An Empirical Study into Use of Dependency Injection in Java

Hong Yul Yang, Ewan Tempero, Hayden Melton
Department of Computer Science
University of Auckland
Auckland, New Zealand
{hongyul|ewan|hayden}@cs.auckland.ac.nz

Abstract

Over the years many guidelines have been offered as to how to achieve good quality designs. We would like to be able to determine to what degree these guidelines actually help. To do that, we need to be able to determine when the guidelines have been followed. This is often difficult as the guidelines are often presented as heuristics or otherwise not completely specified. Nevertheless, we believe it is important to gather quantitative data on the effectiveness of design guidelines wherever possible.

In this paper, we examine the use of "Dependency Injection", which is a design principle that is claimed to increase software design quality attributes such as extensibility, modifiability, testability, and reusability. We develop operational definitions for it and analysis techniques for detecting its use. We demonstrate these techniques by applying them to 34 open source Java applications.

1 Introduction

Design principles [23, 7, 16, 17, 25] influence the internal structure of a software system. Particularly, they guide the decisions we make as developers about the organisation of the entities in a system's source code. These decisions are inherent to the activity of programming — for instance, in adding some particular functionality to a system should we write the code as a new method, generalise an existing method, create a whole new class, or some combination of the above? Design principles help us to choose the "best" option.

Design principles are important because we believe that the internal structure of a system, as reflected in its source code, affects its maintainability, understandability, testability, modifiability, performance and so on, that is, its *software quality attributes*. [20, 25, 4]. Thus the "best" decision we can make in organising source code entities (i.e., methods, classes, packages etc) is the one that most improves the attributes of software quality that are important for a particular system. In order to determine which is "best", we need to be able to quantify the benefit due to the application of any given design principle. We need to understand what the trade-offs are and how different design principles interact.

We can determine the benefit achieved by applying a design principle by applying it and measuring the change to all the quality attributes. Measuring quality attributes is difficult enough but by itself does not tell us what the benefit is if we cannot be sure that the design principle has been applied correctly (or at all). Without reliable and objective means to determine when a design principle has been applied, we cannot be sure what caused the effects on quality attributes we observe.

A difficulty in reliably and objectively determining the use of most design principles is that they usually are not expressed in an *operational* manner. We believe that developing operational definitions of design principles is a necessary step in empirical validating their use. In this paper, we look at developing an operational definition for the the design principle sometimes known as *Dependency Inversion Principle (DIP)* [16] and carry out an empirical study of its use.

We think the DIP is worthy of further study because its proponents argue its application leads to systems that are more extensible [16, 12, 24], testable [16, 15, 28] [14, p.388], modifiable [16] [14, p.330] and reusable [16, 12]. In the work described in this paper we discuss a specific structural form of the DIP — what Fowler terms *Dependency Injection* (DI) [5]. We have developed an operation definition for DI, developed a tool that measures the use of DI according to our definition, and have applied the tool to 34 open source Java applications.

The rest of the paper is organised as follows. In section 2, we summarise the arguments for using DI, in particular the anticipated benefits having classes designed by applying the DI principle. From this, in section 3, we determine the structural characteristics of code that result from such an application, which leads to the definitions of four structural forms representing possible DI use. From this we develop

1530-0803/08 $25.00 © 2008 IEEE
DOI 10.1109/ASWEC.2008.35

our analysis techniques. In section 4 we present the results of our study and discuss our interpretation of these results in section 5. Section 6 then presents our conclusions.

2 Background

The phrase *Dependency Inversion Principle* was first coined by Martin in 1996 [16] although the concept it represents has been discussed by many others under the guise of different names. Fowler [6] dates the concept back to Johnson and Foote's discussion of *Inversion of Control (IOC)* [12] in 1988, and he notes that Sweet also alludes to it in 1985 with the more "colourful" phrase the *Hollywood's Law* [27]. Lakos [14, ch.6] also discusses the DIP under the guise of *insulation*.

While the DIP is easily stated at a conceptual level, defining it concretely, in terms of entities in Java source code, is less straightforward. A conceptual statement of the DIP is that by Martin: "High-level modules should not depend upon low-level modules. Both should depend upon abstractions" [16]. If we glean the examples given by Martin we might take this to mean that in Java a class should depend on interface or abstract types, not concrete types, although there are some benefits that accrue even with concrete types.

Besides the issue of whether we should depend on interface, abstract or concrete types we must deal with the "problem of instantiation" [18], or as the Gang of Four state "you have to instantiate concrete classes (that is, specify a particular implementation) somewhere in your system" [7, p.18]. This is another challenge in concretely stating, and measuring the DIP — we need to know the mechanism by which concrete classes are instantiated and passed in to their DIP exhibiting clients.

There are actually many ways to instantiate and pass in concrete classes to those exhibiting the DIP. Fowler identifies the Dependency Injection and Service Locator approach [5]. In this work, we concentrate on the *Dependency Injection* form of the DIP.

In the *Dependency Injection* (DI) form of the DIP, as it is discussed by Fowler [5], the object assigned to the field of a class is passed in through one of that class' constructors or methods. A simple illustration of dependency injection is as follows:

```
class A {
  B b;
  public A(B b) {
    this.b = b;
  }
  //...
}
```

In the above code example `A` is exhibiting dependency injection because the object that gets assigned to its field 'b' is passed in as a parameter in `A`'s constructor. We will, for the moment, avoid a discussion of whether `B` should be an interface type, abstract type or concrete type. The key observation is that `A` does not depend on a particular implementation of B. Particularly, when clients instantiate A, they get to specify the particular subtype of B to be assigned to 'b' at runtime. This can have beneficial consequences to several software design quality attributes.

2.1 Effects on Quality

It has been argued that Dependency Injection affects many quality attributes, in particular extensibility, testability, and reusability.

Extensibility can be defined as "the ease with which a system or component can be modified to increase its storage or functional capacity" [10]. In the above snippet `A` is arguably more extensible because it can be used with different implementations of `B` without modifying the source code of `A`. Indeed this is why DI is used at the "plug-points" of application frameworks [12, 24].

Testability can be defined as "the degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met" [10]. Dependency Injection supports the use of *mock objects* to help unit test a class [15, 28]. Mock objects can be used to both provide control and observe the class under test.

Reusability can be defined as "the degree to which a software module or other work product can be used in more than one computer program or software system" [10]. Dependency injection can improve reuse by (1) improving flexibility and (2) breaking transitive dependencies.

DI can improve flexibility because different implementations can be used with the class we want to reuse, improving the degree to which that class can be used in multiple situations, as discussed discussed above in extensibility. Dependency injection can reduce the number of classes we have to deploy in the context of a new system by breaking transitive dependencies. This is important because to effectively reuse a class it should not be tied to a large block of unnecessary code [14, p.14]. If a class we reuse depends on a class type, from the perspective of reuse, it also transitively depends on any types that appear in the private part of that class type. If the type it depends on is an interface type then it does not transitively depend on any private parts of that interface's implementation.

```
class CNDeg {
 B b;
 public CNDeg(B b){
   this.b = b;
 }
}

class CWDeg {
 B b;
 public CWDeg() {
   b = new BImpl();
 }
 public CWDeg(B b){
   this.b = b;
 }
}
```

```
class MNDeg {
 B b;
 public void setB(B b){
   this.b = b;
 }
}

class MWDeg {
 B b;
 public MWDeg() {
   b = new BImpl();
 }
 public void setB(B b){
   this.b = b;
 }
}
```

Figure 1. Examples of the forms of DI

3 Characterising Dependency Injection

3.1 Definitions

In the Dependency Injection form of the DIP the value assigned to a class' field is passed in through a setter or constructor, rather than created within the class. We can use this as the basis for an operational definition for DI. For each field in a class, we determine what values are assigned to the field and where those values came from. If they do not come from outside the class, then that is inconsistent with the intent of DI (although we identify one special case below). We have identified 4 forms of DI for fields in Java code, which we define below.

3.1.1 Constructor No Default (CND)

The *only* object a field in a class can be assigned comes through the parameter of the class' constructors. That is, the only objects a field is assigned are passed in from *outside* the class, through the class' constructor(s). Class `CNDeg` in Figure 1 shows an example.

Rationale: In the above code the *only* way an object can be assigned to field 'b' is by passing that object through the constructor. This means `CNDeg` can be tested with a mock object of supertype `B`. Similarly, `CNDeg` is potentially more extensible because it can be used with different implementations of `B`. If `B` is an interface type it means that `CNDeg` can be reused in another system independently from any implementations of `B`.

3.1.2 Method No Default (MND)

The object a field in a class can be assigned comes through the parameter of one of either the constructor, or the class' non-private methods. That is, the only objects a field is assigned are passed in from *outside* the class, through the class' non-private methods(s). Class `MNDeg` in Figure 1 shows an example.

Rationale: The above code is similar to that given for CND, except the object 'b' is assigned is passed in through a setter method. The use of a setter method allows the object assigned to the field to change over the object of the lifetime purporting improved flexibility over CND. On the other hand it is also possible we forget to assign an object to field 'b' (not possible in CND), and this will likely cause a `NullPointerException` at runtime. Which is better is subject to some debate. Beck recommends the constructor based approach [3], saying it is immediately clear what a class requires when it is instantiated, and furthermore it is impossible to instantiate the class without passing in the field's objects. However, a recent empirical study by Stylos and Clarke seems to contradict Beck's argument. Their study found that that programmers found it easier to pass references through setter methods rather than constructors [26]. Consequently we have chosen to measure both forms. Apart from this the reusability, testability and extensibility are the same as CND.

3.1.3 Constructor With Default (CWD)

The object assigned to a field *can* be passed in through a constructor but this does not happen exclusively. The field is also assigned a "default" object from within the class. Class `CWDeg` in Figure 1 shows an example.

Rationale: The above code is similar to that given for CND, except there is a *default* implementation of `B` referenced in it. This potentially hinders reusability because we must now deploy `B` *and* `BImpl` in order to compile `CWDeg` in the context of a new system. We must also deploy anything `BImpl` depends on — we could end up copying a very large chunk of code in this transitive fashion. That said, CWD has no significant difference in flexibility, extensibility and testability than in CND, and furthermore users of such classes do not have the burden of having to provide an implementation for `B` for every use of `CWDeg`.

3.1.4 Method With Default (MWD)

The object assigned to a field *can* be passed in through a constructor or non-private method but this does not happen exclusively. The field is also assigned a "default" object from within the class. Class `MWDeg` in Figure 1 shows an example.

Rationale: This situation is analogous to CWD — reuse is inhibited because a concrete type (`BImpl`) is referred to in the body of `MWDeg`.

3.1.5 Completeness

There are a number of ideas that have been labelled "dependency injection" or something similar (as we will mention further below). As this is the first study of its kind, we have chosen not to try to capture all possible variations. Instead, we have limited our study to these relatively simple forms of DI. We believe these forms are representative of the presentations of DI in the literature, in particular, in the trade press and tutorials likely to be accessible to developers. As such, we believe that if there is widespread adoption of DI, then the forms we have identified should be prevalent.

3.2 Practical Considerations

The definitions above give the general structures that indicate the use of DI. There are, however, some consequences and practical issues that require further discussion.

In this study, we require that types of fields be non-concrete (that is, either interfaces or abstract classes). This means the use of any concrete type for a field rules out that class as using DI. As discussed earlier, from the point of view of, for example, testing, such fields might be considered an acceptable form of DI and so we intend to look at such forms in future work.

The creation of concrete values (that is, calls to a constructor) also rule out the class as using DI provided the creation occurs outside the class' constructor (since creation of concrete values within a constructor could indicate one of the "default" cases). It is also possible that concrete values can be assigned to fields, even though they are not created in the class. For example, a parameter of concrete type can be assigned to the field. Such definitions rule out the class from using DI.

The use of constructors of arrays depends on the base types of the arrays — if the result requires the use of concrete values then it rules out the use of DI, otherwise it is neutral.

A value assigned as a result of a method invocation on another class also rules out the use of DI, as in the general case we cannot be sure what the type of that value will be. The use of a service locator is a special case that we believe can be identified, and we will consider this in future work.

One last form of field definition that we must consider is the assignment of `null` to a field. This form of defining a value for a field does not impact use of DI and so is ignored.

We also ignore fields that are of primitive type, or of type from the Java Standard API ("built-ins"), in that their presence did not impact our classification of a class. We ignore fields of primitive types since there is no opportunity for a developer to allow alternative implementations to be provided for such fields. It could be argued that the object wrapper types, such as `java.lang.Boolean` could have been used instead, however this choice has other issues, being both final classes and types supplied by the Standard API. In the case of types from the Standard API, there is the opportunity to use appropriate interfaces (e.g., `java.util.List`) or abstract classes (e.g., `java.io.Reader`). However there are also classes for which there is no convenient interface or abstract parent (e.g., `java.lang.String`) meaning, again, the developer has no alternative. This is also something we wish to consider further in future work, that is, whether these built-in types support are being used to provide DI.

```
public class A {
  protected B b;
  public A(B ba) {
    setB(ba);
  }
  public A() {
    setB(getDefault());
  }
  protected void setB(B b) {
    this.b = b;
  }
  protected B getDefault() {
    return new BImpl();
  }
}
```

Figure 2. Non-trivial assignment examples

Final types, that is types that cannot have subtypes (classes declared "final" in Java), cannot be used for DI, and so their presence disqualifies a class from using DI. We also note that, should we consider classes with fields of concrete types to be acceptable for using DI, final types would still be a problem. If we consider built-in types further, final types such as `String` might have to be treated specially.

3.3 Measurement

3.3.1 Analysis Procedure and Algorithm

The analysis of dependency injection in application code is performed by extending a part of our existing tool [30], which operates on Jimple – a static single assignment typed 3-addressed intermediate representation of Java bytecode from the Soot framework [29]. It also utilises static analysis features of the Indus project [11], which is based on Soot. The tool is limited to Java 1.4 source code.

The overall measurement actually comprises several steps. The first step is to analyse the source (represented in Jimple) for "use-def" information and generate a graph data-structure comprising the usage/definition sites and data flows among them. The algorithm for computing this employs standard inter-procedural data flow analysis techniques such as those found in the slicing literature (e.g.

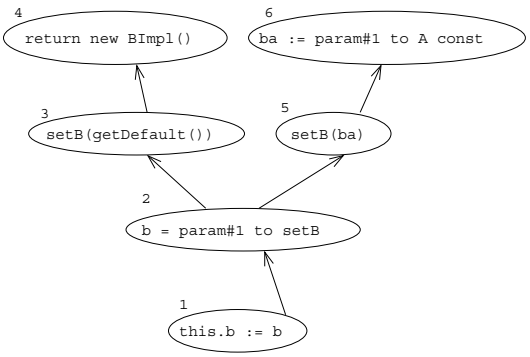

Figure 3. data-flow graph

[1]), and in particular the data- and object-flow analyses that Indus provides, as described in [22]. The precision of the analysis is thus dependent on the underlying techniques, which deal with well-known issues such as polymorphism and array aliasing to a certain extent.

The aim of the analysis for each field of a class is to determine all of its possible *definition* values. In the simplest case, the definition of a field is a direct assignment, e.g. `field := value`. But it is usually the case that the value is in turn defined through a preceding statement, and so on, which effectively results in a chain of definitions, thus referred to as *use-def chains*, that eventually affect the value of the original field.

Whereas in the examples we have shown so far, the definitions of fields have been of a simple nature (direct assignments), there are many instances where the value is defined through indirect means, as illustrated in figure 2, thereby requiring an analysis along use-def chains. Here, field `b` is defined through a parameter to the method `setB`, which is called locally by both of `A`'s constructors. The first constructor simply passes down its parameter `ba` to `setB`. The second constructor on the other hand additionally calls `getDefault`, which constructs and returns a concrete value that is then passed to `setB`.

To obtain the definitions of each field, the tool applies a depth-first traversal algorithm on a graph representing statements and data flows between them. This graph, for a given class `C`, is constructed such that:

- its vertices represent statements within the *"boundary"* of `C` (we define this as any program element contained in `C` or its superclasses)
- edge exists from v1 to v2 iff both data and control flow exist from a value used in v2 to value used in v1, i.e. the direction of the edge is opposite to that of data/control flow.

The algorithm begins traversing from the statements (vertices) that directly assign a value to any field of `C`, then follows the edges until either all vertices have been visited or no more vertices can be visited. During the traversal, the algorithm records each parameter value or concrete value it encounters. Figure 3 demonstrates this process being applied class `A` in figure 2 – note that parameter passing are shown as implicit statements to clarify the data flow. In this example, the algorithm begins from the source vertex labelled `1` (the assignment statement) and traces along the edges to eventually reach vertices numbers `4` (concrete value) and `6` (parameter to `A`'s constructor). These vertices represent the definition sites of `b` that are relevant to DI and hence are recorded.

For each definition site, the following are recorded:

- the name and type of the field that is being defined
- the class that it belongs to
- location (class, method and line number) of the definition
- the type of the value defining the field
- The nature of definition: 'is the value from a parameter? Or a concrete value? Or is it through some other means?'

The above details are aggregated for each class and used to determine the class's conformance to the DI definitions outlined previously.

3.3.2 Scope of Analysis

It is worth noting that we are deliberately limiting the use-def analysis to within the boundary of each given class. This effectively reduces the search space for the analysis, thereby improving performance. Also in the interest of the forms of dependency injection under investigation, analysing within classes is sufficient in obtaining the necessary results. However in the future we could extend the tool by tracking the use-def chain further to outside the boundary to cater for more system-wide forms of DIP such as service locators.

A consequence of our decision is that many of the issues that face other forms of data flow analysis, such as inheritance, polymorphism, aliasing, and the like do not apply to our analysis. For example, if a subclass of `A` from figure 2 directly assigns to the field `b`, then in our analysis that assignment will be treated as if `b` were a field of the subclass (as indeed it is).

An issue with polymorphic calls in data flow analysis is not being sure which code can actually be executed. However, since we regard values assigned to fields as the result of *any* method invocation to rule out the use of DI, the fact that the method invocation might be polymorphic is irrelevant.

Object aliasing is when two or more references (pointers) refer to the same runtime instance. Aliasing can present

Application	Type	Size	N	CND	MND	CWD	MWD	P	B	Not	%
ant-1.4.1	App	178	138	4	0	0	0	10	67	57	7%
antlr-2.7.5	Emb	209	142	2	1	1	0	23	35	80	5%
aoi-2.2	App	346	297	5	3	0	0	30	34	225	3%
argouml-0.18.1	App	1210	613	18	3	1	0	16	296	279	7%
axion-1.0-M2	App	237	148	13	13	0	0	8	40	74	26%
azureus-2.0.4.0	App	346	199	16	12	0	1	21	65	84	26%
colt-1.2.0	Emb	195	104	9	0	6	0	42	11	36	29%
fitjava-1.1	App	37	21	0	0	0	0	3	7	11	0%
fitlibraryforfitnesse-20051216	Emb	156	70	0	0	0	0	6	38	26	0%
ganttproject-1.11.1	App	310	197	28	4	0	0	9	50	106	23%
hibernate-3.1	Emb	895	529	62	22	0	0	24	232	189	31%
hsqldb-1.8.0.2	App	218	165	3	1	0	0	21	34	106	4%
ireport-0.5.2	App	347	289	0	0	0	0	41	131	117	0%
jag-5.0.1	Emb	121	99	1	0	0	0	6	51	41	2%
jaga-1.0.b	Emb	100	55	1	2	0	2	12	21	17	23%
james-2.2.0	App	259	150	4	4	0	0	9	79	54	13%
jchempaint-2.0.12	App	83	50	0	0	0	0	1	22	27	0%
jeppers-20050607	App	84	48	1	0	0	0	7	18	22	4%
jext-5.0	App	211	108	1	0	0	0	1	43	63	2%
jfreechart-1.0.0-rc1	App	469	313	10	8	2	1	33	141	118	15%
jgraph-5.7.4.3	Emb	50	32	2	1	0	0	1	9	19	14%
jhotdraw-6.0.1	Emb	300	171	15	11	0	1	20	59	65	29%
jmeter-1.8.1	Emb	216	153	3	3	0	0	4	81	62	9%
jparse-0.96	Emb	69	61	1	0	0	0	1	9	50	2%
jung-1.7.1	Emb	378	236	24	9	2	0	28	86	87	29%
junit-3.8.1	Emb	47	28	4	0	0	0	5	13	6	40%
lucene-1.4.3	Emb	170	134	18	2	0	0	19	38	57	26%
megamek-2005.10.11	App	455	314	12	4	0	0	52	94	152	10%
picocontainer-1.3	Emb	82	52	5	2	11	0	3	17	14	56%
poi-2.5.1	Emb	385	284	1	1	0	0	121	58	103	2%
rssowl-1.2	App	189	135	0	0	0	0	3	50	82	0%
sablecc-3.1	Emb	198	112	1	1	0	0	4	21	85	2%
scala-1.4.0.3	App	399	62	1	0	0	0	27	9	25	4%
spring_framework-1.1.5	Emb	905	508	22	30	4	16	22	295	119	38%

Table 1. Number of classes meeting each DI definition

a problem because it allows the state of one object to be changed from multiple syntactic locations. We are only concerned as to where any object that is assigned to a field originated, specifically inside or outside of the class boundary. The state of that object, or how that state might change, is therefore not relevant to that determination.

4 Results

We have analysed a subset of a Java Corpus we have compiled[2, 19, 21] consisting of open-source java applications, looking for evidence of the use of DI.

Of the 34 applications in our study, 17 could be classified as true applications, in that they are intended to be deployed as is, whereas the other 17 were designed with the intent that they be embedded within other applications. For some it is difficult to draw the line, as some frameworks come with ready-made useful applications as examples (e.g., JMeter) and some applications provide APIs to allow programmatic customisation (e.g., JFreeChart). Our reason for classifying applications this way was the hypothesis that we would see more use of DI in systems intended to be embedded, in particular, frameworks.

We classified each class in an application according to the definitions given in section 3 and determined the totals for each category for each application. The results are shown in Table 1. The application name includes the version number we analysed. The **Type** column shows our

classification of applications into those intended to be embedded (`Emb`) versus those that can be deployed stand-alone (`App`). The next column shows the size of the application in terms of number of top-level classes. The third column shows the number of top-level classes that appear in the analysis.

Classes that were not analysed include classes with no fields (e.g., interfaces, classes with only static methods), or subclasses whose only fields are those inherited from ancestors and do not directly assign to them. In some cases, the difference is quite surprising (ArgoUML for example), and is worth further study.

The columns **CND**, **MND**, **CWD**, **MWD** show the number of classes in the application obeying the different forms of DI as discussed in section 3.1. The **Not** column gives the number of classes analysed that have some form of field assignment that means they cannot be using DI as we have defined it. The **P** column shows classes that contain only fields that are of primitive type and the **B** column shows classes that contain only fields that are of a type from the Standard API or a primitive type. We report these separately as we cannot classify them in the 4 DI categories, and, since we ignore the effect of fields of primitive and built-in types, we felt it was mis-leading to classify such classes as not using DI. The last column shows the number of classes meeting one of sets of the DI criteria as a proportion of those classes that are "eligible" to meet the criteria, that is, the proportion of $\mathbf{N} - \mathbf{P} - \mathbf{B}$.

Of the 34 applications we analysed, 5 have no classes that meet our criteria and a further 5 had only 1 class (classified as CND in all cases). All applications that had any class meeting the criteria had classes classified as CND, only two applications (`jaga` and `spring_framework`) had fewer CND classes than some other category, and for only two further applications (`jfreechart` and `picocontainer`) where there fewer (and only just) CND classes that all other DI categories put together. The application with the most classes satisfying at least one DI form was `hibernate` (84) with the next largest number being that for `spring_framework` (72).

The application with the highest proportion of eligible classes meeting one of the sets of DI criteria is `picocontainer`, with 12 applications overall having almost one quarter of their classes meeting the criteria.

5 Discussion

In this section we attempt to develop conclusions based on our data. Our goal is to determine to what degree DI is being used. The main issue in doing this, however, is determining intent. The structures we measure, while they may be consistent with the use of DI, may also be developed without intending to use DI or indeed without the knowledge of DI. Determining intent from source code is difficult. The rationale for decisions, particularly design decision, cannot be divined from code, and often is not provided in what documentation there may be available. Since this is the first study of its kind, we have no baseline on which to make some comparisons, and so some of our statements are necessarily speculative.

The overall sense is that DI is not being widely applied. There is no obvious difference between `Emb` and `App` type applications. Of the applications that had a small number of classes appearing to use DI, several of those classes appeared to met the requirements only by accident, that is, a false positive.

An example of a likely false positive is `JagBlockViewer` in `jag`. This is the only class that meets any of our criteria in this application, and it is documented as being a test class. It seems unlikely that DI was intended to be used in this case. Another example is `Node` in `sablecc`, which is classified as MND. This is an abstract class whose sole field is of its own type, that is, `Node`, which suggests it is unlikely to have been designed with DI in mind. This application also has a single class classified as CND, namely `ParserException`, an exception class.

For other applications with small numbers, it is more difficult to rule out the use of DI, but we think it unlikely that anyone familiar with DI would deliberately use it so few times. For example `ant` has four classes classified as CND. They are all in the same package (`org.apache.tools.ant.taskdefs`), a package that has more than 70 classes, including more than 25 that are eligible to meeting DI criteria but which do not. While it is conceivable that no other classes meet ID criteria due to the nature of the application, we think it is unlikely, and so suspect that the four classes that do, so do accidentally.

One possible explanation for small numbers is that the relevant classes were not developed with DI in mind, but in order to support a design pattern. Many common implementations of design patterns, especially the original set [7], use DI-like structures to achieve their goal. We speculate that someone not familiar with DI but implementing a design pattern would thus create classes that meet our definitions.

An example of this is JHotDraw. JHotDraw has a number of classes (about 26%) meeting the criteria, but looking at the classes we find many instances of classes with names involving "Command", "Handle", "Visitor", "Listener", and "Enumerator". These names suggest these classes were designed not so much with DI in mind, but a consequence of the respective design patterns.

In fact, given JHotDraw's history, DI was almost certainly intended, but it does illustrate the fact that since some design patterns implementations do mimic the DI patterns

we study, it is conceivable that developers create classes without being aware of DI. This raises the issue of whether design patterns should be taught without reference to the underlying principles that make the patterns effective.

Determining whether the numbers we see are indicative of high or low use of DI is difficult without studying each and every class. The application `junit` provides a useful case study, being small enough for it to be feasible to do exactly that. On the surface, 4 of 28 classes seems a small number of classes to be using DI, and given Junit's development history we might expect to see DI used extensively. In fact, one class (`TestDecorator`) appears the consequence of a design pattern, and one class (`FailureRunView`) has a sufficiently complex "setter" method that we wonder whether DI was intended. That said, none of the classes classified as not involving use of DI could easily be changed to do so. We are left with the conclusion that either DI was not a significant consideration when designing Junit, or the level of use that we have measured is in fact indicative of good use of DI. Further study is needed in this regard.

The results for `spring_framework` [13], `hibernate` [9], and `picocontainer` [8] are of particular interest as all three have been described as being based around DI. That their results are three of the top four proportions (the 4th being the relatively small application `junit`) suggests that our methods for analysing software for use of DI are sound.

5.1 Threats to Validity

As already discussed, divining intent from code is problematic, and our conclusions must be interpreted in that light.

As we have indicated above, we believe there are false positives, meaning our results may overstate the actual usage of DI.

We have not considered service locators, mainly to limit the scope of the study to something that can be done in a reasonable amount of time. As we said earlier, our choice of DI structures was motivated by the material describing DI commonly available to developers. It would be very interesting indeed if the use of service locators was significantly higher than the structures we have studied.

Whether there are false negatives in our study is a matter of definition. There is some debate within the industry as to what is "proper" use of DI, or even whether DI is a concept separate from other concepts, such as design patterns. Indeed, it has been suggested that what we are calling DI, is really just a particular Design Pattern. Rather than argue the point, we can only observe that Fowler, Martin, and others clearly consider DI as a distinct concept, and that alone makes it worthy of study.

Our decision to ignore fields with types from the Standard API needs to be revisited. It is conceivable that a number of classes containing just fields of such types may be a consequence of using DI.

Finally, how widely applicable our results are depends on the representativeness of our corpus. We do cover a variety of domains, although limitations of our analysis tools means we have been somewhat limited in the size of application we can consider. Nevertheless, we believe our results do indicate a significant trend, although it remains to be seen how widespread it is.

6 Conclusions

Dependency Injection is widely touted in the trade literature as a way to improve the structure of code. We have presented the analysis of 34 open-source Java applications for the evidence of the use of Dependency Injection. This represents the first study of this kind. To do so, we have identified four patterns of code structure that are consistent with DI use and developed analysis tools to recognise these structures.

Our conclusion is that, while there are individual pockets, there is not a great deal of evidence to suggest widespread use of DI. Why there is so little use of DI is a matter of conjecture. It may be that the benefits resulting from its use are not as good as claimed. It is possible that other mechanisms, such as service locators, are being used. The most likely explanation is that it is simply not taught as a matter of course in software design courses and so consequently is not that well known as a design principle.

The measurements we have obtained provide a useful starting point for developing a benchmark for DI use, however we see our main contribution as being the fact that we can make the measurements at all. Having an operational definition of DI means we can now more reliably do studies on the actual benefits of using this design principle.

There is ample future work to be done. As we have mentioned, we would like to determine how to measure the use of service locators. We would also like to make our tool more accessible. It grew out of other research we are doing and its current form is one that is sufficient to prove the concept but not useful for distribution. Ultimately we would like to carry out studies to quantify the benefits of using dependency injection.

References

[1] M. Allen and S. Horwitz. Slicing java programs that throw and catch exceptions. In *PEPM'03*, 2003.

[2] G. Baxter, M. Frean, J. Noble, M. Rickerby, H. Smith, M. Visser, H. Melton, and E. Tempero. Understanding the

shape of Java software. In W. Cook, editor, *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 397–412, Portland, OR, U.S.A, Oct. 2006.

[3] K. Beck. *Smalltalk: best practice patterns*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.

[4] F. P. Brooks. *The Mythical Man-Month (Anniversary ed.)*. Addison-Wesley Longman Publishing Co. Inc., Boston, MA, USA, 1995.

[5] M. Fowler. Inversion of control containers and the dependency injection pattern. `http://martinfowler.com/articles/injection.html`, Jan. 2004.

[6] M. Fowler. Inversion of control. `http://www.martinfowler.com/bliki/InversionOfControl.html`, Jan. 2005.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[8] A. Hellesoy and J. Tirsen. Picocontainer introduction. `http://www.picocontainer.org/introduction.html`. Accessed October 2007., 2007.

[9] Preparing daos with manual dependency injection. `http://www.hibernate.org/328.html#A5` Red Hat Middleware, LLC. Accessed October 2007, 2006.

[10] IEEE standard glossary of software engineering terminology. IEEE Std 610.12-1990.

[11] Indus project site. http://indus.projects.cis.ksu.edu/.

[12] R. E. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988.

[13] B. Jose. The spring framework. `http://javaboutique.internet.com/tutorials/spring_frame/article.html`. Accessed October 2007, 2007.

[14] J. Lakos. *Large-scale C++ software design*. Addison Wesley Longman Publishing Co. Inc., Redwood City, CA, USA, 1996.

[15] T. Mackinnon, S. Freeman, and P. Craig. Endo-testing: Unit testing with mock objects. In *eXtreme Programming and Flexible Processes in Software Engineering - XP2000*, 2000.

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

[17] R. C. Martin. Granularity [object-oriented design]. *C++ Report*, 8(10):57–62, Nov.-Dec. 1996.

[18] H. Melton and E. Tempero. The CRSS metric for package design quality. In G. Dobbie, editor, *Australasian Computer Science Conference*, pages 201–210, Ballarat, Australia, Jan. 2007. Australian Computer Science Communications. Published as CRPIT 62.

[19] H. Melton and E. Tempero. An empirical study of cycles among classes in Java. *Empirical Software Engineering*, 12(4):389–415, Aug. 2007.

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

[21] Qualitas Research Group. Qualitas corpus. `http://www.cs.auckland.ac.nz/~ewan/corpus/`, Oct. 2007.

[22] V. P. Ranganath. Object-flow analysis for optimizing finite-state models of java softw are. Master's thesis, Kansas State University, 2002.

[23] A. J. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[24] D. C. Schmidt, A. Gokhale, and B. Natarajan. Leveraging application frameworks. *Queue*, 2(5):66–75, 2004.

[25] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Syst. J.*, 13(2):115–139, 1974.

[26] J. Stylos and S. Clarke. Usability implications of requiring parameters in objects' constructors. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 529–539, Washington, DC, USA, 2007. IEEE Computer Society.

[27] R. E. Sweet. The Mesa programming environment. In *Proceedings of the ACM SIGPLAN 85 symposium on Language issues in programming environments*, pages 216–229, New York, NY, USA, 1985. ACM Press.

[28] D. Thomas and A. Hunt. Mock objects. *IEEE Software*, 19(3):22–24, 2002.

[29] R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

[30] H. Y. Yang, E. Tempero, and R. Berrigan. Detecting indirect coupling. In *Proceedings of the 2005 Australian Software Engineering Conference*, pages 212–221, 2005.