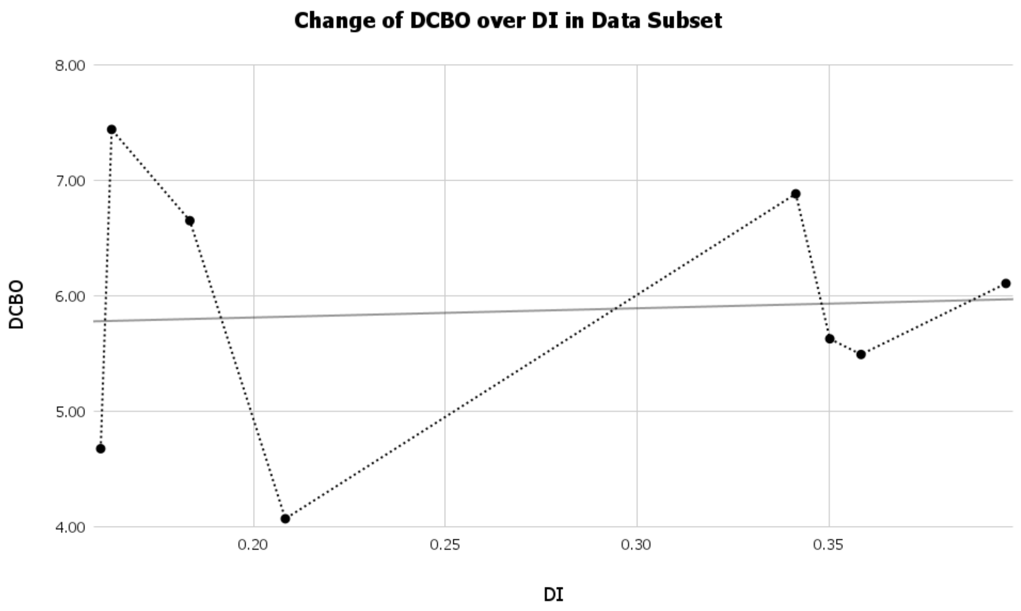

Figure 4. Impact of DI on DCBO in data subset.

5. Answering Research Questions

In this section, we discuss the implications of the experimental and open-source project results by answering the questions posed in Section 3.

- **Answer** to *"Does change in DI proportion impact CBO?"* Experimental results in Table 1 show that XML DI using Spring Framework [2] introduces a small coupling overhead, which increases CBO in small projects. The experimental projects controlled for all other metrics and LOC, but larger projects will be helpful in determining whether large-scale DI use significantly impacts CBO. After controlling for LOC, open-source projects also observed an increase in CBO as DI increased. While the study gave an overall analysis on the impact of DI on coupling, it did not identify projects using Spring Framework or other DI frameworks. While the use of a DI framework would suggest developer intent in using DI, it leaves out other projects that use DI without a framework. More research should be conducted in determining whether developers intended to use DI, as discussed by Yang et al. [4], and whether a DI threshold can be quantified in order to identify a project as utilizing DI.
- **Answer** to *"How effective is CBO in quantifying maintainability?"* CBO alone does not appear to be sufficient in quantifying maintainability. Other metrics have been used to assess maintainability, including lack of cohesion of methods (LCOM) and response for class (RFC) as analyzed by Razina and Janzen [3], but difficulties remain in determining how effective the values are as a robust metric [12]. More research will be needed to determine whether other metrics can better inform the maintainability of a software system.
- **Answer** to *"How should DI inform coupling in the software system?"* DCE and DCBO inform additional developer effort required when DI is not used. This paper covers a partial aspect of how DI can impact the maintainability of a project. In particular, we argued that DI via parameter or XML injection instead of `new` object (default object) construction improves maintainability by reducing the number of files that need to be changed should the injected class' constructor parameters change. CKJM-Analyzer, DCE and DCBO explicitly incorporate DI as a factor in weighing coupling differently. One future work of CKJM-Analyzer would be determining which portions of the software are using the CWD/MWD patterns and replacing them with the CND/MND patterns. Another work would be making a cross-platform CKJM-Analyzer, as right now, it is only usable on the Windows OS.

6. Related Work

This section describes existing works related to DI and coupling. We will start with a literature review on various papers regarding coupling, and finish with a literature review on DI.

6.1. Coupling Literature Review

The work by Anwer et al. [21] discussed how coupling metrics could be useful in indicating software faults. The authors analyzed three coupling metrics, coupling between objects (CBO), afferent couplings (Ca), and efferent couplings (Ce). Statistical analysis results showed that Ce was the greatest metric in determining defects, with CBO being the next most significant metric. This work suggests that reducing coupling—especially coupling that leads to software faults—can help in making the software more maintainable.

The work by Okike and Osofisan [12] proposed normalization approaches for CKJM metrics, particularly LCOM. The authors discussed how CKJM metrics are not normalized, and often do not give very helpful values. Using various normalization techniques, they found that the best-fit normalized LCOM (BFNLCOM) helped remove outliers more than other normalization techniques (Bowles and Sigmoid). They also discussed a formula for module complexity, a normalization technique revolving around coupling metrics. Our work uses these formulas to normalize the metrics we analyze for maintainability.

The work by Újházi et al. [22] proposed a novel metric coined Conceptual Coupling Between Objects (CCBO). They defined CCBO as "the sum of the parameterized conceptual similarities between a class c and all the other classes in the system" [22], and found that CCBO was effective in detecting fault proneness and developing bug prediction models.

The work by Saidulu [23] proposed a metric coined weighted coupling between objects (WCBO), used to better predict software fault proneness. They ranked the weight of class coupling with bipartite graphs and used the metric to assess fault proneness. If one class is coupled to only one other class, it is weighed much lower than a class that is coupled to multiple other classes—should the one class coupled to multiple other classes contain an error, many other classes will also inherit that error. Through WCBO, Saidulu was able to increase fault proneness sensitivity to over 90% as opposed to the 40% sensitivity given by CBO alone.

The work by Muhammad et al. [24] proposed a metric coined Vovel, a weighted metric based on volume and levels of coupling within the system. They found that Vovel metrics are more useful in predicting software faults than currently used coupling metrics, such as CBO, RFC, FanIn (CE) and FanOut (CA).

Our work differs from the existing work on coupling and its impact on maintainability through our proposal of new metrics extending on the CBO and CE metrics. We weigh the metrics via DI instead of the number of couplings [23], conceptual similarities [22], or volume and levels of coupling [24]. We target change in developer "effort" when DI is used by reducing the weight of efferent couplings that have been injected via DI, as there is significantly less work required in the code base when a change in the dependency is needed (a new concrete class is created, new parameters are added, etc.).

6.2. Dependency Injection Literature Review

The work by Razina and Janzen [3] studied the impact of DI on maintainability. They counted the number of DIs in a Spring project (i.e., a project built on Spring Framework, which is built on DI [25]) and divided the number by the sum of CBOs of the project for normalization. Maintainability is measured in terms of CBO, RFC, and LCOM using the CKJM tool [9]. They formed two groups of projects—Spring projects and non-Spring projects with an assumption that Spring projects use the DI pattern and non-Spring projects do not. They conducted ANOVA analysis on those groups to find out the correlation between the use of DI and maintainability. However, they found no obvious correlation between the use of DI and maintainability. We think that their hypothesis was rejected because it is likely that non-Spring projects also have DI. This is difficult to assess in Razina and Janzen's paper because they did not describe how DI was counted, which is critical for determining whether the non-Spring projects had DI considered in the same way as

Spring projects. For example, if Razina and Janzen only analyzed XML files for DI, it is more likely that non-Spring projects can implement DI in non-XML configured methods. They considered non-weighted CBO to evaluate the effect of DI.

The work by Yang et al. [4] provided technical definitions for DI, which we use later in the paper. They analyzed Java projects in light of those definitions and counted the number of instances of each pattern reflecting DI. Their results reveal that fewer projects utilized DI than expected, and they discussed whether their analysis was able to determine whether specific projects had an "intention" to use DI, or if they simply followed the DI pattern without realizing. While they described a tool they developed to count instances of DI, the source code does not exist and is therefore unable to be used in this paper.

The work by Crasso et al. [6] suggested that the use of DI can provide cleaner web-based interfaces. The model they developed demonstrates a significant improvement in the precision of interfaces for web applications with DI applied. An improvement was observed in web service queries, and the authors think it is due to DI code containing more "meaningful terms" or having higher accuracy in words/phrases that relate to search queries. Additionally, the authors observed a small performance hit and overhead increase when using DI, primarily because of the DI pattern requiring more memory in service adaptors. The authors argued that the slight increase in memory and overhead is minimal, especially because DI-based web services provide more accurate queries. The authors did not directly mention maintainability, but suggested that DI makes it easier to outsource web service development.

While most work focuses on benefits of DI, some argue that improper use of DI can lead to a decrease in maintainability. The work by Roubtsov et al. [7] presented bad smells caused by DI on modularity. They observed that a specialized form of DI using syntactic metadata has a higher probability of violating modularity rules, which leads to less cohesion. Similarly, the work by Laigner et al. [8] presented a catalog of DI anti-patterns (e.g., *framework coupling*, *intransigent injection*, *non used injection*) which increase coupling (e.g., by DI annotations, and unnecessary dependencies) while decreasing maintainability as opposite to "good" patterns (e.g., GoF patterns [26]).

Our work differs from the existing work on DI and its impact on maintainability through our open-source DI detection solution, CKJM-Analyzer [10]. There is currently no accessible open-source tool that can count instances of the DI pattern—Laigner et al. [8] developed a tool specifically detecting DI anti-patterns, not DI patterns. We developed a tool that can quickly detect DI proportion in Java projects, and provide general software quality metric analysis.

7. Conclusions

We presented CKJM-Analyzer, a command line tool to analyze DI and maintainability, as well as DCBO and DCE, weighted coupling metrics that explicitly consider DI as a more maintainable framework. The algorithm used to detect DI is defined. Experimental results show that small projects see an increase in coupling due to overhead caused by DI injection, while open-source results reflect an increase in coupling as DI increases. More research is needed to determine how to differentiate between projects intentionally utilizing DI and those simply injecting parameters. We argue that DI may contribute to more maintainable software by reducing the developer "effort" when a dependency change is required.

Author Contributions: Conceptualization, P.S.; methodology, P.S.; validation, P.S.; investigation, P.S.; writing—original draft preparation, P.S.; writing—review and editing, D.-K.K.; supervision, D.-K.K.; supervision, H.M.; supervision, L.L. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: All data were presented in the main text.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Murgia, A.; Tonelli, R.; Counsell, S.; Concas, G.; Marchesi, M. An empirical study of refactoring in the context of FanIn and FanOut coupling. In Proceedings of the 18th Working Conference on Reverse Engineering, Limerick, Ireland, 17–20 October 2011; pp. 372–376.
2. Johnson, R.; Hoeller, J. *Expert One-on-One J2EE Development without EJB*; Wiley Publishing: Hoboken, NJ, USA, 2004.
3. Razina, E.; Janzen, D.S. Effects of dependency injection on maintainability. In Proceedings of the 11th IASTED International Conference on Software Engineering and Applications, Cambridge, MA, USA, 19–21 November 2007; pp. 7–12.
4. Yang, H.Y.; Tempero, E.; Melton, H. An empirical study into use of dependency injection in java. In Proceedings of the 19th Australian Conference on Software Engineering, Perth, WA, Australia, 26–28 March 2008.
5. Lee, Y.; Chang, K.H. Reusability and maintainability metrics for object-oriented software. In Proceedings of the 38th Annual on Southeast Regional Conference, Clemson, SC, USA, 7–8 April 2000; pp. 88–94.
6. Crasso, M.; Mateos, C.; Zunino, A.; Campo, M. Empirically assessing the impact of dependency injection on the development of Web Service applications. *J. Web Eng.* **2010**, *9*, 66–94.
7. Roubtsov, S.; Serebrenik, A.; van den Brand, M. Detecting modularity "smells" in dependencies injected with Java annotations. In Proceedings of the 14th European Conference on Software Maintenance and Engineering, Madrid, Spain, 15–18 March 2010; pp. 244–247.
8. Laigner, R.; Kalinowski, M.; Carvalho, L.; Mendonça, D.; Garcia, A. Towards a Catalog of Java Dependency Injection Anti-Patterns. In Proceedings of the 33rd Brazilian Symposium on Software Engineering, Salvador, Brazil, 23–27 September 2019; pp. 104–113.
9. Spinellis, D. Chidamber and Kemerer Java Metrics. Available online: http://www.spinellis.gr/sw/ckjm/ (accessed on 22 September 2020).
10. CKJM-Analyzer. Available online: https://github.com/Narnian12/ckjm-analyzer (accessed on 1 June 2022).
11. Fowler, M. Inversion of Control Containers and the Dependency Injection Pattern. 2004. Available online: https://martinfowler.com/articles/injection.html (accessed on 6 August 2020) .
12. Okike, E. An Evaluation of Chidamber and Kemerer's Lack of Cohesion in Method (LCOM) Metric Using Different Normalization Approaches. *Afr. J. Comput. ICT* **2008**, *1*, 35–54.
13. ckjm-di. Available online: https://github.com/Narnian12/ckjm-di (accessed on 1 August 2022).
14. Lincke, R.; Lundberg, J.; Löwe, W. Comparing Software Metrics Tools. In Proceedings of the 2008 International Symposium on Software Testing and Analysis, Seattle, WA, USA, 20–24 July 2008; pp. 131–142.
15. Oman, P.; Hagemeister, J. Metrics for assessing a software system's maintainability. In Proceedings of the Proceedings Conference on Software Maintenance, Orlando, FL, USA, 9–12 November 1992; pp. 337–344.
16. Wagey, B.C.; Hendradjaya, B.; Mardiyanto, M.S. A proposal of software maintainability model using code smell measurement. In Proceedings of the International Conference on Data and Software Engineering, Yogyakarta, Indonesia, 25–26 November 2015; pp. 25–30.
17. Emam, K.E.; Benlarbi, S.; Goel, N.; Rai, S.N. The confounding effect of class size on the validity of object-oriented metrics. *IEEE Trans. Softw. Eng.* **2001**, *27*, 630–650. [CrossRef]
18. ckjm-di-projects. Available online: https://github.com/Narnian12/ckjm-di-projects (accessed on 1 August 2022).
19. Tempero, E.; Anslow, C.; Dietrich, J.; Han, T.; Li, J.; Lumpe, M.; Melton, H.; Noble, J. The Qualitas Corpus: A curated collection of Java code for empirical studies. In Proceedings of the 2010 Asia Pacific Software Engineering Conference, Sydney, NSW, Australia, 30 November–3 December 2010; pp. 336–345.
20. Garcia, S.; Herrera, F. An Extension on 'Statistical Comparisons of Classifiers over Multiple Data Sets' for all Pairwise Comparisons. *J. Mach. Learn. Res.* **2008**, *9*, 2677–2694.
21. Anwer, S.; Adbellatif, A.; Alshayeb, M.; Anjum, M.S. Effect of coupling on software faults: An empirical study. In Proceedings of the 2017 International Conference on Communication, Computing and Digital Systems, Islamabad, Pakistan, 8–9 March 2017; pp. 211–215.
22. Újházi, B.; Ferenc, R.; Poshyvanyk, D.; Gyimóthy, T. New conceptual coupling and cohesion metrics for object-oriented systems. In Proceedings of the 2010 10th IEEE Working Conference on Source Code Analysis and Manipulation, Timisoara, Romania, 12–13 September 2010; pp. 33–42.
23. Saidulu, A. Assessing Weight of the Coupling between Objects towards Defect Forecasting in Object Oriented Programming. *Glob. J. Comput. Sci. Technol.* **2014**, *14*, 35–39.
24. Muhammad, R.; Nadeem, A.; Sindhu, M.A. Vovel metrics—novel coupling metrics for improved software fault prediction. *PeerJ Comput. Sci.* **2021**, *7*, e590. [CrossRef] [PubMed]
25. Johnson, R.; Hoeller, J.; Donald, K.; Sampaleanu, C.; Harrop, R.; Risberg, T.; Arendsen, A.; Davison, D.; Kopylenko, D.; Pollack, M.; et al. The spring framework-reference documentation. *Interface* **2004**, *21*, 27 .
26. Gamma, E.; Helm, R.; Johnson, R.; Johnson, R.E.; Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*; Addison-Wesley: Boston, MA, USA, 1995.