Detecting modularity "smells" in dependencies injected with Java annotations

2010 14th European Conference on Software Maintenance and Reengineering

Detecting Modularity "Smells" in Dependencies Injected with Java Annotations

Serguei Roubtsov, Alexander Serebrenik and Mark van den Brand
Faculty of Mathematics and Computer Science
Technische Universiteit Eindhoven,
Den Dolech 2, P.O. Box 513, 5600 MB Eindhoven, The Netherlands
{*s.roubtsov,a.serebrenik,m.g.j.v.d.brand*}*@tue.nl*

***Abstract*—Dependency injection is a recent programming mechanism reducing dependencies among components by delegating them to an external entity, called a dependency injection framework. An increasingly popular approach to dependency injection implementation relies upon using Java annotations, a special form of syntactic metadata provided by the dependency injection frameworks. However, uncontrolled use of annotations may lead to potential violations of well-known modularity principles.**

In this paper we catalogue "bad smells", i.e., modularity-violating annotations defined by the developer or originating from the popular dependency injection frameworks. For each violation we discuss potential implications and propose means of resolving it. By detecting modularity bad smells in Java annotations our approach closes the gap between the state-of-the-art programming practice and currently available analysis techniques.

***Keywords*-dependency injection, modularity, Java annotations, coding smells, configuration**

I. Introduction

Recent years have seen a growth of multiple Java component middleware platforms called "dependency injection frameworks" [18], [21], [15]. These frameworks support a novel form of component deployment and configuration generative techniques implementing *dependency injection*. Intuitively, dependency injection is a mechanism of delegating handling dependencies among components to an external entity. An increasingly popular approach to dependency injection implementation relies upon using Java annotations (see Section II), provided by dependency injection frameworks or defined by the application developer.

Flexibility provided by dependency injection should not, however, result in poorly maintainable software. In this paper we focus on *modularity*, known to be important for maintenance as recognized, e.g., in [11], [6]. Furthermore, it has also been observed in [2] that given two artifacts of the same value (functionality), users and designers prefer more modular artifacts, i.e., those with less costly structure.

In this paper we investigate the implications of Java annotations provided by modern dependency injection frameworks on the quality of the resulting software products. To this end we consider one of the commonly accepted good software design practices, namely modularity. We present a *catalogue* of annotations violating the modularity principles found in the literature, including discussion of the violation impact and resolution suggestions.

Example 1: Let Polygon be a public interface annotated with @ImplementedBy(Square.class). This annotation indicates that Square.class provides a default implementation of the interface being annotated. Since multiple implementations may be available and the choice between them may depend on different deployment aspects, we consider associating interfaces with implementation to be a configuration task. The use of @ImplementedBy implies duplication of the information that the class Square implements interface Polygon: this information is present in the @ImplementedBy annotation as well as in the class definition: public class Square implements Polygon.

Moreover, explicit reference to Square in the @ImplementedBy annotation makes the Polygon aware of its implementation. Not only does it tighten coupling between the interface and its implementation, it also replaces the unidirectional link from Polygon to Square by a loop. Circular dependencies are known to hinder program comprehension and, therefore, maintenance [7].□

Reminder of the paper is organized as follows. Section II is dedicated to the definition of dependency injection. In Section III we provide the catalogue of annotations violating the modularity principles. Section IV reviews related work and concludes the paper.

II. Dependency injection

Intuitively speaking, dependency injection is concerned with getting rid of external dependencies as much as possible. By "external dependencies" we understand dependencies of an object on other objects. To our best knowledge, there is no commonly accepted definition of dependency injection in literature. Although all the authors [8], [23], [14] agree that its underlying idea is simple, the definitions they give are explanatory, and can be restated as *Dependency injection means providing an object with its instance variables by an external entity.*

Till recently a common approach to injection resolution involved creating and supporting configuration entities recording data settings required by multiple components. This centralized approach implied that configuration entities, typically XML files, should be maintained as well. For large

1534-5351/10 $26.00 © 2010 IEEE
DOI 10.1109/CSMR.2010.45

applications maintenance of configuration entities might become prohibitive. A more recent approach to specifying dependency injection consists in using annotations, i.e., additional information about the program. Java annotations, originally introduced in Java 2 Platform Standard Edition (J2SE) 5.0 [17], affect the way programs are treated by tools and libraries. Annotations are increasingly popular: e.g., SEAM [15] provides almost fifty different annotations, while EJB3 [18] counts more than seventy.

Working with annotations usually involves two phases: defining annotation types and using annotations. Typical enterprise application programmers seldom have to define an annotation type as they are usually provided by dependency injection frameworks such as EJB3 [18], Guice [21] or SEAM [15]. Still, in the exceptional cases, when an annotation type should be defined by the developer, this can be easily done as the defining an annotation type is akin to an interface definition, where each method declaration defines an element of the annotation type. Once an annotation type is defined, it can be used to annotate a program. An annotation is a special kind of modifier, similar to public or static, and can be used anywhere where other modifiers can be used. Annotations consist of an @ character followed by an annotation type and a parenthesized list of element-value pairs. For one-parameter annotations, such as @ImplementedBy, the list of element-value pairs can be reduced to the corresponding value: @ImplementedBy(Square.class).

Semantics of the annotation is provided by tools interpreting it, e.g., an annotation processing part of the Java compiler or the Java virtual machine or a framework.

III. Annotations catalogue

Principles of good software design have been propagated by many researches since 1970s [13].In Table I we present a catalogue of annotations violating the modularity principles found in the literature. We group the violations by the modularity principle being violated. For each group we start by an explaining which principle is violated and how; give an example illustrating the violation; list the violation implications on the system modularity; provide a number of examples of annotations from popular application frameworks [18], [21], [15] violating the principle; propose means of resolving the violation.

We recognize that our concerns for modularity violating annotations mean in particular arguing against the new J2EE persistence model, which rely heavily on annotations (e.g., @Table, @Column, @ManyToMany) and has been widely accepted as very convenient and powerful. However, software design is always a trade-off, in this case, a trade-off between the convenience and power of this model, on the one hand, and possible threads to modularity and maintainability, on the other. Even if the developers prefer the former, they have to be aware of the latter.

We stress that the dependency injection implementation by multiple frameworks is allowing to embed more and more configuration data into the code. Although it seems to facilitate the development process, it comes with the price of hindering maintainability of the resulting code and compromising the spirit of dependency injection, i.e., the separation between object configuration and object implementation [8]. In fact, multiple on-line discussions [5], [10] indicate developers' concerns pertaining to the use of annotations. While the annotation champions stress development and debugging facilitation [21], their opponents voice the considerations with regards to maintainability of the resulting code. Indeed, one can envisage development to be facilitated as an effort spend on unit-testing can be expected to diminish due the top-level wiring of components. It should, however, be noted that modularity-violating annotations can compromise this benefit should the configuration be changed: propagating modification through multiple annotations might become prohibitive. Moreover, mixture of annotations and configuration files incurs consistency keeping burden for developers. To the best of our knowledge, no support for annotation vs. configuration files consistency management is provided by current development environments.

IV. Related work and conclusions

Dependency injection is recognized by many researches (see, e.g. [8], [23]) as an implementation of Dependency Inversion design principle [12] or, as Fowler [8] coined it, the Inversion of Control principle. Despite the industrial popularity of dependency injection, as witnessed by series of implementations and industry-oriented publications [8], [21], it has not deserved much attention inside the research community with exception of [14], [23], [16]. In [14], the authors analyze the impact of dependency injection on software maintainability metrics. After completing the analysis of 20 open source projects, the authors have found no apparent correlation between the use of dependency injection and coupling and cohesion metrics. The authors of [23] provide a classification of four different kinds of dependency injection: constructor and method (setter) dependency injection with and without default implementation of an injected object. They investigate 34 open source projects (including the most popular dependency injection frameworks) counting each kind of dependency injection implemented in those projects. Finally, [16] focuses on interceptors and proposed a reverse engineering mechanism allowing to visualize the intricate interplay of the interceptors as a sequence diagram. Although the three papers do conjecture the possible impact of dependency injection on such software design quality attributes as extensibility and reusability, they do not perform any comprehensive analysis of possible negative effects of applying different dependency injection techniques on software design.

In recent years a variety of tools has been developed helping programmers to identify code "bad smells" and potential bugs. However none of the multitude of Java static analysis tools, with exception of FindBugs [9], provides any support for Java annotations analysis, since annotations are not considered by those tools as part of native Java code. FindBugs provides analysis of Java annotations, both included in Annotations for Software Defect Detection [19] and its own, but not annotations from Java EJB standard specification [18] and not in the context of determining design "bad smells". Another group of tools help identify Java application's design "bad smells" by means of measuring coupling and cohesion metrics. However, modern techniques of dependency injection in Java programs remain out of reach of such tools: they just cannot see injected dependencies in the analyzed code.

In this paper we focused on annotations implementing dependency injection and investigated the impact of uncontrolled usage of such annotations on software modularity. Our contribution consists of comprehensive *catalogue* of "bad smells", i.e., violations of modularity principles. For each violation we provided an example, discussed potential implications and proposed means of resolution. The following modularity principles can be violated:

Configuration should be separated from functionality, i.e., one should avoid annotations implying dependencies on the application server (@Install, @Startup), web server (@Path, @RequestMapping) and database server (@Table, @Column) configurations. *Information should not be duplicated*: this principle is violated by interface annotations mentioning the interface implementation (@ImplementedBy, @ProvidedBy) and annotations duplicating the database structure (@Id, @OneToOne, @OneToMany, @ManyToOne and @ManyToMany). Finally, *interfaces should be explicit*. Java interceptor annotations can violate this principle since interceptor invocations and the interceptor invocation order might be implicit.

We have observed that dependency injection with Java annotations still has to get the necessary attention from the research community. As an important direction of the *future work* we consider studying possible impact of Java and C# annotations and configuration files on software comprehension, maintenance and evolution.

REFERENCES

[1] S.W. Ambler. Mapping Objects To Relational Databases. An AmbySoft Inc. White Paper. 1997.

[2] C.Y. Baldwin and K.B. Clark. Design Rules. vol. 1: The Power of Modularity. The MIT Press, 2000.

[3] C. Bauer and G. King. Hibernate in Action. Manning, 2005.

[4] M. van den Brand, S. Roubtsov, and A. Serebrenik. SQuAVisiT: A Flexible Tool for Visual Software Analytics. *13th European Conference on Soft. Maintenance and Reengineering,* 2009:222-224.

[5] R. Burke. Annotations...Good, Bad, or Worse? http://www.javalobby.org/java/forums/t101604.html Consulted on Sept. 28, 2009.

[6] K. Kaur Chahal and H. Singh. Metrics to study symptoms of bad software designs. ACM SIGSOFT Soft. Eng. Notes 34(1): 1–4 (2009).

[7] M. Fowler. Reducing Coupling. IEEE Soft., 18(4):102–104, 2001.

[8] M. Fowler. Module Assembly. IEEE Soft., 21(2):65–67, 2004.

[9] D. Hovemeyer and W. Pugh. Finding Bugs Is Easy. ACM SIGPLAN Notices. Volume 39 , Issue 12, 2004:92 - 106

[10] Jeff vs. Software. An example of how not to use annotations. http://jeffvssoftware.blogspot.com/2009/ 01/example-of-how-not-to-use-java.html. Consulted on 19 May 2009.

[11] B.P. Lientz, E. Burton Swanson, G.E. Tompkins. Characteristics of Applications Software Maintenance. *CACM* 21(6):466–471 (1978).

[12] R.C. Martin. The Dependency Inversion Principle. Report, 8(6):61-66,1996.

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

[14] E. Razina and D.S. Janzen. Effects of Dependency Injection on Maintainability. *11th IASTED Int. Conf. on Soft. Eng. and Applications*, 2007

[15] Red Hat, Inc. Seam - Contextual Components. A Framework for Java EE 5, 2007.

[16] A. Serebrenik, S. Roubtsov, E. Roubtsova and M. van den Brand. Reverse Engineering Sequence Diagrams for Enterprise Java Beans with Business Method Interceptors. *16th Working Conf. on Reverse Eng.*, IEEE Computer Society 2009.

[17] Sun Microsystems, Inc. Java™2 Platform Standard Edition 5.0. API Specification 2004.

[18] Sun Microsystems, Inc. JSR 220 Enterprise JavaBeans™, Version 3.0 EJB Core Contracts and Requirements, 2006.

[19] Sun Microsystems, Inc. JSR 305: Annotations for Software Defect Detection. http: http://jcp.org/en/jsr/detail?id=305, 2007. Consulted on Sept. 15, 2009.

[20] Sun Microsystems, Inc. RESTful Web Services Developer's Guide, April 2009.

[21] R. Vanbrabant. Google Guice: Agile Lightweight Dependency Injection Framework. APress, 2008.

[22] C. Walls and R. Breidenbach. Spring in Action. Manning Publications, 2007.

[23] H.Y. Yang, E. Tempero and H. Melton. An Empirical Study into Use of Dependency Injection in Java. *19th Australian Conf. on Soft. Eng.* 2008: 239-247.

Modularity principle	Violation	Example	Impact	Other annotations	Resolution
Configuration should be separated from functionality	Dependency on the application server organization: Explicit reference to another component	In presence of @Install(dependencies=XXX,...) the annotated component should be installed only if the component XXX has also been installed.	Redeployment of the software on a new server is hindered by presence of explicit dependencies on other components.	@Startup [15].	Issues related to installation and start up of systems should be addressed in separate application server configuration and build files.

	Dependency on the web-server organization: Explicit reference to a web-page	`@Path("/users/username")`
		`public class UserResource {...}`
		
		The @Path annotation [20] is a relative (with respect to the application context) URI path indicating where the Java class will be hosted.
			Redeployment of the software on a new server is hindered by presence of explicit dependencies on web-pages. Moreover, reorganization of the information storage at the web-server, e.g., replacement of /users/ by /private/ and /business/, requires modification of the software implementation.	@RequestMapping [22].	Association of a web-page with a specific Java class should be recorded in configuration files, known as web-deployment descriptors, e.g., those described in [18].

	Dependency on the database organization: Explicit reference to database tables	`@Table(name="ORDER_TABLE")`
		`public class Order {`
		`  private int id;`
		`  @Id`
		`  @Column(name="ORDER_ID")`
		`  public int getId() {...}`
		`}`
		
		The @Table annotation indicates that the class Order implements the functionality associated with the table ORDER_TABLE.
			Redeployment of the software on a new server is hindered by presence of explicit dependencies on the database tables. Moreover, reorganization of the information storage, e.g., to speed up database querying, requires modification of the functionality implementation.	@Column, @DiscriminatorColumn, @SecondaryTable [18].	These annotations aim at providing an object-relational mapping [1]. This aim, however, does not justify mixing the mapping with the implementation. Such object-relational mapping libraries as Hibernate [3] allow the developers to express the mapping in separate configuration files.

Information should not be duplicated.	Interface annotation mentioning the interface implementation.	@ImplementedBy (see Example 1) associates a class implementing an interface to the interface, i.e., results in information duplication.	Potential inconsistency due to maintenance and an undesirable circular dependency [7]involving the interface and its implementation.	@ProvidedBy [21].	Bindings should be taken care of in a separate configuration class. This class should extend com.google.inject.AbstractModule and provide a protected void method configure, consisting of a series of bindings: bind(Interface).to(Implementation) or bind(Service).toProvider(Provider).
	Database structure duplication	@Id (see above) indicates that the annotated field is or the annotated method allows to retrieve a key of the corresponding table. This information should, however, be present in the database definition itself.	Potential inconsistency due to maintenance.	@OneToOne, @OneToMany, @ManyToOne and @ManyToMany.	Object-relational mapping libraries such as Hibernate [3] provide means of identifying columns as keys by means of an XML tag <id>. Similarly, relations between different entities can be expressed using tags <one-to-one>, <one-to-many>, etc.
Interfaces should be explicit	EJB [18] provides a powerful mechanism for business method invocations and life cycle events interposition, namely, *interceptors*. Some of the calls to interceptors and their call order are implicit.	Interceptors, introduced by @Interceptors and @AroundInvoke, can be regarded as implementing a form of aspect-oriented programming as a part of dependency injection. "Implicit" interceptors result from defaults (provided by @Interceptors annotation at the class level) and inheritance of the interceptor classes.	Unexpected behavior and convoluted invocation order caused by implicit invocations, defaults and inheritance.	@ExcludeDefaultInterceptors, @ExcludeClassInterceptors [18].	Inheritance in combination with interceptors should be used very cautiously as it can easily lead to a convoluted behavior. To clarify the bean-interceptor interfaces reverse engineering [16] might be used. Furthermore, one should extract @AroundInvoke methods in non-interceptor classes (beans) in a separate class; avoid default interceptors and if interceptors are combined with inheritance, then @AroundInvoke methods in interceptor classes inheriting from other classes should override the corresponding methods from their superclasses.

Table I
CLASSIFICATION OF ANNOTATIONS VIOLATING MODULARITY PRINCIPLES