

Catalog of Java Dependency Injection Anti-Patterns

Rodrigo Laigner, Marcos Kalinowski, Diogo Mendonça, Alessandro Garcia

Introduction

This work presents an updated version of the catalog of Java dependency injection (DI) anti-patterns found in our previous work [1, 2], fully embracing practitioners perceptions over the initial version of the catalog. The instances of the catalog were observed and conjectured based on the experience of the author while maintaining software in industry settings and evolved through discussions with researchers and extensive feedback from practitioners.

Each anti-pattern presented provides the following structural elements: a name, description, context where it is applied, drawbacks observed, pattern of occurrence, and resolution. Furthermore, we classify the DI anti-patterns into four different classes of problems: *Architecture*, *Design*, *Performance*, and *Standardization*.

Architecture concerns an architectural violation, such as taking control over the dependency injection process. In other words, these instances are directly related to violating Dependency Inversion (DIP) and Inversion of Control (IoC) principles. *Design* problems are related to the presence of design issues, such as design smells. Such pitfalls contrast with what is pursued in dependency injection, such as strong modularity and low coupling. *Performance* problem concerns impact on memory usage or response time, such as loading unnecessary dependencies. Finally, *Standardization* is related to sticking to a DI coding style, such as following a defined specification (e.g., JSR-330). In addition, each anti-pattern shows the pattern of occurrence and a suggested refactoring procedure separated by a dashed line. A summary of the catalog is exhibited in Table 1 and the catalog is further described along Sections .

Design Anti-Patterns

Concrete class injection

Description: Concrete class injection concerns a dependence requested via dependency injection on which the

Figure 1: Concrete class injection

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

-----
public class ConcreteClassExample
    implements IExampleInterface {
    @Override
    public void doSomething() {
        // code omitted for brevity
    }
}

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

Table 1: Catalog of Java DI Anti-Patterns

Name	Description	Category
Concrete class injection	Reference on concrete class for injection	Design
Open window injection	An injected instance is passed as parameter to another class method or opened for external accessing (e.g., get method)	
Open door injection	An injection request is fulfilled by a DI container, however, the instance is opened for modification by an external element (e.g., set method)	
Fat DI class	Provision of a high number of dependencies to a class, hindering the modularity	
Multiple assigned injection	An injected instance is assigned to multiple attributes (may include external attributes)	
Long Producer method	Method that performs activities that are out of the scope of providing a dependence, which is its main objective	
Static dependence provider	Usage of static fabrics or Service Locator to obtain a dependence	Architecture
Direct container call	Relying directly on the DI container to obtain a dependence	
Intransigent injection	Dependencies that are not needed on construction time, however, are provided by the DI container, introducing additional workload and memory consumption	Performance
Useless injection	Dependency requested via DI that is not used	
Framework coupling	Elements on source code that are dependent on a given DI framework implementation	Standardization
Multiple forms of injection	Refers to the use of multiple forms of injection to a given element, e.g., attribute-and-constructor-level injections	

element type of the dependence is a concrete class.

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

Drawbacks: As a design problem, this anti-pattern yields the following negative consequences: (i) violation of IoC principle, once the class requesting its dependence acknowledges an implementation detail (i.e., the concrete class); (ii) Less flexibility on testing, once a mock object would need to be an inherited class of the given concrete class in order to modify desired behavior; (iii) According to Gamma et al. [3], coupling to a concrete class can increase maintenance efforts.

Pattern of occurrence: A class relies on a concrete class injection to request a dependence. Figure 1 exhibits the class *B* depending on the concrete class *ConcreteExample*, configuring a high coupling.

Resolution: Gamma et al. [3] advocates for programming to an interface, which is a natural solution to this anti-pattern. The resolution shows a code transformation, on which an interface (see *IExampleInterface*) is created so that the class *ConcreteExample* implements it.

Open window injection

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

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

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

Pattern of occurrence: Figure 3 show an example of occurrence, on which the inject element *parser* is passed as parameter to another method.

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

Figure 2: Open door injection

```

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

-----

public class H_Without_Anti_Pattern {
    @Inject private Parser parser;
    // code omitted for brevity
}

```

Open door injection

Description: This anti-pattern is applied when an inject request is provided by a DI container, however, the instance requested is open for modification by an external element. It usually happens when the developer lacks appropriate knowledge about DI.

Context: Any context.

Drawbacks: Open door injection can configure a hard to follow traceability, hindering program comprehension. Also, bugs are another possibility, since concrete implementation is open to chance by an external class.

Pattern of occurrence: Figure 2 depicts the presence of a public set method that allows changing of the injected instance of *parser* in runtime. In details, the example depicts a public set method (*setParser*), which allows for modification of the instance of an injected element (*parser*) by an external class.

Resolution: The resolution concerns the removal of the element on source code (e.g., public set method) that enables changing injected element.

Fat DI class

Description: This anti-pattern concerns the injection of a substantial number of dependencies in a class. This anti-pattern is primarily concerned over injected instances that are often inconsequentially introduced by developers without reasoning over the increased dependence of the class with other components. Thus, it is configured as a design problem.

Context: Along with a substantial number of injected dependencies, the class also shows a multiple set of responsibilities that would be better handled if distributed properly.

Drawbacks: Increased efforts on maintenance tasks operated in the class.

Pattern of occurrence: Figure 6 depicts an excerpt of a class with high level of complexity, in terms of number

Figure 3: Open window injection

```

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

-----

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

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

```

of injected element instances, and number of methods.

Resolution: Figure 6 depicts a refactoring that removes the anti-pattern, dividing dependencies and behavior into different classes. The resolution example depicts a code transformation applied to previous class *D*, on which a refactoring type called *Extract Class* [4] was employed three times in order to reduce the complexity of class *D*.

Multiple assigned injection

Description: This anti-pattern occurs when the reference to an injected instance is spread among multiple attributes.

Context: This anti-pattern is not applied in cases where

Figure 4: Multiple assigned injection

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

```
-----
abstract class GenericBusinessImpl {
    abstract IDAO getGenericDAO();
}
```

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

the class receiving an injection also assigns it to an element of its superclass (which is not managed by the DI container), thus, forcing the assignment to multiple attributes.

Drawbacks: This anti-pattern is correlated to *Open door injection*, since it opens a gap for an undesirable modification of the injected object at run time.

Pattern of occurrence: Figure 4 provides an example of occurrence, where the assignment of an injected instance of *ExampleDAO* to an attribute of a parent class (*GenericBusinessImpl*).

Resolution: In the case of injection instance being assigned to an attribute of super-class, a better approach would be overriding an abstract method. This way, the overridden abstract method would provide the instance injected, not incurring on reference duplication. Thus, the resolution shown in Figure 4 depicts a code transformation that removes the assignment of an injected instance to an attribute presented in a parent class. The removal makes room for an abstract method in the parent class, which still allow the reference to the original injected instance.

Long producer method

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

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

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

Pattern of occurrence: Figure 5 shows a high complex method that should be simple, once the main concern of a Producer method (see *@Produces* annotation) is to provide a given dependency. The DI container, when it identifies the existence of a Producer method for a given type, transfer the responsibility for dependence provision to the Provider method.

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

Figure 5: Long producer method

```

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

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

```

Figure 6: Fat DI class

```

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

-----
public class D_Part_1 {
    @Inject private IExample1 one;
    @Inject private IExample2 two;
    @Inject private IExample3 three;
    void methodOne() { /* code omitted */ }
}

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

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

```

Architectural Anti-Patterns

Static dependence provider

Description: Static dependence providers are related to Fabrics and Service Locators. The first refers to a class that has the objective to provide a given concrete implementation, not being a *Provider* class. On the other side, Service Locator pattern also applies to this context, since it is a class that has the responsibility for serving all dependencies that might be required on run time. Both classes of problem concerns architectural problems, since both violate DIP and IoC principle.

Context: The use of static dependence providers in projects that employ a DI framework where the dependence provision is not related to fulfill a third-party component or a class outside the container.

Drawbacks: It incurs a high coupling to a fabric class in the source code. In case of Service Locator, the dependence on this pattern is even worse due to its widespread usage in the project. Indeed, inversion of control is not achieved in both cases.

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

Resolution: Figure 7 enforces the use of DI container for dependency injection at run time by relying on a *Producer* method in order to provide an instance of *IDataSource*. Particularly, the resolution example above shows a code transformation, in which the logic for creating an instance of *IDataSource* is modularized within a *Producer* method. This way, the class *E_Without_Service_Locator* is not coupled to a service locator class anymore.

Direct container call

Description: Direct container calls can provide a concrete implementation at any point of the system. The nature of this anti-pattern is similar to using a static fabric or a Service Locator. Once DI is chosen as an architectural standard for the project, employing container call for dependence resolution conveys an architectural violation.

Context: The use of static dependence providers in projects that employ a DI framework where the dependence provision is not related to fulfill a third-party component or a class outside the container. In cases where dependencies must be dynamically resolved at runtime with the support of the DI container, this anti-pattern is not applied.

Figure 7: Static dependence provider

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

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

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

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

Figure 8: Direct container call

```

public class F {
    @Inject private Parser parser;
    @Inject private ApplicationContext context;

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

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

```

Figure 9: Intransigent injection

```

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

```

Drawbacks: High coupling to framework specifics, since it relies directly on the framework to provide the dependency. In addition, inversion of control principle is not achieved in this context.

Pattern of occurrence: The Figure 8 shows the class (*E*) with a dependence provision made by a direct container call. In other words, rather than relying on the DI container for injecting an instance of *IDataSource* type on *dataSource* attribute, the code relies on a direct container call.

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

Performance Anti-Patterns

Intransigent Injection

Description: Intransigent injection concerns dependencies that are not needed on construction time, however,

Figure 10: Useless injection

```

public class E {
    @Inject private ExampleType one;
    public void foo() { /* no reference to one */ }
    public void bar() { /* no reference to one */ }
}

-----

public class E_Without_Unused {
    public void foo() {
        // code omitted for brevity
    }
    public void bar() {
        // code omitted for brevity
    }
}

```

they are decidedly provided by the DI container on construction time.

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

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

Pattern of occurrence: Figure 9 shows that the injected attribute *example1* is not used in construction time, thus, the process of injecting a given instance in this attribute might require additional workload to DI container.

Suggested resolution: The resolution concerns the use of a *Provider*, an interface type defined by JSR-330 that allows for obtaining a given dependence when necessary. Thus, the resolution provided in Figure 9 concerns relying on a *Provider* to obtain from the DI container the given instance when its use is requested (see *example1*).

Useless injection

Description: This anti-pattern regards a dependency requested via dependency injection that is actually not used in the class. This way, unused injection is categorized as a performance problem.

Context: Only applied if the unused injection is not used by any inherited class. The problem is more related to current integrated development environments (IDEs) which, once annotated, even though the attributed is not used, do not warn the developer about the issue.

Drawbacks: It overloads the DI container with the incumbency to provide the non used dependency on run

Figure 11: Framework coupling

```

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

-----

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

```

time. Worst case scenario if it is not a lightweight object, or if it is not a singleton scope, impacting on performance.

Pattern of occurrence: The Figure 10 shows a class (*E*) with an injected instance that is not used though any method of the class.

Resolution: The solution concerns removing the unused injection element.

Standardization Anti-Patterns

Framework coupling

Description: It refers to elements on source code that are dependent on a given framework implementation. As the name of the anti-pattern expose, it can be represented as annotations or method calls to framework configuration classes along the source code. We categorize this anti-pattern as part of standardization category.

Context: Avoiding this anti-pattern is better applied in the context of new software projects. In legacy systems, the complexity entailed by removing this anti-pattern may not be worthy the effort.

Drawbacks: In the context of Java, which presents a specification for DI, a framework specific annotation, for example, incurs in high coupling to the framework. In addition, in case where compatibility is a requirement, this anti-pattern can lead to greater effort in maintenance activities, framework change or framework version

Figure 12: Multiple forms of injection

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

-----

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

update.

Pattern of occurrence: Figure 11 depicts a class that employs Spring framework *@Autowired* annotation.

Resolution: A suitable option for removing coupling from a given DI framework is relying on the adoption of annotations presented in the specification.

Multiple forms of injection

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

Context: Any context.

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

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

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

References

[1] Rodrigo Laigner, Marcos Kalinowski, Luiz Carvalho, Diogo S. Mendonça, and Alessandro Garcia. Towards a catalog of java dependency injection anti-patterns. In *Proceedings of the XXXIII Brazilian Symposium on Software Engineering, SBES 2019*,

Salvador, Brazil, September 23-27, pages 104–113, 2019. doi: 10.1145/3350768.3350771.

- [2] Rodrigo Laigner. Cataloging dependency injection anti-patterns in software systems, 2020. PUC-Rio. Master’s dissertation.
- [3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., 1995.
- [4] Diego Cedrim. *Understanding and Improving Batch Refactoring in Software Systems*. PhD thesis, PUC-Rio, 2018.

This document was created on and last updated on April 11, 2021. Source files are at <https://www.overleaf.com/read/xmyythnqhymz>.
